# IRLbot: Scaling to 6 Billion Pages and Beyond

HSIN-TSANG LEE, DEREK LEONARD, XIAOMING WANG, and DMITRI LOGUINOV
Texas A&M University

This article shares our experience in designing a Web crawler that can download billions of pages using a single-server implementation and models its performance. We first show that current crawling algorithms cannot effectively cope with the sheer volume of URLs generated in large crawls, highly branching spam, legitimate multimillion-page blog sites, and infinite loops created by server-side scripts. We then offer a set of techniques for dealing with these issues and test their performance in an implementation we call IRLbot. In our recent experiment that lasted 41 days, IRLbot running on a single server successfully crawled 6.3 billion valid HTML pages (7.6 billion connection requests) and sustained an average download rate of 319 mb/s (1,789 pages/s). Unlike our prior experiments with algorithms proposed in related work, this version of IRLbot did not experience any bottlenecks and successfully handled content from over 117 million hosts, parsed out 394 billion links, and discovered a subset of the Web graph with 41 billion unique nodes.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*; H.3.3 [**Information Storage And Retrieval**]: Information Search and Retrieval—*Search process*; C.2.0 [**Computer-Communication Networks**]: General

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: IRLbot, large scale, crawling

**ACM Reference Format:**

Lee, H.-T., Leonard, D., Wang, X., and Loguinov, D. 2009. IRLbot: Scaling to 6 billion pages and beyond. ACM Trans. Web. 3, 3, Article 8 (June 2009), 34 pages.
DOI = 10.1145/1541822.1541823 http://doi.acm.org/10.1145/1541822.1541823

## 1. INTRODUCTION

Over the last decade, the World Wide Web (WWW) has evolved from a handful of pages to billions of diverse objects. In order to harvest this enormous data repository, search engines download parts of the existing Web and offer Internet

An earlier version of the article appeared in *the World Wide Web Conference (WWW'08)* 2008.
Authors' addresses: H.-T. Lee, Microsoft Corp., Redmond, WA 98052; D. Leonard, X. Wang, D. Loguinov, Department of Computer Science, Texas A&M University, College Station, TX 77843; email: {h0l9314, dleonard, xmwang, dmitri}@cs.tamu.edu.

DOI 10.1145/1541822.1541823 http://doi.acm.org/10.1145/1541822.1541823

users access to this database through keyword search. Search engines consist of two fundamental components: *Web crawlers*, which find, download, and parse content in the WWW, and *data miners*, which extract keywords from pages, rank document importance, and answer user queries. This article does not deal with data miners, but instead focuses on the design of Web crawlers that can scale to the size of the current[1] and future Web, while implementing consistent per-Web site and per-server rate-limiting policies and avoiding being trapped in spam farms and infinite Webs. We next discuss our assumptions and explain why this is a challenging issue.

## 1.1 Scalability

With the constant growth of the Web, discovery of user-created content by Web crawlers faces an inherent trade-off between *scalability*, *performance*, and *resource usage*. The first term refers to the number of pages $N$ a crawler can handle without becoming "bogged down" by the various algorithms and data structures needed to support the crawl. The second term refers to the speed $S$ at which the crawler discovers the Web as a function of the number of pages already crawled. The final term refers to the CPU and RAM resources $\Sigma$ that are required to sustain the download of $N$ pages at an average speed $S$. In most crawlers, larger $N$ implies higher complexity of checking URL uniqueness, verifying robots.txt, and scanning the DNS cache, which ultimately results in lower $S$ and higher $\Sigma$. At the same time, higher speed $S$ requires smaller data structures, which often can be satisfied only by either lowering $N$ or increasing $\Sigma$.

Current research literature [Boldi et al. 2004a; Brin and Page 1998; Cho et al. 2006; Eichmann 1994; Heydon and Najork 1999; Internet Archive; Koht-arsa and Sanguanpong 2002; McBryan 1994; Najork and Heydon 2001; Pinkerton 2000, 1994; Shkapenyuk and Suel 2002] generally provides techniques that can solve a subset of the problem and achieve a combination of any two objectives (i.e., large slow crawls, small fast crawls, or large fast crawls with unbounded resources). They also do not analyze how the proposed algorithms scale for very large $N$ given fixed $S$ and $\Sigma$. Even assuming sufficient Internet bandwidth and enough disk space, the problem of designing a Web crawler that can support large $N$ (hundreds of billions of pages), sustain reasonably high speed $S$ (thousands of pages/s), and operate with fixed resources $\Sigma$ remains open.

## 1.2 Reputation and Spam

The Web has changed significantly since the days of early crawlers [Brin and Page 1998; Najork and Heydon 2001; Pinkerton 1994], mostly in the area of dynamically generated pages and Web spam. With server-side scripts that can create infinite loops, high-density link farms, and unlimited number of hostnames, the task of Web crawling has changed from simply doing a BFS scan of

[1]Adding the size of all top-level domains using site queries (e.g., "site:.com", "site:.uk"), Google's index size in January 2008 can be estimated at 30 billion pages and Yahoo's at 37 billion. Furthermore, Google recently reported [Official Google Blog 2008] that its crawls had accumulated links to over 1 trillion unique pages.

the WWW [Najork and Wiener 2001] to deciding in real time which sites contain useful information and giving them higher priority as the crawl progresses.

Our experience shows that BFS eventually becomes trapped in useless content, which manifests itself in multiple ways: (a) the queue of pending URLs contains a nonnegligible fraction of links from spam sites that threaten to overtake legitimate URLs due to their high branching factor; (b) the DNS resolver succumbs to the rate at which new hostnames are dynamically created within a single domain; and (c) the crawler becomes vulnerable to the delay attack from sites that purposely introduce HTTP and DNS delays in all requests originating from the crawler's IP address.

No prior research crawler has attempted to avoid spam or document its impact on the collected data. Thus, designing low-overhead and robust algorithms for computing site reputation during the crawl is the second open problem that we aim to address in this work.

### 1.3 Politeness

Even today, Web masters become easily annoyed when Web crawlers slow down their servers, consume too much Internet bandwidth, or simply visit pages with "too much" frequency. This leads to undesirable consequences including blocking of the crawler from accessing the site in question, various complaints to the ISP hosting the crawler, and even threats of legal action. Incorporating per-Web site and per-IP hit limits into a crawler is easy; however, preventing the crawler from "choking" when its entire RAM gets filled up with URLs pending for a small set of hosts is much more challenging. When $N$ grows into the billions, the crawler ultimately becomes bottlenecked by its own politeness and is then faced with a decision to suffer significant slowdown, ignore politeness considerations for certain URLs (at the risk of crashing target servers or wasting valuable bandwidth on huge spam farms), or discard a large fraction of backlogged URLs, none of which is particularly appealing.

While related work [Boldi et al. 2004a; Cho et al. 2006; Heydon and Najork 1999; Najork and Heydon 2001; Shkapenyuk and Suel 2002] has proposed several algorithms for rate-limiting host access, none of these studies has addressed the possibility that a crawler may stall due to its politeness restrictions or discussed management of rate-limited URLs that do not fit into RAM. This is the third open problem that we aim to solve in this article.

### 1.4 Our Contributions

The first part of the article presents a set of Web crawler algorithms that address the issues raised earlier and the second part briefly examines their performance in an actual Web crawl. Our design stems from three years of Web crawling experience at Texas A&M University using an implementation we call IRLbot [IRLbot 2007] and the various challenges posed in simultaneously: (1) sustaining a fixed crawling rate of several thousand pages/s; (2) downloading billions of pages; and (3) operating with the resources of a single server.

The first performance bottleneck we faced was caused by the complexity of verifying uniqueness of URLs and their compliance with robots.txt. As $N$ scales into many billions, even the disk algorithms of Najork and Heydon [2001] and Shkapenyuk and Suel [2002] no longer keep up with the rate at which new URLs are produced by our crawler (i.e., up to 184K per second). To understand this problem, we analyze the URL-check methods proposed in the literature and show that all of them exhibit severe performance limitations when $N$ becomes sufficiently large. We then introduce a new technique called *Disk Repository with Update Management* (DRUM) that can store large volumes of arbitrary hashed data on disk and implement very fast `check`, `update`, and `check+update` operations using bucket-sort. We model the various approaches and show that DRUM's overhead remains close to the best theoretically possible as $N$ reaches into the trillions of pages and that for common disk and RAM size, DRUM can be thousands of times faster than prior disk-based methods.

The second bottleneck we faced was created by multimillion-page sites (both spam and legitimate), which became backlogged in politeness rate-limiting to the point of overflowing the RAM. This problem was impossible to overcome unless politeness was tightly coupled with site reputation. In order to determine the legitimacy of a given domain, we use a very simple algorithm based on the number of incoming links from assets that spammers cannot grow to infinity. Our algorithm, which we call *Spam Tracking and Avoidance through Reputation* (STAR), dynamically allocates the budget of allowable pages for each domain and all of its subdomains in proportion to the number of in-degree links from other domains. This computation can be done in real time with little overhead using DRUM even for millions of domains in the Internet. Once the budgets are known, the rates at which pages can be downloaded from each domain are scaled proportionally to the corresponding budget.

The final issue we faced in later stages of the crawl was how to prevent livelocks in processing URLs that exceed their budget. Periodically rescanning the queue of over-budget URLs produces only a handful of good links at the cost of huge overhead. As $N$ becomes large, the crawler ends up spending all of its time cycling through failed URLs and makes very little progress. The solution to this problem, which we call *Budget Enforcement with Anti-Spam Tactics* (BEAST), involves a dynamically increasing number of disk queues among which the crawler spreads the URLs based on whether they fit within the budget or not. As a result, almost all pages from sites that significantly exceed their budgets are pushed into the last queue and are examined with lower frequency as $N$ increases. This keeps the overhead of reading spam at some fixed level and effectively prevents it from "snowballing."

The aforesaid algorithms were deployed in IRLbot [IRLbot 2007] and tested on the Internet in June through August 2007 using a single server attached to a 1gb/s backbone of Texas A&M. Over a period of 41 days, IRLbot issued 7,606,109,371 connection requests, received 7,437,281,300 HTTP responses from 117,576,295 hosts in 33,755,361 domains, and successfully downloaded $N = 6{,}380{,}051{,}942$ unique HTML pages at an average rate of 319 mb/s (1,789 pages/s). After handicapping quickly branching spam and over 30 million low-ranked domains, IRLbot parsed out 394,619,023,142 links and found 41,502,195,631 unique pages residing on 641,982,061 hosts, which explains our interest in crawlers that scale to tens and hundreds of billions of pages as we believe a good fraction of 35B URLs not crawled in this experiment contains useful content.

The rest of the article is organized as follows. Section 2 overviews related work. Section 3 defines our objectives and classifies existing approaches. Section 4 discusses how checking URL uniqueness scales with crawl size and proposes our technique. Section 5 models caching and studies its relationship with disk overhead. Section 6 discusses our approach to ranking domains and Section 7 introduces a scalable method of enforcing budgets. Section 8 summarizes our experimental statistics and Section 10 concludes.

## 2. RELATED WORK

There is only a limited number of articles describing detailed Web crawler algorithms and offering their experimental performance. First-generation designs [Eichmann 1994; McBryan 1994; Pinkerton 2000, 1994], were developed to crawl the infant Web and commonly reported collecting less than 100,000 pages. Second-generation crawlers [Boldi et al. 2004a; Cho et al. 2006; Heydon and Najork 1999; Hirai et al. 2000; Najork and Heydon 2001; Shkapenyuk and Suel 2002] often pulled several hundred million pages and involved multiple agents in the crawling process. We discuss their design and scalability issues in the next section.

Another direction was undertaken by the Internet Archive [Burner 1997; Internet Archive], which maintains a history of the Internet by downloading the same set of pages over and over. In the last 10 years, this database has collected over 85 billion pages, but only a small fraction of them are unique. Additional crawlers are Brin and Page [1998], Edwards et al. [2001], Hafri and Djeraba [2004], Koht-arsa and Sanguanpong [2002], Singh et al. [2003], and Suel et al. [2003]; however, their focus usually does not include the large scale assumed in this article and their fundamental crawling algorithms are not presented in sufficient detail to be analyzed here.

