Self Adjusting Refresh Time Based Architecture for Incremental Web Crawler

A.K. Sharma[1], *Ashutosh Dixit*[2]

Abstract— Due to the deficiency in their refresh techniques [12], current crawlers add unnecessary traffic to the already overloaded Internet. Moreover there exist no certain ways to verify whether a document has been updated or not. In this paper, an efficient approach is being proposed for building an effective incremental web crawler [13]. It selectively updates its database and/ or local collection of web pages instead of periodically refreshing the collection in batch mode thereby improving the "freshness" of the collection significantly and bringing new pages in more timely manner. It also detects web pages which frequently undergo up-dation and dynamically calculates the refresh time of the page for its next update.

Index Terms - World Wide Web, Search engine, Incremental Crawler, Hypertext, Browser.

I. INTRODUCTION

The World Wide Web [1, 3, 5] is a global, large repository of text documents, images, multimedia and many other items of information, referred to as information resources. It is estimated that www contains more than 2000 billion visible pages and five times more lying in the hidden web. Due to the extremely large number of pages present on Web, the search engine depends upon crawlers for the collection of required pages [2]. The general architecture of a crawler is shown in Fig.1.

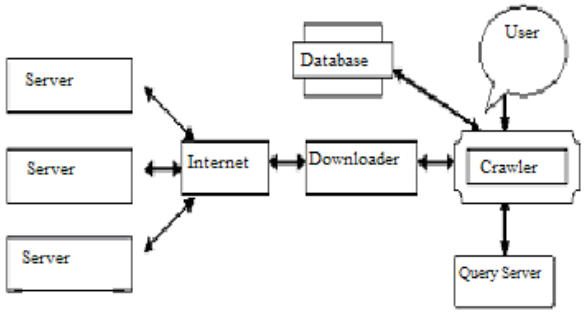

Fig.1: Architecture of a Typical Crawler

1. Prof. and Head Computer Engineering Deptt. YMCA Institute of Engineering Faridabad Haryana India.
2. Lecturere Computer Engineering Deptt. YMCA Institute of Engineering Faridabad Haryana India.

Manuscript received December 5, 2008
Manuscript revised December 20, 2008

In order to download a document, the crawler picks up its seed URL, and depending on the host protocol, downloads the document from the web server. For instance when a user accesses an HTML page using its URL, the documents is transferred to the requesting machine using Hyper Text Transfer Protocol (HTTP) [4, 5, 7]. The browser parses the document and makes it available to the user. Roughly, a crawler starts off by placing an initial set of URLs, in a queue, where all URLs to be retrieved are kept and prioritized. From this queue, the crawler extracts a URL, downloads the page, extracts URLs from the downloaded pages, and places the new URLs in the queue. This process is repeated and the collected pages are later used by other applications, such as a Web search engine.

The algorithm of the typical crawler is given below:

1: Read a URL from the set of seed URLs.
2: Determine the IP address for the host name.
3: Download the Robot.txt file which carries downloading permissions and also specifies the files to be excluded by the crawler.
4: Determine the protocol of underlying host like http, ftp, gopher etc.
5: Based on the protocol of the host, download the document.
6: Identify the document format like doc, html, or pdf etc.
7: Check whether the document has already been downloaded or not.
8: If the document is fresh one
Then
Read it and extract the links or references to the other cites from that documents.
Else Continue;
9: Convert the URL links into their absolute IP equivalents.
10: Add the URLs to set of seed URLs.

II. LITRATURE SURVEY

With the exponential growth of the information stored on www, a high performance crawling strategy based on following design issues is need to be addressed.

i. Keep the local collection fresh

Freshness of a collection can vary depending on the strategy used [6]. Thus, the crawler should use the best policies to keep pages fresh. This includes adjusting the revisit frequency for a page based on its estimated change frequency..

ii. Improve quality of the local collection

The crawler should increase the quality of the local collection by replacing less important pages with more important ones [6]. This updation is necessary for two reasons- firstly pages are constantly created and removed. Some of the new pages can be more important than existing pages in the collection, so the crawler should replace the old and less important pages with the new and more important pages. Second, the importance of existing pages changes over time. When some of the existing pages become less important than previously ignored pages, the crawler should replace the existing pages with the previously ignored pages.

A. INCREMENTAL CRAWLER

In order to refresh its collection, a traditional crawler periodically replaces the old documents with the newly downloaded documents. On the contrary, based upon the estimate as to how often pages change, an incremental crawler [18] incrementally refreshes the existing collection of pages by visiting them frequently. It also replaces less important pages by new and more important pages.

