BUbiNG: Massive Crawling for the Masses*

Paolo Boldi
Dipartimento di Informatica
Università degli Studi di
Milano, Italy
boldi@di.unimi.it

Andrea Marino
Dipartimento di Informatica
Università degli Studi di
Milano, Italy
andrea.marino1@unimi.it

Massimo Santini
Dipartimento di Informatica
Università degli Studi di
Milano, Italy
santini@di.unimi.it

Sebastiano Vigna
Dipartimento di Informatica
Università degli Studi di
Milano, Italy
vigna@acm.org

ABSTRACT

Although web crawlers have been around for twenty years by now, there is virtually no freely available, open-source crawling software that guarantees high throughput, overcomes the limits of single-machine tools and at the same time scales linearly with the amount of resources available. This paper aims at filling this gap.

We describe BUbiNG, our next-generation web crawler built upon the authors' experience with UbiCrawler [8] and on the last ten years of research on the topic. BUbiNG is an open-source Java fully distributed crawler (no central coordination), and single BUbiNG agents using sizeable hardware can crawl several thousands pages (per agent) per second respecting strict politeness constraints, both host- and IP-based. Unlike existing open-source distributed crawlers that rely on batch techniques (like MapReduce), BUbiNG job distribution is based on modern high-speed protocols so to achieve very high throughput.

1. INTRODUCTION

A *web crawler* (sometimes also known as a *(ro)bot* or *spider*) is a system that downloads systematically a large number of web pages starting from a seed. Web crawlers are, of courses, used by search engines, but also by companies selling "Search–Engine Optimization" services, archival projects such as the Internet Archive, surveillance systems (e.g., that scan the web looking for cases of plagiarism), and by entities performing statistical studies of the structure and the content of the web, just to name a few.

*The authors were supported by the EU-FET grant NADINE (GA 288956).

The basic inner working of a crawler is surprisingly simple form a theoretical viewpoint: it is a form of traversal (for example, a breadth-first visit). Starting from a given *seed* of URLs, a set of associated pages is downloaded, their content is parsed, and the resulting links are used iteratively to collect new pages.

Albeit in principle a crawler just performs a visit of the web, there are a number of factors that make the visit of a crawler inherently different from a textbook algorithm. The first and most important difference is that the size of the graph to be explored is unknown and huge; in fact, infinite. The second difference is that visiting a node (i.e., downloading a page) is a complex process that has intrinsic limits due to network speed, latency, and *politeness*—the requirement of not overloading servers during the download. Not to mention the countless problems (errors in DNS resolutions, protocol or network errors, presence of traps) that the crawler may find on its way.

In this paper we describe the design and implementation of BUbiNG, our new web crawler built upon the experience with UbiCrawler [8] and on the last ten years of research on the topic. BUbiNG aims at filling an important gap in the range of available crawlers. In particular:

- It is a pure-Java, open-source crawler released under the Gnu GPLv3.
- It is fully distributed: multiple agents perform the crawl concurrently and handle the necessary coordination without the need of any central control; given enough bandwidth, the crawling speed grows linearly with the number of agents.
- Its design acknowledges that CPUs and OS kernels have become extremely efficient in handling a large number of threads, in particular if they are mainly I/O-bound, and that large amounts of RAM are by now easily available at a moderate cost.
- More in detail, we assume that the memory used by an agent must be *constant* in the number of discovered URLs, but that it can *scale linearly* in the number of discovered hosts. This assumption simplifies and makes several data structures more efficient.
- It is very fast: on a 64-core, 64GB workstation it can download hundreds of million of pages at more than

9 000 pages per second respecting politeness both by host and by IP, analyzing, compressing and storing more than 140 MB/s of data.

- It is extremely configurable: beyond choosing the sizes of the various data structures and the communication parameters involved, implementations can be specified by reflection in a configuration file and the whole dataflow followed by a discovered URL can be controlled by arbitrary user-defined filters, that can further be combined with standard Boolean-algebra operators.

- It guarantees that hostwise the visit is an exact breadth-first visit (albeit the global policy can be customized).

- It guarantees that politeness intervals are satisfied both at the host and the IP level, that is, that two data fetch to the same host or IP are separated by at least a specified amount of time. The two intervals can be set independently, and, in principle, customized per host or IP.