The largest prior crawl using a fully disclosed implementation appeared in Najork and Heydon [2001], where Mercator downloaded 721 million pages in 17 days. Excluding non-HTML content, which has a limited effect on scalability, this crawl encompassed $N = 473$ million HTML pages. The fastest reported crawler was Hafri and Djeraba [2004] with 816 pages/s, but the scope of their experiment was only $N = 25$ million. Finally, to our knowledge, the largest Web graph used in any article was AltaVista's 2003 crawl with 1.4B pages and 6.6B links [Gleich and Zhukov 2005].

## 3. OBJECTIVES AND CLASSIFICATION

This section formalizes the purpose of Web crawling and classifies algorithms in related work, some of which we study later in the article.

### 3.1 Crawler Objectives

We assume that the ideal task of a crawler is to start from a set of seed URLs $\Omega_0$ and eventually crawl the set of all pages $\Omega_\infty$ that can be discovered from $\Omega_0$ using HTML links. The crawler is allowed to dynamically change the order in

Table I. Comparison of Prior Crawlers and Their Data Structures

| | | `URLseen` | | `RobotsCache` | | | |
|---|---|---|---|---|---|---|---|
| Crawler | Year | RAM | Disk | RAM | Disk | DNScache | $Q$ |
| WebCrawler [Pinkerton 1994] | 1994 | database | | – | | – | database |
| Internet Archive [Burner 1997] | 1997 | site-based | – | site-based | – | site-based | RAM |
| Mercator-A [Heydon and Najork 1999] | 1999 | LRU | seek | LRU | – | – | disk |
| Mercator-B [Najork and Heydon 2001] | 2001 | LRU | batch | LRU | – | – | disk |
| Polybot [Shkapenyuk and Suel 2002] | 2001 | tree | batch | database | | database | disk |
| WebBase [Cho et al. 2006] | 2001 | site-based | – | site-based | – | site-based | RAM |
| UbiCrawler [Boldi et al. 2004a] | 2002 | site-based | – | site-based | – | site-based | RAM |

which URLs are downloaded in order to achieve a reasonably good coverage of "useful" pages $\Omega_U \subseteq \Omega_\infty$ in some finite amount of time. Due to the existence of legitimate sites with hundreds of millions of pages (e.g., ebay.com, yahoo.com, blogspot.com), the crawler cannot make any restricting assumptions on the maximum number of pages per host, the number of hosts per domain, the number of domains in the Internet, or the number of pages in the crawl. We thus classify algorithms as *nonscalable* if they impose hard limits on any of these metrics or are unable to maintain crawling speed when these parameters become very large.

We should also explain why this article focuses on the performance of a single server rather than some distributed architecture. If one server can scale to $N$ pages and maintain speed $S$, then with sufficient bandwidth it follows that $m$ servers can maintain speed $mS$ and scale to $mN$ pages by simply partitioning the set of all URLs and data structures between themselves (we assume that the bandwidth needed to shuffle the URLs between the servers is also well provisioned) [Boldi et al. 2004a; Cho and Garcia-Molina 2002; Heydon and Najork 1999; Najork and Heydon 2001; Shkapenyuk and Suel 2002]. Therefore, the aggregate performance of a server farm is ultimately governed by the characteristics of individual servers and their local limitations. We explore these limits in detail throughout the work.

## 3.2 Crawler Operation

The functionality of a basic Web crawler can be broken down into several phases: (1) removal of the next URL $u$ from the queue $Q$ of pending pages; (2) download of $u$ and extraction of new URLs $u_1, \ldots, u_k$ from $u$'s HTML tags; (3) for each $u_i$, verification of uniqueness against some structure `URLseen` and checking compliance with robots.txt using some other structure `RobotsCache`; (4) addition of passing URLs to $Q$ and `URLseen`; (5) update of `RobotsCache` if necessary. The crawler may also maintain its own `DNScache` structure when the local DNS server is not able to efficiently cope with the load (e.g., its RAM cache does not scale to the number of hosts seen by the crawler or it becomes very slow after caching hundreds of millions of records).

A summary of prior crawls and their methods in managing `URLseen`, `RobotsCache`, `DNScache`, and queue $Q$ is shown in Table I. The table demonstrates that two approaches to storing visited URLs have emerged in the literature: *RAM-only* and *hybrid RAM-disk*. In the former case [Boldi et al. 2004a;

Burner 1997; Cho et al. 2006], crawlers keep a small subset of hosts in memory and visit them repeatedly until a certain depth or some target number of pages have been downloaded from each site. URLs that do not fit in memory are discarded and sites are assumed to never have more than some fixed volume of pages. This methodology results in truncated Web crawls that require different techniques from those studied here and will not be considered in our comparison. In the latter approach [Heydon and Najork 1999; Najork and Heydon 2001; Pinkerton 1994; Shkapenyuk and Suel 2002], URLs are first checked against a buffer of popular links and those not found are examined using a disk file. The RAM buffer may be an LRU cache [Heydon and Najork 1999; Najork and Heydon 2001], an array of recently added URLs [Heydon and Najork 1999; Najork and Heydon 2001], a general-purpose database with RAM caching [Pinkerton 1994], and a balanced tree of URLs pending a disk check [Shkapenyuk and Suel 2002].

Most prior approaches keep `RobotsCache` in RAM and either crawl each host to exhaustion [Boldi et al. 2004a; Burner 1997; Cho et al. 2006] or use an LRU cache in memory [Heydon and Najork 1999; Najork and Heydon 2001]. The only hybrid approach is used in Shkapenyuk and Suel [2002], which employs a general-purpose database for storing downloaded robots.txt. Finally, with the exception of Shkapenyuk and Suel [2002], prior crawlers do not perform DNS caching and rely on the local DNS server to store these records for them.

## 4. SCALABILITY OF DISK METHODS

This section describes algorithms proposed in prior literature, analyzes their performance, and introduces our approach.

### 4.1 Algorithms

In Mercator-A [Heydon and Najork 1999], URLs that are not found in memory cache are looked up on disk by seeking within the `URLseen` file and loading the relevant block of hashes. The method clusters URLs by their site hash and attempts to resolve multiple in-memory links from the same site in one seek. However, in general, locality of parsed out URLs is not guaranteed and the worst-case delay of this method is one seek/URL and the worst-case read overhead is one block/URL. A similar approach is used in WebCrawler [Pinkerton 1994], where a general-purpose database performs multiple seeks (assuming a common B-tree implementation) to find URLs on disk.

Even with RAID, disk seeking cannot be reduced to below 3 to 5 ms, which is several orders of magnitude slower than required in actual Web crawls (e.g., 5 to 10 microseconds in IRLbot). General-purpose databases that we have examined are much worse and experience a significant slowdown (i.e., 10 to 50 ms per lookup) after about 100 million inserted records. Therefore, these approaches do not appear viable unless RAM caching can achieve some enormously high hit rates (i.e., 99.7% for IRLbot). We examine whether this is possible in the next section when studying caching.

Mercator-B [Najork and Heydon 2001] and Polybot [Shkapenyuk and Suel 2002] use a so-called *batch disk check*; they accumulate a buffer of URLs in

Table II. Parameters of the Model

| Variable | Meaning | Units |
|---|---|---|
| $N$ | Crawl scope | pages |
| $p$ | Probability of URL uniqueness | – |
| $U$ | Initial size of URLseen file | pages |
| $R$ | RAM size | bytes |
| $l$ | Average number of links per page | – |
| $n$ | Links requiring URL check | – |
| $q$ | Compression ratio of URLs | – |
| $b$ | Average size of URLs | bytes |
| $H$ | URL hash size | bytes |
| $P$ | Memory pointer size | bytes |

memory and then merge it with a sorted URLseen file in one pass. Mercator-B stores only hashes of new URLs in RAM and places their text on disk. In order to retain the mapping from hashes to the text, a special pointer is attached to each hash. After the memory buffer is full, it is sorted in place and then compared with blocks of URLseen as they are read from disk. Nonduplicate URLs are merged with those already on disk and written into the new version of URLseen. Pointers are then used to recover the text of unique URLs and append it to the disk queue.

Polybot keeps the entire URLs (i.e., actual strings) in memory and organizes them into a binary search tree. Once the tree size exceeds some threshold, it is merged with the disk file URLseen, which contains compressed URLs already seen by the crawler. Besides being CPU intensive, this method has to perform more frequent scans of URLseen than Mercator-B due to the less-efficient usage of RAM.

## 4.2 Modeling Prior Methods

Assume the crawler is in some steady state where the probability of uniqueness $p$ among new URLs remains constant (we verify that this holds in practice later in the article). Further assume that the current size of URLseen is $U$ entries, the size of RAM allocated to URL checks is $R$, the average number of links per downloaded page is $l$, the average URL length is $b$, the URL compression ratio is $q$, and the crawler expects to visit $N$ pages. It then follows that $n = lN$ links must pass through URL check, $np$ of them are unique, and $bq$ is the average number of bytes in a compressed URL. Finally, denote by $H$ the size of URL hashes used by the crawler and $P$ the size of a memory pointer (Table II summarizes this notation). Then we have the following result.

THEOREM 1. *The combined read-write overhead of URLseen batch disk check in a crawl of size $N$ pages is $\omega(n, R) = \alpha(n, R)bn$ bytes, where for Mercator-B*

$$\alpha(n, R) = \frac{2(2UH + pHn)(H + P)}{bR} + 2 + p \qquad (1)$$

*and for Polybot*

$$\alpha(n, R) = \frac{2(2Ubq + pbqn)(b + 4P)}{bR} + p. \qquad (2)$$

PROOF. To prevent locking on URL check, both Mercator-B and Polybot must use two buffers of accumulated URLs (i.e., one for checking the disk and the other for newly arriving data). Assume this half-buffer allows storage of $m$ URLs (i.e., $m = R/[2(H + P)]$ for Mercator-B and $m = R/[2(b+4P)]$ for Polybot) and the size of the initial disk file is $f$ (i.e., $f = UH$ for Mercator-B and $f = Ubq$ for Polybot).

For Mercator-B, the $i$th iteration requires writing/reading of $mb$ bytes of arriving URL strings, reading the current `URLseen`, writing it back, and appending $mp$ hashes to it, namely, $2f + 2mb + 2mpH(i - 1) + mpH$ bytes. This leads to the following after adding the final overhead to store $pbn$ bytes of unique URLs in the queue:

$$\begin{aligned}\omega(n) &= \sum_{i=1}^{n/m} \bigl(2f + 2mb + 2mpHi - mpH\bigr) + pbn \\ &= nb\left(\frac{2(2UH + pHn)(H + P)}{bR} + 2 + p\right). \qquad (3)\end{aligned}$$

For Polybot, the $i$th iteration has overhead $2f + 2mpbq(i - 1) + mpbq$, which yields

$$\begin{aligned}\omega(n) &= \sum_{i=1}^{n/m} \bigl(2f + 2mpbqi - mpbq\bigr) + pbn \\ &= nb\left(\frac{2(2Ubq + pbqn)(b + 4P)}{bR} + p\right) \qquad (4)\end{aligned}$$

and leads to (2). □

This result shows that $\omega(n, R)$ is a product of two elements: the number of bytes $bn$ in all parsed URLs and how many times $\alpha(n, R)$ they are written to/read from disk. If $\alpha(n, R)$ grows with $n$, the crawler's overhead will scale superlinearly and may eventually become overwhelming to the point of stalling the crawler. As $n \to \infty$, the quadratic term in $\omega(n, R)$ dominates the other terms, which places Mercator-B's asymptotic performance at

$$\omega(n, R) = \frac{2(H + P)pH}{R} n^2 \qquad (5)$$

and that of Polybot at

$$\omega(n, R) = \frac{2(b + 4P)pbq}{R} n^2. \qquad (6)$$

The ratio of these two terms is

$$\frac{(H + P)H}{bq(b + 4P)}, \qquad (7)$$

which for the IRLbot case with $H = 8$ bytes/hash, $P = 4$ bytes/pointer, $b = 110$ bytes/URL, and using very optimistic $bq = 5.5$ bytes/URL shows that Mercator-B is roughly 7.2 times faster than Polybot as $n \to \infty$.

