Accelerating Structured Web Crawling without Losing Data

Boutros R. El-Gamil
Vienna University of Technology, Institute for Software Technology and Interactive Systems
Favoritenstraße 9-11
1040 Vienna, Austria
boutros@ims.tuwien.ac.at

Werner Winiwarter
University of Vienna, Research Group Data Analytics and Computing
Währinger Straße 29
1090 Vienna, Austria
werner.winiwarter@univie.ac.at

ABSTRACT

Size of retrieved data versus crawling time formulate a well-known dilemma in the structured Web crawling community. The real challenge within this dilemma is to optimize the settings of a given wrapper to obtain maximum available data in shortest possible time. In this paper, we try to tune these settings, by introducing a threaded algorithm that guarantees accessing all available detail pages within crawling scope; and using this algorithm, we try to reduce the time consumed by the crawler, via simple adjustments of sleeping time after each detail page visit.

Categories and Subject Descriptors

H.1.2 [**User/Machine Systems**]: Human information processing; H.2.5 [**Heterogeneous Databases**]: Data translation; H.3.3 [**Information Search and Retrieval**]: Human information processing; H.3.5 [**Online Information Services**]: Commercial services, Web-based services

General Terms

Algorithms, Experimentation

Keywords

Structured Web Crawling, Web Wrappers, Online Databases.

1. INTRODUCTION

Structured Web crawling represents daily homework for many industrial sectors. Spanning from shopping comparisons and publication tracking to social Web analysis and competitive intelligence, collecting information from Web databases has attracted considerable research efforts. An interesting, yet challenging factor in this research field, is that the design and implementation of deep Web extraction systems has been manipulated by different scientific perspectives, such as natural language processing and machine learning. These different considerations generated a variety of Web extraction systems, which consequently introduced new perspectives of comparing and assessing such systems [3, 5, 6]. Ongoing research in this field concentrates on certain open issues, such as *wrapper maintenance*, *degree of automation*, *crawling ethics*, and *user privacy*.

In the last decade, many applications have been developed to tackle enterprise deep Web mining. Systems like Lixto [1], Wargo [11], and WICCAP [9] provide visual platforms of Web database querying. Samples of wrapper induction systems were introduced in [4, 13]; and useful approaches in reducing Web crawling traffic can be found in [2, 10].

Whether the end user is a small firm that fetches few Websites in a specific domain, or a large organization that spends millions of dollars to build robust Web spiders for multiple usages, data gathering optimization is a basic issue. That is, the main goal behind Web database crawling is to pick up every little piece of data from each available results/detail page. This introduces an important question: does the strategy of one crawling session fulfils this goal? In other words, do we need auxiliary crawling session(s) to optimize the performance of Web database crawling? In this paper we try to answer this question, by exploring the existence of unvisited detail pages during the basic crawling session. We experimentally prove that for a few high-speed Web databases and a back-end server with 12.5MB/s download speed for the crawler, the basic crawling session failed to visit all available detail pages.

The paper is organized as follows: Section 2 studies the problem of dropped links of the single-session Web crawlers. In Section 2.2 we introduce **S**tructured-**W**eb **T**hreaded **C**rawler (**SWTC**), a simple algorithm that handles this problem by providing auxiliary sessions to the wrapper to optimize its output. Section 2.3 gives an experimental test of SWTC. In Section 3 we try (using SWTC algorithm) to accelerate crawling time, by adjusting the sleeping time intervals of each Web database. We conclude our work in Section 4.

2. CATCHING UNVISITED LINKS WITHIN STRUCTURED WEB CRAWLING

There exist many technical factors that prevent Web wrappers from accessing all required data in one crawling session, but almost all of them can be summarized into the fact of *Web servers' heterogeneity.* Most of commercial Web wrappers are designed to fetch multiple Web databases in parallel, which forces the wrapper to construct connections to multiple Web servers with different bandwidths. As a result, wrappers which are running with fixed crawling speed are definitely threatened with losing data, simply due to different connection and download speeds of the crawled sites.