When designing a crawler, one should always ponder over the specific usage the crawler is intended for. This decision influences many of the design details that need to be taken. Our main goal is to provide a crawler that can be used out-of-the-box as an archival crawler, but that can be easily modified to accomplish other tasks. Being an archival crawler, it does not perform any refresh of the visited pages, and moreover it tries to perform a visit that is as close to breadth-first as possible (more about this below). Both behaviors can in fact be modified easily in case of need, but this discussion (on the possible ways to customize BUbiNG) is out of the scope of this paper.

We plan to use BUbiNG to provide new data sets for the research community. Datasets crawled by UbiCrawler have been used in hundreds of scientific publications, but BUbiNG makes it possible to gather data orders of magnitude larger.

2. RELATED WORKS

Web crawlers have been developed since the birth of the web. The first generation crawler dates back to the early 90s: World Wide Web Worm [24], RBSE spider [16], MOMspider [18], WebCrawler [30]. One of the main contributions of these works has been that of pointing out some of the main algorithmic and design issues of crawlers. In the meanwhile, several commercial search engines, having their own crawler (e.g., AltaVista) were born. In the second half of the 90s, the fast growth of the web called for the need of large-scale crawlers, like the crawler of Internet Archive Module [11] and the first generation of the Google crawler [9]. This generation of spiders was able to download efficiently tens of millions of pages. At the beginning of 2000, the scalability, the extensibility, and the distribution of the crawlers become a key design point: this was the case of the Java crawler Mercator [28] (distributed version of [19]), Polybot [32], IBM WebFountain [15], and UbiCrawler [8]. These crawlers were able to produce snapshots of the web of hundreds of millions of pages.

Recently, a new generation of crawlers was designed, aiming to download billions of pages, like [22]. Nonetheless, none of them is freely available and open source: BUbiNG is the first open-source crawler designed to be fast, scalable and runnable on commodity hardware.

For more details about previous works or in the main issues in the design of crawlers, we refer the reader to [29, 26, 31].

2.1 Open-source crawlers

Although web crawlers have been around for twenty years by now (since the spring of 1993, according to [29]), the area of freely available ones, let alone open-source, is still quite narrow. With the few exceptions that will be discussed below, most stable projects we are aware of (GNU wget, Ht//Dig, mngoGoSearch, to cite a few) do not (and are not designed to) scale to download more than few thousands or tens of thousands pages. They can be useful to build an intranet search engine, but not for web-scale experiments.

Heritrix [1, 27] is one of the few examples of an open-source search engine designed to download large datasets: it was developed starting from 2003 by Internet Archive [2] (a non-profit corporation aiming to keep large archival-quality historical records of the world-wide web) and it has been since actively developed. Heritrix (available under the Apache license) is a single-machine crawler, although it is of course multi-threaded, which is the main hindrance to its scalability. The default crawl order is breadth-first, as suggested by the archival goals behind its design. On the other hand, it provides a powerful checkpointing mechanism and a flexible way of filtering and processing URLs after and before fetching. It is worth noting that Internet Archive proposed, implemented (in Heritrix) and fostered a standard format for archiving web content, called WARC, that is now an ISO standard [4] and that BUbiNG is also adopting for storing the downloaded pages.

Nutch [21] is one of the best known existing open-source web crawlers; in fact, the goal of Nutch itself is much broader in scope, because it aims at offering a full-fledged search engine under all respects: besides crawling, Nutch implements features such as (hyper)text-indexing, link analysis, query resolution, result ranking and summarization. It is natively distributed (using Apache Hadoop as task-distribution backbone) and quite configurable; it also adopts breadth-first as basic visit mechanism, but can be optionally configured to go depth-first or even largest-score first, where scores are computed using some scoring strategy which is itself configurable. Scalability and speed are the main design goals of Nutch; for example, Nutch was used to collect TREC ClueWeb09 dataset[1], the largest web dataset publicly available as of today consisting of 1 040 809 705 pages, that were downloaded at the speed of 755.31 pages/s [3], but to do this they used a Hadoop cluster of 100 machines [12], so their real throughput was of about 7.55 pages/s *per machine*. This figure is not unexpected: using Hadoop to distribute the crawling jobs is easy, but not efficient, because it constrains the crawler to work in a batch fashion. It shouldn't be surprising that using a modern job-distribution framework like BUbiNG does increases the throughput by orders of magnitude.

[1]The new ClueWeb12 dataset, that is going to be released soon, was collected using Heritrix, instead: five instances of Heritrix, running on five Dell PowerEdge R410, were run for three months, collecting 1.2 billions of pages. The average speed was of about 38.6 pages per second per machine.

3. ARCHITECTURE OVERVIEW

BUbiNG stands on a few architectural choices which in some cases contrast the common folklore wisdom. We took our decisions after carefully benchmarking several options and gathering the hands-on experience of similar projects.