The best performance of any method that stores the text of URLs on disk before checking them against `URLseen` (e.g., Mercator-B) is $\alpha_{min} = 2+p$, which is

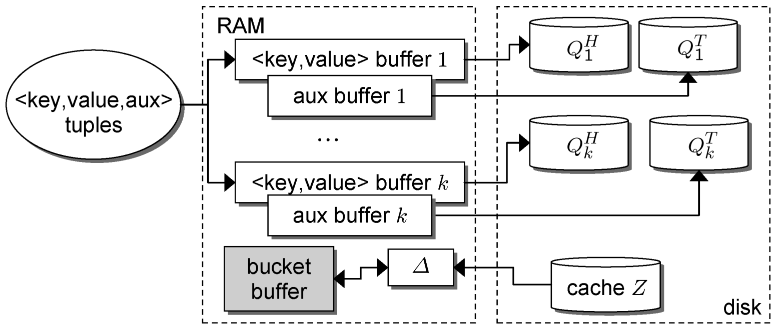


Fig. 1. Operation of DRUM.

the overhead needed to write all *bn* bytes to disk, read them back for processing, and then append *bpn* bytes to the queue. Methods with memory-kept URLs (e.g., Polybot) have an absolute lower bound of $\alpha'_{min} = p$, which is the overhead needed to write the unique URLs to disk. Neither bound is achievable in practice, however.

## 4.3 DRUM

We now describe the URL-check algorithm used in IRLbot, which belongs to a more general framework we call *Disk Repository with Update Management* (DRUM). The purpose of DRUM is to allow for efficient storage of large collections of `<key,value>` pairs, where `key` is a unique identifier (hash) of some data and `value` is arbitrary information attached to the key. There are three supported operations on these pairs: `check`, `update`, and `check+update`. In the first case, the incoming set of data contains keys that must be checked against those stored in the disk cache and classified as being duplicate or unique. For duplicate keys, the value associated with each key can be optionally retrieved from disk and used for some processing. In the second case, the incoming list contains `<key,value>` pairs that need to be merged into the existing disk cache. If a given key exists, its value is updated (e.g., overridden or incremented); if it does not, a new entry is created in the disk file. Finally, the third operation performs both check and update in one pass through the disk cache. Also note that DRUM may be supplied with a mixed list where some entries require just a check, while others need an update.

A high-level overview of DRUM is shown in Figure 1. In the figure, a continuous stream of tuples `<key,value,aux>` arrives into DRUM, where `aux` is some auxiliary data associated with each `key`. DRUM spreads pairs `<key,value>` between $k$ disk buckets $Q_1^H, \ldots, Q_k^H$ based on their `key` (i.e., all keys in the same bucket have the same bit-prefix). This is accomplished by feeding pairs `<key,value>` into $k$ memory arrays of size $M$ each and then continuously writing them to disk as the buffers fill up. The `aux` portion of each key (which usually contains the text of URLs) from the $i$th bucket is kept in a separate file $Q_i^T$ in the same FIFO order as pairs `<key,value>` in $Q_i^H$. Note that to maintain fast sequential writing/reading and avoid segmentation in the file-allocation table, all buckets are preallocated on disk before they are used.

Once the largest bucket reaches a certain size $r < R$, the following process is repeated for $i = 1, \ldots, k$: (1) bucket $Q_i^H$ is read into the *bucket buffer* shown in Figure 1 and sorted; (2) the disk file $Z$ is sequentially read in chunks of $\Delta$ bytes and compared with the keys in bucket $Q_i^H$ to determine their uniqueness; (3) those `<key,value>` pairs in $Q_i^H$ that require an `update` are merged with the contents of the disk cache and written to the updated version of $Z$; (4) after all unique keys in $Q_i^H$ are found, their original order is restored, $Q_i^T$ is sequentially read into memory in blocks of size $\Delta$, and the corresponding `aux` portion of each unique key is sent for further processing (see the following). An important aspect of this algorithm is that all buckets are checked in *one* pass through disk file $Z$.[2]

We now explain how DRUM is used for storing crawler data. The most important DRUM object is `URLseen`, which implements only one operation: `check+update`. Incoming tuples are `<URLhash,-,URLtext>`, where the key is an 8-byte hash of each URL, the value is empty, and the auxiliary data is the URL string. After all unique URLs are found, their text strings (`aux` data) are sent to the next queue for possible crawling. For caching robots.txt, we have another DRUM structure called `RobotsCache`, which supports asynchronous check and update operations. For checks, it receives tuples `<HostHash,-,URLtext>` and for updates `<HostHash,HostData,->`, where `HostData` contains the robots.txt file, IP address of the host, and optionally other host-related information. The last DRUM object of this section is called `RobotsRequested` and is used for storing the hashes of sites for which a robots.txt has been requested. Similar to `URLseen`, it only supports simultaneous `check+update` and its incoming tuples are `<HostHash,-,HostText>`.

Figure 2 shows the flow of new URLs produced by the crawling threads. They are first sent directly to `URLseen` using `check+update`. Duplicate URLs are discarded and unique ones are sent for verification of their compliance with the budget (both STAR and BEAST are discussed later in the article). URLs that pass the budget are queued to be checked against robots.txt using `RobotsCache`. URLs that have a matching robots.txt file are classified immediately as passing or failing. Passing URLs are queued in $Q$ and later downloaded by the crawling threads. Failing URLs are discarded.

URLs that do not have a matching robots.txt are sent to the back of queue $Q_R$ and their hostnames are passed through `RobotsRequested` using `check+update`. Sites whose hash is not already present in this file are fed through queue $Q_D$ into a special set of threads that perform DNS lookups and download robots.txt. They subsequently issue a batch `update` to `RobotsCache` using DRUM. Since in steady state (i.e., excluding the initial phase) the time needed to download robots.txt is much smaller than the average delay in $Q_R$, each URL makes no more than one cycle through this loop. In addition, when `RobotsCache` detects that certain robots.txt or DNS records have become outdated, it marks all corresponding URLs as “unable to check, outdated records,” which forces

[2]Note that disk bucket-sort is a well-known technique that exploits uniformity of keys; however, its usage in checking URL uniqueness and the associated performance model of Web crawling has not been explored before.

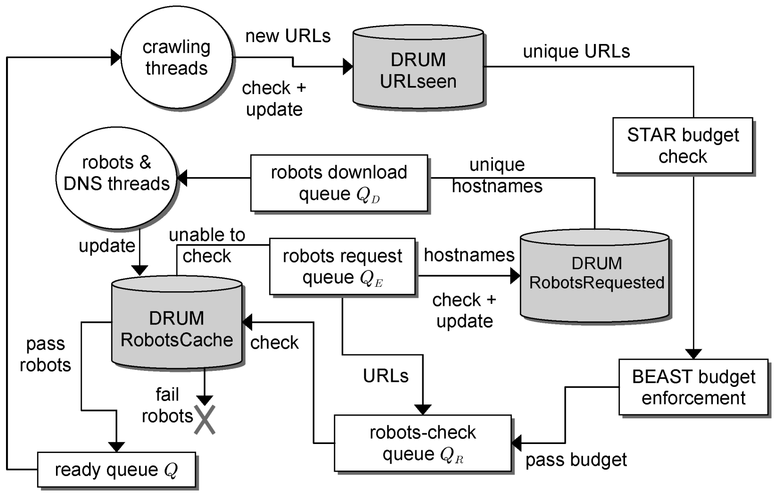


Fig. 2. High-level organization of IRLbot.

`RobotsRequested` to pull a new set of exclusion rules and/or perform another DNS lookup. Old records are automatically expunged during the update when `RobotsCache` is rewritten.

It should be noted that all queues in Figure 2 are stored on disk and can support as many hostnames, URLs, and robots.txt exception rules as disk space allows.

## 4.4 DRUM Model

Assume that the crawler maintains a buffer of size $M = 256$KB for each open file and that the hash bucket size $r$ must be at least $\Delta = 32$MB to support efficient reading during the check-merge phase. Further assume that the crawler can use up to $D$ bytes of disk space for this process. Then we have the following result.

THEOREM 2. *Assuming that $R \geq 2\Delta(H+P)/H$, DRUM's `URLseen` overhead is $\omega(n, R) = \alpha(n, R)bn$ bytes, where*

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

*and $\Lambda = 8MD(H+P)/(H+b)$.*

PROOF. Memory $R$ needs to support $2k$ open file buffers and one block of URL hashes that are loaded from $Q_i^H$. In order to compute block size $r$, recall that it gets expanded by a factor of $(H+P)/H$ when read into RAM due to the addition of a pointer to each hash value. We thus obtain that $r(H+P)/H + 2Mk = R$ or

$$r = \frac{(R-2Mk)H}{H+P}. \tag{9}$$

Our disk restriction then gives us that the size of all buckets $kr$ and their text $krb/H$ must be equal to $D$.

$$kr + \frac{krb}{H} = \frac{k(H+b)(R-2Mk)}{H+P} = D \tag{10}$$

It turns out that not all pairs $(R, k)$ are feasible. The reason is that if $R$ is set too small, we are not able to fill all of $D$ with buckets since $2Mk$ will leave no room for $r \geq \Delta$. Rewriting (10), we obtain a quadratic equation $2Mk^2 - Rk + A = 0$, where $A = (H+P)D/(H+b)$. If $R^2 < 8MA$, we have no solution and thus $R$ is insufficient to support $D$. In that case, we need to maximize $k(R-2Mk)$ subject to $k \leq k_m$, where

$$k_m = \frac{1}{2M}\left(R - \frac{\Delta(H+P)}{H}\right) \tag{11}$$

is the maximum number of buckets that still leave room for $r \geq \Delta$. Maximizing $k(R-2Mk)$, we obtain the optimal point $k_0 = R/(4M)$. Assuming that $R \geq 2\Delta(H+P)/H$, condition $k_0 \leq k_m$ is always satisfied. Using $k_0$ buckets brings our disk usage to $D' = (H+b)R^2/[8M(H+P)]$, which is always less than $D$.

In the case $R^2 \geq 8MA$, we can satisfy $D$ and the correct number of buckets $k$ is given by two choices.

$$k = \frac{R \pm \sqrt{R^2 - 8MA}}{4M} \tag{12}$$

The reason why we have two values is that we can achieve $D$ either by using few buckets (i.e., $k$ is small and $r$ is large) or many buckets (i.e., $k$ is large and $r$ is small). The correct solution is to take the smaller root to minimize the number of open handles and disk fragmentation. Putting things together,

$$k_1 = \frac{R - \sqrt{R^2 - 8MA}}{4M}. \tag{13}$$

Note that we still need to ensure $k_1 \leq k_m$, which holds when

$$R \geq \frac{\Delta(H+P)}{H} + \frac{2MAH}{\Delta(H+P)}. \tag{14}$$

Given that $R \geq 2\Delta(H+P)/H$ from the statement of the theorem, it is easy to verify that (14) is always satisfied.

Next, for the $i$th iteration that fills up all $k$ buckets, we need to write/read $Q_i^T$ once (overhead $2krb/H$) and read/write each bucket once as well (overhead $2kr$). The remaining overhead is reading/writing `URLseen` (overhead $2f + 2krp(i-1)$) and appending the new URL hashes (overhead $krp$). We thus obtain that we need $nH/(kr)$ iterations and

$$\begin{aligned}\omega(n,R) &= \sum_{i=1}^{nH/(kr)} \left(2f + \frac{2krb}{H} + 2kr + 2krpi - krp\right) + pbn \\ &= nb\left(\frac{(2UH + pHn)H}{bkr} + 2 + p + \frac{2H}{b}\right). \end{aligned} \tag{15}$$

Table III. Overhead $\alpha(n, R)$ for $R = 1$GB, $D = 4.39$TB

| $N$ | Mercator-B | Polybot | DRUM |
|---|---|---|---|
| 800M | 11.6 | 69 | 2.26 |
| 8B | 93 | 663 | 2.35 |
| 80B | 917 | 6,610 | 3.3 |
| 800B | 9,156 | 66,082 | 12.5 |
| 8T | 91,541 | 660,802 | 104 |

Recalling our two conditions, we use $k_0 r = HR^2/[8M(H+P)]$ for $R^2 < 8MA$ to obtain

$$\omega(n, R) = nb\left(\frac{8M(H+P)(2UH + pHn)}{bR^2} + 2 + p + \frac{2H}{b}\right). \tag{16}$$

For the other case $R^2 \geq 8MA$, we have $k_1 r = DH/(H+b)$ and thus get

$$\omega(n, R) = nb\left(\frac{(H+b)(2UH + pHn)}{bD} + 2 + p + \frac{2H}{b}\right), \tag{17}$$

which leads to the statement of the theorem. □

It follows from the proof of Theorem 2 that in order to match $D$ to a given RAM size $R$ and avoid unnecessary allocation of disk space, we should operate at the optimal point given by $R^2 = \Lambda$.

$$D_{opt} = \frac{R^2(H+b)}{8M(H+P)} \tag{18}$$

For example, $R = 1$GB produces $D_{opt} = 4.39$TB and $R = 2$GB produces $D_{opt} = 17$TB. For $D = D_{opt}$, the corresponding number of buckets is $k_{opt} = R/(4M)$, the size of the bucket buffer is $r_{opt} = RH/[2(H+P)] \approx 0.33R$, and the leading quadratic term of $\omega(n, R)$ in (8) is now $R/(4M)$ times smaller than in Mercator-B. This ratio is $1,000$ for $R = 1$GB and $8,000$ for $R = 8$GB. The asymptotic speedup in either case is significant.