The architecture of a simple incremental crawler is shown in Fig.2.

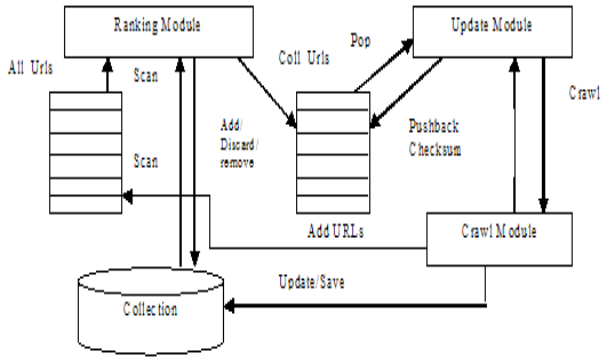

Fig.2: Architecture of Incremental Crawler

Architecture is composed of the following Modules/Data Structures:

All_Urls: A set of all URLs known

Coll_Urls: A set of URLs in the local collection. (it is assumed that this list is full from the beginning)

Collection: Documents corresponding to the URLs of Coll_Urls list.

Ranking Module: It constantly scans through the All_Urls and Coll_Urls to make the refinement decisions.

Update Module: It takes a url from the top of the Coll_Urls and decides whether the document corresponding to that URL has been refreshed or not.

Crawl Module: Add URLs to All_Urls and updates the Collection.

Based on some previous history or estimate the ranking module picks a URL from All_Urls list and assigns rank to it. The ranked URLs are placed on the list called Coll_Urls. The update module picks the URLs from Coll_Urls list in the order of their rank and crawl/ updates the corresponding document. Links contained in the fresh downloaded documents are added in to the All_Urls list and so on.

B. MANAGEMENT OF VOLATILE INFORMATION

When the information contained in a document changes very frequently, the crawler downloads the document as often as possible and updates it into its database so that fresh information could be maintained for the potential users. If the number of such documents is large, the crawling process becomes hopelessly slow and inefficient because it puts a tremendous pressure on the internet traffic.

In order to reduce the traffic on the web and increase the efficiency of the crawler a novel approach for managing the volatile information was introduced [19]. The technique introduces volatile information tags in HTML documents to store the changing information also called as volatile information. While storing the document at server side the *Vol#* Tags are extracted from the document along with their associated volatile information. The Tags and the Information are stored separately in a file having same name but with different extension (*.TVI*) [10]. The *TVI* (Table of variable information) file is updated every time the changes are made to the hypertext document. This file containing the changed contents of a document is substantially smaller in size as compared to the whole document. PARCAHYD [8, 9] downloads the TVI files to incrementally refresh its collection of pages.

A critical look at the available literature indicates that for the purpose of refreshing the document, at present, the frequency of visit by the existing crawler for a particular domain is fixed. Whereas, the frequency is important because the frequency of the visit to a site is directly proportional to the relevance of the site depending upon whether it houses more of dynamic pages or static pages, i.e. higher the frequency, higher the traffic.

In this paper an alternate approach for optimizing the frequency of visits to sites is being proposed. The proposed approach adjusts the frequency of visit by dynamically assigning a priority to a site. A mechanism for computing the dynamic priority for any site has been developed.

III. PROPOSED WORK

Based upon up-dation activity, documents can be categorized as follows.

(i). A static web page, which is not updated regularly.

(ii). Dynamically generated web pages e.g. database driven web page.

(iii). Very frequently updated parts of web pages e.g. News Website, Share Market website.

(iv). Updated pages generated when website administrator updates or modifies its website.

Keeping in view the above categorization, it is proposed that crawler may visit a site frequently and after every visit its frequency of future visits may be adjusted according to the category of the site. The adjusted refresh rate/frequency can be computed by the following formula:

$$t_{n+1} = t_n + \Delta t \qquad \ldots eq^n (1)$$

Where t_n: is current refresh time for any site.
t_{n+1}: is adjusted refresh time.
Δt: is change in refresh time calculated dynamically.

The value of Δt may be positive or negative, based upon the degree of success (p_c) that the site contains the volatile documents. The degree of success is computed in terms of no. of hits by detecting the frequency of changes occurred in the documents on a site. For example, if the crawler encounters a document being updated 6 times out of its 10 visits, the degree of success (p_c) is assigned as 0.6 to that site.

