A New Approach for Verifying URL Uniqueness in Web Crawlers

Wallace Favoreto Henrique[1], Nivio Ziviani[1], Marco Cristo[2], Edleno Silva de Moura[2], Altigran Soares da Silva[2], and Cristiano Carvalho[1]

[1] Universidade Federal de Minas Gerais,
Department of Computer Science, Belo Horizonte, Brazil,
{wallace,nivio,cristiano.dcc}@dcc.ufmg.br
[2] Universidade Federal do Amazonas,
Department of Computer Science, Manaus, Brazil
{marco.cristo,edleno,alti}@dcc.ufam.edu.br

Abstract. The Web has become a huge repository of pages and search engines allow users to find relevant information in this repository. Web crawlers are an important component of search engines. They find, download, parse content and store pages in a repository. In this paper, we present a new algorithm for verifying URL uniqueness in a large-scale web crawler. The verifier of uniqueness must check if a URL is present in the repository of unique URLs and if the corresponding page was already collected. The algorithm is based on a novel policy for organizing the set of unique URLs according to the server they belong to, exploiting a locality of reference property. This property is inherent in Web traversals, which follows from the skewed distribution of links within a web page, thus favoring references to other pages from a same server. We select the URLs to be crawled taking into account information about the servers they belong to, thus allowing the usage of our algorithm in the crawler without extra cost to pre-organize the entries. We compare our algorithm with a state-of-the-art algorithm found in the literature. We present a model for both algorithms and compare their performances. We carried out experiments using a crawling simulation of a representative subset of the Web which show that the adopted policy yields to a significant improvement in the time spent handling URL uniqueness verification.

1 Introduction

In July 2008, Google has reported more than 1 trillion unique URLs in its scheduler queue[3]. To get to this number of unique URLs, the crawler starts at a set of initial pages and follow each of their links to new pages. Then it follows the links on those new pages to more pages, in a continuous fashion. In fact, they found more than 1 trillion individual URLs, but not all of them lead to unique web pages, as many pages are duplicates or auto-generated copies of each other.

[3] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

Search engines consist of web crawlers (which find, download, parse content and store each page), indexers (which construct an inverted file index for fast retrieval of pages), and query processors (which rank documents by relevance to answer user queries). Search engines download and index only a subset of the pages related to the set of unique URLs.

This paper focuses on the problem of verifying URL uniqueness in a web crawler. The verifier of uniqueness must check if a URL is present in the repository of unique URLs and if the corresponding page was already collected. For a repository of URLs stored in a disk file, URLs can be checked against a buffer of popular URLs and only those not present are searched in the disk file. In [2] and [4], the RAM buffer is a LRU cache with an array of recently added URLs, in [5], a general purpose database is used with a RAM caching, and in [6], a balanced tree of URLs is used for disk checking.

A better idea is to accumulate URLs into a RAM buffer and check several URLs sequentially in one pass. This *batch disk check* strategy is used by DRUM (*Disk Repository with Update Management*), which is part of the IRLBot web crawler [3]. DRUM can store large volumes of arbitrary hashed data on disk and implement very fast check, update, and check+update operations using bucket sort. Through a set of comparison experiments it is shown that DRUM outperforms all the previously cited ones. Thus, in this work, we will use DRUM as our comparison baseline to evaluate our proposed strategy. While DRUM can be applied to other tasks such as handling "robots.txt" files and DNS caching, we here focus on its use for verifying URL uniqueness.

For a large number of pages, the task of verifying URL uniqueness becomes very complex. As shown in [3], the complexity of verifying URL uniqueness is quadratic on the number of links that must pass through URL check. In the *batch disk check* strategy large sets of URLs are overwritten in lexicographic order in disk files, which is a time consuming operation.

In this paper, we present VEUNIQ (VErifier of UNIQueness). VEUNIQ is based on a novel policy for organizing the set of known URLs according to the server they belong. Following such a policy, the algorithm exploits a locality of reference property, which is inherent in Web traversals. This property follows from the skewed distribution of links within a web page, which favors references to other pages from a same server. Previous work [1, 7] have observed and exploited this property for ranking purposes, and, in here, we take advantage of such property to speed up the crawling process.