Finally, observe that the best possible performance of any method that stores both hashes and URLs on disk is $\alpha''_{min} = 2 + p + 2H/b$.

## 4.5 Comparison

We next compare disk performance of the studied methods when nonquadratic terms in $\omega(n, R)$ are nonnegligible. Table III shows $\alpha(n, R)$ of the three studied methods for fixed RAM size $R$ and disk $D$ as $N$ increases from 800 million to 8 trillion ($p = 1/9$, $U = 100$M pages, $b = 110$ bytes, $l = 59$ links/page). As $N$ reaches into the trillions, both Mercator-B and Polybot exhibit overhead that is thousands of times larger than the optimal and invariably become "bogged down" in rewriting `URLseen`. On the other hand, DRUM stays within a factor of 50 from the best theoretically possible value (i.e., $\alpha''_{min} = 2.256$) and does not sacrifice nearly as much performance as the other two methods.

Since disk size $D$ is likely to be scaled with $N$ in order to support the newly downloaded pages, we assume for the next example that $D(n)$ is the maximum of 1TB and the size of unique hashes appended to `URLseen` during the crawl

Table IV. Overhead $\alpha(n, R)$ for $D = D(n)$

| | $R = 4$GB | | $R = 8$GB | |
|---|---|---|---|---|
| $N$ | Mercator-B | DRUM | Mercator-B | DRUM |
| 800M | 4.48 | 2.30 | 3.29 | 2.30 |
| 8B | 25 | 2.7 | 13.5 | 2.7 |
| 80B | 231 | 3.3 | 116 | 3.3 |
| 800B | 2,290 | 3.3 | 1,146 | 3.3 |
| 8T | 22,887 | 8.1 | 11,444 | 3.7 |

of $N$ pages, namely, $D(n) = \max(pHn, 10^{12})$. Table IV shows how dynamically scaling disk size allows DRUM to keep the overhead virtually constant as $N$ increases.

To compute the average crawling rate that the previous methods support, assume that $W$ is the average disk I/O speed and consider the next result.

THEOREM 3. *Maximum download rate (in pages/s) supported by the disk portion of URL uniqueness checks is*

$$S_{disk} = \frac{W}{\alpha(n, R)bl}. \tag{19}$$

PROOF. The time needed to perform uniqueness checks for $n$ new URLs is spent in disk I/O involving $\omega(n, R) = \alpha(n, R)bn = \alpha(n, R)blN$ bytes. Assuming that $W$ is the average disk I/O speed, it takes $N/S$ seconds to generate $n$ new URLs and $\omega(n, R)/W$ seconds to check their uniqueness. Equating the two entities, we have (19). □

We use IRLbot's parameters to illustrate the applicability of this theorem. Neglecting the process of appending new URLs to the queue, the crawler's read and write overhead is symmetric. Then, assuming IRLbot's 1GB/s read speed and 350MB/s write speed (24-disk RAID-5), we obtain that its average disk read-write speed is equal to 675MB/s. Allocating 15% of this rate for checking URL uniqueness,[3] the effective disk bandwidth of the server can be estimated at $W = 101.25$MB/s. Given the conditions of Table IV for $R = 8$GB and assuming $N = 8$ trillion pages, DRUM yields a sustained download rate of $S_{disk} = 4,192$ pages/s (i.e., 711mb/s using IRLbot's average HTML page size of 21.2KB). In crawls of the same scale, Mercator-B would be $3,075$ times slower and would admit an average rate of only 1.4 pages/s. Since with these parameters Polybot is 7.2 times slower than Mercator-B, its average crawling speed would be 0.2 pages/s.

## 5. CACHING

To understand whether URL caching in memory provides improved performance, we must consider a complex interplay between the available CPU capacity, spare RAM size, disk speed, performance of the caching algorithm, and crawling rate. This is a three-stage process: We first examine how cache size and crawl speed affect the hit rate, then analyze the CPU restrictions of caching, and

[3]Additional disk I/O is needed to verify robots.txt, perform reputation analysis, and enforce budgets.

Table V. LRU Hit Rates Starting at $N_0$ Crawled Pages

| Cache elements $E$ | $N_0 = 1\text{B}$ | $N_0 = 4\text{B}$ |
|---|---|---|
| 256K | 19% | 16% |
| 4M | 26% | 22% |
| 8M | 68% | 59% |
| 16M | 71% | 67% |
| 64M | 73% | 73% |
| 512M | 80% | 78% |

finally couple them with RAM/disk limitations using analysis in the previous section.

## 5.1 Cache Hit Rate

Assume that $c$ bytes of RAM are available to a `URLseen` cache whose entries incur fixed overhead $\gamma$ bytes. Then $E = c/\gamma$ is the maximum number of elements stored in the cache at any time. Then define $\pi(c, S)$ to be the cache miss rate under crawling speed $S$ pages/s and cache size $c$. The reason why $\pi$ depends on $S$ is that the faster the crawl, the more pages it produces between visits to the same site, which is where duplicate links are most prevalent. Defining $\tau_h$ to be the per-host visit delay, common sense suggests that $\pi(c, S)$ should depend not only on $c$, but also on $\tau_h l S$, which is the number of links parsed from all downloaded pages before the crawler returns to the same Web site.

Table V shows LRU cache hit rates $1 - \pi(c, S)$ during several stages of our crawl. We seek in the trace file to the point where the crawler has downloaded $N_0$ pages and then simulate LRU hit rates by passing the next $10E$ URLs discovered by the crawler through the cache. As the table shows, a significant jump in hit rates happens between 4M and 8M entries. This is consistent with IRLbot's peak value of $\tau_h l S \approx 7.3$ million. Note that before cache size reaches this value, most hits in the cache stem from redundant links within the same page. As $E$ starts to exceed $\tau_h l S$, popular URLs on each site survive between repeat visits and continue staying in the cache as long as the corresponding site is being crawled. Additional simulations confirming this effect are omitted for brevity.

Unlike Broder et al. [2003], which suggests that $E$ be set 100 to 500 times larger than the number of threads, our results show that $E$ must be slightly larger than $\tau_h l S$ to achieve a 60% hit rate and as high as $10\tau_h l S$ to achieve 73%.

## 5.2 Cache Speed

Another aspect of keeping a RAM cache is the speed at which potentially large memory structures must be checked and updated as new URLs keep pouring in. Since searching large trees in RAM usually results in misses in the CPU cache, some of these algorithms can become very slow as the depth of the search increases. Define $0 \le \phi(S) \le 1$ to be the average CPU utilization of the server crawling at $S$ pages/s and $\mu(c)$ to be the number of URLs/s that a cache of size $c$ can process on an unloaded server. Then, we have the following result.

Table VI. Insertion Rate and Maximum Crawling Speed from (22)

| Method | $\mu(c)$ URLs/s | Size $c$ | $S_{cache}(c)$ |
|---|---|---|---|
| Balanced tree (strings) | 113K | 2.2GB | 1, 295 |
| Tree-based LRU (8-byte int) | 185K | 1.6GB | 1, 757 |
| Balanced tree (8-byte int) | 416K | 768MB | 2, 552 |
| CLOCK (8-byte int) | 2M | 320MB | 3, 577 |

THEOREM 4. *Assuming $\phi(S)$ is monotonically nondecreasing, the maximum download rate $S_{cache}$ (in pages/s) supported by URL caching is*

$$S_{cache}(c) = g^{-1}(\mu(c)), \tag{20}$$

*where $g^{-1}$ is the inverse of $g(x) = lx/(1 - \phi(x))$.*

PROOF. We assume that caching performance linearly depends on the available CPU capacity, that is, if fraction $\phi(S)$ of the CPU is allocated to crawling, then the caching speed is $\mu(c)(1 - \phi(S))$ URLs/s. Then, the maximum crawling speed would match the rate of URL production to that of the cache, namely,

$$lS = \mu(c)(1 - \phi(S)). \tag{21}$$

Rewriting (21) using $g(x) = lx/(1 - \phi(x))$, we have $g(S) = \mu(c)$, which has a unique solution $S = g^{-1}(\mu(c))$ since $g(x)$ is a strictly increasing function with a proper inverse. □

For the common case $\phi(S) = S/S_{max}$, where $S_{max}$ is the server's maximum (i.e., CPU-limited) crawling rate in pages/s, (20) yields a very simple expression.

$$S_{cache}(c) = \frac{\mu(c)S_{max}}{lS_{max} + \mu(c)} \tag{22}$$

To show how to use the preceding result, Table VI compares the speed of several memory structures on the IRLbot server using $E = 16$M elements and displays model (22) for $S_{max} = 4{,}000$ pages/s. As can be seen in the table, insertion of text URLs into a balanced tree (used in Polybot [Shkapenyuk and Suel 2002]) is the slowest operation that also consumes the most memory. The speed of classical LRU caching (185K/s) and search trees with 8-byte keys (416K/s) is only slightly better since both use multiple (i.e., $\log_2 E$) jumps through memory. CLOCK [Broder et al. 2003], which is a space- and time-optimized approximation to LRU, achieves a much better speed (2M/s), requires less RAM, and is suitable for crawling rates up to 3,577 pages/s on this server. The important lesson of this section is that caching may be detrimental to a crawler's performance if it is implemented inefficiently or $c$ is chosen too large, which would lead to $S_{cache}(c) < S_{disk}$ and a lower crawling speed compared to the noncaching scenario.

After experimentally determining $\mu(c)$ and $\phi(S)$, we can easily compute $S_{cache}(c)$ from (20); however, this metric by itself does not determine whether caching should be enabled or even how to select the optimal cache size $c$. Even though caching reduces the disk overhead by sending $\pi n$ rather than $n$ URLs to be checked against the disk, it also consumes more memory and leaves less space for the buffer of URLs in RAM, which in turn results in more scans

Table VII. Overhead $\alpha(\pi n, R - c)$ for $D = D(n), \pi = 0.33, c = 320$MB

| | $R = 4$GB | | $R = 8$GB | |
|---|---|---|---|---|
| $N$ | Mercator-B | DRUM | Mercator-B | DRUM |
| 800M | 3.02 | 2.27 | 2.54 | 2.27 |
| 8B | 10.4 | 2.4 | 6.1 | 2.4 |
| 80B | 84 | 3.3 | 41 | 3.3 |
| 800B | 823 | 3.3 | 395 | 3.3 |
| 8T | 8, 211 | 4.5 | 3, 935 | 3.3 |

through disk to determine URL uniqueness. Understanding this trade-off involves careful modeling of hybrid RAM-disk algorithms, which we perform next.

## 5.3 Hybrid Performance

We now address the issue of how to assess the performance of disk-based methods with RAM caching. Mercator-A improves performance by a factor of $1/\pi$ since only $\pi n$ URLs are sought from disk. Given common values of $\pi \in [0.25, 0.35]$ in Table V, this optimization results in a 2.8 to 4 times speedup, which is clearly insufficient for making this method competitive with the other approaches.

Mercator-B, Polybot, and DRUM all exhibit new overhead

$$\omega(n, R) = \alpha(\pi(c, S)n, R - c)b\pi(c, S)n \tag{23}$$

with $\alpha(n, R)$ taken from the appropriate model. As $n \to \infty$ and assuming $c \ll R$, all three methods decrease $\omega$ by a factor of $\pi^{-2} \in [8, 16]$ for $\pi \in [0.25, 0.35]$. For $n \ll \infty$, however, only the linear factor $b\pi(c, S)n$ enjoys an immediate reduction, while $\alpha(\pi(c, S)n, R - c)$ may or may not change depending on the dominance of the first term in (1), (2), and (8), as well as the effect of reduced RAM size $R - c$ on the overhead. Table VII shows one example where $c = 320$MB ($E = 16$M elements, $\gamma = 20$ bytes/element, $\pi = 0.33$) occupies only a small fraction of $R$. Notice in the table that caching can make Mercator-B's disk overhead close to optimal for small $N$, which nevertheless does not change its scaling performance as $N \to \infty$.