A unit step function $u(p_c)$ has been employed for the computation of Δt, which can be defined as follows

$$\Delta t = \{(1-p_c/p_g)*u(p_c-p_g) + (1-p_c/p_l)*u(p_l-p_c)\} * t_n \ldots eq^n (2)$$

Where: p_g, p_l are the boundary conditions i.e. upper and lower threshold values of p_c respectively,

$$\text{And } u(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{otherwise} \end{cases}$$

In the following examples the refresh time has been computed for three cases by taking three different data sets.

Case 1: consider the data given below:

t_n = 100 units
P_l = 0.3
P_g = 0.7
P_c = say **0.6**

Δt for a site S would be computed as given below:

Δt= {(1-0.6/0.7)*0 + (1-0.6/0.3)*0} * 100 = 0

The new refresh time

- ⇨ $t_{n+1} = t_n + \Delta t$
- ⇨ t_{n+1} = 100 + 0 =**100 units**
- ⇨ **No Change in refresh time**

Case 2: now consider the data given below:

t_n = 100 units
P_l = 0.3
P_g = 0.7
P_c = say **0.85**

Δt for a site S would be computed as given below:

Δt= {(1-0.85/0.7)*1+(1-0.85/0.3)*0}*100 = - 150/7 = - 21.42

The new refresh time

- ⇨ $t_{n+1} = t_n + \Delta t$
- ⇨ t_{n+1} = 100 – 21.42 = **78.58 units**
- ⇨ **Refresh time is decreased**

Case 3: now consider the data given below:

t_n = 78.58 units
P_l = 0.3
P_g = 0.7
P_c = say **0.25**

Δt for a site S would be computed as given below

Δt= {(1-0.25/0.7)*0 + (1-0.25/0.3)*1} * 78.58 = 13.09

The new refresh time

- ⇨ $t_{n+1} = t_n + \Delta t$
- ⇨ t_{n+1} = 78.58+13.09 = **91.67 units**
- ⇨ **Refresh time is increased**

From the examples it may be observed a value of p_c greater than p_g results in decrease in refresh time i.e. increase in refresh rate. Similarly values of p_c less than P_l results in increase in refresh time i.e. decrease in refresh rate. Thus the refresh rate gets self adjusted periodically.

In order to incorporate the proposed mechanism, the scheduling policy of the incremental crawler needs to be modified, so that it self adjust its refresh time periodically. The modified architecture of the incremental crawler is discussed in the next section.

A. *SELF ADJUSTING REFRESH TIME BASED ARCHITECTURE FOR INCREMENTAL WEB CRAWLER*

The system architecture of the modified incremental crawler is given in Fig.3.

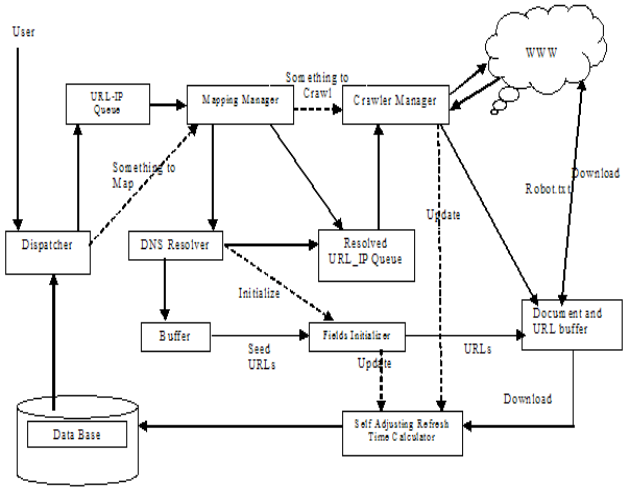

Fig.3: Architecture of Self adjusting Refresh Time Based Incremental Crawler

The architecture is composed of the following Modules/Data Structures:

URL_IP Queue: A set of seed URL-IP pairs.

Resolved URL_IP Queue: Set of URLs which have been resolved for their IP addresses.

Data Base: It contains a database of downloaded documents and their URL-IP pairs. In this work priority queue is used to store URL_IP pairs.

Refresh Time Controller: This component uses the proposed formula for computing the next refresh time. Algorithm for this component is given in next section.

Dispatcher: It reads a URL from database which may be a priority queue, and loads them in to URL_IP queue.

DNS Resolver: The DNS resolver uses the services of internet for translating the URLs to their corresponding IP addresses and stores them in to the resolved URL_IP queue. The new URLs with the IP addresses are stored in the buffer. A signal *initialize* is sent to the Field initializer.

Buffer : It stores the new URLs with their corresponding IP address.