VEUNIQ is part of a larger web crawling project developed by us. Our scheduling strategy in this crawler also selects URLs taking into account information about the servers they belong to, thus allowing the usage of VEUNIQ in the crawler without extra cost to pre-organize the entries.

We present a model for both algorithms, DRUM and VEUNIQ, and compare their performance. We carried out experiments using a crawling simulation of a representative subset of the Web which show that the adopted policy yields to an improvement in the time spent handling URL uniqueness verification in comparison to the baseline algorithm.

2 Related Work

Few works in literature have addressed the problem of ensuring that a certain page will not be collected more than once during a crawling cycle. A very first approach to deal with this problem was proposed by Pinkerton [5]. He used the B-Tree structure of a database management system (DBMS) to verify the uniqueness of each page URL, thus avoiding repeated occurrences in the list of URLs to be collected. This strategy is simple to implement since it takes advantage of the large availability of commercial DBMS systems. However, the use of a DBMS to store and retrieve a massive number of URLs leads to poor disk performance with large impact over the whole crawling process.

To cope with this problem, Heydon and Najork [2] proposed the use of a cache in memory to minimize random disk access operations. Their algorithm, called Mercator-A, would only perform disk accesses if the current URL being checked was not found in the cache. In such a case, to minimize the probability of new misses, a new set of URLs would be retrieved from the disk instead of only the missing one. The disadvantage of Mercator-A lies in the fact that, in the worst case, it requires a disk access for each URL and that access operation comprises much more data to be transferred.

A better strategy to minimize random disk-accesses operations is to check several URLs sequentially in one pass, a strategy called *batch disk check*. The idea is to accumulate URLs into a memory buffer. When this buffer is full, the URLs are sorted in place. They are then merged with blocks of (already sorted) URLs retrieved from the disk, so that duplicates are not stored. This is the main idea behind the approaches referred to as Mercator-B [4] and Polybot [6]. Polybot avoids the sorting step because it uses a binary tree as memory buffer.

A more recent algorithm, called DRUM (*Disk Repository with Update Management*), was proposed for the IRLBot web crawler [3]. As this algorithm was used as the baseline for an experimental comparison with VEUNIQ, a detailed description of DRUM is provided in Section 4.

3 Crawler Architecture

In this section, we present a high-level description of our crawler. It has four main components: fetcher, extractor of URLs, verifier of uniqueness and scheduler. Figure 1 illustrates the crawl cycle involving the four components. The fetcher is the component that sees the Web. In step 1, the fetcher receives from the scheduler a set of URLs to download web pages. In step 2, the extractor of URLs parses each downloaded page and obtains a set of new URLs. In step 3, the uniqueness verifier checks each URL against the repository of unique URLs. In step 4, the scheduler chooses a new set of URLs to be sent to the fetcher, thus finishing one crawl cycle.

Considering cycle i, the *fetcher* locates and downloads web pages. It receives from the *scheduler* a set of candidate URLs to be crawled and returns a set L_i of URLs actually downloaded. The set of candidate URLs is small, determined

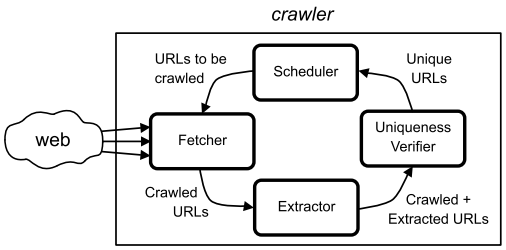

Fig. 1. Web page crawling cycle.

by the amount of memory space available to VEUNIQ. The downloaded pages are stored on disk for efficient access and retrieval.

Important to notice that there are many different policies in the literature to select the set of candidates to be crawled from a given set of servers at each cycle. Also important is that the interval between two accesses to the same server must follow politeness rules, which might cause a significant slowdown in the whole process. In this paper, we will not address these issues but rather concentrate on the algorithm for verifying URL uniqueness.