Since $\pi(c, S)$ depends on $S$, determining the maximum speed a hybrid approach supports is no longer straightforward.

THEOREM 5. *Assuming $\pi(c, S)$ is monotonically nondecreasing in $S$, the maximum download rate $S_{hybrid}$ supported by disk algorithms with RAM caching is*

$$S_{hybrid}(c) = h^{-1}\left(\frac{W}{bl}\right), \tag{24}$$

*where $h^{-1}$ is the inverse of*

$$h(x) = x\alpha(\pi(c, x)n, R - c)\pi(c, x).$$

PROOF. From (19), we have

$$S = \frac{W}{\alpha(\pi(c, S)n, R - c)b\pi(c, S)l}, \tag{25}$$

Table VIII. Maximum Hybrid Crawling Rate $\max_c S_{hybrid}(c)$ for $D = D(n)$

| | $R = 4$GB | | $R = 8$GB | |
|---|---|---|---|---|
| $N$ | Mercator-B | DRUM | Mercator-B | DRUM |
| 800M | 18,051 | 26,433 | 23,242 | 26,433 |
| 8B | 6,438 | 25,261 | 10,742 | 25,261 |
| 80B | 1,165 | 18,023 | 2,262 | 18,023 |
| 800B | 136 | 18,023 | 274 | 18,023 |
| 8T | 13.9 | 11,641 | 27.9 | 18,023 |

which can be written as $h(S) = W/(bl)$. The solution to this equation is $S = h^{-1}(W/(bl))$ where as before the inverse $h^{-1}$ exists due to the strict monotonicity of $h(x)$. □

To better understand (24), we show an example of finding the best cache size $c$ that maximizes $S_{hybrid}(c)$ assuming $\pi(c, S)$ is a step function of hit rates derived from Table V. Specifically, $\pi(c, S) = 1$ if $c = 0$, $\pi(c, S) = 0.84$ if $0 < c < \gamma\tau_h l S$, 0.41 if $c < 4\gamma\tau_h l S$, 0.27 if $c < 10\gamma\tau_h l S$, and 0.22 for larger $c$. Table VIII shows the resulting crawling speed in pages/s after maximizing (24) with respect to $c$. As before, Mercator-B is close to optimal for small $N$ and large $R$, but for $N \to \infty$ its performance degrades. DRUM, on the other hand, maintains at least 11,000 pages/s over the entire range of $N$. Since these examples use large $R$ in comparison to the cache size needed to achieve nontrivial hit rates, the values in this table are almost inversely proportional to those in Table VII, which can be used to ballpark the maximum value of (24) without inverting $h(x)$.

Knowing function $S_{hybrid}$ from (24), we need to couple it with the performance of the caching algorithm to obtain the true optimal value of $c$. We have

$$c_{opt} = \arg\max_{c\in[0,R]} \min(S_{cache}(c), S_{hybrid}(c)), \tag{26}$$

which is illustrated in Figure 3. On the left of the figure, we plot some hypothetical functions $S_{cache}(c)$ and $S_{hybrid}(c)$ for $c \in [0, R]$. Assuming that $\mu(0) = \infty$, the former curve always starts at $S_{cache}(0) = S_{max}$ and is monotonically nonincreasing. For $\pi(0, S) = 1$, the latter function starts at $S_{hybrid}(0) = S_{disk}$ and tends to zero as $c \to R$, but not necessarily monotonically. On the right of the figure, we show the supported crawling rate $\min(S_{cache}(c), S_{hybrid}(c))$ whose maximum point corresponds to the pair $(c_{opt}, S_{opt})$. If $S_{opt} > S_{disk}$, then caching should be enabled with $c = c_{opt}$; otherwise, it should be disabled. The most common case when the crawler benefits from disabling the cache is when $R$ is small compared to $\gamma\tau_h l S$ or the CPU is the bottleneck (i.e., $S_{cache} < S_{disk}$).

## 6. SPAM AND REPUTATION

This section explains the necessity for detecting spam during crawls and proposes a simple technique for computing domain reputation in real time.

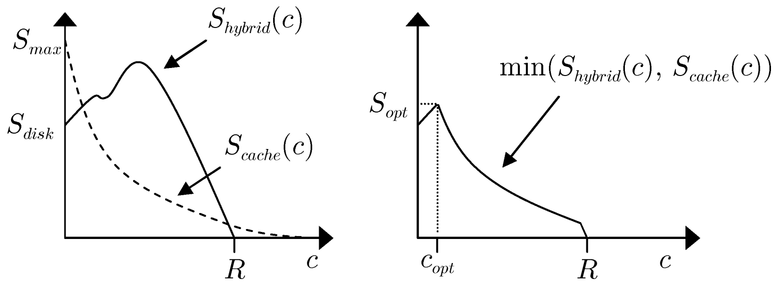


Fig. 3. Finding optimal cache size $c_{opt}$ and optimal crawling speed $S_{opt}$.

## 6.1 Problems with Breadth-First Search

Prior crawlers [Cho et al. 2006; Heydon and Najork 1999; Najork and Heydon 2001; Shkapenyuk and Suel 2002] have no documented spam-avoidance algorithms and are typically assumed to perform BFS traversals of the Web graph [Najork and Wiener 2001]. Several studies [Arasu et al. 2001; Boldi et al. 2004b] have examined in simulations the effect of changing crawl order by applying bias towards more popular pages. The conclusions are mixed and show that PageRank order [Brin and Page 1998] can be sometimes marginally better than BFS [Arasu et al. 2001] and sometimes marginally worse [Boldi et al. 2004b], where the metric by which they are compared is the rate at which the crawler discovers popular pages.

While BFS works well in simulations, its performance on infinite graphs and/or in the presence of spam farms remains unknown. Our early experiments show that crawlers eventually encounter a quickly branching site that will start to dominate the queue after 3 to 4 levels in the BFS tree. Some of these sites are spam-related with the aim of inflating the page rank of target hosts, while others are created by regular users sometimes for legitimate purposes (e.g., calendars, testing of asp/php engines), sometimes for questionable purposes (e.g., intentional trapping of unwanted robots), and sometimes for no apparent reason at all. What makes these pages similar is the seemingly infinite number of dynamically generated pages and/or hosts within a given domain. Crawling these massive Webs or performing DNS lookups on millions of hosts from a given domain not only places a significant burden on the crawler, but also wastes bandwidth on downloading largely useless content.

Simply restricting the branching factor or the maximum number of pages/hosts per domain is not a viable solution since there is a number of legitimate sites that contain over a hundred million pages and over a dozen million virtual hosts (i.e., various blog sites, hosting services, directories, and forums). For example, Yahoo currently reports indexing 1.2 billion objects *just within its own domain* and blogspot claims over 50 million users, each with a unique hostname. Therefore, differentiating between legitimate and illegitimate Web "monsters" becomes a fundamental task of any crawler.

Note that this task does not entail assigning popularity to each potential page as would be the case when returning query results to a user; instead, the crawler needs to decide *whether a given domain or host should be allowed to massively branch or not*. Indeed, spam sites and various auto-generated Webs with a handful of pages are not a problem as they can be downloaded with very little effort and later classified by data miners using PageRank or some other appropriate algorithm. The problem only occurs when the crawler assigns to domain $x$ download bandwidth that is disproportionate to the value of $x$'s content.

Another aspect of spam classification is that it must be performed with very little CPU/RAM/disk effort and run in real time at speed $SL$ links per second, where $L$ is the number of unique URLs per page.

## 6.2 Controlling Massive Sites

Before we introduce our algorithm, several definitions are in order. Both *host* and *site* refer to Fully Qualified Domain Names (FQDNs) on which valid pages reside (e.g., `motors.ebay.com`). A *server* is a physical host that accepts TCP connections and communicates content to the crawler. Note that multiple hosts may be colocated on the same server. A *Top-Level Domain* (TLD) or a *country-code TLD* (cc-TLD) is a domain one level below the root in the DNS tree (e.g., `.com`, `.net`, `.uk`). A *Pay-Level Domain* (PLD) is any domain that requires payment at a TLD or cc-TLD registrar. PLDs are usually one level below the corresponding TLD (e.g., `amazon.com`), with certain exceptions for cc-TLDs (e.g., `ebay.co.uk`, `det.wa.edu.au`). We use a comprehensive list of custom rules for identifying PLDs, which have been compiled as part of our ongoing DNS project.

While PageRank [Arasu et al. 2001; Brin and Page 1998; Kamvar et al. 2003b], BlockRank [Kamvar et al. 2003a], SiteRank [Feng et al. 2006; Wu and Aberer 2004], and OPIC [Abiteboul et al. 2003] are potential solutions to the spam problem, and these methods become extremely disk intensive in large-scale applications (e.g., 41 billion pages and 641 million hosts found in our crawl) and arguably with enough effort can be manipulated [Gyöngyi and Garcia-Molina 2005] by huge link farms (i.e., millions of pages and sites pointing to a target spam page). In fact, strict page-level rank is not absolutely necessary for controlling massively branching spam. Instead, we found that spam could be "deterred" by budgeting the number of allowed pages per PLD based on domain reputation, which we determine by domain in-degree from resources that spammers must pay for. There are two options for these resources: PLDs and IP addresses. We chose the former since classification based on IPs (first suggested in Lycos [Mauldin 1997]) has proven less effective since large subnets inside link farms could be given unnecessarily high priority and multiple independent sites cohosted on the same IP were improperly discounted.

While it is possible to classify each site and even each subdirectory based on their PLD in-degree, our current implementation uses a coarse-granular approach of only limiting spam at the PLD level. Each PLD $x$ starts with a default budget $B_0$, which is dynamically adjusted using some function $F(d_x)$ as $x$'s in-degree $d_x$ changes. Budget $B_x$ represents the number of pages that are

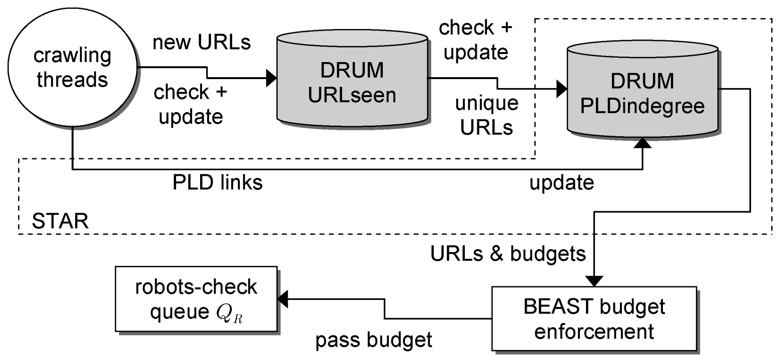


Fig. 4. Operation of STAR.

allowed to pass from $x$ (including all hosts and subdomains in $x$) to crawling threads every $T$ time units.

Figure 4 shows how our system, which we call *Spam Tracking and Avoidance through Reputation* (STAR), is organized. In the figure, crawling threads aggregate PLD-PLD link information and send it to a DRUM structure `PLDindegree`, which uses a batch `update` to store for each PLD $x$ its hash $h_x$, in-degree $d_x$, current budget $B_x$, and hashes of all in-degree neighbors in the PLD graph. Unique URLs arriving from `URLseen` perform a batch `check` against `PLDindegree`, and are given $B_x$ on their way to BEAST, which we discuss in the next section.

Note that by varying the budget function $F(d_x)$, it is possible to implement a number of policies: crawling of only popular pages (i.e., zero budget for low-ranked domains and maximum budget for high-ranked domains), equal distribution between all domains (i.e., budget $B_x = B_0$ for all $x$), and crawling with a bias toward popular/unpopular pages (i.e., budget directly/inversely proportional to the PLD in-degree).

## 7. POLITENESS AND BUDGETS

This section discusses how to enable polite crawler operation and scalably enforce budgets.

