Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AN APPROACH TO DESIGN INCREMENTAL PARALLEL WEBCRAWLER

DIVAKAR YADAV[1], AK SHARMA[2], SONIA SANCHEZ-CUADRADO[3], JORGE MORATO[4]

[1]Assistant Professor, Department of Computer Science & Engg. and IT, JIIT, Noida (India)

[2]Professor and Dean, Department of Computer Science & Engg., YMCA University, Faridabad (India)

[3, 4] Associate Professor, Department of Computer Science & Engg., UC3, Madrid (Spain)

Email: [1]dsy99@rediffmail.com, [2]ashokkale2@rediffmail.com, [3]ssanchec@ie.inf.uc3m.es, [4]jorge@kr.inf.uc3m.es

ABSTRACT

World Wide Web (WWW) is a huge repository of interlinked hypertext documents known as web pages. Users access these hypertext documents via Internet. Since its inception in 1990, WWW has become many folds in size, and now it contains more than 50 billion publicly accessible web documents distributed all over the world on thousands of web servers and still growing at exponential rate. It is very difficult to search information from such a huge collection of WWW as the web pages or documents are not organized as books on shelves in a library, nor are web pages completely catalogued at one central location. Search engine is basic information retrieval tool, used to access information from WWW. In response to the search query provided by users, Search engines use their database to search the relevant documents and produce the result after ranking on the basis of relevance. In fact, the Search engine builds its database, with the help of WebCrawlers. To maximize the download rate and to retrieve the whole or significant portion of the Web, search engines run multiple crawlers in parallel.

Overlapping of downloaded web documents, quality, network bandwidth and refreshing of web documents are the major challenging problems faced by existing parallel WebCrawlers that are addressed in this work. A Multi Threaded (MT) server based novel architecture for incremental parallel web crawler has been designed that helps to reduce overlapping, quality and network bandwidth problems. Additionally, web page change detection methods have been developed to refresh the web document by detecting the structural, presentation and content level changes in web documents. These change detection methods help to detect whether version of a web page, existing at Search engine side has got changed from the one existing at Web server end or not. If it has got changed, the WebCrawler should replace the existing version at Search engine database side to keep its repository up-to-date

Keywords: *World Wide Web (WWW), Uniform Resource Locator (URLs), Search engine, WebCrawler, Checksum, Change detection, Ranking algorithms.*

1. INTRODUCTION

About 20% of the world's population use Web [28] which is increasing exponentially day by day and a large majority thereof uses web search engines to find information. Search engines consist of three major components: indexer, query processor and crawlers. Indexer processes pages, decides which of them to index and builds various data structures (inverted index, web graph etc) representing the pages. Query processor processes user queries and returns matching answers in an order determined by ranking algorithms. Web Crawlers are responsible to maintain indexed database for search engines which are used by the users indirectly. Every time one searches the internet using a service such as Google, Alta Vista, Excite, Lycos etc, he is making use of an index that is based on the output of WebCrawlers. WebCrawlers, also known as spiders, robots, or wanderers are software programs that automatically

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

traverse the web [4]. Search engines use it to find what is on the web. It starts by parsing a specified web page and noting any hypertext links on that page that point to other web pages. They recursively parse those pages for new links.

The size of WWW is enormous and there are no known methods available to find its exact size, but it may be approximated by observing the indexes maintained by key search engines. In 1994, one of the first web search engine, the World Wide Web Worm (WWWW) had an index of 110,000 web pages and web accessible documents [1]. Google, the most popular search engine had 26 million web pages indexed in 1998 that by 2000 reached one billion mark and now it indexes around 50 billion web pages [2]. Similar is the case with other search engines too. So major difficulty for any search engine is to create the repository of high quality web pages and keep it up-to-date. It is not possible in time to create such a huge database by a single WebCrawler because it may take moths or even more time to create it and in mean time large fractions of web pages would have been changed and thus not so useful for end users. To minimize down load time search engines execute multiple crawlers simultaneously known as parallel WebCrawlers [4, 8, 26].

The other major problem associated with any web crawler is refreshing policies used to keep indexes of web pages up to date with its copy maintained at owner's end. Two policies are used for this purpose. First policy is based on fixed frequency and the second on variable frequency [6]. In fixed frequency scheme all web pages are revisited after fixed interval (say once in 15 days) irrespective of how often they change, updating all pages in the collection of WebCrawler whereas in variable frequency scheme, revisit policy is based on how frequently web documents change. The more frequently changing documents are revisited in shorter span of time where as less frequently changing web pages have longer revisit frequency.

The aim of this paper is to propose a design for parallel WebCrawler and change detection techniques for refreshing web documents in variable frequency scheme. The crawling scheme, discussed in this paper for parallel WebCrawler, is such that the important URLs/documents are crawled first and thus the fraction of web that is visited is more meaningful and up-to date. This paper is divided as follows: Section 2 discusses related work on parallel WebCrawlers and page refreshment techniques. Section 3 discusses about the proposed design architecture for parallel WebCrawler. Section 4 discusses about the proposed change detection algorithms and finally section 5 concludes the work along with some future directions followed by references.

2. RELATED WORKS

Though most of the popular search engines nowadays use parallel/distributed crawling schemes but not many literatures are available on it because they generally do not disclose the internal working of their systems.

Junghoo Cho and Hector Garcia-Molina [3] proposed architecture for parallel crawler and discussed fundamental issues related with it. It concentrated mainly on three issues of parallel crawlers namely overlap, quality and communication bandwidth. In [9] P Boldi et all have described about UbiCrawler: a scalable fully distributed web crawler. The main features mentioned are: platform independence, linear scalability, graceful degradation in the presence of faults, an effective assignment function based on consistent hashing for partitioning the domain to crawl and complete decentralization of every task. In [10] authors have discussed about design and implementation of distributed web crawler. In [5] Junghoo Cho et all have discussed about order in which a crawler should visit the URLs in order to obtain more important pages first. This paper defined several important metrics, ordering schemes and performance evaluation measures for this problem. Its results show that a crawler with a good ordering scheme can obtain important pages significantly faster than one without it.

Another major problem associated with web crawler is dynamic nature of web documents. No methods till date are known that may provide the actual change frequencies of web documents because changes in web pages follow Poisson process [7] but few researchers [6, 11-14] have discussed about its dynamic nature and change frequency. According to [27] it takes approximately 6 months for a new page to be indexed by popular search engine. Junghoo Cho and H G Molina [6] have performed some experiments to find the change frequencies among web pages mainly from .com, .netorg, .edu and .gov domains. After observing the web pages from these four domains for 4 continuous months, it was found that web pages from .com domain change at highest frequency and .edu and .gov pages are static in

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

nature. In [14] the authors have discussed analytically as well as experimentally about the effective page refresh policies for web crawlers. According to it the freshness of a local database *S* with *N* elements at time *t* is given as $F(S: t) = M/N$ where *M* (<*N*) are up-to-date elements at time *t*. This paper also proposed a metric, age to calculate how old a database is.

Papers [12, 15-23, 25, 29] discuss about the methods for change detection in HTML and XML documents. In [12] researchers of AT & T and Lucent technologies, Bell laboratories have discussed details about the internet difference search engine (AIDE) that finds and displays changes to pages on World Wide Web. In [15] Ling Liu et all have discussed about WebCQ: a prototype system for large scale web information monitoring and delivery. It consists of four main components: a change detection robot that discovers and detects changes, a proxy cache service that reduces communication traffic to the original information servers, a personalized presentation tool that highlights changes detected by WebCQ sentinels and a change notification service that delivers the fresh information to the right users at right times. Ntoulas [16] collected a historical database for the web by downloading 154 popular web sites (e.g., acm.org, hp.com and oreilly.com) every week from October 2002 until October 2003, for a total of 51 weeks. The experiments show that significant fractions (around 50%) of web pages remain completely unchanged during the entire period of observation. To measure the degree of changes, it computed the shingles of each document and measured the difference of shingles between different versions of web documents. Fretterly [23] performed a large crawl that downloaded about 151 million HTML pages. Then it was attempted to fetch each of these 151 million HTML pages ten more times over a span of ten weeks during Dec. 2002 to Mar. 2003. For each version of documents, checksum and shingles were computed to measure the degree of change. The degree of change was categorized into 6 groups: complete change, large change, medium change, small change, no text change, and no change. Experiments show that about 76% of all pages fall into the groups of no text change and no change. The percentage for the group of small change is around 16% while the percentage for groups of complete change and large change is only 3%. The above results are very supportive to study the change behaviors of web documents. The paper suggests that incremental method may be very effective in updating web indexes, and that searching for new information appearing on the web by retrieving the changes will require a small amount of data processing as compared to the huge size of the web.

3. PROPOSED ARCHITECTURE OF PARALLEL CRAWLER AND CHANGE DETECTION METHODS

The proposed crawler (see Figure 1) has a client server based architecture consisting of following main components:

- Multi Threaded server
- Client crawlers
- Change detection module

The Multi Threaded (MT) server is the main coordinating component of the architecture. On its own, it does not download any web document but manages a connection pool with client machines which actually download the web documents.

The client crawlers collectively refer to all the different instances of client machines interacting with each other through server. The number of clients may vary depending on the availability of resources and the scale of actual implementation.

Change detection module helps to identify whether the target page has changed or not and consequently only the changed documents are stored in the repository in search/insert fashion. Thus the repository is kept up-to-date having fresh and latest information available at Search engine database end.