The *extractor of URLs* parses each downloaded page and output two sets: the set E_i of URLs just extracted and the set M_i with auxiliary metadata for each page such as the repository disk file address and its offset.

The *verifier of uniqueness* receives as input the three sets L_i, E_i and M_i and creates as output an updated repository of unique URLs. It identifies from L_i and E_i the following sets: (i) URLs crawled in previous cycles; (ii) URLs seen in previous cycles but not crawled yet; (iii) URLs crawled in the current cycle; (iv) New URLs. The URLs from sets (i) and (ii) are discarded (they are already in the set of unique URLs), the set M_i and the information on disk location for URLs from set (iii) are updated, and new URLs from set (iv) are merged to the repository of unique URLs.

4 The Baseline Algorithm (DRUM)

The verifier of uniqueness must check if a URL is present in the repository of unique URLs. Before presenting our approach VEUNIQ for checking URL uniqueness we describe DRUM, which we use as a comparison baseline approach.

DRUM (Disk Repository with Update Management) is a technique for efficient storage of key/value pairs (KV-pairs) with asynchronous support to three operations: *check* (a key is checked against a disk repository and, if found, its corresponding value can be retrieved), *update* (KV-pairs are merged into the disk repository), and *check+update* (the two previous operations combined). While DRUM is a more general solution to external hashing, as far as we know it is also the best solution published in literature, to verify uniqueness of URLs in a web crawler. Further, as it can be seen in the article where DRUM was proposed, verifying uniqueness in web crawlers is its main application focus.

DRUM keeps the set of unique URLs in a large persistent repository, which is sorted by means of a bucket sort strategy, described as follows. Input received by DRUM is inserted into two sets of memory arrays, according to their key values. The KV-pairs are stored in the first set while auxiliaries are sent to the second one. For each array, two corresponding buckets are kept in disk, one for KV-pairs (KV-bucket) and another for auxiliaries (A-bucket). Once an array is filled, their values are moved to the corresponding disk buckets. The arrays continue to receive KV-pairs until one of the KV-buckets reaches a certain size r. When this happens, all the KV-buckets are merged with the central repository of URLs. To accomplish this, each KV-bucket is loaded into a memory buffer and sorted. The buffer content is then merged with the central repository. After the pairs have been processed, the buffer is restored to its original order so that KV-pairs match those in the corresponding A-bucket. Key, value and auxiliary data are then dispatched for further processing. All these steps are repeated for all buckets sequentially.

Note that sorting is only done when a KV-bucket is read into memory, which allows an efficient synchronization of all buckets with the central repository in a single pass. Further, to ensure fast sequential writing/reading operations, all buckets are pre-allocated on disk before they are used.

DRUM Model

As shown by Lee et al. [3], after some time, a crawler reaches a steady state where the probability p of a page to be unique remains constant. Assume that in such state: (i) the crawler is scheduled to visit $\mathcal{N}$ pages, each one with an average of ℓ links. Thus, $n = \mathcal{N}\ell$ pages have to be checked to verify their uniqueness; (ii) a RAM of $\mathcal{R}$ bytes is allocated to this task which, as previously described, will require the merge of n input URLs with U unique URLs stored in a central disk repository; (iii) the average URL length is b, H is the size of the URL hashes used by the crawler, and P is the size of a memory pointer.

As previously mentioned, DRUM starts a merge each time one of the KV-buckets reaches a size r. If we assume that the hashing distributes the URLs evenly into k buckets, when one of them reaches size r, a total of kr disk bytes was filled with URL hashes. Thus, DRUM performs $\frac{nH}{kr}$ merges to check n URLs. To fill the KV-buckets, in each iteration i, it is necessary:

- Read/write the i-th A-bucket once to load into memory the URL text of $\frac{kr}{H}$ URLs, for a total amount of $2\frac{krb}{H}$ bytes.
- Read/write all the KV-buckets, i.e., $2kr$ bytes.
- Read/write URLs from the central repository, i.e., $2UH + (i-1)2krp$ bytes.
- Append new URL hashes, i.e., krp bytes.