2.1 Significance of Auxiliary Sessions in Structured Web Crawling

One approach to crawl Web servers with different bandwidths is to slow down the wrapper, for instance, by sleeping for a certain time interval after each page visit. However, this strategy has two main defects. Firstly, the total duration of a crawling session will be significantly expanded, which is not preferable for many end users, as in the field of Business and Competitive Intelligence, where a company needs timely-based analysis of the market. Secondly, as we experimentally explain in Section 3.2, some servers (like Amazon's) react negatively to moderate/low speed crawlers. On the other hand, if we speed up the crawler, by adding more crawling threads, the system may be blocked by some sites with limited bandwidth usage, or lose connections to Web servers with moderate/low download speeds. Some commercial Web wrappers (such as Visual Web Ripper[1], iMacros[2], Screen-scraper Enterprise[3], and Heliumscraper[4]) tried to resolve the consequences of speeding up the crawling session, by adding proxies as crawling interfaces, to prevent the wrapper of being detected and blocked by Web servers. However, this strategy has also its disadvantages, such as the occasional unavailability to access some/all proxies, and the need for regular update of the proxies list.

In order to overcome this dilemma, we introduce Structured-Web Threaded Crawler (or SWTC), a very simple, yet effective algorithm to catch unvisited links during the main crawling session. It achieves this goal by adding multiple auxiliary sessions to the main crawler to optimize the number of visited detail pages and consequently the total amount of retrieved data.

2.2 SWTC Algorithm

Numerous algorithms and techniques have been introduced to detect and extract different types of data residing in Web databases. The most well-known one is *the simple tree matching algorithm*, which makes use of the availability of representing a Web page as labeled ordered rooted tree. The algorithm inspires its robustness and popularity from the simple tree matching algorithm [12] and its variants. Another class of solutions to explore and mine Web databases relies on algorithms adapted from machine learning, as it provides variable methodologies to build training classifiers/sessions, which enables the given wrapper to explore Web databases upon predefined expertise within the same/different domains of interest. Because of great motivation to investigate this area of research, many Web data mining systems based on machine learning techniques have been implemented so far, such as SoftMealy [7] and WIEN [8].

The aim behind SWTC algorithm introduced in this paper, however, is not to give a particular model of detecting and extracting data from Web databases. Rather, it introduces a multi-threaded crawling environment that guarantees sufficient access to all available detail pages within the crawling scope. The SWTC algorithm (see Figure 1) starts by defining input and output variables of the threaded crawler, such as queue of queries, queue of Web sites, maximum number of allowed threads and sleeping time after accessing each detail page. It then calls $NavigateDomain()$ function, which launches the multi-threaded environment of the crawler. The function then enters a nested loop of both queries and Web sites to submit each query to each site. After each query submission, the function navigates through intermediate result pages of the given Web site, if any exist, until reaching the link to the targeted detail page. After obtaining this link, it passes it to the $OpenPage()$ function, which tries to open this link and read target data inside it. If failed, both given query and site are stored in 2 auxiliary queues, to be re-processed by the main algorithm. After $NavigateDomain()$ has completed the main crawling session (i.e. crawling basic set of queries), the main algorithm checks if auxiliary queues are not empty. If so, it calls $Navigate\text{-}Domain()$ repeatedly, until either all failed queries have successfully obtained their detail pages or maximum iterations have been achieved. SWTC is a linear algorithm, as it basically runs two nested loops of its input parameters (queries and Websites) in $NavigateDomain()$. Besides, steps 9 - 12 of the SWTC guarantee the algorithm to terminate.

