Foundations and Trends® in
Information Retrieval
Vol. 4, No. 3 (2010) 175–246

DOI: 10.1561/1500000017

now
the essence of knowledge

Web Crawling

By Christopher Olston and Marc Najork

Contents

Foundations and Trends® in
Information Retrieval
Vol. 4, No. 3 (2010) 175–246

DOI: 10.1561/1500000017

now
the essence of knowledge

Web Crawling

Christopher Olston[1] and Marc Najork[2]

[1] *Yahoo! Research, 701 First Avenue, Sunnyvale, CA, 94089, USA*
olston@yahoo-inc.com
[2] *Microsoft Research, 1065 La Avenida, Mountain View, CA, 94043, USA*
najork@microsoft.com

Abstract

This is a survey of the science and practice of web crawling. While at first glance web crawling may appear to be merely an application of breadth-first-search, the truth is that there are many challenges ranging from systems concerns such as managing very large data structures to theoretical questions such as how often to revisit evolving content sources. This survey outlines the fundamental challenges and describes the state-of-the-art models and solutions. It also highlights avenues for future work.

1

Introduction

A *web crawler* (also known as a *robot* or a *spider*) is a system for the bulk downloading of web pages. Web crawlers are used for a variety of purposes. Most prominently, they are one of the main components of web search engines, systems that assemble a corpus of web pages, index them, and allow users to issue queries against the index and find the web pages that match the queries. A related use is web archiving (a service provided by e.g., the Internet archive [77]), where large sets of web pages are periodically collected and archived for posterity. A third use is web data mining, where web pages are analyzed for statistical properties, or where data analytics is performed on them (an example would be Attributor [7], a company that monitors the web for copyright and trademark infringements). Finally, web monitoring services allow their clients to submit standing queries, or *triggers*, and they continuously crawl the web and notify clients of pages that match those queries (an example would be GigaAlert [64]).

The raison d'être for web crawlers lies in the fact that the web is not a centrally managed repository of information, but rather consists

of hundreds of millions of independent web content providers, each one providing their own services, and many competing with one another. In other words, the web can be viewed as a federated information repository, held together by a set of agreed-upon protocols and data formats, such as the Transmission Control Protocol (TCP), the Domain Name Service (DNS), the Hypertext Transfer Protocol (HTTP), the Hypertext Markup Language (HTML) and the robots exclusion protocol. So, content aggregators (such as search engines or web data miners) have two choices: They can either adopt a pull model where they will proactively scour the web for new or updated information, or they could try to establish a convention and a set of protocols enabling content providers to push content of interest to the aggregators. Indeed, the Harvest system [24], one of the earliest search services, adopted such a push model. However, this approach did not succeed, and virtually all content aggregators adopted the pull approach, with a few provisos to allow content providers to exclude all or part of their content from being crawled (the robots exclusion protocol) and to provide hints about their content, its importance and its rate of change (the Sitemaps protocol [110]).

There are several reasons why the push model did not become the primary means of acquiring content for search engines and other content aggregators: The fact that web servers are highly autonomous means that the barrier of entry to becoming a content provider is quite low, and the fact that the web protocols were at least initially extremely simple lowered the barrier even further — in fact, this simplicity is viewed by many as the reason why the web succeeded where earlier hypertext systems had failed. Adding push protocols would have complicated the set of web protocols and thus raised the barrier of entry for content providers, while the pull model does not require any extra protocols. By the same token, the pull model lowers the barrier of entry for content aggregators as well: Launching a crawler does not require any a priori buy-in from content providers, and indeed there are over 1,500 operating crawlers [47], extending far beyond the systems employed by the big search engines. Finally, the push model requires a trust relationship between content provider and content aggregator, something that is not given on the web at large — indeed, the relationship between

content providers and search engines is characterized by both mutual dependence and adversarial dynamics (see Section 6).

1.1 Challenges

The basic web crawling algorithm is simple: Given a set of seed Uniform Resource Locators (URLs), a crawler downloads all the web pages addressed by the URLs, extracts the hyperlinks contained in the pages, and iteratively downloads the web pages addressed by these hyperlinks. Despite the apparent simplicity of this basic algorithm, web crawling has many inherent challenges:

- **Scale.** The web is very large and continually evolving. Crawlers that seek broad coverage and good freshness must achieve extremely high throughput, which poses many difficult engineering problems. Modern search engine companies employ thousands of computers and dozens of high-speed network links.
- **Content selection tradeoffs.** Even the highest-throughput crawlers do not purport to crawl the whole web, or keep up with all the changes. Instead, crawling is performed selectively and in a carefully controlled order. The goals are to acquire high-value content quickly, ensure eventual coverage of all reasonable content, and bypass low-quality, irrelevant, redundant, and malicious content. The crawler must balance competing objectives such as coverage and freshness, while obeying constraints such as per-site rate limitations. A balance must also be struck between exploration of potentially useful content, and exploitation of content already known to be useful.
- **Social obligations.** Crawlers should be "good citizens" of the web, i.e., not impose too much of a burden on the web sites they crawl. In fact, without the right safety mechanisms a high-throughput crawler can inadvertently carry out a denial-of-service attack.
- **Adversaries.** Some content providers seek to inject useless or misleading content into the corpus assembled by

the crawler. Such behavior is often motivated by financial incentives, for example (mis)directing traffic to commercial web sites.

1.2 Outline

Web crawling is a many-faceted topic, and as with most interesting topics it cannot be split into fully orthogonal subtopics. Bearing that in mind, we structure the survey according to five relatively distinct lines of work that occur in the literature:

- Building an efficient, robust and scalable crawler (Section 2).
- Selecting a traversal order of the web graph, assuming content is well-behaved and is interconnected via HTML hyperlinks (Section 4).
- Scheduling revisitation of previously crawled content (Section 5).
- Avoiding problematic and undesirable content (Section 6).
- Crawling so-called "deep web" content, which must be accessed via HTML forms rather than hyperlinks (Section 7).

Section 3 introduces the theoretical crawl ordering problem studied in Sections 4 and 5, and describes structural and evolutionary properties of the web that influence crawl ordering. Section 8 gives a list of open problems.

2

Crawler Architecture

This section first presents a chronology of web crawler development, and then describes the general architecture and key design points of modern scalable crawlers.

2.1 Chronology

Web crawlers are almost as old as the web itself. In the spring of 1993, shortly after the launch of NCSA Mosaic, Matthew Gray implemented the World Wide Web Wanderer [67]. The Wanderer was written in Perl and ran on a single machine. It was used until 1996 to collect statistics about the evolution of the web. Moreover, the pages crawled by the Wanderer were compiled into an index (the "Wandex"), thus giving rise to the first search engine on the web. In December 1993, three more crawler-based Internet Search engines became available: JumpStation (implemented by Jonathan Fletcher; the design has not been written up), the World Wide Web Worm [90], and the RBSE spider [57]. WebCrawler [108] joined the field in April 1994, and MOMspider [61] was described the same year. This first generation of crawlers identified some of the defining issues in web crawler design. For example, MOM-

spider considered *politeness* policies: It limited the rate of requests to each site, it allowed web sites to exclude themselves from purview through the nascent robots exclusion protocol [83], and it provided a "black-list" mechanism that allowed the crawl operator to exclude sites. WebCrawler supported parallel downloading of web pages by structuring the system into a central crawl manager and 15 separate downloading processes. However, the design of these early crawlers did not focus on scalability, and several of them (RBSE spider and WebCrawler) used general-purpose database management systems to store the state of the crawl. Even the original Lycos crawler [89] ran on a single machine, was written in Perl, and used Perl's associative arrays (spilt onto disk using the DBM database manager) to maintain the set of URLs to crawl.

The following few years saw the arrival of several commercial search engines (Lycos, Infoseek, Excite, AltaVista, and HotBot), all of which used crawlers to index tens of millions of pages; however, the design of these crawlers remains undocumented.

Mike Burner's description of the Internet Archive crawler [29] was the first paper that focused on the challenges caused by the scale of the web. The Internet Archive crawling system was designed to crawl on the order of 100 million URLs. At this scale, it is no longer possible to maintain all the required data in main memory. The solution proposed by the IA paper was to crawl on a site-by-site basis, and to partition the data structures accordingly. The list of URLs to be crawled was implemented as a disk-based queue per web site. To avoid adding multiple instances of the same URL to the queue, the IA crawler maintained an in-memory Bloom filter [20] of all the site's URLs discovered so far. The crawl progressed by dequeuing a URL, downloading the associated page, extracting all links, enqueuing freshly discovered on-site links, writing all off-site links to disk, and iterating. Each crawling process crawled 64 sites in parallel, using non-blocking input/output (I/O) and a single thread of control. Occasionally, a batch process would integrate the off-site link information into the various queues. The IA design made it very easy to throttle requests to a given host, thereby addressing politeness concerns, and DNS and robot exclusion lookups for a given web site were amortized over all the site's URLs crawled in a single round. However, it is not clear whether the batch

process of integrating off-site links into the per-site queues would scale to substantially larger web crawls.

Brin and Page's 1998 paper outlining the architecture of the first-generation Google [25] system contains a short description of their crawler. The original Google crawling system consisted of a single URLserver process that maintained the state of the crawl, and around four crawling processes that downloaded pages. Both URLserver and crawlers were implemented in Python. The crawling process used asynchronous I/O and would typically perform about 300 downloads in parallel. The peak download rate was about 100 pages per second, with an average size of 6 KB per page. Brin and Page identified social aspects of crawling (e.g., dealing with web masters' complaints) as a major challenge in operating a crawling system.

With the Mercator web crawler, Heydon and Najork presented a "blueprint design" for web crawlers [75, 94]. Mercator was written in Java, highly scalable, and easily extensible. The first version [75] was non-distributed; a later distributed version [94] partitioned the URL space over the crawlers according to host name, and avoided the potential bottleneck of a centralized URL server. The second Mercator paper gave statistics of a 17-day, four-machine crawl that covered 891 million pages. Mercator was used in a number of web mining projects [27, 60, 71, 72, 95], and in 2001 replaced the first-generation AltaVista crawler.

Shkapenyuk and Suel's Polybot web crawler [111] represents another "blueprint design." Polybot is a distributed system, consisting of a crawl manager process, multiple downloader processes, and a DNS resolver process. The paper describes scalable data structures for the URL frontier and the "seen-URL" set used to avoid crawling the same URL multiple times; it also discusses techniques for ensuring politeness without slowing down the crawl. Polybot was able to download 120 million pages over 18 days using four machines.

The IBM WebFountain crawler [56] represented another industrial-strength design. The WebFountain crawler was fully distributed. The three major components were multi-threaded crawling processes ("Ants"), duplicate detection processes responsible for identifying downloaded pages with near-duplicate content, and a central controller

process responsible for assigning work to the Ants and for monitoring the overall state of the system. WebFountain featured a very flexible crawl scheduling mechanism that allowed URLs to be prioritized, maintained a politeness policy, and even allowed the policy to be changed on the fly. It was designed from the ground up to support incremental crawling, i.e., the process of recrawling pages regularly based on their historical change rate. The WebFountain crawler was written in C++ and used MPI (the Message Passing Interface) to facilitate communication between the various processes. It was reportedly deployed on a cluster of 48 crawling machines [68].

UbiCrawler [21] is another scalable distributed web crawler. It uses consistent hashing to partition URLs according to their host component across crawling machines, leading to graceful performance degradation in the event of the failure of a crawling machine. UbiCrawler was able to download about 10 million pages per day using five crawling machines. UbiCrawler has been used for studies of properties of the African web [22] and to compile several reference collections of web pages [118].

Recently, Yan et al. described IRLbot [84], a single-process web crawler that is able to scale to extremely large web collections without performance degradation. IRLbot features a "seen-URL" data structure that uses only a fixed amount of main memory, and whose performance does not degrade as it grows. The paper describes a crawl that ran over two months and downloaded about 6.4 billion web pages. In addition, the authors address the issue of crawler traps (web sites with a large, possibly infinite number of low-utility pages, see Section 6.2), and propose ways to ameliorate the impact of such sites on the crawling process.

Finally, there are a number of open-source crawlers, two of which deserve special mention. Heritrix [78, 93] is the crawler used by the Internet Archive. It is written in Java and highly componentized, and its design is quite similar to that of Mercator. Heritrix is multithreaded, but not distributed, and as such suitable for conducting moderately sized crawls. The Nutch crawler [62, 81] is written in Java as well. It supports distributed operation and should therefore be suitable for very large crawls; but as of the writing of [81] it has not been scaled beyond 100 million pages.

2.2 Architecture Overview

Figure 2.1 shows the high-level architecture of a prototypical distributed web crawler. The crawler consists of multiple processes running on different machines connected by a high-speed network. Each crawling process consists of multiple worker threads, and each worker thread performs repeated work cycles.

At the beginning of each work cycle, a worker obtains a URL from the *Frontier* data structure, which dispenses URLs according to their priority and to politeness policies. The worker thread then invokes the *HTTP fetcher*. The fetcher first calls a DNS sub-module to resolve the host component of the URL into the IP address of the corresponding web server (using cached results of prior resolutions if possible), and then connects to the web server, checks for any robots exclusion rules (which typically are cached as well), and attempts to download the web page.

If the download succeeds, the web page may or may not be stored in a repository of harvested web pages (not shown). In either case, the page is passed to the *Link extractor*, which parses the page's HTML content and extracts hyperlinks contained therein. The corresponding URLs are then passed to a *URL distributor*, which assigns each URL to a crawling process. This assignment is typically made by hashing the URLs host component, its domain, or its IP address (the latter requires additional DNS resolutions). Since most hyperlinks refer to pages on the same web site, assignment to the local crawling process is the common case.

Next, the URL passes through the *Custom URL filter* (e.g., to exclude URLs belonging to "black-listed" sites, or URLs with particular file extensions that are not of interest) and into the *Duplicate URL eliminator*, which maintains the set of all URLs discovered so far and passes on only never-before-seen URLs. Finally, the *URL prioritizer* selects a position for the URL in the Frontier, based on factors such as estimated page importance or rate of change.[1]

[1] Change rates play a role in *incremental* crawlers (Section 2.3.5), which route fetched URLs back to the prioritizer and Frontier.