### 7.1 Rate Limiting

One of the main goals of IRLbot from the beginning was to adhere to strict rate-limiting policies in accessing poorly provisioned (in terms of bandwidth or server load) sites. Even though larger sites are much more difficult to crash, unleashing a crawler that can download at 500mb/s and allowing it unrestricted access to individual machines would generally be regarded as a denial-of-service attack.

Prior work has only enforced a certain per-host access delay $\tau_h$ (which varied from 10 times the download delay of a page [Najork and Heydon 2001] to 30 seconds [Shkapenyuk and Suel 2002]), but we discovered that this presented a major problem for hosting services that colocated thousands of virtual hosts on the same physical server and did not provision it to support simultaneous access to all sites (which in our experience is rather common in the current Internet). Thus, without an additional per-server limit $\tau_s$, such hosts could be easily crashed or overloaded.

We keep $\tau_h = 40$ seconds for accessing all low-ranked PLDs, but then for high-ranked PLDs scale it down proportional to $B_x$, up to some minimum value $\tau_h^0$. The reason for doing so is to prevent the crawler from becoming "bogged down" in a few massive sites with millions of pages in RAM. Without this rule, the crawler would make very slow progress through individual sites in addition to eventually running out of RAM as it becomes clogged with URLs from a few "monster" networks. For similar reasons, we keep per-server crawl delay $\tau_s$ at the default 1 second for low-ranked domains and scale it down with the average budget of PLDs hosted on the server, up to some minimum $\tau_s^0$.

Crawling threads organize URLs in two heaps: the IP heap, which enforces delay $\tau_s$, and the host heap, which enforces delay $\tau_h$. The URLs themselves are stored in a searchable tree with pointers to/from each of the heaps. By properly controlling the coupling between budgets and crawl delays, it is possible to ensure that the rate at which pages are admitted into RAM is no less than their crawl rate, which results in no memory backlog.

We should also note that threads that perform DNS lookups and download robots.txt in Figure 2 are limited by the IP heap, but not the host heap. The reason is that when the crawler is pulling robots.txt for a given site, no other thread can be simultaneously accessing that site.

## 7.2 Budget Checks

We now discuss how IRLbot's budget enforcement works in a method we call *Budget Enforcement with Anti-Spam Tactics* (BEAST). The goal of this method is not to discard URLs, but rather to delay their download until more is known about their legitimacy. Most sites have a low rank because they are not well linked to, but this does not necessarily mean that their content is useless or they belong to a spam farm. All other things equal, low-ranked domains should be crawled in some approximately round-robin fashion with careful control of their branching. In addition, as the crawl progresses, domains change their reputation and URLs that have earlier failed the budget check need to be re-budgeted and possibly crawled at a different rate. Ideally, the crawler should shuffle URLs without losing any of them and eventually download the entire Web if given infinite time.

A naive implementation of budget enforcement in prior versions of IRLbot maintained two queues $Q$ and $Q_F$, where $Q$ contained URLs that had passed the budget and $Q_F$ those that had failed. After $Q$ was emptied, $Q_F$ was read in its entirety and again split into two queues – $Q$ and $Q_F$. This process was then repeated indefinitely.

We next offer a simple overhead model for this algorithm. As before, assume that $S$ is the number of pages crawled per second and $b$ is the average URL size. Further define $E[B_x] < \infty$ to be the expected budget of a domain in the Internet, $V$ to be the total number of PLDs seen by the crawler in one pass through $Q_F$, and $L$ to be the number of *unique* URLs per page (recall that $l$

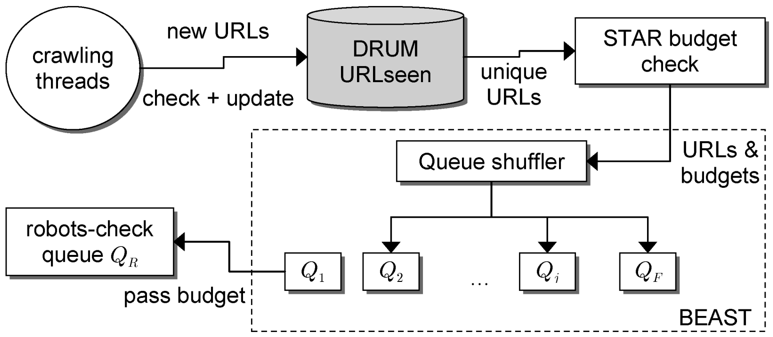


Fig. 5. Operation of BEAST.

in our earlier notation allowed duplicate links). The next result shows that the naive version of BEAST must increase disk I/O performance with crawl size $N$.

THEOREM 6. *Lowest disk I/O speed (in bytes/s) that allows the naive budget-enforcement approach to download $N$ pages at fixed rate $S$ is*

$$\lambda = 2Sb(L-1)\alpha_N, \tag{27}$$

*where*

$$\alpha_N = \max\Big(1, \frac{N}{E[B_x]V}\Big). \tag{28}$$

PROOF. Assume that $N \geq E[B_x]V$. First notice that the average number of links allowed into $Q$ is $E[B_x]V$ and define interval $T$ to be the time needed to crawl these links, that is, $T = E[B_x]V/S$. Note that $T$ is a constant, which is important for the analysis that follows. Next, by the $i$th iteration through $Q_F$, the crawler has produced $TiSL$ links and $TiS$ of them have been consumed through $Q$. Thus, the size of $Q_F$ is $TiS(L-1)$. Since $Q_F$ must be both read and written in $T$ time units for any $i$, the disk speed $\lambda$ must be $2TiS(L-1)/T = 2iS(L-1)$ URLs/s. Multiplying this by URL size $b$, we get $2ibS(L-1)$ bytes/s. The final step is to realize that $N = TSi$ (i.e., the total number of crawled pages) and substitute $i = N/(TS)$ into $2ibS(L-1)$.

For $N < E[B_x]V$ observe that queue size $E[B_x]V$ must be no larger than $N$ and thus $N = E[B_x]V$ must hold since we cannot extract from the queue more elements than have been placed there. Combining the two cases, we get (28). □

This theorem shows that $\lambda \sim \alpha_N = \Theta(N)$ and that rechecking failed URLs will eventually overwhelm *any* crawler regardless of its disk performance. For IRLbot (i.e., $V = 33$M, $E[B_x] = 11$, $L = 6.5$, $S = 3,100$ pages/s, and $b = 110$), we get $\lambda = 3.8$MB/s for $N = 100$ million, $\lambda = 83$MB/s for $N = 8$ billion, and $\lambda = 826$MB/s for $N = 80$ billion. Given other disk-intensive tasks, IRLbot's bandwidth for BEAST was capped at about 100MB/s, which explains why this design eventually became a bottleneck in actual crawls.

The correct implementation of BEAST rechecks $Q_F$ at exponentially increasing intervals. As shown in Figure 5, suppose the crawler starts with $j \geq 1$

queues $Q_1, \ldots, Q_j$, where $Q_1$ is the current queue and $Q_j$ is the last queue. URLs are read from the current queue $Q_1$ and written into queues $Q_2, \ldots, Q_j$ based on their budgets. Specifically, for a given domain $x$ with budget $B_x$, the first $B_x$ URLs are sent into $Q_2$, the next $B_x$ into $Q_3$, and so on. BEAST can always figure out where to place URLs using a combination of $B_x$ (attached by STAR to each URL) and a local array that keeps for each queue $Q_j$ the leftover budget of each domain. URLs that do not fit in $Q_j$ are all placed in $Q_F$ as in the previous design.

After $Q_1$ is emptied, the crawler moves to reading the next queue $Q_2$ and spreads newly arriving pages between $Q_3, \ldots, Q_j, Q_1$ (note the wrap-around). After it finally empties $Q_j$, the crawler rescans $Q_F$ and splits it into $j$ additional queues $Q_{j+1}, \ldots, Q_{2j}$. URLs that do not have enough budget for $Q_{2j}$ are placed into the new version of $Q_F$. The process then repeats starting from $Q_1$ until $j$ reaches some maximum OS-imposed limit or the crawl terminates.

There are two benefits to this approach. First, URLs from sites that exceed their budget by a factor of $j$ or more are pushed further back as $j$ increases. This leads to a higher probability that good URLs with enough budget will be queued and crawled ahead of URLs in $Q_F$. The second benefit, shown in the next theorem, is that the speed at which the disk must be read does not skyrocket to infinity.

THEOREM 7. *Lowest disk I/O speed (in bytes/s) that allows BEAST to download N pages at fixed rate S is*

$$\lambda = 2Sb\left[\frac{2\alpha_N}{1+\alpha_N}(L-1)+1\right] \le 2Sb\,(2L-1). \tag{29}$$

PROOF. Assume that $N \ge E[B_x]V$ and suppose one iteration involves reaching $Q_F$ and doubling $j$. Now assume the crawler is at the end of the $i$th iteration ($i = 1$ is the first iteration), which means that it has emptied $2^{i+1} - 1$ queues $Q_i$ and $j$ is currently equal to $2^i$. The total time taken to reach this stage is $T = E[B_x]V(2^{i+1} - 1)/S$. The number of URLs in $Q_F$ is then $TS(L - 1)$, which must be read/written together with $j$ smaller queues $Q_1, \ldots, Q_j$ in the time it takes to crawl these $j$ queues. Thus, we get that the speed must be at least

$$\lambda = 2\frac{TS(L-1) + jE[B_x]V}{jT_0} \text{ URL/s}, \tag{30}$$

where $T_0 = E[B_x]V/S$ is the time to crawl one queue $Q_i$. Expanding, we have

$$\lambda = 2S[(2 - 2^{-i})(L-1) + 1] \text{ URL/s}. \tag{31}$$

To tie this to $N$, notice that the total number of URLs consumed by the crawler is $N = E[B_x]V(2^{i+1} - 1) = TS$. Thus,

$$2^{-i} = \frac{2E[B_x]V}{N + E[B_x]V} \tag{32}$$

and we directly arrive at (29) after multiplying (31) by URL size $b$. Finally, for $N < E[B_x]V$, we use the same reasoning as in the proof of the previous theorem to obtain $N = E[B_x]V$, which leads to (28). □

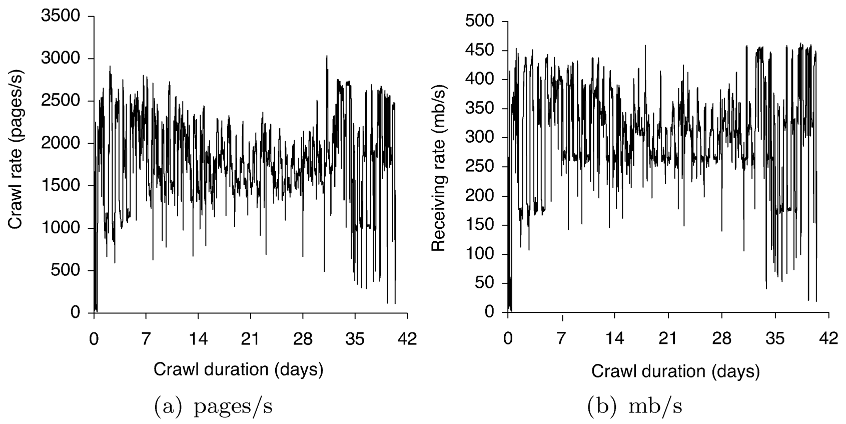


Fig. 6. Download rates during the experiment.

For $N \to \infty$ and fixed $V$, disk speed $\lambda \to 2Sb\,(2L-1)$, which is roughly four times the speed needed to write all unique URLs to disk as they are discovered during the crawl. For the examples used earlier in this section, this implementation needs $\lambda \le 8.2$MB/s *regardless of crawl size N*. From the preceding proof, it also follows that the last stage of an $N$-page crawl will contain

$$j = 2^{\lceil \log_2(\alpha_N+1)\rceil - 1} \tag{33}$$

queues. This value for $N = 8$B is 16 and for $N = 80$B only 128, neither of which is too imposing for a modern server.