2.3 Testing SWTC Algorithm

2.3.1 Settings

In order to evaluate SWTC algorithm, we set a testing environment under Gentoo Linux, with a download speed of roughly 12.5MB/s. To simulate the average capacity of commercial server bandwidths of small firms, we set the maximum number of threads to 30 in all of our experiments. We chose online bookshops as our target domain, with limited size of queries and Web databases. The list of queries consists of 1000 **ISBN**s (International Standard Book Numbers) and **ASIN**s (Amazon Standard Identification Numbers). The targeted Web databases consist of 8 online bookshops, namely *amazon.co.uk*, *amazon.de*, *amazon.fr*, *amazon.it*, *amazon.es*, *weltbild.de*, *mayersche.de* and *thalia.de.*

2.3.2 Targeted Data

In order to obtain real assessment of our algorithm, we aimed at extracting certain types of data within the crawling sessions. We adopted the XPath language as our approach to locate and extract different pieces of data. The goal behind our choice is the popularity of XPath as a standard tool in addressing specific elements in an XML document (and HTML page as one of its dialects). Besides, XPath language provides a powerful syntax to extract data in a simple manner. Our set of targeted data includes textual strings (like Title and Author Name), URLs (Also Bought links), numbers (price and sales rank), and number of occurrences of certain element (e.g. Review List Customers in *thalia.de* using XPath `count()` function). Table 1 gives a detailed description of the target data of our experiments along with its data types.

2.3.3 Evaluation and Results

It is likely with any threaded crawler to drop some links through running, either because of visiting Web sites with different server speeds, or due to connection time out errors. As the aim is to prove the importance of adding auxiliary

[1]www.visualwebripper.com
[2]www.iopus.com/iMacros
[3]www.screen-scraper.com
[4]www.heliumscraper.com

Function OpenPage(q,s,l,st)

```
1. try
2.     open link $l$ and read target data from it
3.     sleep for time interval $st$
4. except
5.     $auxiliaryQuery \leftarrow q$
6.     $auxiliarySite \leftarrow s$
```

Function NavigateDomain(Q,S,T,st)

```
1. launch threaded environment of maximum threads $T$
2. for $s$ in $S$ do
3.     for $q$ in $Q$ do
4.         submit query $q$ to Website $s$
5.         navigate inside $s$ until obtain the link $l$ of detail
           page
6.         OpenPage$(q,s,l,st)$
7.     end for
8. end for
```

Algorithm SWTC

```
1.  $basicQuery \leftarrow$ list of queries
2.  $basicSite \leftarrow$ list of Websites
3.  $auxiliaryQuery \leftarrow$ auxiliary list of queries
4.  $auxiliarySite \leftarrow$ auxiliary list of Websites
5.  $maxThreads \leftarrow$ maximum number of threads
6.  $st \leftarrow$ sleeping time after accessing each detail page
7.  $maxIteration \leftarrow$ maximum trials of obtaining $auxiliaryQuery$ detail pages within $auxiliarySite$
8.  NavigateDomain$(basicQuery, basicSite, maxThreads, st)$
9.  set $counter = 0$
10. while ($auxiliaryQuery$ is not empty or $counter < maxIteration$) do
11.     NavigateDomain$(auxiliaryQuery, auxiliarySite,$
         $maxThreads, st)$
12.     increment $counter$ by 1
13. end while
```

Figure 1: The SWTC Algorithm.

crawling sessions to the algorithm, we decided to test the SWTC algorithm under different sleeping time intervals after accessing each detail page, varying between 0 and 1.2 seconds.

To execute the algorithm, we translated it as Python script and added it to our Linux server as a cron job. The script was running for 20 days continuously, and the average number of detail pages (for basic and auxiliary crawling sessions) was calculated. The results show that regardless of changing the sleeping time between visiting detail pages, the algorithm misses links during its basic session. Figure 2 displays the performance of SWTC in different sleeping time intervals. The algorithm misses 8 detail pages in average within small sleeping times (0 - 0.7 sec.). Detail pages missed by the basic crawling session relatively decreased to 5 or 6 pages within larger sleeping time scale (0.8 - 1.2 sec.)