3.1 Multi Threaded (MT) Server

The Multi Threaded server is the main coordinating component of the proposed architecture (see Figure 2), directly involved in interaction with client processes ensuring that there is no need for direct communication among them.

The sub-components of MT server are:

- URL dispatcher
- Ranking module
- URL distributor
- URL allocator
- Indexer
- Repository

3.1.1 Url Dispatcher

The URL dispatcher is initiated by seed URLs. In the beginning, both unsorted as well as sorted URL queues [see Figure 2] are empty and the seed URL received from user is stored in sorted URLs queue and a message called distribute_URLs is sent to URL

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

distributor. URL distributor selects the seed URL from the sorted queue and puts it in first priority list (P_list 1) as shown in Figure 2. URL allocator picks the URL from priority list and assigns it to client crawler to download web document. After downloading the document, the client crawler parses it to extract the embedded URLs within it and stores the web document and corresponding URLs in document and URL buffer. Every URL, retrieved by the URL dispatcher from document and URL buffer, is verified from repository before putting it in the unsorted queue, to know whether it has already been downloaded or not. If it is found that the URL is already downloaded and the corresponding document exists in the repository then the retrieved URL is discarded and not stored in unsorted queue. By doing this, URL dispatcher ensures that the same URL is not used multiple times to download its corresponding document from WWW and thus it helps to save network bandwidth. URL dispatcher keeps all new URLs in the queue of unsorted URLs in the order they are retrieved from the buffer and sends rank_URL message to ranking module. This process is repeated till the queue of sorted URLs becomes empty.

3.1.2 Ranking Module

After receiving Rank_URLs message from the dispatcher, the ranking module retrieves URLs from the unsorted queue and computes their priority. For computing priority of the URLs to be crawled, it considers both their forward as well as back link counts where forward link count is the number of URLs present in the web page, pointing to other web pages of WWW and back link count is the number of URLs from search engine's local repository pointing to this URL. The forward link count is computed at the time of parsing of the web pages. However, to estimate the number of back link count, the server refers to its existing database and checks as to how many pages of current database refer to this page. Once the forward and back link counts are known, the priority (Pvalue) is computed using the following developed formula.

$$Pvalue = FwdLk - BkLk \quad \text{---------- (1)}$$

Where, Pvalue = priority value
FwdLk = forward link counts
BkLk = back link counts

URLs having higher difference between FwdLk and BkLk are assigned higher Pvalue. Initially, the forward link count holds higher weightage as there are no or very few URLs pointing to current URLs, as the database is still in a nascent stage but as the database of indexed web pages grows, the weightage of back link count gradually increases. This results in more number of URLs having lower Pvalue and thus more URLs holding lower priority. This is important since it is not desired to assign high priority to a page which is already downloaded and update it very frequently as indicated by the high BkLk value but we do want to prioritize addition of new pages to our repository of indexed pages, which has a nil BkLk, but high FwdLk as it leads to a higher number of pages. A similar ranking method is discussed in [5] in which only back link counts are considered. After calculating priority, the ranking module sorts the list in descending order of priority, stores them in the sorted queue and sends signal to URLs distributor.

This method is particularly useful as it also gives weightage to current database and builds a quality database of indexed web pages even when the focus is to crawl whole of the Web. It works efficiently in both, when the database is growing or in maturity stage. Also, the method works well for broken links i.e. the URLs having zero value for forward link. Even if it is referred from pages in the database, priority will always be negative resulting in low priority value.

The advantage of above ranking method over others is that it does not require the image of the entire Web to know the relevance of an URL as "forward link counts" are directly computed from the downloaded web pages where as "back link counts" are obtained from in-built repository.

3.1.3 Url Distributor

URLs distributor retrieves URLs from the sorted queue maintained by the ranking module. Sometimes, a situation may arise wherein highly referenced URL with a low forward link counts, may always find itself at the bottom in the sorted queue of URLs. To get rid of such situations, the URLs list is divided into three almost equal parts on the basis of priority in such a way that top one-third of the URLs are sent to p_list1, middle one-third are sent to p_list2 and bottom one-third are sent to p_list3 from where the URL allocator assign them to the respective client crawlers, as shown in Figure 2.

For example, if the queue of unsorted URLs contains URLs list as shown in Table 1 then after calculating the ranking of URLs, same are divided in three lists as shown in Table 2. From the Table 2 one

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

can see that the number of back link count is zero for all URLs as no links from the local repository of search engines are pointing to them, being the repository in initial state and is almost empty.

Table 1: Unsorted URLs list received from client crawlers

URL	Forward Link count	Back link count	Priority value (Pvalue)
http://www.jiit.ac.in	20	0	20
http://www.jiit.ac.in/jiit/files/RTI.htm	18	0	18
http://www.jiit.ac.in/jiit/files/department.htm	11	0	11
http://www.jiit.ac.in/jiit/files/PROJ_SPONS.htm	4	0	4
http://www.jiit.ac.in/jiit/files/curri_obj.htm	3	0	3
http://www.google.co.in/	40	0	40
http://www.rediffmail.com	79	0	79
http://www.gmail.com	65	0	65
http://www.yahoo.com	155	0	155
http://www.ugc.ac.in/	60	0	60
http://www.ugc.ac.in/orgn/regional_offices.html	21	0	21
http://www.ugc.ac.in/policy/fac_dev.html	13	0	13
http://www.ugc.ac.in/policy/payorder.html	52	0	52
http://www.ugc.ac.in/policy/modelcurr.html	60	0	60
http://www.ugc.ac.in/inside/uni.html	15	0	15
http://www.ugc.ac.in/contact/index.html	5	0	5
http://www.ugc.ac.in/inside/fakealerts.html	16	0	16
http://www.ugc.ac.in/orgn/directory.html	53	0	53

Table 2: URLs after sorting on basis of Pvalue and assigned in respective lists

URL	Forward Link count	Back link count	Pvalue	P_list
http://www.yahoo.com	155	0	155	1
http://www.rediffmail.com	79	0	79	1
http://www.gmail.com	65	0	65	1
http://www.ugc.ac.in/	60	0	60	1
http://www.ugc.ac.in/policy/modelcurr.html	60	0	60	1
http://www.ugc.ac.in/policy/payorder.html	52	0	52	1
http://www.ugc.ac.in/orgn/directory.html	53	0	53	2
http://www.google.co.in/	40	0	40	2
http://www.ugc.ac.in/orgn/regional_offices.html	21	0	21	2
http://www.jiit.ac.in	20	0	20	2
http://www.jiit.ac.in/jiit/files/RTI.htm	18	0	18	2
http://www.ugc.ac.in/inside/fakealerts.html	16	0	16	2
http://www.ugc.ac.in/inside/uni.html	15	0	15	3
http://www.ugc.ac.in/policy/fac_dev.html	13	0	13	3
http://www.jiit.ac.in/jiit/files/department.htm	11	0	11	3
http://www.ugc.ac.in/contact/index.html	5	0	5	3
http://www.jiit.ac.in/jiit/files/PROJ_SPONS.htm	4	0	4	3
http://www.jiit.ac.in/jiit/files/curri_obj.htm	3	0	3	3

3.1.4 Url Allocator

The basic function of URL allocator is to select appropriate URLs from sub-lists (P_list1, P_list 2, P_list3) and assign them to client crawlers for downloading the respective web documents. According to the ranking algorithm discussed above, P_liast1 contains higher relevant URLs but we cannot completely ignore the P_list2 & P_list3. P_list 2 contains the URLs which either belongs to a web document having higher forward and backward link counts or both forward link count and back link count are less for these URLs. Similarly P_list3 consists of mostly those URLs which have higher back link count value and less forward link count. Therefore, to avoid complete ignorance for URLs present in these both lists (P_list1 & P_list2), the strategy used is such that for every 4 URLs selected from p-list1, 2 URLs from p-list2 and 1 URL from p-list3 are assigned to client crawlers for downloading.

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

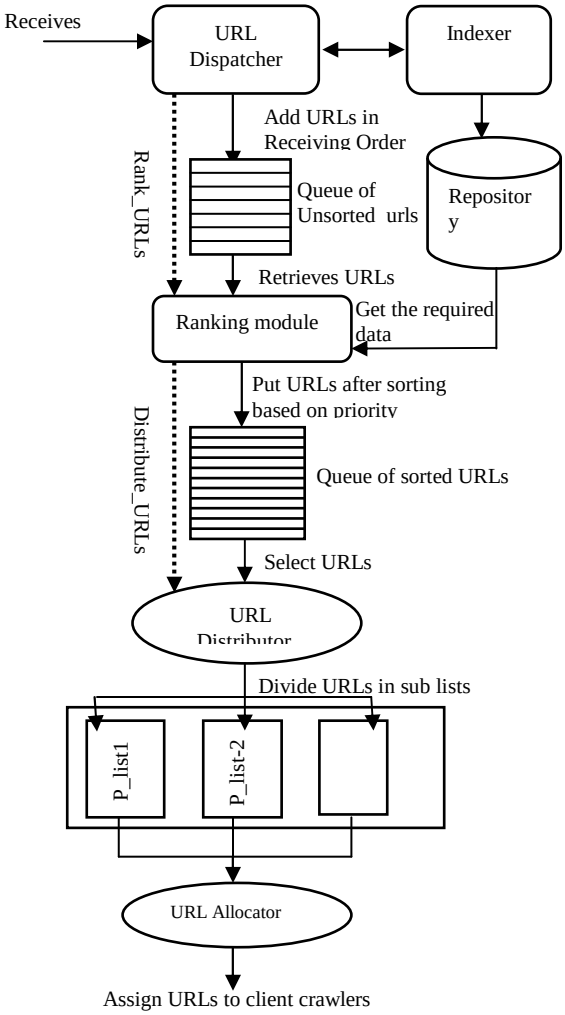


Figure 2: Architecture of Multi threaded server

3.1.5 Indexer

Indexer module retrieves web documents and corresponding URLs from document and URL buffer. The documents are stored in repository in search/insert fashion and corresponding index is created for them. The indexes created for the documents mainly consist of keywords present in the document, address of document in repository, corresponding URL through which the web document was downloaded, checksum values and so many other information. Later, all these indexed information for web documents help to produce appropriate result when users fire search query using search engine as well as to detect whether the current downloaded version is same or changed from its previous version existing in the repository.