## 8. EXPERIMENTS

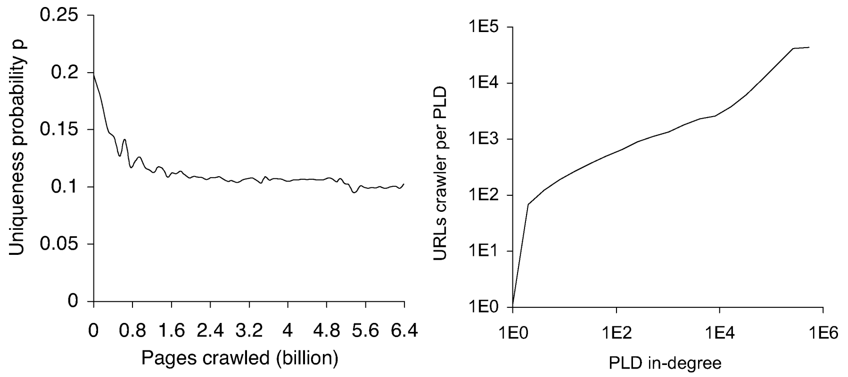
This section briefly examines the important parameters of the crawl and highlights our observations.

### 8.1 Summary

Between June 9 and August 3, 2007, we ran IRLbot on a quad-CPU AMD Opteron 2.6 GHz server (16GB RAM, 24-disk RAID-5) attached to a 1gb/s link at the campus of Texas A&M University. The crawler was paused several times for maintenance and upgrades, which resulted in the total active crawling span of 41.27 days. During this time, IRLbot attempted 7,606,109,371 connections and received 7,437,281,300 valid HTTP replies. Excluding non-HTML content (92M pages), HTTP errors and redirects (964M), IRLbot ended up with $N =$ 6,380,051,942 responses with status code 200 and content-type `text/html`.

We next plot average 10-minute download rates for the active duration of the crawl in Figure 6, in which fluctuations correspond to day/night bandwidth limits imposed by the university.[4] The average download rate during this crawl was 319mb/s (1,789 pages/s) with the peak 10-minute average rate of 470mb/s (3,134 pages/s). The crawler received 143TB of data, out of which 254GB were

[4]The day limit was 250mb/s for days 5 through 32 and 200mb/s for the rest of the crawl. The night limit was 500mb/s.



(a) fraction of unique pages (b) effectiveness of STAR

Fig. 7. Evolution of $p$ throughout the crawl and effectiveness of budget control in limiting low-ranked PLDs.

robots.txt files, and transmitted 1.8TB of HTTP requests. The parser processed 161TB of HTML code (i.e., 25.2KB per uncompressed page) and the gzip library handled 6.6TB of HTML data containing 1,050,955,245 pages, or 16% of the total. The average compression ratio was 1:5, which resulted in the peak parsing demand being close to 800mb/s (i.e., 1.64 times faster than the maximum download rate).

IRLbot parsed out 394,619,023,142 links from downloaded pages. After discarding invalid URLs and known non-HTML extensions, the crawler was left with $K = 374{,}707{,}295{,}503$ potentially "crawlable" links that went through URL uniqueness checks. We use this number to obtain $l = K/N \approx 59$ links/page used throughout the article. The average URL size was 70.6 bytes (after removing "http://"), but with crawler overhead (e.g., depth in the crawl tree, IP address and port, timestamp, and parent link) attached to each URL, their average size in the queue was $b \approx 110$ bytes. The size of `URLseen` on disk was 332GB and it contained $M = 41{,}502{,}195{,}631$ unique pages hosted by $641, 982, 061$ different sites. This yields $L = M/N \approx 6.5$ unique URLs per crawled page.

As promised earlier, we now show in Figure 7(a) that the probability of uniqueness $p$ stabilizes around 0.11 once the first billion pages have been downloaded. The fact that $p$ is bounded away from 0 even at $N = 6.3$B suggests that our crawl has discovered only a small fraction of the Web. While there are certainly at least 41 billion pages in the Internet, the fraction of them with useful content and the number of additional pages not seen by the crawler remain a mystery at this stage.

## 8.2 Domain Reputation

The crawler received responses from 117,576,295 sites, which belonged to 33,755,361 Pay-Level Domains (PLDs) and were hosted on 4,260,532 unique IPs. The total number of nodes in the PLD graph was 89,652,630 with the number of PLD-PLD edges equal to $1, 832, 325, 052$. During the crawl, IRLbot performed 260,113,628 DNS lookups, which resolved to 5,517,743 unique IPs.

Table IX. Top-Ranked PLDs, Their PLD In-Degree, Google PageRank, and Total Pages Crawled

| Rank | Domain | In-degree | PageRank | Pages |
|---|---|---|---|---|
| 1 | microsoft.com | 2,948,085 | 9 | 37,755 |
| 2 | google.com | 2,224,297 | 10 | 18,878 |
| 3 | yahoo.com | 1,998,266 | 9 | 70,143 |
| 4 | adobe.com | 1,287,798 | 10 | 13,160 |
| 5 | blogspot.com | 1,195,991 | 9 | 347,613 |
| 7 | wikipedia.org | 1,032,881 | 8 | 76,322 |
| 6 | w3.org | 933,720 | 10 | 9,817 |
| 8 | geocities.com | 932,987 | 8 | 26,673 |
| 9 | msn.com | 804,494 | 8 | 10,802 |
| 10 | amazon.com | 745,763 | 9 | 13,157 |

Without knowing how our algorithms would perform, we chose a conservative budget function $F(d_x)$ where the crawler would give only moderate preference to highly ranked domains and try to branch out to discover a wide variety of low-ranked PLDs. Specifically, top-10K ranked domains were given budget $B_x$ linearly interpolated between 10 and 10K pages. All other PLDs received the default budget $B_0 = 10$. Figure 7(b) shows the average number of downloaded pages per PLD $x$ based on its in-degree $d_x$. IRLbot crawled on average 1.2 pages per PLD with $d_x = 1$ incoming link, 68 pages per PLD with $d_x = 2$, and 43K pages per domain with $d_x \geq 512$K. The largest number of pages pulled from any PLD was 347,613 (blogspot.com), while 90% of visited domains contributed to the crawl fewer than 586 pages each and 99% fewer than $3,044$ each. As seen in the figure, IRLbot succeeded at achieving a strong correlation between domain popularity (i.e., in-degree) and the amount of bandwidth allocated to that domain during the crawl.

Our manual analysis of top-1000 domains shows that most of them are highly ranked legitimate sites, which attests to the effectiveness of our ranking algorithm. Several of them are listed in Table IX together with Google Toolbar PageRank of the main page of each PLD and the number of pages downloaded by IRLbot. The exact coverage of each site depended on its link structure, as well as the number of hosts and physical servers (which determined how polite the crawler needed to be). By changing the budget function $F(d_x)$, much more aggressive crawls of large sites could be achieved, which may be required in practical search-engine applications.

We believe that PLD-level domain ranking by itself is not sufficient for preventing *all* types of spam from infiltrating the crawl and that additional fine-granular algorithms may be needed for classifying individual hosts within a domain and possibly their subdirectory structure. Future work will address this issue, but our first experiment with spam-control algorithms demonstrates that these methods are not only necessary, but also very effective in helping crawlers scale to billions of pages.

## 9. CAVEATS

This section provides additional details about the current status of IRLbot and its relationship to other algorithms.

### 9.1 Potential for Distributed Operation

Partitioning of URLs between crawling servers is a well-studied problem with many insightful techniques [Boldi et al. 2004a; Cho and Garcia-Molina 2002; Heydon and Najork 1999; Najork and Heydon 2001; Shkapenyuk and Suel 2002]. The standard approach is to localize URLs to each server based on the hash of the Web site or domain the URL belongs to. Due to the use of PLD budgets, our localization must be performed on PLDs rather than hosts, but the remainder of the approach for distributing IRLbot is very similar to prior work, which is the reason we did not dwell on it in this article. Our current estimates suggest that DRUM with RAM caching can achieve over 97% local hit-rate (i.e., the sought hash is either in RAM or on local disk) in verification of URL uniqueness. This means that only 3% of parsed URL hashes would normally require transmission to other hosts. For actual crawling, over 80% of parsed URLs stay on the same server, which is consistent with the results of prior studies [Najork and Heydon 2001].

Applying a similar technique to domain reputation (i.e., each crawling node is responsible for the in-degree of all PLDs that map to it), it is possible to distribute STAR between multiple hosts with a reasonably small effort. Finally, it is fairly straightforward to localize BEAST by keeping separate copies of queues of backlogged URLs at each crawling server, which only require budgets produced by the local version of STAR. The amount of traffic needed to send PLD-PLD edges between servers is also negligible due to the localization and the small footprint of each PLD-PLD edge (i.e., 16 bytes). Note, however, that more sophisticated domain reputation algorithms (e.g., PageRank [Brin and Page 1998]) may require a global view of the entire PLD graph and may be more difficult to distribute.

### 9.2 Duplicate Pages

It is no secret that the Web contains a potentially large number of syntactically similar pages, which arises due to the use of site mirrors and dynamic URLs that retrieve the same content under different names. A common technique, originally employed in Heydon and Najork [1999] and Najork and Heydon [2001], is to hash page contents and verify uniqueness of each page-hash before processing its links. Potential pitfalls of this method are false positives (i.e., identical pages on different sites with a single link to `/index.html` may be flagged as duplicates even though they point to different targets) and inability to detect volatile elements of the page that do not affect its content (e.g., visitor counters, time of day, weather reports, different HTML/javascript code rendering the same material).

As an alternative to hashing entire page contents, we can rely on significantly more complex analysis that partitions pages into elements and studies their similarity [Bharat and Broder 1999; Broder et al. 1997; Charikar 2002; Henzinger 2006; Manku et al. 2007], but these approaches often require significant processing capabilities that must be applied in real time and create nontrivial storage overhead (i.e., each page hash is no longer 8 bytes, but may consist of a large array of hashes). It has been reported in Henzinger [2006] that 25% to 30% of downloaded pages by Google are identical and an additional 1.7% to 2.2% are near-duplicate. The general consensus between the various datasets [Broder et al. 1997; Manasse and Najork 2003; Henzinger 2006] shows that 20% to 40% of downloaded pages can be potentially eliminated from processing; however, the false positive and false negative rates of these algorithms remain an open question.

Due to the inherent inability of HTML to flag objects as mirrors of other objects, it is impossible for a crawler to automatically detect duplicate content with 100% certainty. Even though the current version of IRLbot does not perform any prevention of duplicates due to its high overhead and general uncertainty about the exact trade-off between false positives and bandwidth savings, future work will examine this issue more closely and some form of duplicate reduction will be incorporated into the crawler.

In general, our view is that the ideal path to combat this problem is for the Internet to eventually migrate to DNS-based mirror redirection (e.g., as done in CDNs such as Akamai) and for Web sites to consolidate all duplicate pages, as well as multiple hostnames, using 301/302 HTTP redirects. One incentive to perform this transition would be the bandwidth savings for Web masters and their network providers. Another incentive would be for commercial search engines to publicly offer benefits (e.g., higher ranking) to sites where each page maps to a single unique URL.

### 9.3 Hash Collisions

Given the size of IRLbot crawls, we may wonder about the probability of collision on 8-byte hashes and the possibility of missing important pages during the crawl. We offer approximate analysis of this problem to show that for $N$ smaller than a few hundred billion pages the collision rate is negligible to justify larger hashes, but high enough that a few pages may indeed be missed in some crawls. Assume that $N$ is the number of crawled URLs and $M = 2^{64}$ is the hash size. A *collision event* happens when two or more pages produce the same hash. Define $V_i$ to be the number of pages with hash $i$ for $i = 0, 1, \dots, M-1$. Assuming $N \ll M$, each $V_i$ is approximately Poisson with rate $\lambda = N/M$ and the probability of a collision on hash $i$ is

$$P(V_i \geq 2) = 1 - e^{-\lambda} - e^{-\lambda}\lambda = \frac{\lambda^2}{2} + O(\lambda^3), \tag{34}$$

where the last step uses Taylor expansion for $\lambda \to 0$. Now define $V = \sum_{i=1}^{M} \mathbf{1}_{V_i \geq 2}$ to be the number of collision events, where $\mathbf{1}_A$ is an indicator variable of event $A$. Neglecting small terms, it then follows that the average number of collisions and therefore missed URLs per crawl is $E[V] \approx N^2/2M$. Using $N = 6.3$ billion, we get $E[V] = 1.075$ pages/crawl. Ignoring the mild dependency in the set $\{V_i\}_{i=1}^{N}$, notice that $V$ also tends to a Poisson random variable with rate $M\lambda^2/2$ as $N/M \to 0$, which means that the deviation of $V$ from the mean is very small. For $N = 6.3$ billion, $P(V = 0) \approx e^{-N^2/2M} = 0.34$, and $P(V \leq 4) \approx 0.995$, indicating that 34% of crawls do not have any collisions and almost every crawl has fewer than four missed URLs.