- The fetching logic of BUbiNG is built around thousands of identical *fetching threads* performing essentially only synchronous (blocking) I/O. Experience with recent Linux kernels and increase in the number of cores per machine shows that this approach consistently outperforms asynchronous I/O. This strategy simplifies significantly the code complexity, and makes it trivial to implement features like HTTP/1.1 "keepalive" multiple-resource downloads.
- *Lock-free* [25] data structures are used to "sandwich" fetching threads, so that they never have to access lock-based data structures. This approach is particularly useful to avoid direct access to synchronized data structures with logarithmic modification time, such as priority queues, as contention between fetching threads can become very significant.
- URL storage (both in memory and on disk) is entirely performed using byte arrays. While this approach might seen anachronistic, the Java `String` class can easily occupy three times the memory used by a URL in byte-array form (both due to additional fields and to 16-bit characters) and doubles the number of objects. BUbiNG aims at exploiting the large memory sizes available today, but garbage collection has a linear cost in the number of objects: this factor must be taken into account.
- Following UbiCrawler's design [8], BUbiNG agents are identical and autonomous. The assignment of URLs to agents is entirely customizable, but by default we use *consistent hashing* as a fault-tolerant, self-configuring assignment function.

In this section, we overview the structure of a BUbiNG agent: the following sections detail the behavior of each component. The inner structure and data flow of an agent is depicted in Figure 1.

The bulk of the work of an agent is carried out by low-priority *fetching threads*, which download pages, and *parsing threads*, which parse and extract information from downloaded pages. Fetching threads are usually thousands, and spend most of their time waiting for network data, whereas one usually allocates as many parsing threads as the number of available cores, because their activity is mostly CPU bound.

Fetching threads are connected to parsing threads using a lock-free *result* list in which they enqueue buffers of fetched data, and wait for a parsing thread to analyze them. Parsing threads poll the result list using an exponential backoff scheme, perform actions such as parsing and link extraction, and signal back to the fetching thread that that the buffer can be filled again.

As parsing threads discover new URLs, they enqueue them to a *sieve* that keeps track of which URLs have been already discovered (we do not want to download the same URL twice). A sieve is a data structure similar to a queue with memory: each enqueued element will be dequeued at some later time, with the guarantee that an element that is enqueued multiple times will be dequeued just once. URLs are added to the sieve as they are discovered by parsing. A cache sits in front of the sieve to avoid that frequently found URLs put the sieve under stress. The cache has also another important goal: it avoids that frequently found URLs assigned to another agent are retransmitted several times.

URLs that come out of the sieve are ready to be visited, and they are taken care of (stored, organized and managed) by the *frontier*, which is actually itself decomposed into several modules.

The most important data structure of the frontier is the *workbench*, an in-memory data structure that keeps track of the visit state of each host currently visited and that can check in constant time whether some host can be accessed for download without violating the politeness constraints. Note that to attain the goal of several thousands downloaded pages per second without violating politeness constraints it is necessary to keep track of the visit state of hundreds of thousands of hosts.

When a host is ready for download, its visit state is extracted from the workbench and moved to a lock-free *todo queue* by a suitable thread. Fetching threads poll the todo queue with an exponential backoff, fetch resources from the retrieved visit state[2] and then put it back on the workbench. Note that we expect that once a large crawl has started, the todo queue will never be empty, so fetching threads will never have to wait. Most of the efforts of the components of the frontier are actually geared towards avoiding that fetching threads ever wait on an empty todo queue.

The only active component (i.e., a thread) of the frontier is the *distributor*: it is a high-priority thread that processes URLs that come out of the sieve (and must therefore be crawled). Assuming for a moment that memory is unbounded, the only task of the distributor is that of iteratively dequeueing a URL from the sieve, checking whether it belongs to a host for which a visit state already exists, and then either creating a new visit state or enqueuing the URL to an existing one. If a new visit state is necessary, it is passed to a set of *DNS threads* that perform DNS resolution and then move the visit state on the workbench.

Since, however, breadth-first visit queue grows exponentially, and the workbench can use only a fixed amount of in-core memory, it is necessary to *virtualize* it, that is, writing on disk part of the URLs coming out of the sieve. To decide whether to keep a visit state entirely in the workbench or to virtualize it, and also to decide when and how URLs should be moved from the virtualizer to the workbench, the distributor uses a complex policy that is described later.