3.1.6 Repository

It is centrally indexed database of web pages, maintained by server which later may be used by search engines to answer the queries of end users. The incremental parallel crawler maintains index of complete web documents in the repository along with other information. The efficiency by which the repository is managed affects the search results. As users need most relevant results for a given search query the repository and corresponding information maintained for these documents play a major role apart from ranking algorithm to produce the result.

3.2 Client Crawlers

Client crawlers collectively refer to all the different client machines interacting with the server. As mentioned earlier, there are no inter client crawler communications, all types of communications are between the MT server and an individual client crawler. In Figure 1 these client crawlers are represented as C-crawler 1, C-crawler 2 and so on. The detailed architecture of a client crawler is shown in Figure 3.

These client crawlers are involved in actual downloading of the web documents. They depend on MT server (URL allocator) for receiving URLs for which they are supposed to download the web pages. After downloading the web documents for the assigned URLs, a client crawler parses and extracts all URLs present in it and puts them back to document and URL buffer from where they are extracted by change detection module as well as URL dispatcher. After performing preprocessing on the received URLs, they are assigned to client crawlers for further downloading. This whole process continues till no more URLs to be crawled are left.

Following are the main components of a client crawler:

- URL buffer
- Crawl Worker

3.2.1 Url Buffer

It stores URLs temporarily which are assigned to the client crawler by URL allocator for downloading the corresponding web documents. This is implemented as simple FIFO (first in first out) queue. The crawl worker retrieves URLs from this queue and starts its downloading process.

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

3.2.2 Crawl Worker

Crawl worker is major component of the client crawler in proposed architecture of incremental parallel web crawler. Working of the whole process of the crawl worker may be represented by the following algorithm:

```
Crawl_worker ()
{
Start
  Repeat
    Pickup a URL from the URL buffer;
    Determine the IP address for the host name;
    Download the Robot.txt file which carries
    downloading permissions and also specifies
    the files to be excluded by the crawler;
    Determine the protocol of underlying host like
    http, ftp, gopher etc;
    Based on the protocol of the host, download
    the document;
    Identify the document format like .doc, .html,
    or .pdf etc;
    Parse the downloaded document and extract
    the links;
    Convert the URL links into their absolute
    URL address;
    Add the URLs and downloaded document in
    "document and URL buffer";
  Until not empty (URL buffer);
End
  }
```

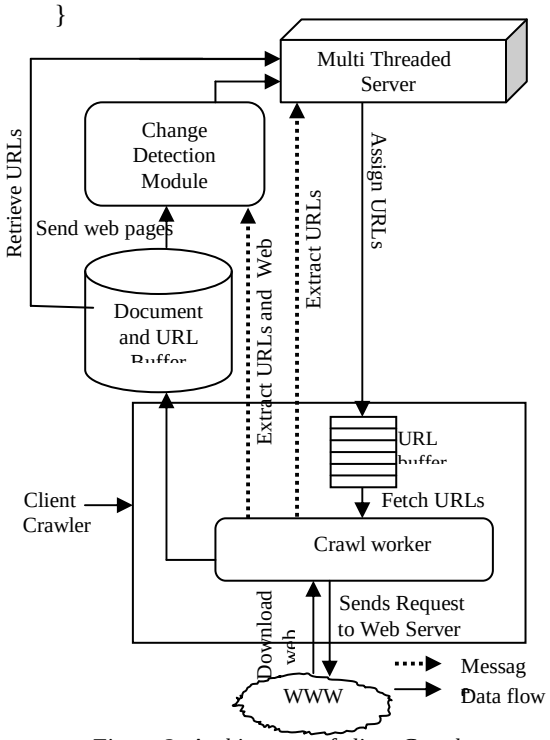


Figure 3: Architecture of client Crawler

After downloading the web documents for assigned URLs, the crawl worker parses its contents with the help of parser for the following further applications:

- Each page has a certain number of links present in it. To maintain the index of back link count, each link on that page is stored in repository along with its source/parent URL in which it appears. The client crawler sends the pair of values (link, parent_URL) to the repository/database. When the same link reappears on some other page, only the name of parent URL is required to be added to the link indexer to the already existing value of the link.
- The crawl worker also extracts all the links present on a web page to compute the number of forward links counts which it sends to the URL dispatcher which are later used by ranking module to compute relevance of uncrawled URLs based on which they are further redistributed among client crawlers for further downloading as discussed in section 3.1.2 to 3.1.4.
- Another motive behind parsing the contents of web pages is to compute the page updating parameters such as checksum etc. that are used by change detection module to check whether the web page has been changed or not.

After parsing, all downloaded web documents are stored in document and URL buffer associated with MT server. After keeping the documents in the document and URL buffer, crawl worker sends extract URLs & web page message to change detection and extract URLs message to MT server (URL dispatcher). After receiving the message, the change detection module and URL dispatcher, extract the web documents along with URLs and other relevant information from the buffer.

The client crawler machines are robust enough to be able to handle all types of web pages appearing on the web and to handle the pages, which are not allowed to be crawled [4]. It also automatically discards URLs that are referenced but do not exist any more on WWW.

3.3 Change Detection Module

The change detection module identifies whether two versions of a web document are same or not and thus helps to decide whether the existing web page in the repository should be replaced with changed one or not. Methods for detecting structural and content level changes in web documents have been proposed. There may be other types of changes also like

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

behavioral, presentation etc. It may not always be required to test both types of changes at the same time. The latter one i.e. content level change detection may be performed only when there are no changes detected by structural change detection methods or the changes detected are minor. The details about proposed methods for change detection are being discussed in the section 4.

3.4 Comparison of Proposed Architecture with Existing Architecture of Parallel WebCrawler

The architecture has been implemented as well as tested for about 2.5 million web pages from various domains. The results obtained thereof establish the fact that the proposed incremental parallel web crawler does not have problem of overlapping and also downloaded pages are of high quality, thereby, proving the efficiency of ranking method developed. The performance of proposed incremental parallel web crawler is compared with that of existing parallel crawlers [3]. The summary of the comparison is as shown in Table 3. The proposed web crawler's performance was found to be comparable with [3].

Table 3: comparison with existing parallel crawler architecture

Features	Existing Parallel Crawler	Incremental parallel web crawlers
Central Coordinator	No central server. It works in distributed environment.	It has central coordinator in the form of MT server as it works on client server principle.
Overlapping	Have overlapping problem associated as individual crawler do not have global image of downloaded web documents.	As MT server assigns the URLs to be crawled to client crawlers and it has global image of crawled/uncrawled URLs and thus, significantly reduces the overlapping.
Quality of Web pages	Being distributed coordination, it may not be aware about others collection, so it decreases the quality of web pages downloaded.	URLs are provided by server to client crawlers on the basis of priority so high priorities pages are downloaded and thus high quality documents are available at Search engine side.
Priority calculation	The crawlers compute priority of pages to be downloaded based on local repository which may be different if computed on global	In our approach all web pages after downloading are sent back to server along with embedded URLs, the priority is computed based on global
	structure of web pages	information.
Scalability	Architecture is scalable as per requirement and resources	It may also be scaled up depending upon resource constraints
Network bandwidth	Due to problem of overlapping more networks bandwidth is consumed in this architecture.	Due to reduction in overlapping problem, network bandwidth is also reduced at the cost of communication between MT server and client crawlers.

4. CHANGE DETECTION MODULE

In this section, novel mechanisms are being proposed that determine whether a web document has been changed from its previous version or not and by what amount. The hallmarks of the proposed mechanisms are that changes may also be known at micro level too i.e. paragraph level. Following 4 types of changes may take place in a web documents [24]:

1. Structural Changes
2. Content level or semantic changes
3. Presentation or cosmetic changes
4. Behavioral changes

Sometimes the structure of web page is changed by addition/deletion of tags. Similarly, addition/deletion/modification in the link structure can also change the overall structure of the document (see Figure 4).