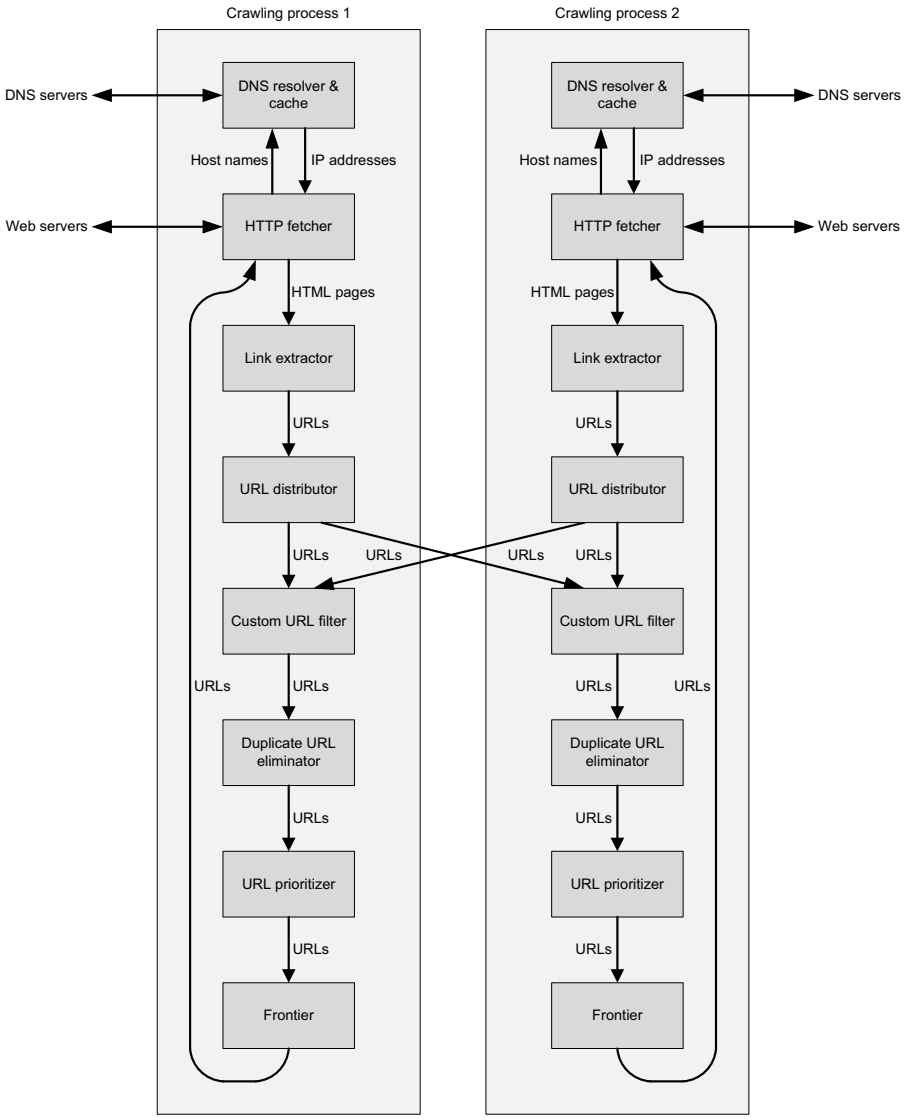

Fig. 2.1 Basic crawler architecture.

2.3 Key Design Points

Web crawlers download web pages by starting from one or more *seed URLs*, downloading each of the associated pages, extracting the

hyperlink URLs contained therein, and recursively downloading those pages. Therefore, any web crawler needs to keep track both of the URLs that are to be downloaded, as well as those that have already been downloaded (to avoid unintentionally downloading the same page repeatedly). The required state is a set of URLs, each associated with a flag indicating whether the page has been downloaded. The operations that must be supported are: Adding a new URL, retrieving a URL, marking a URL as downloaded, and testing whether the set contains a URL. There are many alternative in-memory data structures (e.g., trees or sorted lists) that support these operations. However, such an implementation does not scale to web corpus sizes that exceed the amount of memory available on a single machine.

To scale beyond this limitation, one could either maintain the data structure (e.g., the tree or sorted list) on disk, or use an off-the-shelf database management system. Either solution allows maintaining set sizes that exceed main memory; however, the cost of accessing items in the set (particularly for the purpose of set membership test) typically involves a disk seek, making it a fairly expensive operation. To achieve high performance, a more specialized approach is needed.

Virtually every modern web crawler splits the crawl state into two major data structures: One data structure for maintaining the set of URLs that have been discovered (whether downloaded or not), and a second data structure for maintaining the set of URLs that have yet to be downloaded. The first data structure (sometimes called the "URL-seen test" or the "duplicated URL eliminator") must support set addition and set membership testing, while the second data structure (usually called the *frontier*) must support adding URLs, and selecting a URL to fetch next.

2.3.1 Frontier Data Structure and Politeness

A straightforward implementation of the frontier data structure is a First-in-First-out (FIFO) queue. Such an implementation results in a breadth-first traversal of the web graph. However, this simple approach has drawbacks: Most hyperlinks on the web are "relative" (i.e., refer to another page on the same web server). Therefore, a frontier realized

as a FIFO queue contains long runs of URLs referring to pages on the same web server, resulting in the crawler issuing many consecutive HTTP requests to that server. A barrage of requests in short order is considered "impolite," and may be construed as a denial-of-service attack on the web server. On the other hand, it would be wasteful for the web crawler to space out requests to the same server without doing other useful work in the meantime. This problem is compounded in a multithreaded or distributed crawler that issues many HTTP requests in parallel.

Most web crawlers obey a policy of not issuing multiple overlapping requests to the same server. An easy way to realize this is to maintain a mapping from web servers to crawling threads, e.g., by hashing the host component of each URL.[2] In this design, each crawling thread has a separate FIFO queue, and downloads only URLs obtained from that queue.

A more conservative politeness policy is to space out requests to each web server according to that server's capabilities. For example, a crawler may have a policy to delay subsequent requests to a server by a multiple (say 10×) of the time it took to download the last page from that server. This policy ensures that the crawler consumes a bounded fraction of the web server's resources. It also means that in a given time interval, fewer pages will be downloaded from slow or poorly connected web servers than from fast, responsive web servers. In other words, this crawling policy is biased toward well-provisioned web sites. Such a policy is well-suited to the objectives of search engines, since large and popular web sites tend also to be well-provisioned.

The Mercator web crawler implemented such an adaptive politeness policy. It divided the frontier into two parts, a "front end" and a "back end." The front end consisted of a single queue Q, and URLs were added to the frontier by enqueuing them into that queue. The back

[2] To amortize hardware cost, many web servers use *virtual hosting*, meaning that multiple symbolic host names resolve to the same IP address. Simply hashing the host component of each URL to govern politeness has the potential to overload such web servers. A better scheme is to resolve the URLs symbolic host name to an IP address and use a hash of that address to assign URLs to a queue. The drawback of that approach is that the latency of DNS resolution can be high (see Section 2.3.3), but fortunately there tends to be a high amount of locality in the stream of discovered host names, thereby making caching effective.

end consisted of many separate queues; typically three times as many queues as crawling threads. Each queue contained URLs belonging to a single web server; a table T on the side maintained a mapping from web servers to back-end queues. In addition, associated with each back-end queue q was a time t at which the next URL from q may be processed. These (q, t) pairs were organized into an in-memory priority queue, with the pair with lowest t having the highest priority. Each crawling thread obtained a URL to download by removing the highest-priority entry (q, t) from the priority queue, waiting if necessary until time t had been reached, dequeuing the next URL u from q, downloading it, and finally reinserting the pair $(q, t_{now} + k \cdot x)$ into the priority queue, where t_{now} is the current time, x is the amount of time it took to download u, and k is a "politeness parameter"; typically 10. If dequeuing u from q left q empty, the crawling thread would remove the mapping from $host(u)$ to q from T, repeatedly dequeue a URL u' from Q and enqueue u' into the back-end queue identified by $T(host(u'))$, until it found a u' such that $host(u')$ was not contained in T. At this point, it would enqueue u' in q and update T to map $host(u')$ to q.

In addition to obeying politeness policies that govern the rate at which pages are downloaded from a given web site, web crawlers may also want to prioritize the URLs in the frontier. For example, it may be desirable to prioritize pages according to their estimated usefulness (based for example on their PageRank [101], the traffic they receive, the reputation of the web site, or the rate at which the page has been updated in the past). The page ordering question is discussed in Section 4.

Assuming a mechanism for assigning crawl priorities to web pages, a crawler can structure the frontier (or in the Mercator design described above, the front-end queue) as a disk-based priority queue ordered by usefulness. The standard implementation of a priority queue is a heap, and insertions into a heap of n elements require $\log(n)$ element accesses, each access potentially causing a disk seek, which would limit the data structure to a few hundred insertions per second — far less than the URL ingress required for high-performance crawling.

An alternative solution is to "discretize" priorities into a fixed number of priority levels (say 10 to 100 levels), and maintain a separate URL

FIFO queue for each level. A URL is assigned a discrete priority level, and inserted into the corresponding queue. To dequeue a URL, either the highest-priority nonempty queue is chosen, or a randomized policy biased toward higher-priority queues is employed.

2.3.2 URL Seen Test

As outlined above, the second major data structure in any modern crawler keeps track of the set of URLs that have been previously discovered and added to frontier. The purpose of this data structure is to avoid adding multiple instances of the same URL to the frontier; for this reason, it is sometimes called the *URL-seen test* (UST) or the *duplicate URL eliminator* (DUE). In a simple batch crawling setting in which pages are downloaded only once, the UST needs to support insertion and set membership testing; in a continuous crawling setting in which pages are periodically re-downloaded (see Section 2.3.5), it must also support deletion, in order to cope with URLs that no longer point to a valid page.

There are multiple straightforward in-memory implementations of a UST, e.g., a hash table or Bloom filter [20]. As mentioned above, in-memory implementations do not scale to arbitrarily large web corpora; however, they scale much further than in-memory implementations of the frontier, since each URL can be compressed to a much smaller token (e.g., a 10-byte hash value). Commercial search engines employ distributed crawlers (Section 2.3.4), and a hash table realizing the UST can be partitioned across the machines in the crawling cluster, further increasing the limit of how far such an in-memory implementation can be scaled out.

If memory is at a premium, the state of the UST must reside on disk. In a disk-based hash table, each lookup requires a disk seek, severely limiting the throughput. Caching popular URLs can increase the throughput by about an order of magnitude [27] to a few thousand lookups per second, but given that the average web page contains on the order of a hundred links and that each link needs to be tested for novelty, the crawling rate would still be limited to tens of pages per second under such an implementation.

While the *latency* of disk seeks is poor (a few hundred seeks per second), the *bandwidth* of disk reads and writes is quite high (on the order of 50–100 MB per second in modern disks). So, implementations performing random file accesses perform poorly, but those that perform streaming sequential reads or writes can achieve reasonable throughput. The Mercator crawler leveraged this observation by aggregating many set lookup and insertion operations into a single large batch, and processing this batch by sequentially reading a set of sorted URL hashes from disk and writing them (plus the hashes of previously undiscovered URLs) out to a new file [94].

This approach implies that the set membership is delayed: We only know whether a URL is new after the batch containing the URL has been merged with the disk file. Therefore, we cannot decide whether to add the URL to the frontier until the merge occurs, i.e., we need to retain all the URLs in a batch, not just their hashes. However, it is possible to store these URLs temporarily on disk and read them back at the conclusion of the merge (again using purely sequential I/O), once it is known that they had not previously been encountered and should thus be added to the frontier. Adding URLs to the frontier in a delayed fashion also means that there is a lower bound on how soon they can be crawled; however, given that the frontier is usually far larger than a DUE batch, this delay is imperceptible except for the most high-priority URLs.

The IRLbot crawler [84] uses a refinement of the Mercator scheme, where the batch of URLs arriving at the DUE is also written to disk, distributed over multiple files keyed by the prefix of each hash. Once the size of the largest file exceeds a certain threshold, the files that together hold the batch are read back into memory one by one and merge-sorted into the main URL hash file on disk. At the conclusion of the merge, URLs are forwarded to the frontier as in the Mercator scheme. Because IRLbot stores the batch on disk, the size of a single batch can be much larger than Mercator's in-memory batches, so the cost of the merge-sort with the main URL hash file is amortized over a much larger set of URLs.

In the Mercator scheme and its IRLbot variant, merging a batch of URLs into the disk-based hash file involves reading the entire old hash

file and writing out an updated version. Hence, the time requirement is proportional to the number of discovered URLs. A modification of this design is to store the URL hashes on disk in sorted order as before, but sparsely packed rather than densely packed. The k highest-order bits of a hash determine the disk block where this hash resides (if it is present). Merging a batch into the disk file is done in place, by reading a block for which there are hashes in the batch, checking which hashes are not present in that block, and writing the updated block back to disk. Thus, the time requirement for merging a batch is proportional to the size of the batch, not the number of discovered URLs (albeit with high constant due to disk seeks resulting from skipping disk blocks). Once any block in the file fills up completely, the disk file is rewritten to be twice as large, and each block contains hashes that now share their $k + 1$ highest-order bits.

2.3.3 Auxiliary Data Structures

In addition to the two main data structures discussed in Sections 2.3.1 and 2.3.2 — the frontier and the UST/DUE — web crawlers maintain various auxiliary data structures. We discuss two: The robots exclusion rule cache and the DNS cache.

Web crawlers are supposed to adhere to the *Robots Exclusion Protocol* [83], a convention that allows a web site administrator to bar web crawlers from crawling their site, or some pages within the site. This is done by providing a file at URL `/robots.txt` containing rules that specify which pages the crawler is allowed to download. Before attempting to crawl a site, a crawler should check whether the site supplies a `/robots.txt` file, and if so, adhere to its rules. Of course, downloading this file constitutes crawling activity in itself. To avoid repeatedly requesting `/robots.txt`, crawlers typically cache the results of previous requests of that file. To bound the size of that cache, entries must be discarded through some cache eviction policy (e.g., least-recently used); additionally, web servers can specify an expiration time for their `/robots.txt` file (via the HTTP `Expires` header), and cache entries should be discarded accordingly.

URLs contain a host component (e.g., `www.yahoo.com`), which is "resolved" using the *Domain Name Service* (DNS), a protocol that

exposes a globally distributed mapping from symbolic host names to IP addresses. DNS requests can take quite a long time due to the request-forwarding nature of the protocol. Therefore, crawlers often maintain their own DNS caches. As with the robots exclusion rule cache, entries are expired according to both a standard eviction policy (such as least-recently used), and to expiration directives.

2.3.4 Distributed Crawling

Web crawlers can be distributed over multiple machines to increase their throughput. This is done by partitioning the URL space, such that each crawler machine or *node* is responsible for a subset of the URLs on the web. The URL space is best partitioned across web site boundaries [40] (where a "web site" may refer to all URLs with the same symbolic host name, same domain, or same IP address). Partitioning the URL space across site boundaries makes it easy to obey politeness policies, since each crawling process can schedule downloads without having to communicate with other crawler nodes. Moreover, all the major data structures can easily be partitioned across site boundaries, i.e., the frontier, the DUE, and the DNS and robots exclusion caches of each node contain URL, robots exclusion rules, and name-to-address mappings associated with the sites assigned to that node, and nothing else.