Thus, after adding the final overhead to store pbn bytes of unique URLs, the read/write DRUM overhead to check n URLs is given by (see Eq.(15) from [3]):

$$w(n,\mathcal{R}) = pbn + \sum_{i=1}^{\frac{nH}{kr}} \left(2UH + \frac{2krb}{H} + 2kr + 2krpi - krp\right)$$
$$= nb\left(\frac{(2UH + pHn)H}{bkr} + 2 + p + \frac{2H}{b}\right) \quad (1)$$

Now assume that to support efficient read and write operations, a buffer of size $\mathcal{M}$ is maintained to each opened file, a buffer of size Δ is used to load the repository into memory, and $r \geq \Delta$. Thus, if $\mathcal{R} \geq \frac{2\Delta(H+P)}{H}$ and DRUM can use up to D bytes of disk for the checking, final overhead is $w(n,\mathcal{R}) = \alpha(n,\mathcal{R})bn$, where $\alpha(n,\mathcal{R})$ is a proportion of the input size, given by:

$$\alpha(n,\mathcal{R}) = \begin{cases} \frac{8M(H+P)(2UH+pHn)}{b\mathcal{R}^2} + 2 + p + \frac{2H}{b} & \mathcal{R}^2 < 8\mathcal{M}D\frac{H+P}{H+b} \\ \frac{(H+b)(2UH+pHn)}{bD} + 2 + p + \frac{2H}{b} & \mathcal{R}^2 \geq 8\mathcal{M}D\frac{H+P}{H+b} \end{cases} \quad (2)$$

Finally, to avoid unnecessary allocation of disk space, $D = \frac{\mathcal{R}^2(H+b)}{8\mathcal{M}(H+P)}$ and $.k = \frac{\mathcal{R}}{4\mathcal{M}}$. We refer the interested reader to [3] for a more detailed explanation of this model. Note that DRUM strategy requires that a large set of U URLs be overwritten several times to check n input URLs. This is a time consuming operation with large impact on the overall performance.

VEUNIQ takes advantage of the site order the scheduler is able to provide to maximize URL locality. This allows the overwriting of the disk repository using much less disk accesses, as we will show in the next section. The parameters used in both DRUM and VEUNIQ models are summarized in Table 1.

Variable	Meaning	Unit
b	URL average size	bytes
β	Probability of a link point to a page not in the current repository	-
D	Disk size available for buckets in DRUM	bytes
Δ	Memory buffer size for loading repository of unique URLs	bytes
H	URL hash size	bytes
k	# of URL buckets in DRUM	-
ℓ	Average number of links per page	-
$\mathcal{M}$	Memory buffer size for each opened file in DRUM	bytes
n	# of links requiring URL checking	-
N	# of repositories in VEUNIQ	-
p	Probability of URL uniqueness	-
P	Memory pointer size	bytes
r	Size of buffer used to load URLs to be checked	bytes
$\mathcal{R}$	RAM memory allocated to uniqueness checking	bytes
R_i	Disk repository of unique URLs maintained by VEUNIQ	-
U	# of URLs in the set of unique URLs stored on disk	-

Table 1. Summary of parameters used in DRUM and VEUNIQ models.

5 Algorithm VEUNIQ

In VEUNIQ, the set of unique URLs is stored in N persistent repositories R_i ($0 \leq i < N$). Each repository R_i contains URLs from a set of servers. The servers are distributed according to a hashing strategy whereas the URLs are sorted lexicographically within each server. VEUNIQ benefits from the locality of reference provided by the skewed distribution of links within each web page, which tends to reference other pages in the same server. For instance, in a sample of 400 servers crawled from the Web, the total number of links extracted was 29,580, with 18,712 (63.3%) links pointing to one of the 400 servers.