Content level or semantic changes refer to the situation where the page's contents are changed from reader's point of view (see Figure 5).

```
<html><body> <p><b>---------- --</b> </p>
<p><big>----------</big></p>
<p><i>----------
<ul><li>----------</li>
<li> ----------</li></ul>
<u>----------</u>
</i></p></body></html>
```

(a) *Initial version*

```
<html> <body>
<p><b><font ---------->----------</font> </b> </p>
<p><big>----------</big></p>
<p><b>
----------
-------
</b></p>
```

(*b) Changed version*

Figure 4: Structural changes in versions of a web page

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
<html> <head> <title> this is page title </title> </head>
<body> <center>Times of India News 19-Oct-2009
</center>
<ul><li>Delhi, Hurriyat talks can't succeed without us: Pak
<li>China wants better Indo-Pak ties, denies interference
<li>Ties with China not at India's expense: US
<li>Sachin slams 43rd Test ton; completes 30,000 runs</ul>
```

(a)

```
<html> <head> <title> this is page title </title> </head>
<body> <center>Times of India News 19-Oct-2009
</center>
 <ul>
<li>India-Lanka 1st Test ends in a draw
<li>Maya asked to increase Rahul's security
<li>Ties with China not at India's expense: US
<li>Sachin slams 43rd Test ton; completes 30,000 runs
</ul>
```

(b)

Figure 5: Content/semantic changes (a) Initial version (b) Changed version of a web page

Under the category of presentation or cosmetic changes only the document's appearance is modified whereas the contents within document remain intact. For example, by changing HTML tags, the appearance of document can change without altering its contents (see Figure 6).

```
<html> <body>
<p>India-Lanka 1st Test ends in a draw </p>
<p>Maya asked to increase Rahul's security
</p>
<p> Ties with China not at India's expense: US
</body> </html>
```

(a)

```
<html><body>
<p style="background-color:#00FF00">
<u>India-Lanka 1st Test ends in a draw</u>
</p>
<p style="background-color: rgb(255,255,0)">
<u>Maya asked to increase Rahul's security</u></p>
<p style="background-color:yellow">
<u>Ties with China not at India's expense: US
</u></p>
```

(b)

Figure 6: Presentation/cosmetic changes (a) Initial Version (b) Changed version

Behavioral changes refer to modifications to the active components present in a document. For example, web pages may contain scripts, applets etc as active components. When such hidden components change, the behavior of the document gets changed. However, it is difficult to catch such changes especially when the codes of these active components are hidden in other files.

So, in order to keep the repository of search engines up-to-date it is mandatory to detect the changes that take place in web documents. In the following sections, mechanisms are being proposed that currently identify content and structural changes. The structural change detection methods are also efficient to detect the presentation level changes as these changes occur due to insertion/modification/deletion of tags in the web documents.

4.1 Proposed Methods for Change Detection

This section is divided in two parts: first part discusses the mechanisms that detect structural changes whereas second part discusses the content level change detection mechanisms. It may be noted that content level change detection may be performed only when either there are no changes at structural level or the changes at structural level are minute, thereby saving computational time.

4.1.1 Methods for Detecting Structural Changes

Following two separate methods have been designed for the detection of structural changes occurring within a document:

- Document tree based structural change detection method, and
- Document fingerprint based structural change detection method.

4.1.1.1 Document Tree Based Structural Change Detection Method

This method works in two steps. In the first step document tree is generated for the downloaded web page while in the second step, level by level comparison between the trees is performed. The downloaded page is stored in the repository in search/insert fashion as given below:

Step 1: Search the downloaded document in the repository.

2. If the document is found then compare both the versions of the document for structural changes using level by level comparison of their respective document tree.

3. Else store the document in the repository.

Generation of the tree is based on the nested structures of tags present in the web page. After parsing the downloaded web page, parser extracts all the tags present in it. Then tags are arranged in the order of their hierarchical relationship and finally document tree is generated with the help of tree generator. All

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the tags which are nested at the same level in the web page should be at the same level in document tree too. Each node in the document tree, representing a tag, consist many fields which keep various information about the tag. The structure of a node of the document tree is as given below:

Tag_name	Child	Level_no	No_of_siblings

Where:

- *Tag_name:* This field of node structure stores the name of tags.
- *Child:* This field contains information about the children of each node.
- *Level_no:* This field contains the level number at which the nodes appear in the constructed document tree.
- *No_of_siblings:* This field in a node structure contains total number of nodes present at that level.

For example, consider the document tree given in Figure 8, generated for the initial version of a web page as shown in Figure 7 whose tag structure is shown in Figure 11 (a). Let's assume that later on some structural changes occurred in the web page as shown in Figure 9 whose tag structure is shown in Figure 11 (b) and the resultant document tree for the changed web page is as shown in Figure 10. A level by level comparison of the two trees is carried out and the result obtained thereof has been tabulated in Table 4. The Table contains a listing of number of nodes/siblings at different levels. From Table 4, it can be concluded that number of nodes have changed at level 3, 4 and 5 indicating that the document has changed.

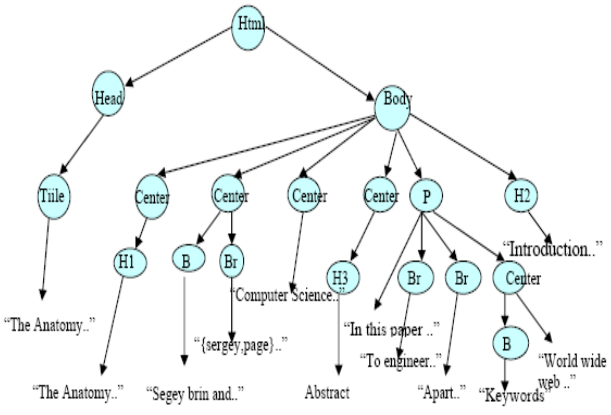
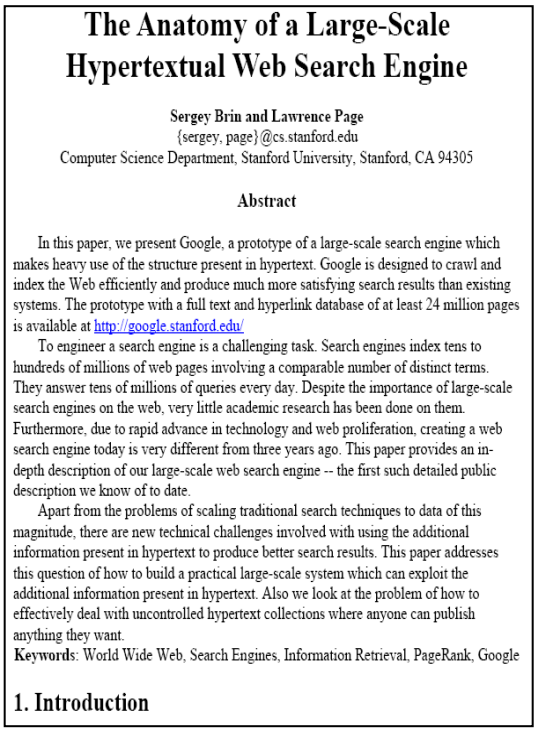


Figure 8: Document tree for initial version of web page



The Anatomy of a Large-Scale Hypertextual Web Search Engine

Sergey Brin and Lawrence Page
{sergey, page}@cs.stanford.edu
Computer Science Department, Stanford University, Stanford, CA 94305

Abstract

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/

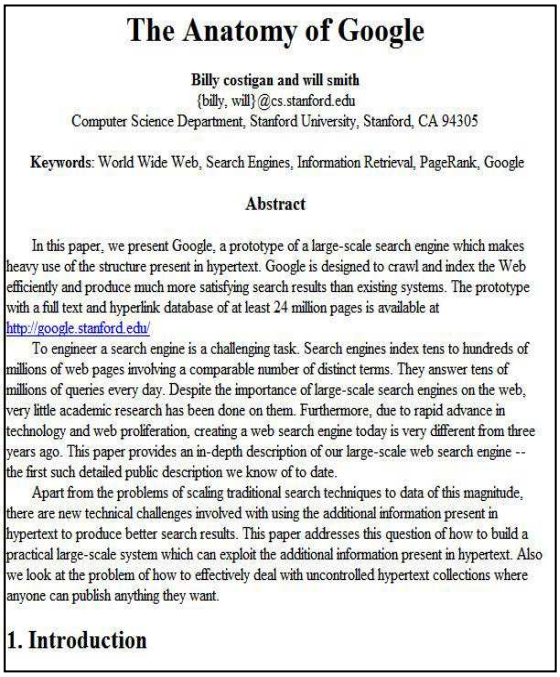
To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and web proliferation, creating a web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale web search engine -- the first such detailed public description we know of to date.

Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want.

Keywords: World Wide Web, Search Engines, Information Retrieval, PageRank, Google

1. Introduction

Figure 7: Initial version of web page



The Anatomy of Google

Billy costigan and will smith
{billy, will}@cs.stanford.edu
Computer Science Department, Stanford University, Stanford, CA 94305

Keywords: World Wide Web, Search Engines, Information Retrieval, PageRank, Google

Abstract

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext. Google is designed to crawl and index the Web efficiently and produce much more satisfying search results than existing systems. The prototype with a full text and hyperlink database of at least 24 million pages is available at http://google.stanford.edu/

To engineer a search engine is a challenging task. Search engines index tens to hundreds of millions of web pages involving a comparable number of distinct terms. They answer tens of millions of queries every day. Despite the importance of large-scale search engines on the web, very little academic research has been done on them. Furthermore, due to rapid advance in technology and web proliferation, creating a web search engine today is very different from three years ago. This paper provides an in-depth description of our large-scale web search engine -- the first such detailed public description we know of to date.

Apart from the problems of scaling traditional search techniques to data of this magnitude, there are new technical challenges involved with using the additional information present in hypertext to produce better search results. This paper addresses this question of how to build a practical large-scale system which can exploit the additional information present in hypertext. Also we look at the problem of how to effectively deal with uncontrolled hypertext collections where anyone can publish anything they want.

1. Introduction

Figure 9: Changed version of web page

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

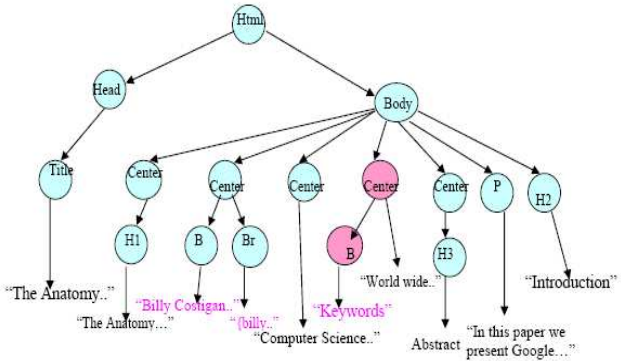


Figure 10: Document tree for changed version of web page