Analysis of the frontier (i.e., pages in the queue that remain to be crawled) shows similar results. For $N = 35$ billion, the average number of missed URLs in the queue is $E[V] = 33$ and $P(V \leq 50) = 0.997$, but the importance of this omission may not be very high since the crawler does not attempt to download any of these pages anyway. The final question is how large a crawl needs to be for 8-byte hashes to be insufficient. Since $E[V]$ grows as a quadratic function of $N$, the collision rate will quickly become noticeable and eventually unacceptable as $N$ increases. This transition likely occurs for $N$ between 100 billion and 10 trillion, where $E[V]$ jumps from 271 to 2.7 million pages. When IRLbot reaches this scale, we will consider increasing its hash size.

## 9.4 Optimality of Disk Sort

It may appear that DRUM's quadratic complexity as $N \to \infty$ is not very exciting since algorithms such as merge-sort can perform the same task in $\Theta(N \log N)$ disk I/Os. We elaborate on this seeming discrepancy next. In general, the performance of sorting algorithms depends on the data being sorted and the range of $N$ under consideration. For example, quick-sort is 2 to 3 times faster than merge-sort on randomly ordered input and selection-sort is faster than quick-sort for very small $N$ despite its $\Theta(N^2)$ complexity. Furthermore, if the array contains uniformly random integers, bucket-sort can achieve $\Theta(N)$ complexity if the number of buckets is scaled with $N$. For RAM-based sorting, this is not a problem since $N \to \infty$ implies that RAM size $R \to \infty$ and the number of buckets can be unbounded.

Returning to our problem with checking URL uniqueness, observe that merge-sort can be implemented using $\Theta(N \log N)$ disk overhead for any $N \to \infty$ [Vitter 2001]. While bucket-sort in RAM-only applications is linear, our results in Theorem 2 show that with *fixed* RAM size $R$, the number of buckets $k$ cannot be unbounded as $N$ increases. This arises due to the necessity to maintain $2k$ open file handles and the associated memory buffers of size $2kM$. As a result, DRUM exhibits $\Theta(N^2)$ complexity for very large $N$; however, Theorem 2 also shows that the quadratic term is almost negligible for $N$ smaller than several trillion pages. Therefore, unless $N$ is so exorbitant that the quadratic term dominates the overhead, DRUM is in fact almost linear and very close to optimal.

As an additional improvement, bucket-sort can be recursively applied to each of the buckets until they become smaller than $R$. This modification achieves $N \log_k(N/R)$ overhead, which is slightly smaller (by a linear term) than that of $k$-way merge-sort. While both algorithms require almost the same number of disk I/O bytes, asymmetry in RAID read/write speed introduces further differences. Each phase of multilevel bucket-sort requires simultaneous writing into $k$ files and later reading from one file. The situation is reversed in $k$-way merge-sort whose phases need concurrent reading from $k$ files and writing into one file. Since reading is usually faster than writing, memory buffer $M$ currently used in DRUM may not support efficient reading from $k$ parallel files. As a result, merge-sort may need to increase its $M$ and reduce $k$ to match the I/O speed of bucket-sort. On the bright side, merge-sort is more general and does not rely on uniformity of keys being sorted. Exact analysis of these trade-offs is beyond the scope of the article and may be considered in future work.

### 9.5 Robot Expiration

IRLbot's user-agent string supplied Web masters with a Web page describing the project and ways to be excluded from the crawl (both manual through email and automatic through robots.txt files). While the robots draft RFC specifies the default expiration delay of 7 days, we used an adaptive approach based on the changes detected in the file. The initial delay was set to one day, which was then doubled at each expiration (until it reached 7 days) if the file did not change and reset to one day otherwise. We also honored the `crawl-delay` parameter and wildcards in robots.txt, even though they are not part of the draft RFC.

Interestingly, Web masters had conflicting expectations about how often the file should be loaded; some wanted us to load it for every visit, while others complained that only one time per crawl was sufficient since doing otherwise wasted their bandwidth. In the end, a compromise between the two extremes seemed like an appropriate solution.

### 9.6 Caching

The article has shown that URL checks with RAM-caching can speed up the crawl if the system is bottlenecked on disk I/O and has spare CPU capacity. In our case, the disk portion of DRUM was efficient enough for download rates well above our peak 3, 000 pages/s, which allowed IRLbot to run without caching and use the spare CPU resources for other purposes. In faster crawls, however, it is very likely that caching will be required for all major DRUM structures.

## 10. CONCLUSION

This article tackled the issue of scaling Web crawlers to billions and even trillions of pages using a single server with constant CPU, disk, and memory speed. We identified several impediments to building an efficient large-scale crawler and showed that they could be overcome by simply changing the BFS crawling order and designing low-overhead disk-based data structures. We experimentally tested our techniques in the Internet and found them to scale much better than the methods proposed in prior literature.

Future work involves refining reputation algorithms, assessing their performance, and mining the collected data.

ACKNOWLEDGMENTS

We are grateful to Texas A&M University and its network administrators for providing the enormous amount of bandwidth needed for this project and patiently handling Web master complaints. We are also thankful to the anonymous ACM Transactions on the Web reviewers for their insightful comments and helpful feedback.

## REFERENCES

Abiteboul, S., Preda, M., and Cobena, G. 2003. Adaptive on-line page importance computation. In *Proceedings of the World Wide Web Conference (WWW'03)*. 280–290.

Arasu, A., Cho, J., Garcia-Molina, H., Paepcke, A., and Raghavan, S. 2001. Searching the Web. *ACM Trans. Internet Technol.1*, 1, 2–43.

Bharat, K. and Broder, A. 1999. Mirror, mirror on the Web: A study of hst pairs with replicated content. In *Proceedings of the World Wide Web Conference (WWW'99)*. 1579–1590.

Boldi, P., Codenotti, B., Santini, M., and Vigna, S. 2004a. Ubicrawler: A scalable fully distributed Web crawler. *Softw. Pract. Exper. 34*, 8, 711–726.

Boldi, P., Santini, M., and Vigna, S. 2004b. Do your worst to make the best: Paradoxical effects in pagerank incremental computations. In *Algorithms and Models for the Web-Graph*. Lecture Notes in Computer Science, vol. 3243. Springer,168–180.

Brin, S. and Page, L. 1998. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the World Wide Web Conference (WWW'98)*. 107–117.

Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. 1997. Syntactic clustering of the Web. *Comput. Netw. ISDN Syst. 29*, 8-13, 1157–1166.

Broder, A. Z., Najork, M., and Wiener, J. L. 2003. Efficient url caching for World Wide Web crawling. In *Proceedings of the World Wide Web Conference (WWW'03)*. 679–689.

Burner, M. 1997. Crawling towards eternity: Building an archive of the World Wide Web. *Web Techn. Mag. 2,* 5.

Charikar, M. S. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC'02)*. 380–388.

Cho, J. and Garcia-Molina, H. 2002. Parallel crawlers. In *Proceedings of the World Wide Web Conference (WWW'02)*. 124–135.

Cho, J., Garcia-Molina, H., Haveliwala, T., Lam, W., Paepcke, A., and Wesley, S. R. G. 2006. Stanford Web base components and applications. *ACM Trans. Internet Technol. 6*, 2, 153–186.

Edwards, J., McCurley, K., and Tomlin, J. 2001. An adaptive model for optimizing performance of an incremental Web crawler. In *Proceedings of the World Wide Web Conference (WWW'01)*. 106–113.

Eichmann, D. 1994. The rbse spider – Balancing effective search against Web load. In *World Wide Web Conference.*

Feng, G., Liu, T.-Y., Wang, Y., Bao, Y., Ma, Z., Zhang, X.-D., and Ma, W.-Y. 2006. Aggregaterank: Bringing order to Web sites. In *Proceedings of the Annual ACM SIGIR Conference on Research and Development in Information Retrieval*. 75–82.

Gleich, D. and Zhukov, L. 2005. Scalable computing for power law graphs: Experience with parallel pagerank. In *Proceedings of SuperComputing*.

Gyöngyi, Z. and Garcia-Molina, H. 2005. Link spam alliances. In *Proceedings of the International Conference on Very Large Databases (VLDB'05)*. 517–528.

Hafri, Y. and Djeraba, C. 2004. High-performance crawling system. In *Proceedings of the ACM International Conference on Multimedia Information Retrieval (MIR'04)*. 299–306.

Henzinger, M. R. 2006. Finding near-duplicate Web pages: A large-scale evaluation of algorithms. In *Proceedings of the Annual ACM SIGIR Conference on Research and Development in Information Retrieval*. 284–291.

Heydon, A. and Najork, M. 1999. Mercator: A scalable, extensible Web crawler. *World Wide Web 2*, 4, 219–229.

Hirai, J., Raghavan, S., Garcia-Molina, H., and Paepcke, A. 2000. Web base: A repository of Web pages. In *Proceedings of the World Wide Web Conference (WWW'00)*. 277–293.

Internet Archive. Internet archive homepage. http://www.archive.org/.

IRLbot. 2007. IRLbot project at Texas A&M. http://irl.cs.tamu.edu/crawler/.

Kamvar, S. D., Haveliwala, T. H., Manning, C. D., and Golub, G. H. 2003a. Exploiting the block structure of the Web for computing pagerank. Tech. rep., Stanford University.

Kamvar, S. D., Haveliwala, T. H., Manning, C. D., and Golub, G. H. 2003b. Extrapolation methods for accelerating pagerank computations. In *Proceedings of the World Wide Web Conference (WWW'03)*. 261–270.

Koht-arsa, K. and Sanguanpong, S. 2002. High-performance large scale Web spider architecture. In *International Symposium on Communications and Information Technology*.

Manasse, D. F. M. and Najork, M. 2003. Evolution of clusters of near-duplicate Web pages. In *Proceedings of the Latin American Web Congress (LAWEB'03)*. 37–45.

Manku, G. S., Jain, A., and Sarma, A. D. 2007. Detecting near duplicates for Web crawling. In *Proceedings of the World Wide Web Conference (WWW'07)*. 141–149.

Mauldin, M. 1997. Lycos: Design choices in an Internet search service. *IEEE Expert Mag. 12*, 1, 8–11.

McBryan, O. A. 1994. Genvl and wwww: Tools for taming the Web. In *World Wide Web Conference (WWW'94)*.

Najork, M. and Heydon, A. 2001. High-performance Web crawling. Tech: rep. 173, Compaq Systems Research Center.

Najork, M. and Wiener, J. L. 2001. Breadth-first search crawling yields high-quality pages. In *Proceedings of the World Wide Web Conference (WWW'01)*. 114–118.

Official Google Blog. 2008. We knew the Web was big...
http://googleblog.blogspot.com/ 2008/07/we- knew- web- was- big.html.

Pinkerton, B. 1994. Finding what people want: Experiences with the Web crawler. In *World Wide Web Conference (WWW'94)*.

Pinkerton, B. 2000. Web crawler: Finding what people want. Ph.D. thesis, University of Washington.

Shkapenyuk, V. and Suel, T. 2002. Design and implementation of a high-performance distributed Web crawler. *In Proceedings of the IEEE International Conference on Data Engineering (ICDE'02)*. 357–368.

Singh, A., Srivatsa, M., Liu, L., and Miller, T. 2003. Apoidea: A decentralized peer-to-peer architecture for crawling the World Wide Web. In *Proceedings of the ACM SIGIR Workshop on Distributed Information Retrieval*. 126–142.

Suel, T., Mathur, C., Wu, J., Zhang, J., Delis, A., Kharrazi, M., Long, X., and Shanmugasundaram, K. 2003. Odissea: A peer-to-peer architecture for scalable Web search and information retrieval. In *Proceedings of the International Workshop on Web and Databases (WebDB'03)*. 67–72.

Vitter, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Comput. Surv. 33*, 2, 209–271.

Wu, J. and Aberer, K. 2004. Using siterank for decentralized computation of Web document ranking. In *Proceedings of the International Conference on Adaptive Hypermedia*, 265–274.

Received March 2008; revised February 2009; accepted March 2009