Crawling processes download web pages and extract URLs, and thanks to the prevalence of relative links on the web, they will be themselves responsible for the large majority of extracted URLs. When a process extracts a URL u that falls under the responsibility of another crawler node, it forwards u to that node. Forwarding of URLs can be done through peer-to-peer TCP connections [94], a shared file system [70], or a central coordination process [25, 111]. The amount of communication with other crawler nodes can be reduced by maintaining a cache of popular URLs, used to avoid repeat forwardings [27].

Finally, a variant of distributed web crawling is *peer-to-peer* crawling [10, 87, 100, 112, 121], which spreads crawling over a loosely collaborating set of crawler nodes. Peer-to-peer crawlers typically employ some form of distributed hash table scheme to assign URLs to crawler

nodes, enabling them to cope with sporadic arrival and departure of crawling nodes.

2.3.5 Incremental Crawling

Web crawlers can be used to assemble one or more static snapshots of a web corpus (*batch crawling*), or to perform *incremental* or *continuous crawling*, where the resources of the crawler are divided between downloading newly discovered pages and re-downloading previously crawled pages. Efficient incremental crawling requires a few changes to the major data structures of the crawler. First, as mentioned in Section 2.3.2, the DUE should support the deletion of URLs that are no longer valid (e.g., that result in a 404 HTTP return code). Second, URLs are retrieved from the frontier and downloaded as in batch crawling, but they are subsequently reentered into the frontier. If the frontier allows URLs to be prioritized, the priority of a previously downloaded URL should be dependent on a model of the page's temporal behavior based on past observations (see Section 5). This functionality is best facilitated by augmenting URLs in the frontier with additional information, in particular previous priorities and compact sketches of their previous content. This extra information allows the crawler to compare the sketch of the just-downloaded page to that of the previous version, for example raising the priority if the page has changed and lowering it if it has not. In addition to content evolution, other factors such as page quality are also often taken into account; indeed there are many fast-changing "spam" web pages.

3

Crawl Ordering Problem

Aside from the intra-site politeness considerations discussed in Section 2, a crawler is free to visit URLs in any order. The crawl order is extremely significant, because for the purpose of crawling the web can be considered infinite — due to the growth rate of new content, and especially due to dynamically generated content [8]. Indeed, despite their impressive capacity, modern commercial search engines only index (and likely only crawl) a fraction of discoverable web pages [11]. The crawler ordering question is even more crucial for the countless smaller-scale crawlers that perform *scoped crawling* of targeted subsets of the web.

Sections 3–5 survey work on selecting a good crawler order, with a focus on two basic considerations:

- **Coverage.** The fraction of desired pages that the crawler acquires successfully.
- **Freshness.** The degree to which the acquired page snapshots remain up-to-date, relative to the current "live" web copies.

Issues related to redundant, malicious or misleading content are covered in Section 6. Generally speaking, techniques to avoid unwanted content

can be incorporated into the basic crawl ordering approaches without much difficulty.

3.1 Model

Most work on crawl ordering abstracts away the architectural details of a crawler (Section 2), and assumes that URLs in the frontier data structure can be reordered freely. The resulting simplified crawl ordering model is depicted in Figure 3.1. At a given point in time, some historical crawl order has already been executed (P_1, P_2, P_3, P_4, P_5 in the diagram), and some future crawl order has been planned ($P_6, P_7, P_4, P_8, \ldots$).[1]

In the model, all pages require the same amount of time to download; the (constant) rate of page downloading is called the *crawl rate*, typically measured in pages/second. (Section 2 discussed how to maximize the crawl rate; here it is assumed to be fixed.) The crawl rate is not relevant to batch crawl ordering methods, but it is a key factor when scheduling page revisitations in incremental crawling.

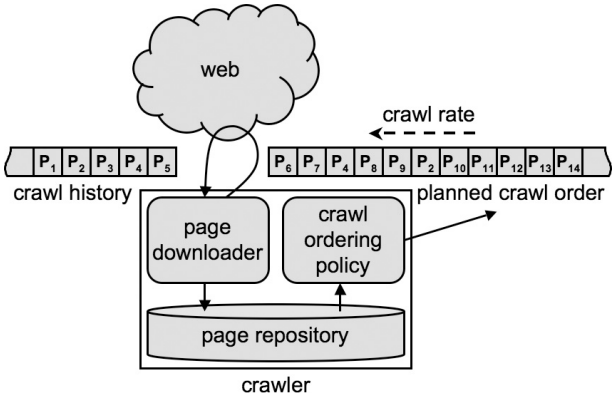

Fig. 3.1 Crawl ordering model.

[1] Some approaches treat the crawl ordering problem hierarchically, e.g., select a visitation order for web sites, and within each site select a page visitation order. This approach helps mitigate the complexity of managing a crawl ordering policy, and is well aligned with policies that rely primarily on site-level metrics such as site-level PageRank to drive crawl ordering decisions. Many of the insights about page-level crawl ordering also apply at the site level.

Pages downloaded by the crawler are stored in a *repository*. The future crawl order is determined, at least in part, by analyzing the repository. For example, one simple policy mentioned earlier, *breadth-first search*, extracts hyperlinks from pages entering the repository, identifies linked-to pages that are not already part of the (historical or planned) crawl order, and adds them to the end of the planned crawl order.

The content of a web page is subject to change over time, and it is sometimes desirable to re-download a page that has already been downloaded, to obtain a more recent snapshot of its content. As mentioned in Section 2.3.5, two approaches exist for managing repeated downloads:

- **Batch crawling.** The crawl order does not contain duplicate occurrences of any page, but the entire crawling process is periodically halted and restarted as a way to obtain more recent snapshots of previously crawled pages. Information gleaned from previous crawl iterations (e.g., page importance score estimates) may be fed to subsequent ones.
- **Incremental crawling.** Pages may appear multiple times in the crawl order, and crawling is a continuous process that conceptually never terminates.

It is believed that most modern commercial crawlers perform incremental crawling, which is more powerful because it allows re-visitation of pages at different rates. (A detailed comparison between incremental and batch crawling is made by Cho and García-Molina [39].)

3.1.1 Limitations

This model has led to a good deal of research with practical implications. However, as with all models, it simplifies reality. For one thing, as discussed in Section 2, a large-scale crawler maintains its frontier data structure on disk, which limits opportunities for reordering. Generally speaking, the approach of maintaining a prioritized ensemble of FIFO queues (see Section 2.3.1) can be used to approximate a desired crawl order. We revisit this issue in Sections 4.3 and 5.3.

Other real-world considerations that fall outside the model include:

- Some pages (or even versions of a page) take longer to download than others, due to differences in size and network latency.
- Crawlers take special care to space out downloads of pages from the same server, to obey politeness constraints, see Section 2.3.1. Crawl ordering policies that assume a single crawl rate constraint can, at least in principle, be applied on a per-server basis, i.e., run n independent copies of the policy for n servers.
- As described in Section 2, modern commercial crawlers utilize many simultaneous page downloader threads, running on many independent machines. Hence rather than a single totally ordered list of pages to download, it is more accurate to think of a set of parallel lists, encoding a partial order.
- Special care must be taken to avoid crawling redundant and malicious content; we treat these issues in Section 6.
- If the page repository runs out of space, and expanding it is not considered worthwhile, is becomes necessary to *retire* some of the pages stored there (although it may make sense to retain some metadata about the page, to avoid recrawling it). We are not aware of any scholarly work on how to select pages for retirement.

3.2 Web Characteristics

Before proceeding, we describe some structural and evolutionary properties of the web that are relevant to the crawl ordering question. The findings presented here are drawn from studies that used data sets of widely varying size and scope, taken at different dates over the span of a decade, and analyzed via a wide array of methods. Hence, caution is warranted in their interpretation.

3.2.1 Static Characteristics

Several studies of the structure of the *web graph*, in which pages are encoded as vertices and hyperlinks as directed edges, have been

conducted. One notable study is by Broder et al. [26], which uncovered a "bowtie" structure consisting of a central strongly connected component (the *core*), a component that can reach the core but cannot be reached from the core, and a component that can be reached from the core but cannot reach the core. (In addition to these three main components there are a number of small, irregular structures such as disconnected components and long "tendrils.")

Hence there exist many ordered pairs of pages (P_1, P_2) such that there is no way to reach P_2 by starting at P_1 and repeatedly following hyperlinks. Even in cases where P_2 is reachable from P_1, the distance can vary greatly, and in many cases hundreds of links must be traversed. The implications for crawling are: (1) one cannot simply crawl to depth N, for a reasonable value of N like $N = 20$, and be assured of covering the entire web graph; (2) crawling "seeds" (the pages at which a crawler commences) should be selected carefully, and multiple seeds may be necessary to ensure good coverage.

In an earlier study, Broder et al. [28] showed that there is an abundance of near-duplicate content of the web. Using a corpus of 30 million web pages collected by the AltaVista crawler, they used the *shingling* algorithm to cluster the corpus into groups of similar pages, and found that 29% of the pages were more than 50% similar to other pages in the corpus, and 11% of the pages were exact duplicates of other pages. Sources of near-duplication include mirroring of sites (or portions of sites) and URL synonymy, see Section 6.1.

Chang et al. [35] studied the "deep web," i.e., web sites whose content is not reachable via hyperlinks and instead can only be retrieved by submitting HTML forms. The findings include: (1) there are over one million deep web sites; (2) more deep web sites have structured (multi-field) query interfaces than unstructured (single-field) ones; and (3) most query interfaces are located within a few links of the root of a web site, and are thus easy to find by shallow crawling from the root page.

3.2.2 Temporal Characteristics

One of the objectives of crawling is to maintain freshness of the crawled corpus. Hence it is important to understand the temporal

characteristics of the web, both in terms of site-level evolution (the appearance and disappearance of pages on a site) and page-level evolution (changing content within a page).

3.2.2.1 Site-Level Evolution

Dasgupta et al. [48] and Ntoulas et al. [96] studied creation and retirement of pages and links inside a number of web sites, and found the following characteristics (these represent averages across many sites):

- New pages are created at a rate of 8% per week.
- Pages are retired at a rapid pace, such that during the course of one year 80% of pages disappear.
- New links are created at the rate of 25% per week, which is significantly faster than the rate of new page creation.
- Links are retired at about the same pace as pages, with 80% disappearing in the span of a year.
- It is possible to discover 90% of new pages by monitoring links spawned from a small, well-chosen set of old pages (for most sites, five or fewer pages suffice, although for some sites hundreds of pages must be monitored for this purpose). However, discovering the remaining 10% requires substantially more effort.

3.2.2.2 Page-Level Evolution

Some key findings about the frequency with which an individual web page undergoes a change are:

- Page change events are governed by a Poisson process, which means that changes occur randomly and independently, at least in the case of pages that change less frequently than once a day [39].[2]
- Page change frequencies span multiple orders of magnitude (sub-hourly, hourly, daily, weekly, monthly, annually), and each order of magnitude includes a substantial fraction of

[2] A Poisson change model was originally postulated by Coffman et al. [46].

pages on the web [2, 39]. This finding motivates the study of non-uniform page revisitation schedules.

- Change frequency is correlated with visitation frequency, URL depth, domain and topic [2], as well as page length [60].
- A page's change frequency tends to remain stationary over time, such that past change frequency is a fairly good predictor of future change frequency [60].

Unfortunately, it appears that there is no simple relationship between the frequency with which a page changes and the cumulative amount of content that changes over time. As one would expect, pages with moderate change frequency tend to exhibit a higher cumulative amount of changed content than pages with a low change frequency. However, pages with high change frequency tend to exhibit *less* cumulative change than pages with moderate change frequency. On the encouraging side, the amount of content that changed on a page in the past is a fairly good predictor of the amount of content that will change in the future (although the degree of predictability varies from web site to web site) [60, 96].

Many changes are confined to a small, contiguous region of a web page [60, 85], and/or only affect transient words that do not characterize the core, time-invariant theme of the page [2]. Much of the "new" content added to web pages is actually taken from other pages [96].

The temporal behavior of (regions of) web pages can be divided into three categories: *Static* (no changes), *churn* (new content supplants old content, e.g., quote of the day), and *scroll* (new content is appended to old content, e.g., blog entries). Simple generative models for the three categories collectively explain nearly all observed temporal web page behavior [99].

Most web pages include at least some static content, resulting in an upper bound on the divergence between an old snapshot of a page and the live copy. The shape of the curve leading to the upper bound depends on the mixture of churn and scroll content, and the rates of churning and scrolling. One simple way to characterize a page is with a pair of numbers: (1) the divergence upper bound (i.e., the amount of non-static content), under some divergence measure such

Technique	Objectives		Factors considered		
	Coverage	*Freshness*	*Importance*	*Relevance*	*Dynamicity*
Breadth-first search [43, 95, 108]	√				
Prioritize by indegree [43]	√		√		
Prioritize by PageRank [43, 45]	√		√		
Prioritize by site size [9]	√		√		
Prioritize by spawning rate [48]	√				√
Prioritize by search impact [104]	√		√	√	
Scoped crawling (Section 4.2)	√			√	
Minimize obsolescence [41, 46]		√	√		√
Minimize age [41]		√	√		√
Minimize incorrect content [99]		√	√		√
Minimize embarrassment [115]		√	√	√	√
Maximize search impact [103]		√	√	√	√
Update capture (Section 5.2)		√		√	√
WebFountain [56]	√	√			√
OPIC [1]	√	√	√		

Fig. 3.2 Taxonomy of crawl ordering techniques.

as shingle [28] or word difference; and (2) the amount of time it takes to reach the upper bound (i.e., the time taken for all non-static content to change) [2].

3.3 Taxonomy of Crawl Ordering Policies

Figure 3.2 presents a high-level taxonomy of published crawl ordering techniques. The first group of techniques focuses exclusively on ordering pages for first-time downloading, which affects *coverage*. These can be applied either in the batch crawling scenario, or in the incremental crawling scenario in conjunction with a separate policy governing re-downloading of pages to maintain *freshness*, which is the focus of the second group of techniques. Techniques in the third group consider the combined problem of interleaving first-time downloads with re-downloads, to balance coverage and freshness.

As reflected in Figure 3.2, crawl ordering decisions tend to be based on some combination of the following factors: (1) *importance* of a page or site, relative to others; (2) *relevance* of a page or site to the purpose served by the crawl; and (3) *dynamicity*, or how the content of a page/site tends to change over time.