The crawler visits and updates the URLs of the repositories using a round robin strategy, so that after processing repository i, the next repository to process is $(i+1) \mod N$. At each cycle, the scheduler loads from current disk repository R_i the seed set of unique URLs. From this set, it derives the set of candidate URLs to be crawled and send it to the fetcher, which returns the set L_i. From $|L_i|$ downloaded pages, the extractor parses new URLs (E_i). These URLs ($L_i \cup E_i$) are then delivered to VEUNIQ. Note that $|L_i|$ is the number of URLs crawled in a cycle. Considering that we have about 10 links per page, we thus should set $|L_i|$ to roughly 1/11 of the number of URLs that we want VEUNIQ to verify in each cycle.

Algorithm 1 describes the usage of VEUNIQ in each crawling cycle. First, each URL found in a cycle is inserted into its corresponding buffer (lines 1 to 8). In line 4, the URLs that correspond to the current repository i are stored in buffer P_M^i and URLs corresponding to other repositories (which are expected to be the minority of them, being about 20% in our experiments) are inserted into their corresponding buffer TMP_M^j, $0 \leq j < N$ and $j \neq i$. Note that the hash function $h'(u)$ assigns each URL a number that corresponds to its appropriate repository. Further, as it takes only the URL server into account, all URLs of a same server are assigned to the same repository. After inserting all URLs in memory, the URLs related to repositories other than R_i are stored in their corresponding temporary disk files $TMP_D^{h \neq i}$ (line 10) and the memory used by them is deallocated.

Further, VEUNIQ recovers from the disk the URLs in TMP_D^i, which should also be stored in R_i and were found in previous cycles that processed other repositories than R_i. URLs from TMP_D^i are loaded and inserted in P_M^i (line 12), obtaining as a result a buffer P_M^i containing all the URLs related to repository R_i available at that moment. We then sort P_M^i in lexicographical order (line 13) and merge its content to the corresponding repository R_i on disk. After this final step, a new crawling cycle starts.

VEUNIQ Model

As in Section 4, we assume that the crawler has reached a steady state and has to check n URLs. VEUNIQ needs $\mathcal{R}$ bytes for buffer P_M^i, which achieves its maximum number of elements in line 4 of the VEUNIQ algorithm. In fact, a few constant-size read/write buffers are used to minimize seek. The size of these

Algorithm 1 Using Algorithm VEUNIQ to verify the uniqueness of URLs found in a crawling cycle and considering that the current repository is R_i.

Input: Set of URLS $L_i \cup E_i$ crawled in a cycle
Output: Disk repository R_i updated
1: **for all** u in $L_i \cup E_i$ **do**
2: Let $\mathcal{H}$ be the value obtained by applying hash function $h'(u)$
3: **if** $\mathcal{H} = i$ **then**
4: Insert u into P_M^i
5: **else**
6: Insert u into $TMP_M^{\mathcal{H}}$
7: **end if**
8: **end for**
9: **for all** $0 \leq j < N$ AND $i \neq j$ **do**
10: Move contents from TMP_M^j to TMP_D^j and delete the memory buffer TMP_M^j
11: **end for**
12: Load TMP_D^i, inserting its content into P_M^i
13: Sort P_M^i in lexicographical order
14: R_i = Merge of P_M^i and R_i (disk merge)

buffers is proportional to the size of a disk block. Since such sizes are small, they will not be considered here.

An interesting property of VEUNIQ is that it does not require much memory to achieve good performance. The only restriction is that the number of URLs in each cycle should be large enough to allow the time required for sequential disk access performed on each cycle to be higher than the time required for disk seek operations related to the change of repositories among cycles. VEUNIQ does not require much memory due to its usage of locality properties to reduce the merge costs each time the memory is filled. Notice that when using DRUM, whenever the memory is filled, it is necessary to perform a merge with the whole set of pages in the repository, an operation that makes the memory requirement a bottleneck.