```
<Html> <Head><Title>----------</Title> </Head>
<Body> <Center><h1>----------</h1> </Center>
<Center><b>----------</b> </Center>
<Center>----------</Center>
<Center> <h3>----------</h3> </Center>
<p>----------
<br>----------
<br>----------
<Center><b>----------</b>----------</Center
>
</p> <h2>----------</h2>
</Body> </Html>
```

(a)

```
<Html> <Head><Title>----------</title>
</head> <Body> <Center><h1>----------</h1>
</Center>
<Center><b>----------</b> </Center>
<Center>----------<br>----------</Cente
r>
<Center><b>----------</b>----------</Cente
r>
<Center><h3>----------</h3></Center>
```

(b)

Figure 11: HTML tag structure of (a) Initial version (b) Changed version of web pages

Table 4: Level structure using BFS for above initial and modified tree

Level_no	No. of siblings in initial web page	No. of siblings in changed web page
1	1	1
2	2	2
3	7	8
4	7	5
5	1	0

For further details about the changes, such as how many tags have been added/deleted and at what level in the document tree etc, the remaining fields of the node structure may be used. The details generated for the document trees (Figure 8 & Figure 10) are as shown in Table 5. After analyzing the document tree method for structural change detection many inferences as tabulated in Table 6 are drawn.

Table 5: Node wise attribute details of the initial and the modified tree using level order traversing

Attributes	For initial version	For changed version
Level_no	1, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5	1, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4
Tag_name	Html, Head, Body, Title, Center, Center, Center, Center, P, H2, H1, B, Br, H3, Br, Br, Center, B	Html, Head, Body, Title, Center, Center, Center, Center, Center, P, H2, H1, B, Br, B, H3
Child	2, 1, 6, null, 1, 2, null, 1, 3, null, null, null, null, null, null, null, null,1, null	2, 1, 7, null, 1, 2, null, 1, 1, null, null, null, null, null, null, null

Table 6: Inferences on structural changes

Change detected	Inference drawn
No sibling added/deleted at any level	No structural change between the two version of the document
Leaf siblings added/deleted	Minor changes between the two version of the document
Siblings having children added/deleted	Major change has occurred*

*This detection helps in locating the area of major change within documents

The document tree based structural change detection method is efficient and guarantees to detect the structural changes perfectly if the constructed tree represents the true hierarchical relationship among tags.

4.1.1.2 Document Fingerprint Method for Structural Change Detection

This method generates two separate fingerprints in the form of strings for each web document based on its structure. To generate the fingerprint, all opening tags present in the web page are arranged in the order of their appearance in the web document whereas all closing tags are discarded. The first fingerprint generated, contains the set of characters appearing at first position in the tag in the order they appear in the web page whereas the second fingerprint contains the characters appearing at last position in the tag for all tags in the order they appear in the web page. For single character tags, same characters are repeated in both the fingerprints.

For example, if the initial version of a web page is as shown in Figure 12 and later it gets changed as shown in Figure 13, the tag structures of the two

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

versions are as shown in Figure 14 and Figure 15 respectively. Applying the above scheme, Table 7 is generated. Comparing the *fingerprint1* of the both versions it may be concluded that the web page gets changed as its fingerprint gets changed. The comparison of *fingerprint2* of both web pages, are required to add surety to the method as the comparison for fingerprint1 may fail in the unlikely case of tags starting with some character being replaced with another tags starting with the same character.

Figure 12: Initial version of web page

Figure 13: Changed version of web page

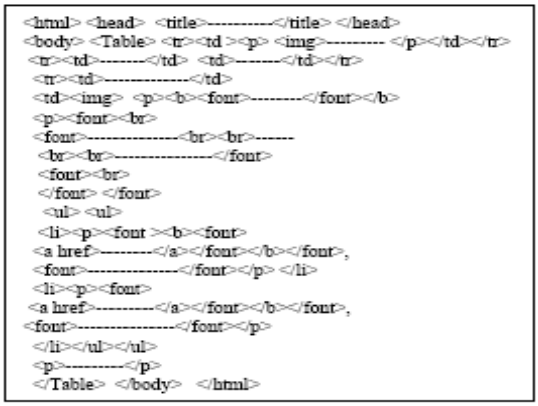


Figure 14: Tag structure for initial version

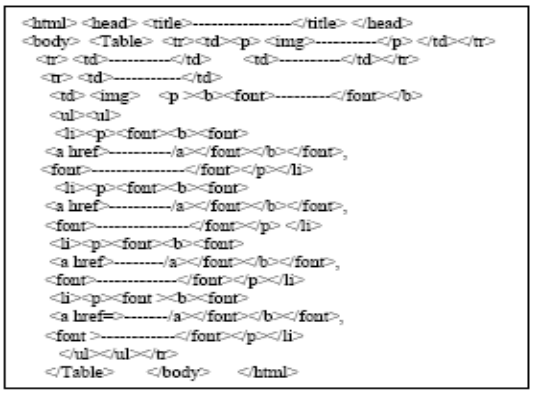


Figure 15: Tag structure for changed version

Table 7: Fingerprints for versions of web page using document fingerprint method

Version	Fingerprint1	Fingerprint2
Initial	hhtbtttpitttttttipbfpfbfbbbbfbuulpfbfaflpfafp	ldeyerdpgrddrddgpbtptrtrrrrtrlliptbtatiptatp
Change	hhtbtttpitttttttipbfuulpfbfaflpfbfaflpfbfaflpfbfaf	ldeyerdpgrddrddgpbtlliptbfatiptbtatipfbtatiptbtat

4.1.1.3 Comparison between Document Tree Based and Fingerprint Based Structural Change Detection Methods

Though both methods discussed for detecting *structural changes* perform efficiently to detect the changes even at minor level but based on certain parameters such as time/space complexity etc, their performance is compared as given in Table 8.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 8: Comparison between document and fingerprint based methods

S. No.	Document tree based method	Document Fingerprint based method
1	This method performs well and is able to detect structural changes even at the minute level.	This method is also able to detect changes even at the minute level.
2	Special attention is required to handle the presence of optional tags in the web document as it becomes difficult to maintain the nested structure among tags.	No effects of optional tags as only opening tags are considered whereas all closing tags are discarded. The fingerprint generated in the form of string contains information about all opening tags in their order of appearance in the document.
3	The presence of misaligned tag structures in the web document may cause problem while establishing the true hierarchical relationship among tag structures.	This method also suffers from the presence of misaligned tag structure as it may produce different fingerprint for the same structure.
4	Time as well as space complexity of document tree based method is higher. To create and compare two trees it consumes more time as compared to creation and comparison of fingerprints. As each node of the tree contains many fields for keeping various information about the tags so it requires more storage space also for document tree.	This method is better in terms of both, time as well as space complexity as compared to document tree based method. Fingerprints are in the form of character strings which require less storage space.
5	This method helps in locating the area of minor/major changes within a document.	With fingerprint based method, it is difficult to locate the area of major/minor change within a document.

4.1.2 Methods for Detecting Content Level Changes

The content level change detection mechanism may be carried out only after the methods as discussed in 4.1.1 for structural changes do not detect any changes. The content level changes can not be captured by the methods discussed for structural level changes. Following two different methods are being proposed for identifying content level changes:

- Root Mean Square (RMS) based content level change detection method, and
- Checksum based content level change detection method.

4.1.2.1 Root Mean Square Based Content Level Change Detection Method

This method computes Root Mean Square (RMS) value as checksum for the entire web page contents as well as for its various paragraphs appearing in the page. While updating the page in the repository, comparison between the checksums of both versions of web pages is performed and in case of an anomaly, it is concluded that the web page at web server end has been modified as compared to the local copy maintained in the repository at search engine end. Thus the document needs to be updated. Paragraph checksums help to detect changes at micro level and help in identifying the locality of changes within the document.

The following formula has been developed to calculate the RMS value.

$$R.M.S = \left[\frac{(a1)^2 + (a2)^2 + \ldots\ldots\ldots\ldots + (a'n)^2}{n} \right]^{1/2} \ldots\ldots\ldots \quad (2)$$

Where a_1, a_2, ----------, a_n are the ASCII code of symbols and n is the number of distinct symbols/characters excluding the white space present in the web page.