Fields Initializer: It initializes the URL_fields in URL record for which object structure is shown in Fig.4.

Doc Id
URL
IP
Status
Refresh Time
Last Crawl Time
P_l & P_g
Document Pointer
Fingerprint key

Fig. 4. URL record

- **Doc ID** field is the unique identifier for each document.
- **RefreshTime** field stores the duration of the document to be refreshed. After initializing by some default value, this field is dynamically updated by refresh time calculator module.
- **LastCrawlTime** field stores the date Time stamp of last time page was crawled.
- **Status** field represent whether URL is present in URL-IP queue or not.
  If Status is '1', the dispatcher does not schedule the URL for Crawling,
  If Status is '0', the dispatcher schedules the URL for crawling.
- **p_l & p_g** are the boundary conditions i.e. upper and lower threshold values of p_c respectively.
- **The document pointer** field contains the pointer to the original document.
- **FingerPrintKey** field stores the finger print key value of the crawled page.

Mapping Manager: It creates multiple worker threads called *URL Mapper[8]*. It extracts URL-IP pairs from the URL-IP Queue and assembles a set of such pairs called URL-IP set. Each URL-Mapper is given a set for mapping. After storing URLs to Rsolved URl_IP queue it sends a signal *something to crawl* to Crawler manager.

Crawl Manager : It creates multiple worker threads named as Crawl Workers. Sets of resolved URLs from Resolved URL Queue are taken and each worker is given a set. It sends a signal *something to update* to refresh time calculator module after storing the documents and URLs in to the documents and URL buffer.

Description of each proposed components are discussed below.

- **Fields Initializer:** It stores the URL into URL_Database and initializes the refresh time for that URL. Its algorithm is given below:

```
Fields_Initializer ( )
  Do
     {
     Read URL from seed url set
     If (URL is present in database)
        Continue;
     Else
        {
         Set Status= 0;
         Set lastCrawlTIme = null;
         Set Refresh Time = AVERAGE_REFRESH_TIME;
         Insert URL record set in database
         }
     }
  Forever
```

- **Self adjusting refresh time calculator module (SARTCM):** it extracts URLs from Document and URL buffer and calculates the new refresh time as per the formula given in eq. (1) and eq. (2). The modified refresh time is updated in URL record, which is later on stored in database.

The algorithm for SARTCM is given below.

```
SARTCM ( )
        Do
         {
           Read URL IP pair from Document and URL buffer
           If (lastCrawlTime ==  null && Status == 0
              {
               Set Status= 1
              Set refresh time equal to default
              Update URL record
              Store URL-IP pair it into priority queue
              }
           Else
              {
              Calculated the new refresh time as per eq^n. (1) and (2)
              If (CURRENT_TIME – new refresh time)>= refresh Time)
                 {
                 Set Refresh time = new refresh time
                 Store URL-IP pair in to priority queue
                 }
              Else
                 Continue;
              }
           }
        Forever
```

The whole mapping and crawling Process can be defined as follows:

Mapping manager takes a URL_IP set from the URL_IP Queue and creates multiple instance of mapper threads named as URL-Mappers [8]. Set of URL_IP taken from URL_IP Queue are assigned to each URL_Mapper. URL-Mapper examines each URL_IP pair and if IP is null, then URL is sent to DNS Resolver. After the URL has been resolved for its IP, it is stored in the Resolved URL_IP Queue. If URL is new then DNS Resolver stores it in the Buffer and a signal "initialize" is sent to Field initiatizer. Field initializer initializes the URL fields and stores them in Document and URL Buffer and sends a signal something to update to SARTCM.

Downloaded documents are examined for modification of contents in it and refresh time is self adjusted accordingly by self adjusting refresh time calculator module. Later these Documents and URLs are stored in Database.

So, the modified crawler optimizes the frequency of refresh rate for an incremental crawler there by improving the quality of collection without incurring much traffic on the network.

IV. PERFORMANCE ANALYSIS

An estimated and approximate performance analysis can be done to compare the conventional search strategies with the proposed one. With the increase in the availability of web pages on the Internet, the major problem faced by the present search engines is difficulty in information retrieval. It is problematic to identify the desired information from amongst the large set of web pages resulted by the search engine. With further increase in the size of the Internet, the problem grows exponentially. The number of web pages which have gone under up-dation increases [11, 12, 13] (see Fig. 5); with this increase in web size the incremental crawler traffic will definitely be more. This results decrease in the quality (see Fig. 5) [13].