In VEUNIQ, the number of URLs crawled in a cycle is chosen based on restrictions related to the scheduling process, since the algorithm can be adjusted to use a large range of RAM memory. For instance, in our crawler, a typical value is 1 million of URLs crawled per cycle, but we run experiments in machines that would allow more than 15 million URLs in memory. When VEUNIQ uses less RAM, one can increase the number of repositories, so that each merge operation in a cycle have its cost reduced. We consider this smaller dependency of RAM as one of the most important properties of VEUNIQ.

Unlike DRUM, the U unique URLs stored on disk by VEUNIQ are divided into N repositories. Thus, VEUNIQ performs a complete merge after N steps. When considering a model where disk seeks are not taken into account, we could say that VEUNIQ runs faster as we increase N. However, in practice, when we increase N two practical problems arise. First, the disk seek operations become a significant portion of the total run time (note that this also happens when

we use a large number of buckets in DRUM). Second, when N increases, the number of servers in each repository might become too small to allow effective scheduling techniques. While we do not discuss scheduling here, we remember that it is important to the crawling process.

Disk seeks play an important role in DRUM and VEUNIQ computational costs. However, disk seek costs heavily depend on the adopted hardware architecture, with many possible scenarios. As the model presented in [3] does not include disk seeks, we also decided do not include disk seeks in our cost model. This decision has no impact on the comparison with the DRUM model. Moreover, we compare VEUNIQ with DRUM using their optimal parameters, and disk seeks would not affect VEUNIQ even if we had taken disk seeks into consideration in both models. Finally, as in [3], we also assume that the URLs requiring uniqueness checking are made available to VEUNIQ as a stream in memory. Thus, its loading into memory is not modeled.

Consider that VEUNIQ is configured to take the maximum amount of memory ($\mathcal{R}$) available. Consider that the *c-th* VEUNIQ cycle processes the *i-th* repository. Then, the costs related to this run for the $\frac{n}{N}$ URLs of one repository is:

- Write temporary buffers TMP_M^j, $0 \leq j < N$, $j \neq i$ to the corresponding TMP_D^j (Algorithm 1, line 10). In each cycle, this corresponds to $\beta\mathcal{R}$, being β the maximum number of links external to current repository R_i.
- Read bucket TMP_D^i into memory (Algorithm 1, line 12). This also corresponds to $\beta\mathcal{R}$ bytes.
- Read/write the current repository R_i and update it (Algorithm 1, line 14). Note that the current repository grows along the c cycles at a rate $p\mathcal{R}$, where p is the uniqueness probability. Thus, the merge cost to update R_i is $\left[p\mathcal{R} + 2\left(\frac{U}{N}b + (c-1)p\mathcal{R}\right)\right]$ bytes.

The total number of cycles depends on the number of bytes required to store each URL in memory, which is about $b+P$. Thus, the number of URLs that can be processed at each cycle is $\frac{\mathcal{R}}{b+P}$, and the total number of cycles necessary to process $\frac{n}{N}$ URLs is $\frac{n}{N}\frac{b+P}{\mathcal{R}}$.

Considering the number of cycles and the cost per cycle, the read/write VEUNIQ overhead to check the n URLs of the N repositories is given by:

$$\begin{aligned} w(n,\mathcal{R}) &= \sum_{i=0}^{N-1} \sum_{c=1}^{\frac{n}{N}\frac{b+P}{\mathcal{R}}} \left[p\mathcal{R} + 2\left(\frac{U}{N}b + (c-1)p\mathcal{R}\right) + 2\beta\mathcal{R}\right] \\ &= nb\left[\frac{(b+P)(bnp + npP + 2bU + 2N\mathcal{R}\beta)}{bN\mathcal{R}}\right] \end{aligned} \tag{3}$$

6 Comparison Between Methods

We now compare the performance of VEUNIQ and DRUM using their computational cost model and a crawling simulation.