Consider the two versions of web page from *Times of India* news site (*http://timesofindia.indiatimes.com/*), taken within three hours of interval, , dated on 28-10-11 as shown in Figure 16 and 17 respectively. Their page checksum (RMS value) using the above method was computed as shown in Table 9. From the table, it can be concluded that the content of both versions of the web page had been changed. The web page, in continuation, was monitored many times from 28-10-11 to 29-10-11 and the data is recorded in Table 10. Due to space constraint it is not possible to include all the monitored web pages.

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 9: Checksum values for two version of web page

Web Page	RMS value of entire page	Distinct symbol counts	Para-graphs	Distinct symbol counts	Paragraph RMS
Initial version	161.176	43	P1	33	355.779
			P2	29	346.451
			P3	27	325.086
			P4	32	261.167
			P5	26	154.898
			P6	24	168.383
Changed version	131.122	41	P1	33	356.838
			P2	29	317.343
			P3	25	291.005
			P4	27	168.39
			P5	25	177.651

Figure 16: Initial version of the web page

Figure 17: changed version of the web page

Table 10: Checksum Table for different versions of web page

V. No	RMS of page	Distinct symbol counts	Para-graphs	Distinct symbol counts	Paragraph RMS
1	161.176	43	P1	33	355.779
			P2	29	346.451
			P3	27	325.086
			P4	32	261.167
			P5	26	154.898
			P6	24	168.383
2	131.122	41	P1	33	356.838
			P2	29	317.343
			P3	25	291.005
			P4	27	168.39
			P5	25	177.651
3	126.053	41	P1	32	381.304
			P2	31	288.686
			P3	31	266.965
			P4	27	160.276
			P5	24	163.306

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4	129.4951	40	P1	30	364.99
			P2	27	341.317
			P3	28	260.205
			P4	27	166.285
			P5	24	162.152
5	121.0349	35	P1	30	364.55
			P2	27	264.664
			P3	24	272.749
			P4	26	168.091
			P5	26	140.293
6	116.56	36	P1	30	346.89
			P2	29	284.38
			P3	26	242.87
			P4	28	142.487
			P5	25	148.874
7	116.21	36	P1	29	337.37
			P2	29	284.38
			P3	26	242.97
			P4	28	142.487
			P5	25	148.87

Based on the data recorded in Table 10, following inferences are drawn about the RMS method for content level change detection:

- It produces unique checksum (RMS value) for entire web page as well as for paragraph level contents of a web document.
- The formula developed is capable to detect minor changes even addition/deletion of few words accurately.
- Paragraph checksum helps to identify changes at smaller level i.e. at paragraph level.
- It may be noted that ASCII values have been used in the formula as each symbol has a unique representation in ASCII Table and thus leading to no ambiguity.

Though the above method performs efficiently and guarantees to detect *content level changes* among different versions of web pages but it assigns uniform weightage to the entire contents of the web document whereas in real life changes in some contents carry more weightage than other. For example changes in main headings of a web document carry more weightage than any other content change. So keeping in mind these points the modified checksum based *content level change* detection method is being proposed.

4.1.2.2 Checksum Based Content Level Change Detection Method

Similar to the previous method, checksum based content level change detection also produces a single checksum for entire web page as well as for each paragraph present in the web document. Paragraphs level checksum help to know the changes at micro level i.e. paragraph level. By comparing the checksums of different versions of a web page, it is known whether the web page content has been changed or not.

The following formula has been developed to calculate the checksum.

$$Checksum = \sum I\_parameter * ASCII\ code * K\text{-}factor \text{ ----- (3)}$$

Where: *I_parameter* = importance parameter,
ASCII code = ASCII code of each symbol (excluding white space)
K-factor = scaling factor

In the implementation, it was font size which was considered *I_parameter*. The purpose of introducing scaling factor is to control the size of checksum value. In this work scaling factor was considered as .0001.

The above method was tested on number of web documents online as well as offline. One such example is shown as in Figure 18 and Figure 19 for two versions of a web document. The checksums for entire page as well as for different paragraphs present within document using the above method was computed as shown in Table 11. From the result shown in the table, it is clear that different checksums are produced if the versions of a web document are not same. By monitoring the paragraphs checksum, it is clear that the changes occurred only in the first paragraph (P1) whereas rests are unchanged and the same is reflected from paragraph checksum recorded in the table.

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

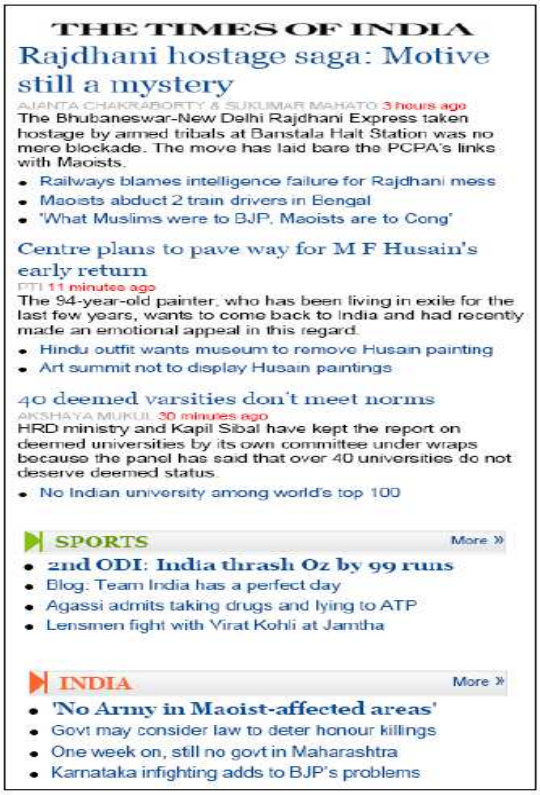


Figure 18: Initial version of web page

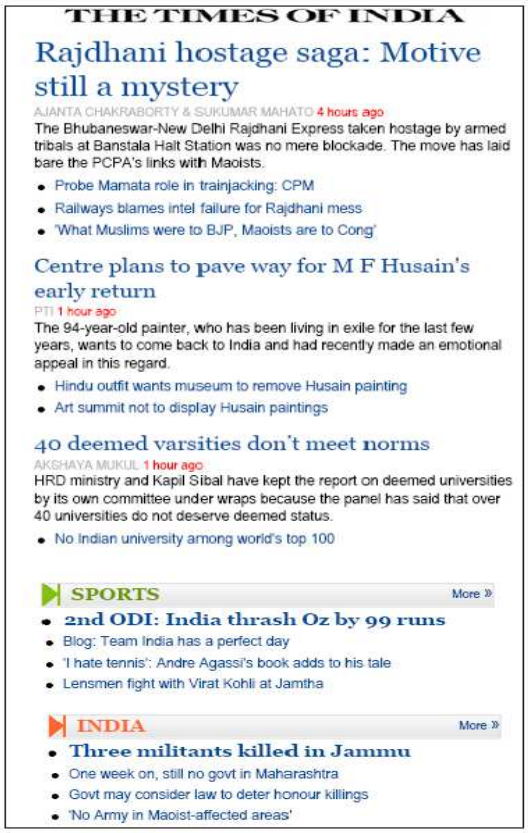


Figure 19: Changed version of the web page

Table 11: Checksum values for web page

Web page	Checksum of page	No of distinct symbols	Para-graph	No of distinct symbols	Paragraph Checksum
Initial version	586.806	36	p1	30	339.439
			p2	29	318.777
			p3	26	312.316
			p4	28	238.419
			p5	25	256.380
Changed version	585.671	36	p1	29	340.564
			p2	29	318.777
			p3	26	312.316
			p4	27	238.419
			p5	24	256.380

This method was also tested on the same set of web pages on which previous (RMS based) method was tested and the results produced are tabulated in Table 12. Monitoring the checksum results recorded in the table and looking at the corresponding contents of the web pages, it may be concluded that the method is efficient and guarantees to detect the content level changes even if there are minor changes among different versions of the web document.

Table 12: Checksum values for web page

V. No	Checksum of page	No of distinct symbols	Para-graph	No of distinct symbols	Checksum
1	625.616	43	p1	33	330.995
			p2	29	339.490
			p3	27	357.407
			p4	32	295.606
			p5	26	258.498
			p6	24	280.597
2	586.280	41	p1	33	330.914
			p2	29	339.495
			p3	25	351.344
			p4	26	262.969
			p5	25	282.021
3	587.837	41	p1	32	353.926
			p2	31	321.996
			p3	31	309.317
			p4	26	262.963
			p5	24	282.439

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4	588.829	40	p1	30	346.411
			p2	27	366.823
			p3	28	312.733
			p4	27	263.958
			p5	23	274.968
5	607.874	35	p1	30	346.317
			p2	27	318.672
			p3	24	345.231
			p4	26	270.575
			p5	26	247.605
6	586.806	36	p1	30	339.439
			p2	29	318.777
			p3	26	312.316
			p4	28	238.419
			p5	25	256.381
7	585.671	36	p1	29	340.564
			p2	29	318.777
			p3	26	312.316
			p4	27	238.419
			p5	24	256.381

Both, RMS based and checksum based methods have been compared with well known message digest algorithms such as MD5 and SHA-X[1], used to calculate checksum for web pages search engine's web crawlers. One such result for an input set [31] is shown in Table 13 and the comparison is made in Table 14. On seeing the table, it can be observed that with MD5 and SHA-X[1] algorithm, the entire checksum gets changed at large scale even though a single character/symbol of the input gets modified, whereas the checksum produced by the *RMS* and *checksum based* methods, do not get changed at that large scale on the modification of single character/symbol in the input set. Even on the modification of few white spaces, the entire message digest gets changed in MD5 and SHA-X[1] algorithm whereas it does not affect the checksum produced by *RMS based* and *checksum based* method as white spaces are not considered while calculating the checksum.

Table 13: Comparison of results with well known message digests algorithms

Name Of Method	Length of message digest (bits)	Initial Input	Message digest for initial input	Changed Input	Message digest for changed input
MD5	128	The quick brown fox jumps over the lazy dog	9e107d9d372bb6826bd81d3542a419d6	The quick brown fox jumps over the lazy cog.	e4d909c90d0fb1ca068ffaddf22cbd0
SHA-1	160	The quick brown fox jumps over the lazy dog	2fd4e1c67a2d28fced849ee1bb76e7391b93eb12	The quick brown fox jumps over the lazy cog	de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3
SHA-224	224	The quick brown fox jumps over the lazy dog	730e109bd7a8a32b1cb9d9a09aa2325d2430587ddbc0c38bad911525	The quick brown fox jumps over the lazy cog	fee755f44a55f20fb3362cdc3c493615b3cb574ed95ce610ee5b1e9b