Finally, every agent stores resources in its *store* (that may possibly reside on a distributed or remote file system). The native BUbiNG store is a compressed file in the Web ARChive (WARC) format (the standard proposed and made popular by Heritrix). This standard specifies how to combine several digital resources with other information into an aggregate archive file. In BUbiNG compression happens in a heavily parallelized way, with parsing threads compressing independently pages and using concurrent primitives to pass compressed data to a flushing thread.

3.1 The sieve

[2]Possibly multiple resources on a single TCP connection using the "keepalive" feature of HTTP 1.1.

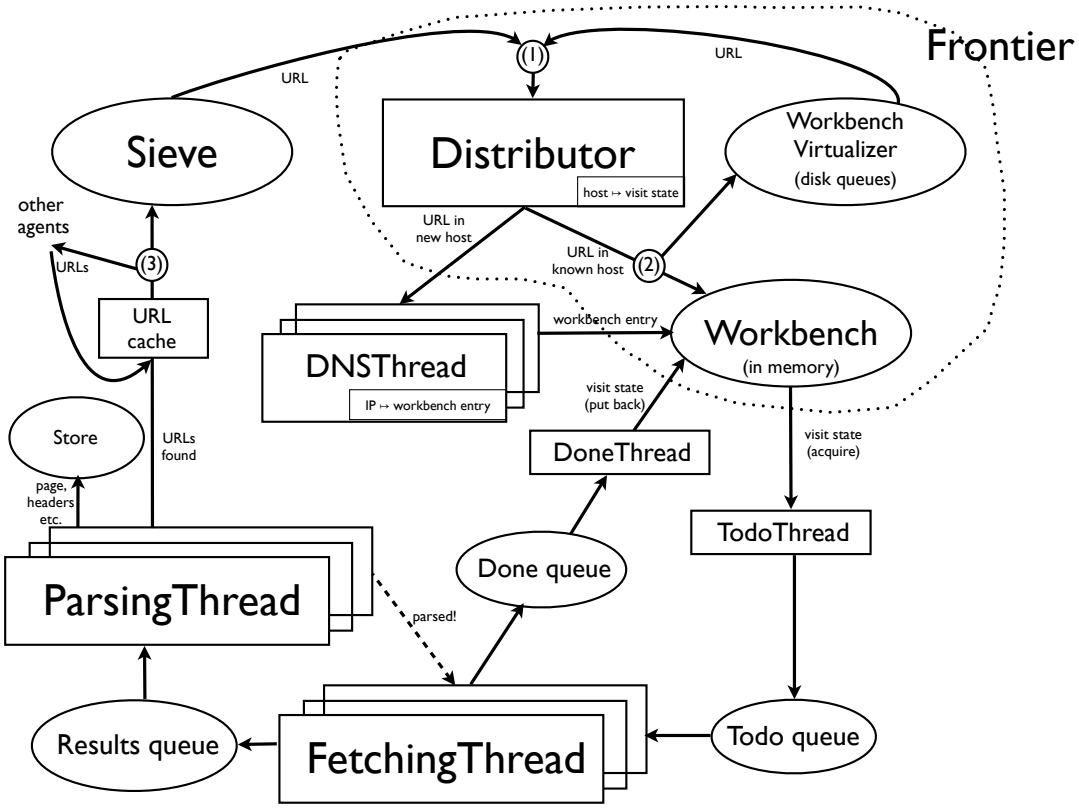

Figure 1: Overview of the architecture of a BUbiNG agent. Ovals represent data structures, whereas rectangles represent threads (or sets of threads).

A *sieve* is a queue with memory: it provides enqueue and dequeue primitives, similarly to a standard queue; each element enqueued to a sieve will be eventually dequeued later. However, a sieve guarantees also that if an element is enqueued multiple times, it will be dequeued just one time. Sieves (albeit not called with this name) have always been recognized as a fundamental basic data structure for a crawler: their main implementation issue lies in the unbounded, exponential growth of the number of discovered URLs. While it is easy to write enqueued elements to a disk file, checking that an element is not returned multiple times requires *ad-hoc* data structures, as standard dictionaries would use too much in-core memory.

The actual sieve implementation used by BUbiNG can be customized, but the default one, called `MercatorSieve`, is similar to the one suggested in [19] (hence its name). Each element known to the sieve is stored as a 64-bit hash in a disk file. Every time a new element is enqueued, its hash is stored in an in-memory array, and the element is saved in an auxiliary file. When the array is full, it is sorted and compared with the set of elements known to the sieve. The auxiliary file is then scanned, and previously unseen elements are stored for later dequeueing. All these operations require only sequential access to all files involved. Note that the output order is guaranteed to be the same of the input order (i.e., elements are appended in the same order in which they appeared the first time).