Some crawl ordering techniques are broader than others in terms of which factors they consider and which objectives they target. Ones that focus narrowly on a specific aspect of crawling typically aim for a "better" solution with respect to that aspect, compared with broader "all-in-one" techniques. On the other hand, to be usable they may need to be extended or combined with other techniques. In some cases a straightforward extension exists (e.g., add importance weights to an importance-agnostic formula for scheduling revisitations), but often not. There is no published work on the best way to combine multiple specialized techniques into a comprehensive crawl ordering approach that does well across the board.

The next two chapters describe the techniques summarized in Figure 3.2, starting with ones geared toward batch crawling (Section 4), and then moving to incremental crawl ordering techniques (Section 5).

4

Batch Crawl Ordering

A batch crawler traverses links outward from an initial *seed set* of URLs. The seed set may be selected algorithmically, or by hand, based on criteria such as importance, outdegree, or other structural features of the web graph [120]. A common, simple approach is to use the root page of a web directory site such as OpenDirectory, which links to many important sites across a broad range of topics. After the seed set has been visited and links have been extracted from the seed set pages, the crawl ordering policy takes over.

The goal of the crawl ordering policy is to maximize the *weighted coverage* (WC) achieved over time, given a fixed crawl rate. WC is defined as:

$$\mathrm{WC}(t) = \sum_{p \in \mathcal{C}(t)} w(p),$$

where t denotes the time elapsed since the crawl began, $\mathcal{C}(t)$ denotes the set of pages crawled up to time t (under the fixed crawl rate assumption, $|\mathcal{C}(t)| \propto t$), and $w(p)$ denotes a numeric weight associated with page p. The weight function $w(p)$ is chosen to reflect the purpose of the crawl. For example, if the purpose is to crawl pages about helicopters, one sets $w(p) = 1$ for pages about helicopters, and $w(p) = 0$ for all other pages.

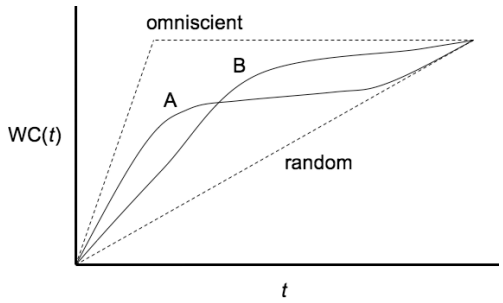

Fig. 4.1 Weighted coverage (WC) as a function of time elapsed (t) since the beginning of a batch crawl.

Figure 4.1 shows some hypothetical WC curves. Typically, $w(p) \geq 0$, and hence $\mathrm{WC}(t)$ is monotonic in t. Under a *random* crawl ordering policy, $\mathrm{WC}(t)$ is roughly linear in t; this line serves as a baseline upon which other policies strive to improve. An *omniscient* policy, which downloads pages in descending order of $w(p)$ yields a theoretical upper-bound curve. (For the helicopter example, the omniscient policy downloads all helicopter pages first, thereby achieving maximal WC before the end of the crawl.) Policies A and B fall in-between the random and omniscient cases, with A performing better in the early stages of the crawl, but B performing better toward the end. The choice between A and B depends on how long the crawl is allowed to run before being stopped (and possibly re-started to bring in a fresh batch of pages), and to what use, if any, pages obtained early in the crawl are put while the crawl is still in flight.

The above framework can be applied to *comprehensive* batch crawling, in which the goal is to achieve broad coverage of all types of content, as well as to *scoped* batch crawling, where the crawler restricts its attention to a relatively narrow slice of the web (e.g., pages about helicopters). This chapter examines the two scenarios in turn, focusing initially on crawl order effectiveness, with implementation and efficiency questions covered at the end.

4.1 Comprehensive Crawling

When the goal is to cover high-quality content of all varieties, a popular choice of weight function is $w(p) = \mathrm{PR}(p)$, where $\mathrm{PR}(p)$ is

p's importance score as measured by PageRank [101].[1] Variations on PageRank used in the crawling context include: Only counting *external* links, i.e., ones that go between two web sites (Najork and Wiener [95] discuss some tradeoffs involved in counting all links versus only external links); biasing the PageRank random jumps to go to a *trusted set* of pages believed not to engage in spamming (see Cho and Schonfeld [45] for discussion); or omitting random jumps entirely, as done by Abiteboul et al. [1].

In view of maximizing coverage weighted by PageRank or some variant, three main types of crawl ordering policies have been examined in the literature. In increasing order of complexity, they are:

- **Breadth-first search [108].** Pages are downloaded in the order in which they are first discovered, where discovery occurs via extracting all links from each page immediately after it is downloaded. Breadth-first crawling is appealing due to its simplicity, and it also affords an interesting coverage guarantee: In the case of PageRank that is biased to a small trusted page set T, a breadth-first crawl of depth d using T as its seed set achieves WC $\geq 1 - \alpha^{d+1}$ [45], where α is the PageRank damping parameter.
- **Prioritize by indegree [43].** The page with the highest number of incoming hyperlinks from previously downloaded pages, is downloaded next. Indegree is sometimes used as a low-complexity stand-in for PageRank, and is hence a natural candidate for crawl ordering under a PageRank-based objective.
- **Prioritize by PageRank (variant/estimate) [1, 43, 45].** Pages are downloaded in descending order of PageRank (or some variant), as estimated based on the pages and links acquired so far by the crawler. Straightforward application

[1] Although this cumulative PageRank measure has been used extensively in the literature, Boldi et al. [23] caution that absolute PageRank scores may not be especially meaningful, and contend that PageRank should be viewed as a way to establish relative page orderings. Moreover, they show that biasing a crawl toward pages with high PageRank in the crawled subgraph leads to subgraphs whose PageRank ordering differs substantially from the PageRank-induced ordering of the same pages in the full graph.

of this method involves recomputing PageRank scores after each download, or updating the PageRank scores incrementally [38]. Another option is to recompute PageRank scores only periodically, and rely on an approximation scheme between recomputations. Lastly, Abiteboul et al. [1] gave an efficient online method of estimating a variant of PageRank that does not include random jumps, designed for use in conjunction with a crawler.

The three published empirical studies that evaluate the above policies over real web data are listed in Figure 4.2 (Najork and Wiener [95] evaluated only breadth-first search). Under the objective of crawling high-PageRank pages early ($w(p) = \mathrm{PR}(p)$), the main findings from these studies are the following:

- Starting from high-PageRank seeds, breadth-first crawling performs well early in the crawl (low t), but not as well as the other policies later in the crawl (medium to high t).
- Perhaps unsurprisingly, prioritization by PageRank performs well throughout the entire crawl. The shortcut of only recomputing PageRank periodically leads to poor performance, but the online approximation scheme by Abiteboul et al. [1] performs well. Furthermore, in the context of repeated batch crawls, it is beneficial to use PageRank values from previous iterations to drive the current iteration.
- There is no consensus on prioritization by indegree: One study (Cho et al. [43]) found that it worked fairly well (almost as well as prioritization by PageRank), whereas another study (Baeza-Yates et al. [9]) found that it performed very

STUDY	DATA SET	DATA SIZE	PUB. YEAR
Cho et al. [43]	Stanford web	10^5	1998
Najork and Wiener [95]	general web	10^8	2001
Baeza-Yates et al. [9]	Chile and Greece	10^6	2005

Fig. 4.2 Empirical studies of batch crawl ordering policies.

poorly. The reason given by Baeza-Yates et al. [9] for poor performance is that it is overly greedy in going after high-indegree pages, and therefore it takes a long time to find pages that have high indegree and PageRank yet are only discoverable via low-indegree pages. The two studies are over different web collections that differ in size by an order of magnitude, and are seven years apart in time.

In addition to the aforementioned results, Baeza-Yates et al. [9] proposed a crawl policy that gives priority to sites containing a large number of discovered but uncrawled URLs. According to their empirical study, which imposed per-site politeness constraints, toward the end of the crawl (high t) the proposed policy outperforms policies based on breadth-first search, indegree, and PageRank. The reason is that it avoids the problem of being left with a few very large sites at the end, which can cause a politeness bottleneck.

Baeza-Yates and Castillo [8] observed that although the web graph is effectively infinite, most user browsing activity is concentrated within a small distance of the root page of each web site. Arguably, a crawler should concentrate its activities there, and avoid exploring too deeply into any one site.

4.1.1 Search Relevance as the Crawling Objective

Fetterly et al. [58] and Pandey and Olston [104] argued that when the purpose of crawling is to supply content to a search engine, PageRank-weighted coverage may not be the most appropriate objective. According to the argument, it instead makes sense to crawl pages that would be viewed or clicked by search engine users, if present in the search index. For example, one may set out to crawl all pages that, if indexed, would appear in the top ten results of likely queries. Even if PageRank is one of the factors used to rank query results, the top result for query Q_1 may have lower PageRank than the eleventh result for some other query Q_2, especially if Q_2 pertains to a more established topic.

Fetterly et al. [58] evaluated four crawl ordering policies (breadth-first; prioritize by indegree; prioritize by trans-domain indegree;

prioritize by PageRank) under two relevance metrics:

- **MaxNDCG:** The total Normalized Distributed Cumulative Gain (NDCG) [79] score of a set of queries evaluated over the crawled pages, assuming optimal ranking.
- **Click count:** The total number of clicks the crawled pages attracted via a commercial search engine in some time period.

The main findings were that prioritization by PageRank is the most reliable and effective method on these metrics, and that imposing per-domain page limits boosts effectiveness.

Pandey and Olston [104] proposed a technique for explicitly ordering pages by expected relevance impact, under the objective of maximizing coverage weighted by the number of times a page appears among the top N results of a user query. The relatively high computational overhead of the technique is mitigated by concentrating on queries whose results are likely to be improved by crawling additional pages (deemed *needy queries*). Relevance of frontier pages to needy queries is estimated from cues found in URLs and referring anchortext, as first proposed in the context of scoped crawling [43, 74, 108], discussed next.

4.2 Scoped Crawling

A scoped crawler strives to limit crawling activities to pages that fall within a particular category or *scope*, thereby acquiring in-scope content much faster and more cheaply than via a comprehensive crawl. Scope may be defined according to *topic* (e.g., pages about aviation), *geography* (e.g., pages about locations in and around Oldenburg, Germany [6]), *format* (e.g., images and multimedia), *genre* (e.g., course syllabi [51]), *language* (e.g., pages in Portuguese [65]), or other aspects. (Broadly speaking, page importance, which is the primary crawl ordering criterion discussed in Section 4.1, can also be thought of as a form of scope.)

Usage scenarios for scoped crawling include mining tasks that call for crawling a particular type of content (e.g., images of animals), personalized search engines that focus on topics of interest to a particular user (e.g., aviation and gardening), and search engines for the

"deep web" that use a surface-web crawler to locate gateways to deep-web content (e.g., HTML form interfaces). In another scenario, one sets out to crawl the full web by deploying many small-scale crawlers, each of which is responsible for a different slice of the web — this approach permits specialization to different types of content, and also facilitates loosely coupled distributed crawling (Section 2.3.4).

As with comprehensive crawling (Section 4.1), the mathematical objective typically associated with scoped crawling is maximization of weighted coverage $\mathrm{WC}(t) = \sum_{p \in \mathcal{C}(t)} w(p)$. In scoped crawling, the role of the weight function $w(p)$ is to reflect the degree to which page p falls within the intended scope. In the simplest case, $w(p) \in \{0, 1\}$, where 0 denotes that p is outside the scope and 1 denotes that p is in-scope. Hence weighted coverage measures the fraction of crawled pages that are in-scope, analogous to the *precision* metric used in information retrieval.

Typically the in-scope pages form a finite set (whereas the full web is often treated as infinite, as mentioned above). Hence it makes sense to measure *recall* in addition to precision. Two recall-oriented evaluation techniques have been proposed: (1) designate a few representative in-scope pages by hand, and measure what fraction of them are discovered by the crawler [92]; (2) measure the overlap among independent crawls initiated from different seeds, to see whether they converge on the same set of pages [34].

Topical crawling (also known as "focused crawling"), in which in-scope pages are ones that are relevant to a particular topic or set of topics, is by far the most extensively studied form of scoped crawling. Work on other forms of scope — e.g., pages with form interfaces [14], and pages within a geographical scope [6, 63] — tends to use similar methods to the ones used for topical crawling. Hence we primarily discuss topical crawling from this point forward.

4.2.1 Topical Crawling

The basic observation exploited by topical crawlers is that relevant pages tend to link to other relevant pages, either directly or via short chains of links. (This feature of the web has been verified empirically

in many studies, including Chakrabarti et al. [34] and Cho et al. [43].) The first crawl ordering technique to exploit this observation was *fish search* [53]. The fish search crawler categorized each crawled page p as either relevant or irrelevant (a binary categorization), and explored the neighborhood of each relevant page up to depth d looking for additional relevant pages.

A second generation of topical crawlers [43, 74, 108] explored the neighborhoods of relevant pages in a non-uniform fashion, opting to traverse the most promising links first. The link traversal order was governed by individual relevance estimates assigned to each linked-to page (a continuous relevance metric is used, rather than a binary one). If a crawled page p links to an uncrawled page q, the relevance estimate for q is computed via analysis of the text surrounding p's link to q (i.e., the anchortext and text near the anchortext[2]), as well as q's URL. In one variant, relevance estimates are smoothed by associating some portion of p's relevance score (and perhaps also the relevance scores of pages linking to p, and so on in a recursive fashion), with q. The motivation for smoothing the relevance scores is to permit discovery of pages that are relevant yet lack indicative anchortext or URLs.

A third-generation approach based on machine learning and link structure analysis was introduced by Chakrabarti et al. [33, 34]. The approach leverages pre-existing topic taxonomies such as the Open Directory and Yahoo!'s web directory, which supply examples of web pages matching each topic. These example pages are used to train a classifier[3] to map newly encountered pages into the topic taxonomy. The user selects a subset of taxonomy nodes (topics) of interest to crawl, and the crawler preferentially follows links from pages that the classifier deems most relevant to the topics of interest. Links from pages that match "parent topics" are also followed (e.g., if the user indicated an interest in bicycling, the crawler follows links from pages about sports in general). In addition, an attempt is made to identify *hub pages* — pages with a collection of links to topical pages — using the HITS link

[2] This aspect was studied in detail by Pant and Srinivasan [107].
[3] Pant and Srinivasan [106] offer a detailed study of classifier choices for topical crawlers.

analysis algorithm [82]. Links from hub pages are followed with higher priority than other links.

The empirical findings of Chakrabarti et al. [34] established topical crawling as a viable and effective paradigm:

- A general web crawler seeded with topical pages quickly becomes mired in irrelevant regions of the web, yielding very poor weighted coverage. In contrast, a topical crawler successfully stays within scope, and explores a steadily growing population of topical pages over time.
- Two topical crawler instances, started from disparate seeds, converge on substantially overlapping sets of pages.