Table 1: Target data

Target Data	Data Type
Title	text
Author	text
Publisher	text
Version	text
Language	text
Price	float
Also Bought Title	text
Also Bought Author	text
Also Bought Link	url
Stars Number	float
Sales Rank	int
Review Number	int

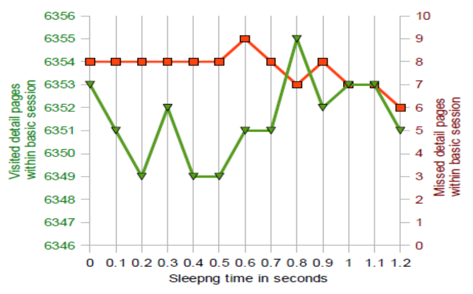

Figure 2: Retrieved detail pages in both basic and auxiliary sessions of SWTC.

It is also worth noting, that for all sleeping time intervals, the SWTC algorithm uses only one auxiliary session to visit the detail pages dropped from the basic session (see Figure 3). This is a normal outcome due to the limited experimental setup, which correspondingly generates a small number of dropped detail pages. However, for larger crawling tasks with Web databases of different bandwidths, the algorithm is expected to consume more auxiliary sessions to optimize its output. In future work we will expand our experimental scope to handle more crawling scenarios with different domains.

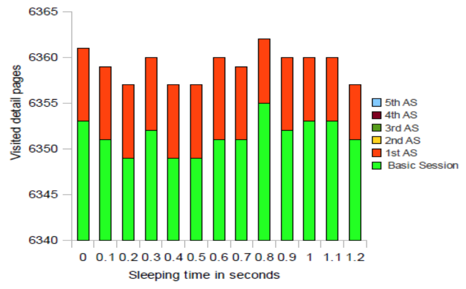

Figure 3: Number of auxiliary sessions (AS) consumed by SWTC.

3. ACCELERATING STRUCTURED WEB CRAWLING

3.1 SWTC Reaction to Different Sleeping Time Intervals

Time is a critical issue in structured Web crawling, either for related front-end applications' robustness and usability, or for back-end running and maintenance. One of the most interesting observations we noticed through our experiments is the number of retrieved detail pages in each sleeping time interval. Figure 4 describes the change in retrieved detail pages (for both basic and auxiliary sessions) according to different sleeping time. The SWTC achieves quite promissing results without sleeping. By increasing the sleeping time by 0.1 second, the crawler performance slightly goes down in sleeping range (0.1 to 0.7) seconds. After that the crawler reaches its best performance at a sleeping time of 0.8 seconds, before stabilizing between 0.9 and 1.1 seconds, until giving its minimum output at a sleeping time of 1.2 seconds. The chart indicates how far the sleeping time in threaded structured Web crawling can affect the system performance. At first glance, running the algorithm without sleeping time at all seems a satisfying solution, since the average difference in retrieved detail pages between 0 and 0.8 seconds (the best outcome) is only 1 page. However, if we run the algorithm on larger domains, targeting Web databases with variable bandwidths, then measuring the optimum sleeping time can be quite useful in catching considerable number of extra detail pages.

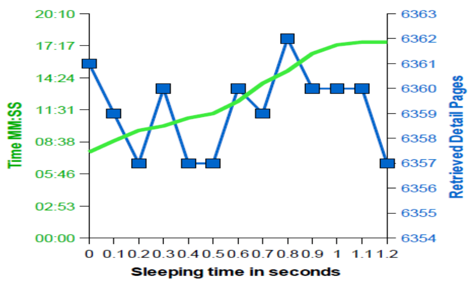

Figure 4: Change in retrieved detail pages over different sleeping time intervals.