6.1 Computational Cost Model

Table 2 shows the overhead of the two methods as RAM size $\mathcal{R}$, disk size D, and number of URLs n increase. The overhead unit is the number of times ($\alpha(n, \mathcal{R})$) that bn bytes are written to/read from disk. The performance was calculated considering $p = 1/9$, $U = 1$ billion URLs, $b = 110$ bytes, $\ell = 59$ links per page, $P = 4$ bytes, $H = 8$ bytes, and $M = 256{,}000$ bytes. In the case of DRUM, optimum recommended values were used for D and k. Note that, despite we did not model disk seek cost, in DRUM, such a cost is proportional to the number of disk bucket files. Unlike DRUM, VEUNIQ seek cost is less sensible to the growth of its set of repositories, given by N, because most of the URLs $(1 - \beta)$ will be sent to the same repository (the current one). Thus, in this analysis, the number of repositories N starts equal k and grows proportionally to $ln(n)$. In this way, we ensure that very few repository reorganizations are carried out as n grows. As we can see, VEUNIQ outperforms DRUM for all sets of parameters.

n (Gb)	N	$\mathcal{R} = 2$ Gb $D = 22$ Tb $k = 2{,}097$		$\mathcal{R} = 4$ Gb $D = 89$ Tb $k = 4{,}194$		$\mathcal{R} = 8$ Gb $D = 354$ Tb $k = 8{,}388$		$\mathcal{R} = 16$ Gb $D = 1{,}417$ Tb $k = 16{,}777$	
		Drum	Veuniq	Drum	Veuniq	Drum	Veuniq	Drum	Veuniq
0.3	2,097	2.26	0.67	2.26	0.64	2.26	0.63	2.26	0.62
3	2,097	2.26	0.68	2.26	0.64	2.26	0.63	2.26	0.62
30	6,926	2.26	0.66	2.26	0.63	2.26	0.62	2.26	0.62
300	38,822	2.27	0.67	2.26	0.63	2.26	0.63	2.26	0.62
3,000	306,991	2.39	0.68	2.29	0.64	2.27	0.63	2.26	0.62

Table 2. Overhead $\alpha(n, \mathcal{R}) = \frac{w(n,\mathcal{R})}{nb}$ calculated for DRUM and VEUNIQ.

6.2 A Crawling Simulation

The computaional cost models for DRUM and VEUNIQ do not consider seek times and other computational costs, such as CPU costs. These points may raise questions about the usefulness of the models for comparison purposes. We thus decided to also perform a practical experiment to give more insight to the reader. However, due to space restrictions, we do not present detailed practical comparison experiments, focusing our comparison in the model, which is more hardware free, and letting detailed experiments for future work.

To evaluate the performance of the algorithms we proposed a crawling simulation. In such a simulation, VEUNIQ and DRUM take as input in each crawl cycle the following: (a) a set of crawled URLs; (b) a set of metadata collected from the pages corresponding to these URLs, and (c) a set of URLs extracted from collected pages.

For this experiment, a simulated crawling is preferable to actually carrying out a crawling over the Web, thus focusing only on the URL uniqueness verifier,

isolating its analysis from the other components of the crawler (i.e., fetcher, scheduler and URL extractor). Furthermore, simulation yields more control over the experiments, enforcing well defined limits and easing its reproduction.

For this simulation we use the ClueWeb09 web page collection[4], containing approximately 1 billion Web pages and occupies approximately 25 terabytes. We simulate the crawling of 350 million URLs, divided into 35 cycles that handle 1 million URLs each. In the simulation, 100,000 of these URLs are collected and 900,000 are extracted from the collected pages. For each cycle we measure the time it takes to update the data structures used by VEUNIQ and DRUM.

The results of this experiment are shown in Table 3. We use a public C++ implementation of DRUM for URL uniqueness verification[5]. This implementation performs the final merge of the repository by storing the URLs in a database using the BerkeleyDB DBMS. However, to ensure fairness, the times reported disregard the time spent accessing the data stored in the database. In the experiment, we also set the value of N to be equal to k, which gives DRUM an advantage in the experiments.