Beyond the basics of topical crawling discussed above, there are two key considerations [92]: Greediness and adaptivity.

4.2.1.1 Greediness

Paths between pairs of relevant pages sometimes pass through one or more irrelevant pages. A topical crawler that is too *greedy* will stop when it reaches an irrelevant page, and never discovers subsequent relevant page(s). On the other extreme, a crawler that ignores relevance considerations altogether degenerates into a non-topical crawler, and achieves very poor weighted coverage, as we have discussed. The question of how greedily to crawl is an instance of the *explore versus exploit* tradeoff observed in many contexts. In this context, the question is: How should the crawler balance exploitation of direct links to (apparently) relevant pages, with exploration of other links that may, eventually, lead to relevant pages?

In the approach of Hersovici et al. [74], a page p inherits some of the relevance of the pages that link to p, and so on in a recursive fashion. This passing along of relevance forces the crawler to traverse irrelevant pages that have relevant ancestors. A *decay factor* parameter controls how rapidly relevance decays as more links are traversed. Eventually, if no new relevant pages are encountered, relevance approaches zero and the crawler ceases exploration on that path.

Later work by Diligenti et al. [54] proposed to classify pages according to their distance from relevant pages. Each uncrawled page p is assigned a distance estimate $d(p) \in [0, \infty)$ that represents the crawler's best guess as to how many links lie between p and the nearest relevant page.[4] Pages are ordered for crawling according to $d(\cdot)$. As long as one or more uncrawled pages having $d(p) = 0$ are available, the crawler downloads those pages; if not, the crawler resorts to downloading $d(p) = 1$ pages, and so on. The threshold used to separate "relevant" pages from "irrelevant" ones controls greediness: If the threshold is strict (i.e., only pages with strong relevance indications are classified as "relevant"), then the crawler will favor long paths to strongly relevant pages over short paths to weakly relevant pages, and vice versa.

A simple meta-heuristic to control the greediness of a crawler was proposed by Menczer et al. [92]: Rather than continuously adjusting the crawl order as new pages and new links are discovered, commit to crawling N pages from the current crawl order before reordering. This heuristic has the attractive property that it can be applied in conjunction with any existing crawl ordering policy. Menczer et al. [92] demonstrated empirically that this heuristic successfully controls the level of greediness, and that there is benefit in not being too greedy, in terms of improved weighted coverage in the long run. The study done by Diligenti et al. [54] also showed improved long-term weighted coverage by not being overly greedy. We are not aware of any attempts to characterize the optimal level of greediness.

4.2.1.2 Adaptivity

In most topical crawling approaches, once the crawler is unleashed, the page ordering strategy is fixed for the duration of the crawl. Some have studied ways for a crawler to *adapt* its strategy over time, in response to observations made while the crawl is in flight. For example, Aggarwal et al. [5] proposed a method to learn on the fly how best to combine

[4] The $d(\cdot)$ function can be trained in the course of the crawl as relevant and irrelevant pages are encountered at various distances from one another. As an optional enhancement to accelerate the training process, ancestors of page p are located using a full-web search engine that services "links-to" queries, and added to the training data.

relevance signals found in the content of pages linking to p, content of p's "siblings" (pages linked to by the same "parent" page), and p's URL, into a single relevance estimate on which to base the crawl order.

Evolutionary algorithms (e.g., genetic algorithms) have been explored as a means to adapt crawl behavior over time [37, 80, 91]. For example, the *InfoSpiders* approach [91] employs many independent crawling *agents*, each with its own relevance classifier that adapts independently over time. Agents reproduce and die according to an evolutionary process: Agents that succeed in locating relevant content multiply and mutate, whereas unsuccessful ones die off. The idea is to improve relevance estimates over time, and also to achieve specialization whereby different agents become adept at locating different pockets of relevant content.

4.3 Efficient Large-Scale Implementation

As discussed in Section 2.3.1, breadth-first crawl ordering can use simple disk-based FIFO queues. Basic scoped crawling methods also afford a fairly simple and efficient implementation: The process of assessing page relevance and assigning priorities to extracted URLs can occur in the main page processing pipeline,[5] and a disk-based priority queue may be used to maintain the crawling frontier (also discussed in Section 2.3.1).

Most of the comprehensive (non-scoped) approaches also operate according to numerical page priorities, but the priority values of enqueued pages are subject to change over time as new information is uncovered (e.g., new links to a page). The more sophisticated scoped crawling approaches also leverage global information (e.g., Chakrabarti et al. [34] used link analysis to identify topical hub pages), and therefore fall into this category as well. With time-varying priorities, one approach is to recompute priority values periodically — either from

[5] In cases where relevance is assessed via a trained classifier, training (and optional periodic retraining) can occur offline and out of the way of the main crawling pipeline.

scratch or incrementally[6] — using distributed disk-based methods similar to those employed in database and map-reduce environments.

Aside from facilitating scalable implementation, delaying the propagation of new signals to the crawl order has a side-effect of introducing an exploration component into an otherwise exploitation-dominated approach, which is of particular significance in the scoped crawling case (see Section 4.2.1.1). On the other hand, some time-critical crawling opportunities (e.g., a new entry of a popular blog) might be compromised. One way to mitigate this problem is to assign initial priority estimates that are not based on global analysis, e.g., using site-level features (e.g., site-level PageRank), URL features (e.g., number of characters or slashes in the URL string), or features of the page from which a URL has been extracted (e.g., that page's PageRank).

The OPIC approach [1] propagates numerical "cash" values to URLs extracted from crawled pages, including URLs already in the frontier. The intention is to maintain a running approximation of PageRank, without the high overhead of the full PageRank computation. If enough memory is (collectively) available on the crawling machines, cash counters can be held in memory and incremented in near-real-time (cash flows across machine boundaries can utilize MPI or another message-passing protocol). Alternatively, following the deferred-updating rubric mentioned above, cash increments can be logged to a side file, with periodic summing of cash values using a disk-based sort-merge algorithm.

[6] Basic computations like counting links can be maintained incrementally using efficient disk-based view maintenance algorithms; more elaborate computations like PageRank tend to be more difficult but offer some opportunities for incremental maintenance [38].

5

Incremental Crawl Ordering

In contrast to a batch crawler, a *continuous* or *incremental* crawler never "starts over." Instead, it continues running forever (conceptually speaking). To maintain freshness of old crawled content, an incremental crawler interleaves revisitation of previously crawled pages with first-time visitation of new pages. The aim is to achieve good freshness and coverage simultaneously.

Coverage is measured according to the same *weighted coverage* metric applied to batch crawlers (Section 4). An analogous *weighted freshness* metric is as follows:

$$\mathrm{WF}(t) = \sum_{p \in \mathcal{C}(t)} w(p) \cdot f(p,t),$$

where $f(p,t)$ is page p's freshness level at time t, measured in one of several possible ways (see below).[1] One is typically interested in the

[1] Pages that have been removed from the web (i.e., their URL is no longer valid) but whose removal has not yet been detected by the crawler, are assigned a special freshness level, e.g., the minimum value on the freshness scale in use.

steady-state average of WF:

$$\overline{\mathrm{WF}} = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathrm{WF}(t)\, dt.$$

At each step, an incremental crawler faces a choice between two basic actions:

(1) **Download a new page.** Consequences include:
   (a) May improve coverage.
   (b) May supply new links, which can lead to discovery of new pages.[2] (New links also contribute to the crawler's estimates of page importance, relevance, and other aspects like likelihood of being spam; cf. Section 6.)

(2) **Re-download an old page.** Consequences include:
   (a) May improve freshness.
   (b) May supply new links or reveal the removal of links, with similar ramifications as 1(b) above.

In the presence of dynamic pages and finite crawling resources, there is a tradeoff between coverage and freshness. There is no consensus about the best way to balance the two. Some contend that coverage and freshness are like apples and oranges, and balancing the two objectives should be left as a business decision, i.e., do we prefer broad coverage of content that may be somewhat out-of-date, or narrower coverage with fresher content? Others have proposed specific schemes for combining the two objectives into a single framework: The approach taken in WebFountain [56] focuses on the freshness problem, and folds in coverage by treating uncrawled pages as having a freshness value of zero. The OPIC approach [1] focuses on ensuring coverage of important pages, and in the process periodically revisits old important pages.

Aside from the two approaches just mentioned, most published work on crawling focuses either uniquely on coverage or uniquely on

[2] Pages can also be discovered via "out-of-band" channels, e.g., e-mail messages, RSS feeds, user browsing sessions.

freshness. We have already surveyed coverage-oriented techniques in Section 4, in the context of batch crawling. In incremental crawling, coverage can be expanded not only by following links from newly crawled pages, but also by monitoring old pages to detect any new links that might be added over time. This situation was studied by Dasgupta et al. [48], who used a set-cover formulation to identify small sets of old pages that collectively permit discovery of most new pages.

The remainder of this chapter is dedicated to techniques that revisit old pages to acquire a fresh version of their content (not just links to new pages). Section 5.1 focuses on maximizing the average freshness of crawled content, whereas Section 5.2 studies the subtly different problem of capturing the history of content updates. Following these conceptual discussions, Section 5.3 considers practical implementation strategies.

5.1 Maximizing Freshness

Here the goal is to maximize time-averaged weighted freshness, $\overline{\mathrm{WF}}$, as defined above. To simplify the study of this problem, it is standard practice to assume that the set of crawled pages is fixed (i.e., $\mathcal{C}(t)$ is static, so we drop the dependence on t), and that each page $p \in \mathcal{C}$ exhibits a stationary stochastic pattern of content changes over time. Freshness maximization divides into three relatively distinct sub-problems:

- **Model estimation.** Construct a model for the temporal behavior of each page $p \in \mathcal{C}$.
- **Resource allocation.** Given a maximum crawl rate r, assign to each page $p \in \mathcal{C}$ a *revisitation frequency* $r(p)$ such that $\sum_{p \in \mathcal{C}} r(p) = r$.
- **Scheduling.** Produce a crawl order that adheres to the target revisitation frequencies as closely as possible.

With model estimation, the idea is to estimate the temporal behavior of p, given samples of the content of p or pages related to p. Cho and García-Molina [42] focused on how to deal with samples of p that are not evenly spaced in time, which may be the case if the samples have been gathered by the crawler itself in the past, while operating

under a non-uniform scheduling regime. Barbosa et al. [15] considered how to use content-based features of a *single* past sample of p to infer something about its temporal behavior. Cho and Ntoulas [44] and Tan et al. [113] focused on how to infer the behavior of p from the behavior of related pages — pages on the same web site, or pages with similar content, link structure, or other features.

We now turn to scheduling. Coffman et al. [46] pursued randomized scheduling policies, where at each step page p is selected with probability $r(p)/r$, independent of the past schedule. Wolf et al. [115] formulated the crawl scheduling problem in terms of network flow, for which prior algorithms exist. Cho and García-Molina [41] studied a special case of the scheduling problem in which pages have the same target revisitation frequency $r(p)$. All three works concluded that it is best to space apart downloads of each page p uniformly in time, or as close to uniformly as possible.

Resource allocation is generally viewed as the central aspect of freshness maximization. We divide work on resource allocation into two categories, according to the freshness model adopted:

5.1.1 Binary Freshness Model

In the *binary* freshness model, also known as *obsolescence*, $f(p,t) \in \{0,1\}$. Specifically,

$$f(p,t) = \begin{cases} 1 & \text{if the cached copy of } p \text{ is identical}^3 \text{ to the live copy} \\ 0 & \text{otherwise} \end{cases}.$$

Under the binary freshness model, if $f(p,t) = 1$ then p is said to be "fresh," otherwise it is termed "stale." Although simplistic, a great deal of useful intuition has been derived via this model.

The first to study the freshness maximization problem were Coffman et al. [46], who postulated a Poisson model of web page change. Specifically, a page undergoes discrete change events, which cause the copy

[3] It is common to replace the stipulation "identical" with "near-identical," and ignore minor changes like counters and timestamps. Some of the techniques surveyed in Section 6.1 can be used to classify near-identical web page snapshots.

cached by the crawler to become stale. For each page p, the occurrence of change events is governed by a Poisson process with rate parameter $\lambda(p)$, which means that changes occur randomly and independently, with an average rate of $\lambda(p)$ changes per time unit.

A key observation by Coffman et al. [46] was that in the case of uniform page weights (i.e., all $w(p)$ values are equal), the appealing idea of setting revisitation frequencies in proportion to page change rates, i.e., $r(p) \propto \lambda(p)$ (called *proportional* resource allocation), can be suboptimal. Coffman et al. [46] also provided a closed-form optimal solution for the case in which page weights are proportional to change rates (i.e., $w(p) \propto \lambda(p)$), along with a hint for how one might approach the general case.

Cho and García-Molina [41] continued the work of Coffman et al. [46], and derived a famously counterintuitive result: In the uniform weights case, a *uniform* resource allocation policy, in which $r(p) = r/|\mathcal{C}|$ for all p, achieves higher average binary freshness than proportional allocation. The superiority of the uniform policy to the proportional one holds under any distribution of change rates ($\lambda(p)$ values).

The optimal resource allocation policy for binary freshness, also given by Cho and García-Molina [41], exhibits the following intriguing property: Pages with a very fast rate of change (i.e., $\lambda(p)$ very high relative to $r/|\mathcal{C}|$) ought never to be revised by the crawler, i.e., $r(p) = 0$. The reason is as follows: A page p_1 that changes once per second, and is revisited once per second by the crawler, is on average only half synchronized ($f(p_1) = 0.5$). On the other hand, a page p_2 that changes once per day, and is revisited once per hour by the crawler, has much better average freshness ($f(p_2) = 24/25$ under randomized scheduling, according to the formula given by Cho and García-Molina [41]). The crawling resources required to keep one fast-changing page like p_1 weakly synchronized can be put to better use keeping several slow-changing pages like p_2 tightly synchronized, assuming equal page weights. Hence, in terms of average binary freshness, it is best for the crawler to "give up on" fast-changing pages, and put its energy into synchronizing moderate- and slow-changing ones. This resource allocation tactic is analogous to *advanced triage* in the field of medicine [3].

The discussion so far has focused on a Poisson page change model in which the times at which page changes occur are statistically independent. Under such a model, the crawler cannot time its visits to coincide with page change events. The following approaches relax the independence assumption.

Wolf et al. [115] studied incremental crawling under a *quasi-deterministic* page change model, in which page change events are non-uniform in time, and the distribution of likely change times is known a priori. (This work also introduced a search-centric page weighting scheme, under the terminology *embarrassment level*. The embarrassment-based scheme sets $w(p) \propto c(p)$, where $c(p)$ denotes the probability that a user will click on p after issuing a search query, as estimated from historical search engine usage logs. The aim is to revisit frequently clicked pages preferentially, thereby minimizing "embarrassing" incidents in which a search result contains a stale page.)