3.2 Quick vs. Slow Web Databases

The significance of measuring the optimum sleeping time within any threaded crawling algorithm is that it reflects the corresponding variance in Web database response. Therefore, it's quite important to measure this variance, in order to make the required sleeping time adjustments to each Web database separately, or to the whole system. To simplify the idea, we measured the number of retrieved detail pages for each Web database separately at three sleeping time intervals; the minimum (0 sec.), the maximum (1.2 sec.), and the optimum (0.8 sec.). Figure 5 illustrates different outcomes of our test Web databases, in response to the three time intervals. While the Amazon group shows quick response, with best outcome at 0 sleeping time, other Web databases (e.g. *thalia.de*) return more detail pages with increased sleeping time. These results confirm our assumption of the importance of measuring the optimum sleeping time within threaded Web crawling, as a critical factor of accessing the maximum available number of detail pages.

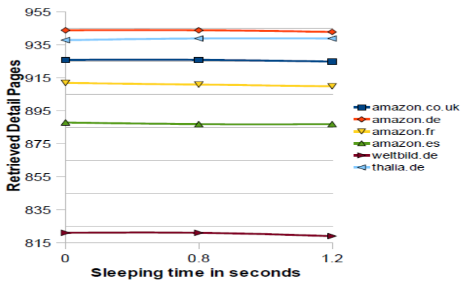

Figure 5: Change in retrieved detail pages for sample set of Web databases.

3.3 Adjust Sleeping Time of SWTC

3.3.1 Settings

Based on results obtained from Section 3.2, we classified our domain Web databases as quick and slow sites. Table 2 gives a list of this classification. In order to accelerate our SWTC algorithm without losing retrieved detail pages, we made a simple adjustment in the sleeping time based on each Web database response. For this set of experiments we re-run the algorithm for the best 2 sleeping time intervals (0 and 0.8 sec.) obtained in Section 2.2. Besides, we run two other scripts of SWTC; one decreases the sleeping time for quick Web databases to 0 sec. and the other increases it for slow databases to 0.8 sec. Sleeping time intervals of the 4 experiments are listed in Table 3. The four scripts of SWTC were running once again on the same server and domain settings for 20 continuous days, and the average time and retrieved detail pages were calculated.

3.3.2 Results

The results of the second set of experiments (Figure 6) give two important outcomes. Primarily, in contrast to the first experimental setup, running the algorithm without sleeping outperforms adding a sleeping time of (0.8 sec.) in terms of retrieved detail pages, which reflects the high variability of threaded Web crawling to sleeping time, and correspondingly the need for regular check and maintenance of this parameter in the long run.

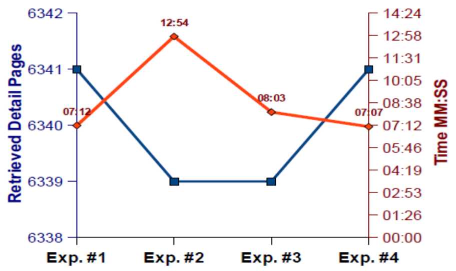

Figure 6: Results of the second set of experiments.

The second observation is that Experiment #4 gives approximately the same retrieving power as Experiment #1

Table 2: Web databases classification according to their response

Web DB	amazon.co.uk	amazon.de	amazon.fr	amazon.it	amazon.es	weltbild.de	mayersche.de	thalia.de
Response	Quick	Quick	Quick	Quick	Quick	Quick	Slow	Slow

Table 3: Sleeping time adjustments (in sec.) for the second set of experiments

Experiment #1	**Experiment #2**	**Experiment #3**	**Experiment #4**
st= 0	*st*= 0.8	*st*= 0 for slow sites *st*= 0.8	*st*= 0.8 for slow sites *st*= 0

(with no sleeping). However, Experiment #4 takes less time than Experiment #1 (5 seconds difference in average). This means that initializing SWTC algorithm at high sleeping time along with suitable time adjustment for quick Web databases gives us the optimum performance of the threaded crawler in terms of both crawling time and retrieved detail pages.