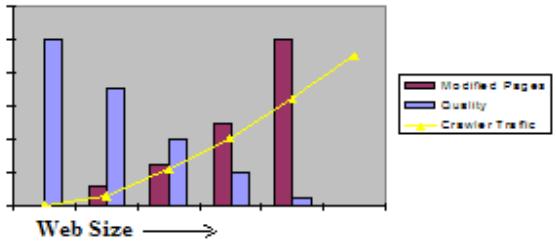

Fig. 5: Effect of growth of web on up-dation of pages and quality

As none of the previous researcher proposed an idea of dynamic refresh time, the architecture given in this work effectively takes into consideration the above-mentioned issue. Being a Self adjusting refresh time based strategy, the proposed download mechanism is independent of the size of the Internet. Since, only those web pages are retrieved which have undergone up-dation , the system would continue to give modified pages only irrespective of the size of the Internet and because the pages are relevant in the sense that they have gone under up-dation, the traffic on network is reduced and hence only quality pages are retrieved (see Fig.6)

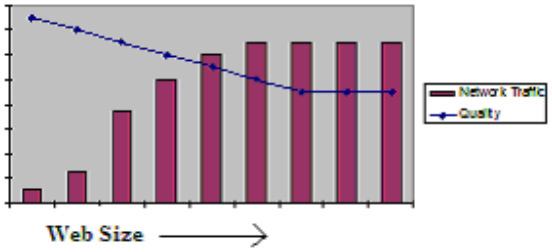

Fig. 6: Performance analysis of proposed architecture

In terms of performance parameters like quantity of web pages downloaded, their quality and the network traffic, the Self Adjusting refresh time based Incremental Crawler definitely holds an edge above the present conventional crawling strategies, which are based on fixed refresh time.

V. CONCLUSION

With the help of the proposed scheme, various design choices for an incremental crawler has been identified to calculate the refresh time of the documents and thus resolving the problem of the freshness of the pages. To perform this task a no. of algorithms and a modified architecture have been proposed that will support the crawler to remove its deficiencies by dynamically adjusting the refresh time and thus improving the efficiency, as it makes the database rich. Only the useful data is provided to the user, thus network traffic is reduced and data enrichment is achieved..

REFERENCES

[1] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hyper textual Web search engine". *Proceedings of the Seventh International World Wide Web Conference*, pages 107—117, April 1998.

[2] Mike Burner, "Crawling towards Eternity: Building an archive of the World Wide Web", *Web Techniques Magazine*, 2(5), May 1997.

[3] L. Page and S. Brin, "The anatomy of a search engine", *Proc. of the 7th International WWW Conference (WWW 98)*, Brisbane, Australia, April 14-18, 1998.

[4] Y. Yang, S. Slattery, and R. Ghani, "A study of approaches to hypertext categorization", *Journal of Intelligent Information Systems*. Kluwer Academic Press, 2001.

[5] S. Chakrabarti, M. van den Berg, and B. Dom, "Distributed hypertext resource discovery through examples", *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, pages 375-386, 1999.

[6] J. Dean and M. Henzinger, "Finding related pages in the world wide web", *Proceedings of the 8th International World Wide Web Conference (WWW8)*, pages 1467-1479, 1999.

[7] www.w3.org/hypertext/WWW/MarkUp/MarkUp.html -- *official HTML specification*.

[8] A. K. Sharma, J. P. Gupta, D. P. Agarwal, "PARCAHYD: A Parallel Crawler based on Augmented Hyper text Documents", *communicated to IASTED International Journal of computer applications*, May. 2005.

[9] A. K. Sharma, J. P. Gupta, D. P. Agarwal, "Augment Hypertext Documents suitable for parallel crawlers", accepted for presentation and inclusion in the *proceedings of WITSA-2003, a National workshop on Information Technology Services and Applications*, Feb'2003, New Delhi.

[10] A. K. Sharma**, J**. P. Gupta, D. P. Agarwal, " A novel approach towards management of Volatile Information" *Journal of CSI*, Vol. 33 No. 1, pp 18-27, Sept' 2003.

[11] C. Dyreson, H.-L. Lin, Y. Wang, "Managing Versions of Web Documents in aTransaction-time Web Server" In Proceedings of the *World-Wide Web Conference*.

[12] Junghoo Cho and Hector Garcia-Molina. Estimating frequency of change, 2000.Submitted to *VLDB 2000*, Research track.

[13] Junghoo Cho, Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In Proceedings of the *8th World-Wide Web Conference*, 2003,