The WebFountain technique by Edwards et al. [56] does not assume any particular page evolution model a priori. Instead, it categorizes pages adaptively into one of n *change rate buckets* based on recently observed change rates (this procedure replaces explicit model estimation). Bucket membership yields a working estimate of a page's present change rate, which is in turn used to perform resource allocation and scheduling.

5.1.2 Continuous Freshness Models

In a real crawling scenario, some pages may be "fresher" than others. While there is no consensus about the best way to measure freshness, several non-binary freshness models have been proposed.

Cho and García-Molina [41] introduced a temporal freshness metric, in which $f(p,t) \propto -age(p,t)$, where

$$age(p,t) = \begin{cases} 0 & \text{if the cached copy of } p \text{ is identical to the live copy} \\ a & \text{otherwise} \end{cases},$$

where a denotes the amount of time the copies have differed. The rationale for this metric is that the longer a cached page remains unsynchronized with the live copy, the more their content tends to drift apart.

The optimal resource allocation policy under this age-based freshness metric, assuming a Poisson model of page change, is given by Cho and García-Molina [41]. Unlike in the binary freshness case, there is no "advanced triage" effect — the revisitation frequency $r(p)$ increases monotonically with the page change rate $\lambda(p)$. Since age increases without bound, the crawler cannot afford to "give up on" any page.

Olston and Pandey [99] introduced an approach in which, rather than relying on time as a proxy for degree of change, the idea is to measure changes in page content directly. A content-based freshness framework is proposed, which constitutes a generalization of binary freshness: A page is divided into a set of *content fragments*[4] $f_1, f_2, \ldots, f_n$, each with a corresponding weight $w(f_i)$ that captures the fragment's importance and/or relevance. Freshness is measured as the (weighted) fraction of fragments in common between the cached and live page snapshots, using the well-known Jaccard set similarity measure.

Under a content-based freshness model, the goal is to minimize the amount of incorrect content in the crawler's cache, averaged over time. To succeed in this goal, Olston and Pandey [99] argued that in addition to characterizing the frequency with which pages change, it is necessary to characterize the *longevity* of newly updated page content. Long-lived content (e.g., today's blog entry, which will remain in the blog indefinitely) is more valuable to crawl than ephemeral content (e.g., today's "quote of the day," which will be overwritten tomorrow), because it stays fresh in the cache for a longer period of time. The optimal resource allocation policy for content-based freshness derived by Olston and Pandey [99] differentiates between long-lived and ephemeral content, in addition to differentiating between frequently and infrequently changing pages.

In separate work, Pandey and Olston [103] proposed a search-centric method of assigning weights to individual content changes, based on the degree to which a change is expected to impact search ranking. The rationale is that even if a page undergoes periodic changes, and

[4] Fragments can be determined in a number of ways, e.g., using logical or visual document structure, or using *shingles* [28]).

the new content supplied by the changes is long-lived, if the search engine's treatment of the page is unaffected by these changes, there is no need for the crawler to revisit it.

5.2 Capturing Updates

For some crawling applications, maximizing average freshness of cached pages is not the right objective. Instead, the aim is to capture as many individual content updates as possible. Applications that need to capture updates include historical archival and temporal data mining, e.g., time-series analysis of stock prices.

The two scenarios (maximizing freshness versus capturing updates) lead to very different page revisitation policies. As we saw in Section 5.1, in the freshness maximization scenario, when resources are scarce the crawler should ignore pages that frequently replace their content, and concentrate on maintaining synchronization with pages that supply more persistent content. In contrast, in the update capture scenario, pages that frequently replace their content offer the highest density of events to be captured, and also the highest urgency to capture them before they are lost.

Early update capture systems, e.g., CONQUER [86], focused on user-facing query languages, algorithms to difference page snapshots and extract relevant tidbits (e.g., updated stock price), and query processing techniques. Later work considered the problem of when to revisit each page to check for updates.

In work by Pandey et al. [105], the objective was to maximize the (weighted) number of updates captured. A suitable resource allocation algorithm was given, which was shown empirically to outperform both uniform and proportional resource allocation.

In subsequent work, Pandey et al. [102] added a *timeliness* dimension, to represent the sensitivity of the application to delays in capturing new information from the web. (For example, historical archival has a low timeliness requirement compared with real-time stock market analysis.) The key insight was that revisitation of pages whose updates do not replace old content can be postponed to a degree permitted by the application's timeliness requirement, yielding a higher total

amount of information captured. A timeliness-sensitive resource allocation algorithm was given, along with a formal performance guarantee.

5.3 Efficient Large-Scale Implementation

One key aspect of page revisitation policy is change model estimation. For certain pages (e.g., newly discovered pages on important sites), model estimation is time-critical; but for the vast majority of pages (e.g., unimportant, well-understood, or infrequently crawled ones) it can be performed lazily, in periodic offline batches. Since most change models deal with each page in isolation, this process is trivially parallelizable. For the time-critical pages, the relevant data tend to be small and amenable to caching: Most techniques require a compact signature (e.g., a few shingle hashes) for two to five of the most recent page snapshots. The search-centric approach of Pandey and Olston [103] requires more detailed information from pages, and incorporates model estimation into the process of re-building the search index.

After model estimation, the other two aspects of revisitation policy are resource allocation and scheduling. While there is quite a bit of work on using global optimization algorithms to assign revisitation schedules to individual pages, it is not immediately clear how to incorporate such algorithms into a large-scale crawler. One scalable approach is to group pages into *buckets* according to desired revisitation frequency (as determined by resource allocation, see below), and cycle through each bucket such that the time to complete one cycle matches the bucket's revisitation frequency target. This approach ensures roughly equal spacing of revisitations of a given page, which has been found to be a good rule of thumb for scheduling, as mentioned in Section 5.1. It also yields an obvious disk-based implementation: Each bucket is a FIFO queue, where URLs removed from the head are automatically appended to the tail; crawler machines pull from the queues according to a weighted randomized policy constructed to achieve the target revisitation rates in expectation.

In the bucket-oriented approach, pages that are identical (or near-identical) from the point of view of the adopted model are placed into the same bucket. For example, in the basic freshness maximization

formulation under the binary freshness model (Section 5.1.1), pages can be bucketized by change frequency. Assignment of a revisitation frequency target to each bucket is done by the resource allocation procedure. The WebFountain crawler by Edwards et al. [56] uses change frequency buckets and computes bucket revisitation frequency targets periodically using a nonlinear problem (NLP) solver. Since the NLP is formulated at the bucket granularity (not over individual pages), the computational complexity is kept in check. Most other resource allocation strategies can also be cast into the bucket-oriented NLP framework. To take a simple example, an appealing variant of the *uniform* resource allocation is to direct more resources toward important pages; in this case, one can bucketize pages by importance, and apply a simple NLP over bucket sizes and importance weights to determine the revisitation rate of each bucket.

6

Avoiding Problematic and Undesirable Content

This section discusses detection and avoidance of content that is redundant, wasteful or misleading.

6.1 Redundant Content

As discussed in Section 3.2.1, there is a prevalence of duplicate and near-duplicate content on the web. Shingling [28] is a standard way to identify near-duplicate pages, but shingling is performed on web page content, and thus requires these pages to have been crawled. As such, it does not help to reduce the load on the crawler; however, it can be used to limit and diversify the set of search results presented to a user.

Some duplication stems from the fact that many web sites allow multiple URLs to refer to the same content, or content that is identical modulo ever-changing elements such as rotating banner ads, evolving comments by readers, and timestamps. Schonfeld et al. proposed the "duplicate URL with similar text" (DUST) algorithm [12] to detect this form of aliasing, and to infer rules for normalizing URLs into a canonical form. Dasgupta et al. [49] generalized DUST by introducing a learning algorithm that can generate rules containing regular

expressions, experimentally tripling the number of duplicate URLs that can be detected. Agarwal et al. attempted to bound the computational complexity of learning rules using sampling [4]. Rules inferred using these algorithms can be used by a web crawler to normalize URLs after extracting them from downloaded pages and before passing them through the duplicate URL eliminator (Section 2.3.2) and into the frontier.

Another source of duplication is *mirroring* [18, 19, 52]: Providing all or parts of the same web site on different hosts. Mirrored web sites in turn can be divided into two groups: Sites that are mirrored by the same organization (for example by having one web server serving multiple domains with the same content, or having multiple web servers provide synchronized content), and content that is mirrored by multiple organizations (for example, schools providing Unix `man` pages on the web, or web sites republishing Wikipedia content, often somewhat reformatted). Detecting mirrored content differs from detecting DUST in two ways: On the one hand, with mirroring the duplication occurs across multiple sites, so mirror detection algorithms have to consider the entire corpus. On the other hand, entire trees of URLs are mirrored, so detection algorithms can use URL trees (suitably compacted e.g., through hashing) as a feature to detect mirror candidates, and then compare the content of candidate subtrees (for example via shingling).

6.2 Crawler Traps

Content duplication inflates the web corpus without adding much information. Another phenomenon that inflates the corpus without adding utility is *crawler traps*: Web sites that populate a large, possibly infinite URL space on that site with mechanically generated content. Some crawler traps are non-malicious, for example web-based calendaring tools that generate a page for every month of every year, with a hyperlink from each month to the next (and previous) month, thereby forming an unbounded chain of dynamically generated pages. Other crawler traps are malicious, often set up by "spammers" to inject large amounts of their content into a search engine, in the hope of having their content show up high in search result pages or providing

many hyperlinks to their "landing page," thus biasing link-based ranking algorithms such as PageRank. There are many known heuristics for identifying and avoiding spam pages or sites, see Section 6.3. Not much research has been published on algorithms or heuristics for detecting crawler traps directly. The IRLbot crawler [84] utilizes a heuristic called "Budget Enforcement with Anti-Spam Tactics" (BEAST), which assigns a budget to each web site and prioritizes URLs from each web site based on the site's remaining budget combined with the domain's reputation.

6.3 Web Spam

Web spam may be defined as "web pages that are crafted for the sole purpose of increasing the ranking of these or some affiliated pages, without improving the utility to the viewer" [97]. Web spam is motivated by the monetary value of achieving a prominent position in search-engine result pages. There is a multi-billion dollar industry devoted to *search engine optimization* (SEO), most of it being legitimate but some of it misleading. Web spam can be broadly classified into three categories [69]: *Keyword stuffing*, populating pages with highly searched or highly monetizable terms; *link spam*, creating cliques of tightly interlinked web pages with the goal of biasing link-based ranking algorithms such as PageRank [101]; and *cloaking*, serving substantially different content to web crawlers than to human visitors (to get search referrals for queries on a topic not covered by the page).

Over the past few years, many heuristics have been proposed to identify spam web pages and sites, see for example the series of AIRweb workshops [76]. The problem of identifying web spam can be framed as a classification problem, and there are many well-known classification approaches (e.g., decision trees, Bayesian classifiers, support vector machines). The main challenge is to identify features that are predictive of web spam and can thus be used as inputs to the classifier. Many such features have been proposed, including hyperlink features [16, 17, 50, 116], term and phrase frequency [97], DNS lookup statistics [59], and HTML markup structure [114]. Combined, these features tend to be quite effective, although web spam detection is a

constant arms race, with both spammers and search engines evolving their techniques in response to each other's actions.

Spam detection heuristics are used during the ranking phase of search, but they can also be used during the corpus selection phase (when deciding which pages to index) and crawling phase (when deciding what crawl priority to assign to web pages). Naturally, it is easier to avoid crawling spam content in a continuous or iterative crawling setting, where historical information about domains, sites, and individual pages is available.[1]

6.4 Cloaked Content

Cloaking refers to the practice of serving different content to web crawlers than to human viewers of a site [73]. Not all cloaking is malicious: For example, many web sites with interactive content rely heavily on JavaScript, but most web crawlers do not execute JavaScript, so it is reasonable for such a site to deliver alternative, script-free versions of its pages to a search engine's crawler to enable the engine to index and expose the content.

Web sites distinguish mechanical crawlers from human visitors either based on their `User-Agent` field (an HTTP header that is used to distinguish different web browsers, and by convention is used by crawlers to identify themselves), or by the crawler's IP address (the SEO community maintains lists of the `User-Agent` fields and IP addresses of major crawlers). One way for search engines to detect that a web server employs cloaking is by supplying a different `User-Agent` field [117]. Another approach is to probe the server from IP addresses not known to the SEO community (for example by enlisting the search engine's user base).

A variant of cloaking is called *redirection spam*. A web server utilizing redirection spam serves the same content both to crawlers and to human-facing browser software (and hence, the aforementioned detection techniques will not detect it); however, the content will cause a

[1] There is of course the possibility of spammers acquiring a site with a good history and converting it to spam, but historical reputation-based approaches at least "raise the bar" for spamming.

browser to immediately load a new page presenting different content. Redirection spam is facilitated either through the HTML `META REFRESH` tag (whose presence is easy to detect), or via JavaScript, which most browsers execute but most crawlers do not. Chellapilla and Maykov [36] conducted a study of pages employing JavaScript redirection spam, and found that about half of these pages used JavaScript's `eval` statement to obfuscate the URLs to which they redirect, or even parts of the script itself. This practice makes static detection of the redirection target (or even the fact that redirection is occurring) very difficult. Chellapilla and Maykov argued for the use of lightweight JavaScript parsers and execution engines in the crawling/indexing pipeline to evaluate scripts (in a time-bounded fashion, since scripts may not terminate) to determine whether redirection occurs.

7

Deep Web Crawling

Some content is accessible only by filling in HTML forms, and cannot be reached via conventional crawlers that just follow hyperlinks.[1] Crawlers that automatically fill in forms to reach the content behind them are called *hidden web* or *deep web* crawlers.

The deep web crawling problem is closely related to the problem known as *federated search* or *distributed information retrieval* [30], in which a mediator forwards user queries to multiple searchable collections, and combines the results before presenting them to the user. The crawling approach can be thought of as an *eager* alternative, in which content is collected in advance and organized in a unified index prior to retrieval. Also, deep web crawling considers structured query interfaces in addition to unstructured "search boxes," as we shall see.

7.1 Types of Deep Web Sites

Figure 7.1 presents a simple taxonomy of deep web sites. Content is either unstructured (e.g., free-form text) or structured (e.g., data

[1] Not all content behind form interfaces is unreachable via hyperlinks — some content is reachable in both ways [35].