SHA-256	256	The quick brown fox jumps over the lazy dog	d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592	The quick brown fox jumps over the lazy cog	e4c4d8f3bf76b692de791a173e05321150f7a345b46484fe427f6acc7ecc81be
SHA-512	512	The quick brown fox jumps over the lazy dog	07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d785436bbb642e93a252a954f23912547d1e8a3b5ed6e1bfd7097821233fa0538f3db854fee6	The quick brown fox jumps over the lazy cog	3eeee1d0e11733ef152a6c29503b3ae20c4f1f3cda4cb26f1bc1a41f91c7fe4ab3bd86494049e201c4bd5155f31ecb7a3c8606843c4cc8dfcab7da11c8ae5045
Root Mean Square Based method	No fixed length message digest	The quick brown fox jumps over the lazy dog	127.329	The quick brown fox jumps over the lazy cog	129.820
Checksum Based method	No fixed length message digest	The quick brown fox jumps over the lazy dog	43.356	The quick brown fox jumps over the lazy cog	43.344

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 14: Performance comparison of proposed methods with well known message digests methods

S No	MD5/SHA-1,224,256,512	Methods developed
1	These message digest algorithms produce fixed length checksum codes which are larger in size.	It does not produce fixed length checksum and also the length of checksum produced are smaller
2.	It considers whole contents present in the input. E.g. even white spaces are counted in input data. So even on insertion/deletion of white spaces, the checksum entirely gets changed.	The contents such as those which are enclosed within tags as well as white spaces are not considered.
3	There is restriction on the input size of data e.g. in MD5 input data should not be greater than 2^{64} bits. Similar is the case with others also.	There is no restriction on the input size of data.
4	These methods are very sensitive. The complete checksum generated gets changed even on changing of single characters in the I/P set. So no conclusion can be drawn about the amount of change on the basis of checksum.	Only little changes occur in the final checksum if input data gets changed at micro level i.e. by few characters. On seeing the checksum, the amount of change in two versions of documents may be estimated..
5	It is more suitable for cryptographic application.	It is not suitable for cryptographic application but suitable for change detection.
6	They generate single checksum for whole web page. So no idea may be drawn about changes at micro level i.e. paragraph level within documents.	They generate checksum for whole page as well as for individual paragraphs present in the web page. So with these methods, changes at micro level too i.e. paragraph level may be identified too.

In brief, the methods proposed for structural as well as content level change detection may be summarized as follows:

- Though document tree based structural change detection method efficiently identifies the structural changes but its time and space complexity is high. If nesting of tag structures is misaligned then it becomes very difficult to handle the situation while constructing the document tree. Though some mechanism has been proposed [30] to handle the misaligned tags but its performance is still questionable.
- There is no impact of optional as well as misaligned tags in the performance of *document fingerprint based* method for *structural change* detection, as it considers the opening tags only and discards all closing tags. Its performance is also better in terms of space and time complexity as it requires less space to store the fingerprint which is in the form of string, than to store document tree. Generation and comparison of fingerprint is much easier than tree.

The change detection module was integrated in the incremental parallel web crawler architecture as shown in Figure 1. It was observed that the methods discussed above for both structural and contextual changes are performing efficiently and guarantee to detect the changes. We also tested the above method offline separately on individual web pages and observed the similar performance.

5. CONCLUSION AND FUTURE WORKS

5.1 Conclusion

In this paper, the complete work done is divided in to two parts. In the first part a novel architecture for incremental parallel crawlers has been proposed whereas in second part methods have been developed which detect whether two versions of a web document have been changed thereby helping to refresh the web documents to keep the repository up-to-date at Search engines side.

The novel architecture proposed for *incremental parallel web crawler*, helps to solve the following challenging problems which are still faced by almost every Search engine while running multiple crawlers in parallel for downloading the web documents for its repository.

- Overlapping of web documents
- Quality of downloaded web documents
- Network bandwidth/traffic

5.1.1 Overlapping of Web Documents

Overlap problem occurs when multiple crawlers running in parallel download the same web document multiple times due to the reason that one web crawler may not be aware of another having already downloaded the page.

In the proposed architecture the entire downloading process of web documents is

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

performed under the coordination of *Multi Threaded server* and therefore no URLs have been assigned simultaneously to more than one client crawler executing in parallel. By applying this approach, server has the global image of the entire downloaded web documents and thus, overlapping problem is reduced.

5.1.2 Quality of Downloaded Web Documents

The quality of downloaded documents can be ensured only when web pages of high relevance are downloaded by the crawlers. Therefore, to download such relevant web pages by earliest, multiple crawlers running in parallel must have global image of collectively downloaded web pages.

In the proposed architecture the ranking algorithm developed, computes the relevance of the URLs to be downloaded on the basis of global image of the collectively downloaded web documents. It computes relevance on the basis of the forward link count as well as back link count whereas the existing method is only based on back link count. The advantage of the proposed ranking method over existing one is that it does not require the image of the entire Web to know the relevance of a URL as "forward link count" is directly computed from the downloaded web pages where as "back link count" is obtained from in-built repository.

5.1.3 Network Bandwidth/Traffic

In order to maintain the quality, the crawling process is carried out using either of the following approaches.

- Crawlers can be generously allowed to communicate among themselves or
- They can not be allowed to communicate among themselves at all.

In the first approach network traffic will increase because crawlers communicate among themselves more frequently to reduce the overlap problem whereas in second approach, if they are not allowed at all to communicate then as a result same web document may be downloaded multiple times thereby consuming the network bandwidth. Thus, both approaches put extra burden on the network traffic.

The proposed architecture helps to reduce the overlapping and because all communications take place through MT Server, there is no direct communication requirement among client crawlers. Both of the above facilities help in reducing network traffic significantly.

5.1.4 Change detection Methods for Refreshing Web Documents

In the second part of the work, the change detection methods have been developed to refresh the web document by detecting the structural, presentation and content level changes.

The structural change detection methods also help in detecting presentation changes as well, as the presentation of web documents gets modified through insertion/deletion/modification of the tag structure.