It's also worth to mention, that during the second set of experiments we could accelerate the SWTC algorithm by 1.1% by adjusting the sleeping time for both quick and slow Web databases. This is a small percentage corresponding to our limited experimental settings, but on commercial long term run over more Web databases, this percentage can increase linearly with crawler inputs, which implicitly reflects the importance of measuring the speed of Web database response for effectively building robust Web wrappers.

4. CONCLUSION AND FUTURE WORK

In this paper we introduced a simple algorithm to optimize the performance of structured Web crawling. We investigated the problem of detail pages that are missed by wrapper basic session, and implemented an algorithm that catches those pages by adding auxiliary sessions. After that we tried to reduce the time consumed by our threaded crawling algorithm, by adjusting the sleeping time intervals between Web databases. The experimental results show promising outcome in terms of both maximization of retrieved data and acceleration of crawling time.

For the future work, we will try to broaden the testing environment to include more domains with different input settings. Domains of interest include Real Estate, Flying Tickets Booking and Hotels Reservations. Our future selection will depend on several factors: (*i*) multiple set of queries used (e.g. departure date, arrival date, destination in Flying Tickets Booking domain), (*ii*) multiple detail page links within each result page, and (*iii*) noticeable changes in HTML structure in the same/different Web database. Using such integrated features shall provide deeper assessment of the algorithm in both design and implementation aspects.

5. REFERENCES

[1] R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In *Proceedings of International Conference on Very Large Data Bases, pp. 119-128*, 2001.

[2] B. B. Cambazoglu, F. Junqueira, V. Plachouras, and L. Telloli. On the feasibility of geographically distributed web crawling. In *proceedings of 3rd International ICST Conference on Scalable Information Systems, ICST*, 2008.

[3] C.-H. Chang, M. Kayed, M. R. Girgis, and K. F. Shaalan. A survey of web information extraction systems. In *IEEE Transactions on Knowledge and Data Engineering Vol. 18 Issue 10, pp. 1411-1428*, 2006.

[4] W. Cohen, M. Hurst, and L. Jensen. A flexible learning system for wrapping tables and lists in html documents. In *Proceedings of International Conference on World Wide Web*, 2002.

[5] B. R. El-Gamil, W. Winiwarter, B. Božić, and H. Wahl. Deep web integrated systems: current achievements and open issues. In *13th Conference on Information Integration and Web-based Applications and Services (iiWAS 11)*, 2011.

[6] E. Ferrara, P. D. Meo, G. Fiumara, and R. Baumgartner. Web data extraction, applications and techniques: A survey. In *ACM Computing Surveys, Vol. V, No. N, pp. 1-48*, 2012.

[7] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. In *Information Systems. Issue 23, Vol. 9, pp. 521-538*, 1998.

[8] N. Kushmerick. Wrapper induction: efficiency and expressiveness. In *Artificial Intellegence. Issue 118, Vol. 1-2, pp. 15-68*, 2000.

[9] Z. Li and W. K. Ng. Wiccap: From semi-structured data to structured data. In *Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, 2004.

[10] S. Mishra, A. Jain, and A. K. Sachan. A query based approach to reduce the web crawler traffic using http get request and dynamic web page. In *International Journal of Computer Applications Vol. 14 - No.3*, 2011.

[11] J. Raposo, A. Pan, M. Álvarez, J. Hidalgo, and A. Vina. The wargo system: semi-automatic wrapper generation in presence of complex data access modes. In *Proceedings of Workshop on Database and Expert Systems Applications*, 2002.

[12] S. Selkow. The tree-to-tree editing problem. In *Information processing letters Vol. 6 Issue 6, pp. 184-186*, 1977.

[13] S. Zheng, R. Song, J. Wen, and C. Giles. Efficient record-level wrapper induction. In *Proceedings of ACM International Conference on Information and knowledge management*, 2009.