	Unstructured Content	Structured Content
Unstructured query interface	News archive (simple search)	Product review site
Structured query interface	News archive (advanced search)	Online bookstore

Fig. 7.1 Deep web taxonomy.

records with typed fields). Similarly, the form interface used to query the content is either unstructured (i.e., a single query box that accepts a free-form query string) or structured (i.e., multiple query boxes that pertain to different aspects of the content).[2]

A news archive contains content that is primarily unstructured (of course, some structure is present, e.g., title, date, author). In conjunction with a simple textual search interface, a news archive constitutes an example of an unstructured-content/unstructured-query deep web site. A more advanced query interface might permit structured restrictions on attributes that are extractable from the unstructured content, such as language, geographical references, and media type, yielding an unstructured-content/structured-query instance.

A product review site has relatively structured content (product names, numerical reviews, reviewer reputation, and prices, in addition to free-form textual comments), but for ease of use typically offers an unstructured search interface. Lastly, an online bookstore offers structured content (title, author, genre, publisher, price) coupled with a structured query interface (typically a subset of the content attributes, e.g., title, author and genre).

For simplicity most work focuses on either the upper-left quadrant (which we henceforth call the *unstructured case*), or the lower-right quadrant (*structured case*).

[2] The unstructured versus structured dichotomy is really a continuum, but for simplicity we present it as a binary property.

7.2 Problem Overview

Deep web crawling has three steps:

(1) **Locate deep web content sources.** A human or crawler must identify web sites containing form interfaces that lead to deep web content. Barbosa and Freire [14] discussed the design of a scoped crawler for this purpose.
(2) **Select relevant sources.** For a scoped deep web crawling task (e.g., crawling medical articles), one must select a relevant subset of the available content sources. In the unstructured case this problem is known as *database* or *resource selection* [32, 66]. The first step in resource selection is to model the content available at a particular deep web site, e.g., using *query-based sampling* [31].
(3) **Extract underlying content.** Finally, a crawler must extract the content lying behind the form interfaces of the selected content sources.

For major search engines, Step 1 is almost trivial, since they already possess a comprehensive crawl of the surface web, which is likely to include a plethora of deep web query pages. Steps 2 and 3 pose significant challenges. Step 2 (source selection) has been studied extensively in the distributed information retrieval context [30], and little has been done that specifically pertains to crawling. Step 3 (content extraction) is the core problem in deep web crawling; the rest of this chapter covers the (little) work that has been done on this topic.

7.3 Content Extraction

The main approach to extracting content from a deep web site proceeds in four steps (the first two steps apply only to the structured case):

(1) Select a subset of form elements to populate,[3] or perhaps multiple such subsets. This is largely an open problem, where the goals are to: (a) avoid form elements that merely

[3] The remaining elements can remain blank, or be populated with a wildcard expression when applicable.

affect the presentation of results (e.g., sorting by price versus popularity); and (b) avoid including correlated elements, which artificially increase the dimensionality of the search space [88].

(2) If possible, decipher the role of each of the targeted form elements (e.g., book author versus publication date), or at least understand their domains (proper nouns versus dates). Raghavan and García-Molina [109] and several subsequent papers studied this difficult problem.

(3) Create an initial database of valid data values (e.g., "Ernest Hemingway" and 1940 in the structured case; English words in the unstructured case). Some sources of this information include [109]: (a) a human administrator; (b) non-deep-web online content, e.g., a dictionary (for unstructured keywords) or someone's list of favorite authors; (c) drop-down menus for populating form elements (e.g., a drop-down list of publishers).

(4) Use the database to issue queries to the deep web site (e.g., publisher = "Scribner"), parse the result and extract new data values to insert into the database (e.g., author = "Ernest Hemingway"), and repeat.

We elaborate on Step 4, which has been studied under (variations of) the following model of deep web content and queries [98, 117]:

A deep web site contains one or more *content items*, which are either unstructured documents or structured data records. A content item contains individual *data values*, which are text terms in the unstructured case, or data record elements like author names and dates in the structured case. Data values and content values are related via a bipartite graph, depicted in Figures 7.2 (unstructured case) and 7.3 (structured case).

A query consists of a single data value[4] V submitted to the form interface, which retrieves the set of content items directly connected

[4] It is assumed that any data value can form the basis of a query, even though this is not always the case in practice (e.g., a bookstore may not permit querying by publisher). Also, multi-value queries are not considered.

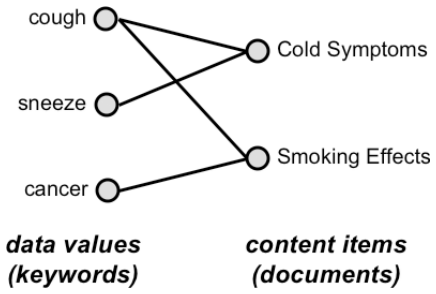

Fig. 7.2 Deep web content model (unstructured content).

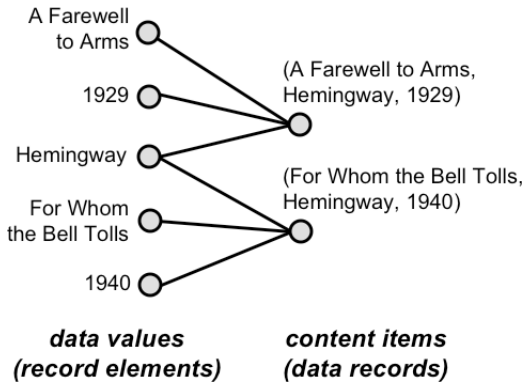

Fig. 7.3 Deep web content model (structured content).

to V via edges in the graph, called V's *result set*. Each query incurs some cost to the crawler, typically dominated by the overhead of downloading and processing each member of the result set, and hence modeled as being linearly proportional to result cardinality.

Under this model, the deep web crawling problem can be cast as a weighted set-cover problem: Select a minimum-cost subset of data values that cover all content items. Unfortunately, unlike in the usual set-cover scenario, in our case the graph is only partially known at the outset, and must be uncovered progressively during the course of the crawl. Hence, adaptive graph traversal strategies are required.

A simple greedy traversal strategy was proposed by Barbosa and Freire [13] for the unstructured case: At each step the crawler issues as a query the highest-frequency keyword that has not yet been issued,

where keyword frequency is estimated by counting occurrences in documents retrieved so far. In the bipartite graph formulation, this strategy is equivalent to selecting the data value vertex of highest degree, according to the set of edges uncovered so far.

A similar strategy was proposed by Wu et al. [117] for the structured case, along with a refinement in which the crawler bypasses data values that are highly correlated with ones that have already been selected, in the sense that they connect to highly overlapping sets of content items.

Ntoulas et al. [98] proposed statistical models for estimating the number of previously unseen content items that a particular data value is likely to cover, focusing on the unstructured case.

Google's deep web crawler [88] uses techniques similar to the ones described above, but adapted to extract a small amount of content from a large number (millions) of sites, rather than aiming for extensive coverage of a handful of sites.

8

Future Directions

As this survey indicates, crawling is a well-studied problem. However, there are at least as many open questions as there are resolved ones. Even in the material we have covered, the reader has likely noticed many open issues, including:

- **Parameter tuning.** Many of the crawl ordering policies rely on carefully tuned parameters, with little insight or science into how best to choose the parameters. For example, what is the optimal level of greediness for a scoped crawler (Section 4.2)?
- **Retiring unwanted pages.** Given finite storage capacity, in practice crawlers discard or retire low-quality and spam pages from their collections, to make room for superior pages. However, we are not aware of any literature that explicitly studies retirement policies. There is also the issue of how much metadata to retain about retired pages, to avoid accidentally rediscovering them, and to help assess the quality of related pages (e.g., pages on the same site, or pages linking to or linked from the retired page).

- **Holistic crawl ordering.** Whereas much attention has been paid to various crawl ordering sub-problems (e.g., prioritizing the crawl order of new pages, refreshing content from old pages, revisiting pages to discover new links), there is little work on how to integrate the disparate approaches into a unified strategy.
- **Integration of theory and systems work.** There is also little work that reconciles the theoretical work on crawl ordering (Sections 3–5) with the pragmatic work on large-scale crawling architectures (Section 2). For example, disk-based URL queues make it infeasible to re-order URLs frequently, which may impede exact implementation of some of the crawl ordering policies. While we have hypothesized scalable implementations of some of the policies (Sections 4.3 and 5.3), a more comprehensive and empirical study of this topic is needed.
- **Deep web.** Clearly, the science and practice of deep web crawling (Section 7) is in its infancy.

There are also several nearly untouched directions:

- **Vertical crawling.** Some sites may be considered important enough to merit crawler specialization, e.g., eBay's auction listings or Amazon's product listings. Also, as the web matures, certain content dissemination structures become relatively standardized, e.g., news, blogs, tutorials, and discussion forums. In these settings the unit of content is not always a web page; instead, multiple content units are sometimes appended to the same page, or a single content unit may span multiple pages (connected via "next page" and "previous page" links). Also, many links lead to redundant content (e.g., shortcut to featured story, or items for sale by a particular user). A crawler that understands these formats can crawl them more efficiently and effectively than a general-purpose crawler. There has been some work on crawling discussion forums [119], but little else. Vertical crawling is not the same as scoped crawling (Section 4.2): Scoped

crawling focuses on collecting semantically coherent content from many sites, whereas vertical crawling exploits syntactic patterns on particular sites.

- **Crawling scripts.** Increasingly, web sites employ scripts (e.g., JavaScript, AJAX) to generate content and links on the fly. Almost no attention has been paid to whether or how to crawl these sites. We are aware of only one piece of preliminary work, by Duda et al. [55].
- **Personalized content.** Web sites often customize their content to individual users, e.g., Amazon gives personalized recommendations based on a user's browsing and purchasing patterns. It is not clear how crawlers should treat such sites, e.g., emulating a generic user versus attempting to specialize the crawl based on different user profiles. A search engine that aims to personalize search results may wish to push some degree of personalization into the crawler.
- **Collaboration between content providers and crawlers.** As discussed in Section 1, crawling is a pull mechanism for discovering and acquiring content. Many sites now publish Sitemaps and RSS feeds, which enable a push-oriented approach. Modern commercial crawlers employ a hybrid of push and pull, but there is little academic study of this practice and the issues involved. Schonfeld and Shivakumar [110] examined tradeoffs between reliance on Sitemaps for content discovery and the classical pull approach.

References

[1] S. Abiteboul, M. Preda, and G. Cobena, "Adaptive on-line page importance computation," in *Proceedings of the 12th International World Wide Web Conference*, 2003.

[2] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas, "The web changes everything: Understanding the dynamics of web content," in *Proceedings of the 2nd International Conference on Web Search and Data Mining*, 2009.

[3] Advanced Triage (medical term), http://en.wikipedia.org/wiki/Triage#Advanced_triage.

[4] A. Agarwal, H. S. Koppula, K. P. Leela, K. P. Chitrapura, S. Garg, P. K. GM, C. Haty, A. Roy, and A. Sasturkar, "URL normalization for de-duplication of web pages," in *Proceedings of the 18th Conference on Information and Knowledge Management*, 2009.

[5] C. C. Aggarwal, F. Al-Garawi, and P. S. Yu, "Intelligent crawling on the world wide web with arbitrary predicates," in *Proceedings of the 10th International World Wide Web Conference*, 2001.

[6] D. Ahlers and S. Boll, "Adaptive geospatially focused crawling," in *Proceedings of the 18th Conference on Information and Knowledge Management*, 2009.

[7] Attributor. http://www.attributor.com.

[8] R. Baeza-Yates and C. Castillo, "Crawling the infinite web," *Journal of Web Engineering*, vol. 6, no. 1, pp. 49–72, 2007.

[9] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez, "Crawling a country: Better strategies than breadth-first for web page ordering," in *Proceedings of the 14th International World Wide Web Conference*, 2005.

[10] B. Bamba, L. Liu, J. Caverlee, V. Padliya, M. Srivatsa, T. Bansal, M. Palekar, J. Patrao, S. Li, and A. Singh, "DSphere: A source-centric approach to

crawling, indexing and searching the world wide web," in *Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[11] Z. Bar-Yossef and M. Gurevich, "Random sampling from a search engine's index," in *Proceedings of the 15th International World Wide Web Conference*, 2006.

[12] Z. Bar-Yossef, I. Keidar, and U. Schonfeld, "Do not crawl in the DUST: Different URLs with similar text," in *Proceedings of the 16th International World Wide Web Conference*, 2007.

[13] L. Barbosa and J. Freire, "Siphoning hidden-web data through keyword-based interfaces," in *Proceedings of the 19th Brazilian Symposium on Databases SBBD*, 2004.

[14] L. Barbosa and J. Freire, "An adaptive crawler for locating hidden-web entry points," in *Proceedings of the 16th International World Wide Web Conference*, 2007.

[15] L. Barbosa, A. C. Salgado, F. de Carvalho, J. Robin, and J. Freire, "Looking at both the present and the past to efficiently update replicas of web content," in *Proceedings of the ACM International Workshop on Web Information and Data Management*, 2005.

[16] L. Becchetti, C. Castillo, D. Donato, S. Leonardi, and R. Baeza-Yates, "Link-based characterization and detection of web spam," in *Proceedings of the 2nd International Workshop on Adversarial Information Retrieval on the Web*, 2006.

[17] A. Benczúr, K. Csalogány, T. Sarlós, and M. Uher, "Spamrank — fully automatic link spam detection," in *Proceedings of the 1st International Workshop on Adversarial Information Retrieval on the Web*, 2005.

[18] K. Bharat and A. Broder, "Mirror, mirror on the web: A study of host pairs with replicated content," in *Proceedings of the 8th International World Wide Web Conference*, 1999.

[19] K. Bharat, A. Broder, J. Dean, and M. Henzinger, "A comparison of techniques to find mirrored hosts on the WWW," *Journal of the American Society for Information Science*, vol. 51, no. 12, pp. 1114–1122, 2000.

[20] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[21] P. Boldi, B. Codenotti, , M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Software — Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[22] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Structural properties of the African web," in *Poster Proceedings of the 11th International World Wide Web Conference*, 2002.

[23] P. Boldi, M. Santini, and S. Vigna, "Paradoxical effects in pagerank incremental computations," *Internet Mathematics*, vol. 2, no. 3, pp. 387–404, 2005.

[24] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz, "The Harvest information discovery and access system," in *Proceedings of the 2nd International World Wide Web Conference*, 1994.

[25] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the 7th International World Wide Web Conference*, 1998.

[26] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," in *Proceedings of the 9th International World Wide Web Conference*, 2000.

[27] A. Broder, M. Najork, and J. Wiener, "Efficient URL caching for World Wide Web crawling," in *Proceedings of the 12th International World Wide Web Conference*, 2003.