Two different schemes, *document tree based* and *document fingerprint based* have been developed for *structural change* detection. The *document tree based* scheme works efficiently and guarantees to detect structural changes. Apart from providing details about the structural changes within documents, it also helps in locating the area of major/minor change as well. For instance, the information about at what level how many nodes have been inserted/deleted is also reported. In *document fingerprint* based scheme, two separate fingerprints in the form of strings are generated on the basis of web pagetag structure. Time and space complexity in document tree based scheme is higher than the *fingerprint based* scheme. Generation and comparison of fingerprints in the form of strings, require less time than to generate and compare two trees. Also, the space required to store the node details (as node consists of multiple fields) in the document tree is high in comparison to space required to store fingerprints. Though fingerprint based scheme is efficient in terms of space and time complexity, it also guarantees to detect structural changes even at micro level but the details about the changes can not be reported, for which document tree based scheme is better.

Similar to structural change detection scheme, two different schemes, *Root Mean Square* based and *Checksum based* have been developed for content level change detection. Both schemes are based on ASCII principle of symbols and both generate checksum for entire web page and also for different paragraphs. The checksum for entire web page helps to draw the picture about content level changes at page level whereas through paragraph checksum, changes at micro level i.e. paragraph level can be detected. Though, both the schemes are efficient and guarantee to detect changes at micro level also but the former scheme considers changes in contents uniformly whereas the later scheme assigns different

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

weightage to different contents present in the web page based on font size of the contents.

5.2 Future Work

The work done in this paper can be extended with the following list of possible future research issues with respect to incremental parallel web crawler:

- Behavioral changes have not been discussed and have been left for future work.
- Ideally, changes in the link of an image hyperlink can be detected through the structural change detection methods discussed but in case the image itself is replaced or modified then that can not be detected using the methods discussed.
- The architecture along with change detection methods can be synchronized with the frequency of change of web documents to get better results, as web documents from various domains change at different intervals.

REFERENCES:

[1] Brin, Sergey, and Page, Lawrence, "*The Anatomy of a large scale hyper textual web Search Engine*", In Proceedings of the Seventh World-Wide Web Conference, 1998.

[2] Maurice de Kunder, "*Size of the World Wide Web*", available at: http://www.worldwidewebsize.com (accessed May 10, 2012).

[3] Cho, Junghoo, and Garcia-Molina Hector, "*Parallel Crawlers*", Proceedings of the 11th international conference on World Wide Web WWW '02", Honolulu, Hawaii, USA. ACM Press, pp.124 – 135, 2002.

[4] Robots exclusion protocol. Available at: http://info.webcrawler.com/mak/projects/robots/exclusion.html

[5] Cho, Junghoo, Garcia-Molina, Hector, and Page Lawrence, "*Efficient crawling through URL ordering*", Proceedings of the 7th World-Wide Web Conference, pp. 161-172, 1998.

[6] Cho, Junghoo, Angeles, Los, and Garcia-Molina, Hector, "*Effective Page Refresh Policies for Web Crawlers*", ACM Transactions on Database Systems, Volume 28, Issue 4, pp. 390 – 426, December 2003.

[7] Jones, George, "*15 World-Widening Years*", InformationWeek, Sept 18, 2006 Issue.

[8] Berners-Lee, Tim, "*The World Wide Web: Past, Present and Future*", MIT USA, Aug 1996, available at: http://www.w3.org/People/Berners-Lee/1996/ppf.html.

[9] Berners-Lee, Tim, and Cailliau, CN, R., "*WorldWideWeb: Proposal for a Hypertext Project*" CERN October 1990, available at: http://www.w3.org/Proposal.html.

[10] Berners-Lee, Tim, Cailliau, CN, R., Groff, J-F, and Pollermann, B., CERN, "*World-Wide Web: The Information Universe*", published in Electronic Networking: Research, Applications and Policy, Vol. 2 No 1, Meckler Publishing, Westport, CT, USA, 1992.

[11] Berners-Lee, Tim, and Cailliau, CN, R., "*World Wide Web*" Invited talk at the conference: Computing in High Energy Physics 92, France, 23027, Sept 1992.

[12] Berners-Lee, Tim, Cailliau, CN, R., Groff, J-F, and Pollermann, B., CERN, "*World-Wide Web: An Information Infrastructure for High-Energy Physics*", Proceeding of Artificial Intelligence and Software Engineering for High Energy Physics, La Londe, France, published by World Scientific, Singapore, January 1992.

[13] CHO, A., J., Garcia-Molina Hector, Paepcke, A., and Raghvan, S., "*Searching the Web*". ACM Transactions on Internet Technology, Vol. 1, No. 1, pp. 2–43, August 2001.

[14] Bar-Yossef, Z., Berg, A., Chien, S., and Weitz, J. F. D., "*Approximating aggregate queries about web pages via random walks*", In Proceedings of the 26th International Conference on Very Large Data Bases, 2000.

[15] Bharat, K. and Border, A., "*Mirror, mirror on the web: A study of host pairs with replicated content*", In Proceedings of the Eighth International Conference on the World-Wide Web, 1999.

[16] Bharat, K., Border, A., Henzinger, M., Kumar, P., and Venkatasubramanian, S., "*The connectivity server: fast access to linkage information on the Web*", Computer Network ISDN Syst. 30, 1-7, 469–477, 1998.

[17] Lawrence, S. and Giles, C., "*Searching the World Wide Web*", Science 280,98–100, 1998.

[18] Douglas C. Engelbart, "*Augmenting Human Intellect: A Conceptual Framework*". Summary Report AFOSR-3223, October 1962.

[19] Smith, L.S. and Hurson, A., R., "*A Search Engine Selection Methodology*", Proceeding of the International Conference on Information Technology: Computers and Communications (ITCC'03), pp.122 – 129, April 2003.

[20] Sharma, A.K., Gupta, J.P., Agarwal D.P., "*Augmented Hypertext Documents suitable for*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

parallel crawlers", Proceeding Of WITSA-2003, a National workshop on Information Technology Services and Applications, New Delhi, Feb 2003.

[21] Saeid, Asadi and Hamid, R. Jamali, "*Shifts in Search and Future Engine Development: A Review of Past, Present Trends in Research on Search Engines*", Webology, Volume 1, Number 2, December, 2004.

[22]Page, Lawrence; Brin, Sergey; Motwani, Rajeev, and Winograd, Terry, "*The PageRank Citation Ranking: Bringing Order to the Web*", Technical Report, Stanford University InfoLab, 1999.

[23] Cho, Junghoo, Garcia-Molina, Hector, "*The Evolution of the Web and Implications for an Incremental Crawler*", Proceedings of the 26th International Conference on Very Large Data Bases, pp. 200 – 209, 2000.

[24] Francisco-Revilla, L., Shipman, F., Furuta, R., Karadkar, U. and Arora, A., "*Managing Change on the Web*", In Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries, pp. 67 – 76, 2001.

[25] P. De Bra, G.-J. Houben, Y. Kornatzky, and R. Post, "*Information retrieval in distributed hypertexts*", Proceedings of RIAO'94, Intelligent Multimedia, Information Retrieval Systems and Management, New York, NY, 1994.

[26] Hersovici, M., Jacovi, M., Maarek, Y., Pelleg, D., Shtalheim, M. and Ur Sigalit, "*The Shark-Search Algorithm – an application: tailored web site mapping*", Computer Networks and ISDN systems, Special Issue on 7th WWW conference, Brisbane, Australia, 30(1-7), 1998.

[27] Salton, G. and McGill, M.J., "*Introduction to Modern Information Retrieval*", Computer Series. McGraw-Hill, New York, NY, 1983.

[28] Pinkerton, B., "*Finding what people want: Experiences with the WebCrawler*", Proceeding of the First World Wide Web Conference, Geneva, Switzerland, 1994.

[29] Heydon, A. and Najork, M., "*Mercator: A scalable, extensible Web crawler*", World Wide Web, 2(4):pp. 219–229, 1999.

[30] Wang, Ziyang, "Incremental Web Search: Tracking Changes in the Web", PhD dissertation, 2007.

[31] "MD5, SHA-1, 224, 256 etc", available at: http://en.wikipedia.org/wiki/MD5 .

Journal of Theoretical and Applied Information Technology
15 September 2012. Vol. 43 No.1

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

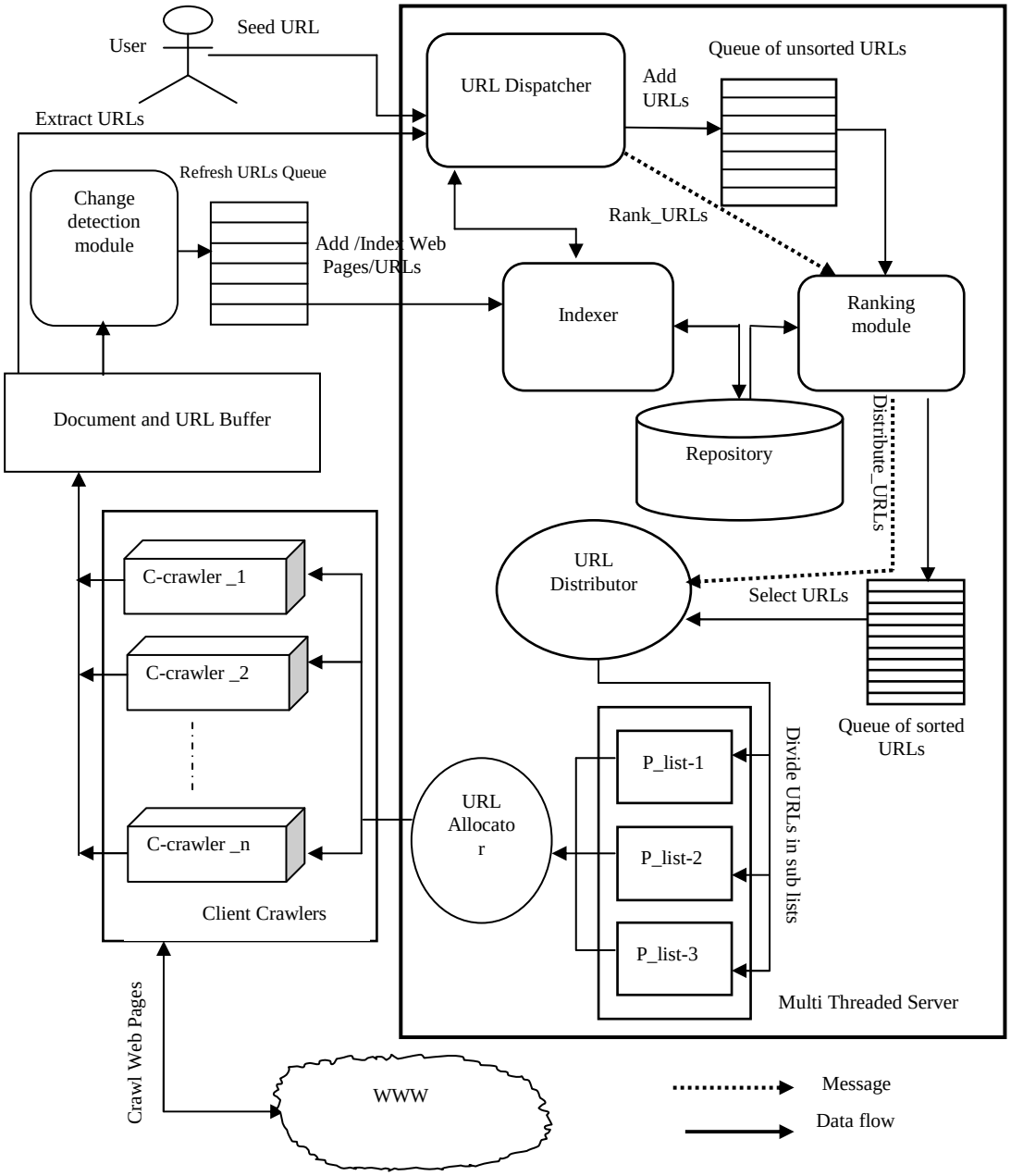


Figure 1: Architecture of Incremental Parallel WebCrawler