Figure 2 illustrates the time spent in each crawl cycle. Notice the improvements achieved by VEUNIQ over DRUM. For instance, with 350 million URLs stored, the baseline requires 82.85 seconds to enter the URLs of the current crawl cycle, while VEUNIQ spent just 9.26 seconds to accomplish the same task. Finally, we observed that the total time spent by VEUNIQ in one cycle to crawl 100,000 URLs was approximately 267.3 seconds, being 240 seconds by the fetcher, 12 seconds by the extractor, 9.3 by the uniqueness verification and 6 seconds by the scheduler. Taking out the time spent by the fetcher, which depends mainly on the bandwidth available, the percentual of the total time (27.3 seconds) spent by the uniqueness verification is approximately 34% (44% for the extractor and 22% for the scheduler).

Millions of URL		1	50	100	150	200	250	300	350
Time (s)	DRUM	16.28	26.39	35.62	44.70	53.99	64.46	73.18	82.85
	VEUNIQ	1.38	2.78	3.62	4.50	5.47	7.20	7.44	9.26

Table 3. Summary of the time required to update the respective data structures – DRUM vs. VEUNIQ.

7 Conclusions

In this paper, we presented VEUNIQ (VErifier of UNIQueness), a new algorithm for verifying URL uniqueness in a web crawler. VEUNIQ uses the idea of accumulating URLs into a RAM buffer and check several URLs sequentially in one pass. This batch disk check strategy is also used by DRUM (Disk Repository with Update Management), which is part of the IRLBot web crawler [3].

[4] ClueWeb09 Dataset, http://www.lemurproject.org/clueweb09.php
[5] http://www.codeproject.com/KB/recipes/cppdrumimplementation.aspx

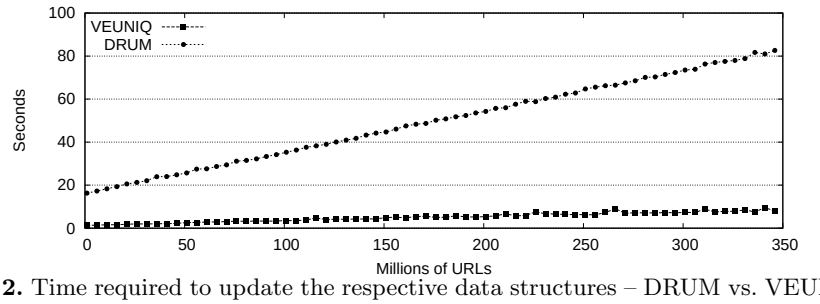

Fig. 2. Time required to update the respective data structures – DRUM vs. VEUNIQ.

The algorithm to verify URL uniqueness in DRUM was used as our comparison baseline to evaluate our proposed strategy.

VEUNIQ is a based on a novel policy for organizing the set of unique URLs according to the server they belong to, exploiting a locality of reference property inherent in Web traversals. We presented a model for both DRUM and VEUNIQ and compare their performances. We also carried out experiments which show that the adopted policy yields to an improvement in the crawling rate in comparison to the baseline algorithm.

8 Acknowledgments

This work was partially sponsored by the Brazilian National Institute of Science and Technology for the Web (grant MCT/CNPq 573871/2008-6) and authors' individual grants and scholarships from CNPq.

References

1. K. Berlt, E. Moura, A. Carvalho, M. Cristo, N. Ziviani, and T. Couto. Modeling the web as a hypergraph to compute page reputation. *Information Systems*, 35(5):530–543, 2010.
2. A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.
3. H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: Scaling to 6 billion pages and beyond. *ACM Transactions on the Web*, 3(3):1–34, 2009.
4. M. Najork and A. Heydon. High-performance web crawling. Technical report, SRC Research Report 173, Compaq Systems Research, Palo Alto, CA, 2001.
5. B. Pinkerton. Finding what people want: Experiences with the web crawler. In *WWW*, pages 30–40, 1994.
6. V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *ICDE*, pages 357–368, 2002.
7. G.-R. Xue, Q. Yang, H.-J. Zeng, Y. Yu, and Z. Chen. Exploiting the hierarchical structure for link analysis. In *SIGIR*, pages 186–193, 2005.