[28] A. Z. Broder, S. C. Glassman, and M. S. Manasse, "Syntactic clustering of the web," in *Proceedings of the 6th International World Wide Web Conference*, 1997.

[29] M. Burner, "Crawling towards eternity: Building an archive of the world wide web," *Web Techniques Magazine*, vol. 2, no. 5, pp. 37–40, 1997.

[30] J. Callan, "Distributed information retrieval," in *Advances in Information Retrieval*, (W. B. Croft, ed.), pp. 127–150, Kluwer Academic Publishers, 2000.

[31] J. Callan and M. Connell, "Query-based sampling of text databases," *ACM Transactions on Information Systems*, vol. 19, no. 2, pp. 97–130, 2001.

[32] J. P. Callan, Z. Lu, and W. B. Croft, "Searching distributed collections with inference networks," in *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1995.

[33] S. Chakrabarti, B. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, and J. Kleinberg, "Automatic resource compilation by analyzing hyperlink structure and associated text," in *Proceedings of the 7th International World Wide Web Conference*, 1998.

[34] S. Chakrabarti, M. van den Berg, and B. Dom, "Focused crawling: A new approach to topic-specific web resource discovery," in *Proceedings of the 8th International World Wide Web Conference*, 1999.

[35] K. C.-C. Chang, B. He, C. Li, M. Patel, and Z. Zhang, "Structured databases on the web: Observations and implications," *ACM SIGMOD Record*, vol. 33, no. 3, pp. 61–70, 2004.

[36] K. Chellapilla and A. Maykov, "A taxonomy of JavaScript redirection spam," in *Proceedings of the 16th International World Wide Web Conference*, 2007.

[37] H. Chen, M. Ramsey, and C. Yang, "A smart itsy bitsy spider for the web," *Journal of the American Society for Information Science*, vol. 49, no. 7, pp. 604–618, 1998.

[38] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar, "Link evolution: Analysis and algorithms," *Internet Mathematics*, vol. 1, no. 3, pp. 277–304, 2003.

[39] J. Cho and H. García-Molina, "The evolution of the web and implications for an incremental crawler," in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[40] J. Cho and H. García-Molina, "Parallel crawlers," in *Proceedings of the 11th International World Wide Web Conference*, 2002.

[41] J. Cho and H. García-Molina, "Effective page refresh policies for web crawlers," *ACM Transactions on Database Systems*, vol. 28, no. 4, pp. 390–426, 2003.

[42] J. Cho and H. García-Molina, "Estimating frequency of change," *ACM Transactions on Internet Technology*, vol. 3, no. 3, pp. 256–290, 2003.
[43] J. Cho, J. García-Molina, and L. Page, "Efficient crawling through URL ordering," in *Proceedings of the 7th International World Wide Web Conference*, 1998.
[44] J. Cho and A. Ntoulas, "Effective change detection using sampling," in *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.
[45] J. Cho and U. Schonfeld, "RankMass crawler: A crawler with high personalized PageRank coverage guarantee," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007.
[46] E. G. Coffman, Z. Liu, and R. R. Weber, "Optimal robot scheduling for web search engines," *Journal of Scheduling*, vol. 1, no. 1, 1998.
[47] CrawlTrack, "List of spiders and crawlers," http://www.crawltrack.net/crawlerlist.php.
[48] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins, "The discoverability of the web," in *Proceedings of the 16th International World Wide Web Conference*, 2007.
[49] A. Dasgupta, R. Kumar, and A. Sasturkar, "De-duping URLs via rewrite rules," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.
[50] B. Davison, "Recognizing nepotistic links on the web," in *Proceedings of the AAAI-2000 Workshop on Artificial Intelligence for Web Search*, 2000.
[51] G. T. de Assis, A. H. F. Laender, M. A. Gonçalves, and A. S. da Silva, "A genre-aware approach to focused crawling," *World Wide Web*, vol. 12, no. 3, pp. 285–319, 2009.
[52] J. Dean and M. Henzinger, "Finding related pages in the world wide web," in *Proceedings of the 8th International World Wide Web Conference*, 1999.
[53] P. DeBra and R. Post, "Information retrieval in the world wide web: Making client-based searching feasible," in *Proceedings of the 1st International World Wide Web Conference*, 1994.
[54] M. Diligenti, F. M. Coetzee, S. Lawrence, C. L. Giles, and M. Gori, "Focused crawling using context graphs," in *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.
[55] C. Duda, G. Frey, D. Kossmann, and C. Zhou, "AJAXSearch: Crawling, indexing and searching web 2.0 applications," in *Proceedings of the 34th International Conference on Very Large Data Bases*, 2008.
[56] J. Edwards, K. S. McCurley, and J. A. Tomlin, "An adaptive model for optimizing performance of an incremental web crawler," in *Proceedings of the 10th International World Wide Web Conference*, 2001.
[57] D. Eichmann, "The RBSE spider — Balancing effective search against web load," in *Proceedings of the 1st International World Wide Web Conference*, 1994.
[58] D. Fetterly, N. Craswell, and V. Vinay, "The impact of crawl policy on web search effectiveness," in *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2009.

[59] D. Fetterly, M. Manasse, and M. Najork, "Spam, damn spam, and statistics: Using statistical analysis to locate spam web pages," in *Proceedings of the 7th International Workshop on the Web and Databases*, 2004.

[60] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener, "A large-scale study of the evolution of web pages," in *Proceedings of the 12th International World Wide Web Conference*, 2003.

[61] R. Fielding, "Maintaining distributed hypertext infostructures: Welcome to MOMspider's web," in *Proceedings of the 1st International World Wide Web Conference*, 1994.

[62] A. S. Foundation, "Welcome to Nutch!," http://lucene.apache.org/nutch/.

[63] W. Gao, H. C. Lee, and Y. Miao, "Geographically focused collaborative crawling," in *Proceedings of the 15th International World Wide Web Conference*, 2006.

[64] GigaAlert, http://www.gigaalert.com.

[65] D. Gomes and M. J. Silva, "Characterizing a national community web," *ACM Transactions on Internet Technology*, vol. 5, no. 3, pp. 508–531, 2005.

[66] L. Gravano, H. García-Molina, and A. Tomasic, "The effectiveness of GlOSS for the text database discovery problem," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

[67] M. Gray, "Internet growth and statistics: Credits and background," http://www.mit.edu/people/mkgray/net/background.html.

[68] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien, "How to build a WebFountain: An architecture for very large-scale text analytics," *IBM Systems Journal*, vol. 43, no. 1, pp. 64–77, 2004.

[69] Z. Gyöngyi and H. García-Molina, "Web Spam Taxonomy," in *Proceedings of the 1st International Workshop on Adversarial Information Retrieval*, 2005.

[70] Y. Hafri and C. Djeraba, "High performance crawling system," in *Proceedings of the 6th ACM SIGMM International Workshop on Multimedia Information Retrieval*, 2004.

[71] M. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork, "Measuring index quality using random walks on the web," in *Proceedings of the 8th International World Wide Web Conference*, 1999.

[72] M. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork, "On near-uniform URL sampling," in *Proceedings of the 9th International World Wide Web Conference*, 2000.

[73] M. R. Henzinger, R. Motwani, and C. Silverstein, "Challenges in web search engines," *SIGIR Forum*, vol. 36, no. 2, pp. 11–22, 2002.

[74] M. Hersovici, M. Jacovi, Y. S. Maarek, D. Pelleg, M. Shtalhaim, and S. Ur, "The shark-search algorithm — An application: Tailored web site mapping," in *Proceedings of the 7th International World Wide Web Conference*, 1998.

[75] A. Heydon and M. Najork, "Mercator: A scalable, extensible web crawler," *World Wide Web*, vol. 2, no. 4, pp. 219–229, 1999.

[76] *International Workshop Series on Adversarial Information Retrieval on the Web*, 2005–.

[77] Internet Archive, http://archive.org/.

[78] Internet Archive, "Heritrix home page," http://crawler.archive.org/.

[79] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Transactions on Information Systems*, vol. 20, no. 4, pp. 422–446, 2002.

[80] J. Johnson, K. Tsioutsiouliklis, and C. L. Giles, "Evolving strategies for focused web crawling," in *Proceedings of the 20th International Conference on Machine Learning*, 2003.

[81] R. Khare, D. Cutting, K. Sitakar, and A. Rifkin, "Nutch: A flexible and scalable open-source web search engine," Technical Report, CommerceNet Labs, 2004.

[82] J. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM*, vol. 46, no. 5, pp. 604–632, 1999.

[83] M. Koster, "A standard for robot exclusion," http://www.robotstxt.org/orig.html, 1994.

[84] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 billion pages and beyond," in *Proceedings of the 17th International World Wide Web Conference*, 2008.

[85] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. C. Agarwal, "Characterizing web document change," in *Proceedings of the International Conference on Advances in Web-Age Information Management*, 2001.

[86] L. Liu, C. Pu, W. Tang, and W. Han, "CONQUER: A continual query system for update monitoring in the WWW," *International Journal of Computer Systems, Science and Engineering*, vol. 14, no. 2, 1999.

[87] B. T. Loo, O. Cooper, and S. Krishnamurthy, "Distributed web crawling over DHTs," UC Berkeley Technical Report CSD-04-1305, 2004.

[88] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's deep-web crawl," in *Proceedings of the 34th International Conference on Very Large Data Bases*, 2008.

[89] M. Mauldin, "Lycos: Design choices in an internet search service," *IEEE Expert*, vol. 12, no. 1, pp. 8–11, 1997.

[90] O. A. McBryan, "GENVL and WWWW: Tools for taming the web," in *Proceedings of the 1st International World Wide Web Conference*, 1994.

[91] F. Menczer and R. K. Belew, "Adaptive retrieval agents: Internalizing local context and scaling up to the web," *Machine Learning*, vol. 39, pp. 203–242, 2000.

[92] F. Menczer, G. Pant, and P. Srinivasan, "Topical web crawlers: Evaluating adaptive algorithms," *ACM Transactions on Internet Technology*, vol. 4, no. 4, pp. 378–419, 2004.

[93] G. Mohr, M. Stack, I. Ranitovic, D. Avery, and M. Kimpton, "An introduction to Heritrix, an open source archival quality web crawler," in *Proceedings of the 4th International Web Archiving Workshop*, 2004.

[94] M. Najork and A. Heydon, "High-performance web crawling," Technical report, Compaq SRC Research Report 173, 2001.

[95] M. Najork and J. L. Wiener, "Breadth-first search crawling yields high-quality pages," in *Proceedings of the 10th International World Wide Web Conference*, 2001.

[96] A. Ntoulas, J. Cho, and C. Olston, "What's new on the web? The evolution of the web from a search engine perspective," in *Proceedings of the 13th International World Wide Web Conference*, 2004.

[97] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly, "Detecting spam web pages through content analysis," in *Proceedings of the 15th International World Wide Web Conference*, 2006.

[98] A. Ntoulas, P. Zerfos, and J. Cho, "Downloading textual hidden web content through keyword queries," in *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries*, 2005.

[99] C. Olston and S. Pandey, "Recrawl scheduling based on information longevity," in *Proceedings of the 17th International World Wide Web Conference*, 2008.

[100] V. J. Padliya and L. Liu, "Peercrawl: A decentralized peer-to-peer architecture for crawling the world wide web," Georgia Institute of Technology Technical Report, 2006.

[101] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Technical Report, Stanford University, 1998.

[102] S. Pandey, K. Dhamdhere, and C. Olston, "WIC: A general-purpose algorithm for monitoring web information sources," in *Proceedings of the 30th International Conference on Very Large Data Bases*, 2004.

[103] S. Pandey and C. Olston, "User-centric web crawling," in *Proceedings of the 14th International World Wide Web Conference*, 2005.

[104] S. Pandey and C. Olston, "Crawl ordering by search impact," in *Proceedings of the 1st International Conference on Web Search and Data Mining*, 2008.

[105] S. Pandey, K. Ramamritham, and S. Chakrabarti, "Monitoring the dynamic web to respond to continuous queries," in *Proceedings of the 12th International World Wide Web Conference*, 2003.

[106] G. Pant and P. Srinivasan, "Learning to crawl: Comparing classification schemes," *ACM Transactions on Information Systems*, vol. 23, no. 4, pp. 430–462, 2005.

[107] G. Pant and P. Srinivasan, "Link contexts in classifier-guided topical crawlers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 1, pp. 107–122, 2006.

[108] B. Pinkerton, "Finding what people want: Experiences with the WebCrawler," in *Proceedings of the 2nd International World Wide Web Conference*, 1994.

[109] S. Raghavan and H. García-Molina, "Crawling the hidden web," in *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[110] U. Schonfeld and N. Shivakumar, "Sitemaps: Above and beyond the crawl of duty," in *Proceedings of the 18th International World Wide Web Conference*, 2009.

[111] V. Shkapenyuk and T. Suel, "Design and implementation of a high-performance distributed web crawler," in *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[112] A. Singh, M. Srivatsa, L. Liu, and T. Miller, "Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web," in *SIGIR Workshop on Distributed Information Retrieval*, 2003.

[113] Q. Tan, Z. Zhuang, P. Mitra, and C. L. Giles, "A clustering-based sampling approach for refreshing search engine's database," in *Proceedings of the 10th International Workshop on the Web and Databases*, 2007.

[114] T. Urvoy, T. Lavergne, and P. Filoche, "Tracking web spam with hidden style similarity," in *Proceedings of the 2nd International Workshop on Adversarial Information Retrieval on the Web*, 2006.

[115] J. L. Wolf, M. S. Squillante, P. S. Yu, J. Sethuraman, and L. Ozsen, "Optimal crawling strategies for web search engines," in *Proceedings of the 11th International World Wide Web Conference*, 2002.

[116] B. Wu and B. Davison, "Identifying link farm spam pages," in *Proceedings of the 14th International World Wide Web Conference*, 2005.

[117] P. Wu, J.-R. Wen, H. Liu, and W.-Y. Ma, "Query selection techniques for efficient crawling of structured web sources," in *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[118] Yahoo! Research Barcelona, "Datasets for web spam detection," http://www.yr-bcn.es/webspam/datasets.

[119] J.-M. Yang, R. Cai, C. Wang, H. Huang, L. Zhang, and W.-Y. Ma, "Incorporating site-level knowledge for incremental crawling of web forums: A list-wise strategy," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[120] S. Zheng, P. Dmitriev, and C. L. Giles, "Graph-based seed selection for web-scale crawlers," in *Proceedings of the 18th Conference on Information and Knowledge Management*, 2009.

[121] K. Zhu, Z. Xu, X. Wang, and Y. Zhao, "A full distributed web crawler based on structured network," in *Asia Information Retrieval Symposium*, 2008.