A generalization of the idea of a sieve, which adds the possibility of associating values with elements, is the DRUM (Disk Repository with Update Management) structure used by IRLBot and described in [22]. A DRUM provides additional operations that retrieve or update the values associated with elements. From an implementation viewpoint, DRUM is a Mercator sieve with multiple arrays, called *buckets*, in which a careful orchestration of in-memory and on-disk data makes it possible to sort in one shot sets of elements an order of magnitude larger than what the Mercator sieve would allow using the same in-core memory. However, to do so DRUM must sacrifice breadth-first order: due to the inherent randomization of key placement in the buckets, there is no guarantee that URLs will be crawled in breadth-first order, not even per host. Finally, the tight analysis in [22] about the properties of DRUM is unavoidably bound to the single-agent approach of IRLBot: for example, the authors conclude that a URL cache to reduce the number of insertions in the DRUM is not useful, but the same cache reduces significantly network transmissions. Once the cache is in place, the Mercator sieve becomes much more competitive.

There are several other implementations of the sieve logic currently used. A quite common choice is to use an explicit queue and a *Bloom filter* [7] to remember enqueued elements. Albeit popular, this choice has no theoretical guarantee: while it is possible to decide *a priori* the maximum number of pages that will ever be crawled, it is very difficult to bound in advance the size of the *discovered* URLs, and this number is essential in sizing the Bloom filter. If

the discovered URLs are significantly more than expected, several pages are likely to be lost because of false positives. A better choice is to use a dictionary of fixed-size *fingerprints* obtained from URLs using a suitable hash function. The disadvantage is that the structure would no longer use constant memory.

3.2 The workbench

The *workbench* is an in-memory data structure that contains the next URLs to be visited, and can check in constant time whether a URL is ready for download without violating politeness limits. It is one of the main novel ideas in BUbiNG's design, and it is one of the main reasons why we can attain a very high throughput.

First of all, URLs associated with a specific host[3] are kept in a structure called *visit state*, containing a FIFO queue of the next URLs to be crawled for that host along with a `next-fetch` field that specifies the first instant in time when a URL from the queue can be downloaded, according to the per-host politeness configuration. Note that inside a visit state we only store a byte-array representation of the path and query of a URL: this approach significantly reduces object creation, and provides a simple form of compression by prefix omission.

Visit states are further gathered by IP address in *workbench entries*; every time the first URL for a given host is found, a new visit state is created and then the IP address is determined (by one of the *DNS threads*): the new visit state is either put in a new workbench entry (if no known host was as yet associated to that IP), or in an existing one.

A workbench entry contains a queue of visit states prioritized by their `next-fetch` field. In other words, a workbench entry contains all visit states associated with the same IP, along with an IP-specific `next-fetch`, containing the first instant in time when the IP address can be accessed again, according to the per-IP politeness configuration. The *workbench* is the queue of all workbench entries, prioritized on the `next-fetch` field of each entry *maximized* with the `next-fetch` field on the top element of its queue of visit states. In other words, the workbench is a priority queue of priority queues of FIFO queues.

We remark that due to our choice of priorities *there is a host that can be visited without violating host or IP politeness if and only if the first URL of the top visit state of the top workbench entry can be visited*. Moreover, if there is no such host, the delay after which a host will be ready is given by the priority of the top workbench entry minus the current time.

The workbench acts as a *delay queue*: its dequeue operation waits, if necessary, until a host is ready to be visited. At that point, the top entry E is removed from the workbench and the top visit state is removed from E. The visit state and the associated workbench entry act as a *token* that is virtually passed between BUbiNG's components to guarantee that no component is working on the same workbench entry at the same time (in particular, this forces both kinds of politeness). In practice, as we mentioned in the overview, dequeueing is performed by a high-priority thread, the *todo thread*, that constantly dequeues visit states from the workbench and enqueue them to a lock-free *todo queue*, which is then accessed by fetching threads. This approach, besides avoiding contention by thousands of threads on a relatively slow structure, makes the number of visit states that are ready for downloads easily measurable: it is just the size of the todo queue. The downside is that, in principle, using very skewed per-host or per-IP politeness delays might cause the order of the todo queue not to reflect the actual priority of the visit state contained therein.

3.3 Fetching threads

A *fetching thread* is a very simple thread that iteratively extracts visit states from the todo queue. If the todo queue is empty, a standard exponential backoff procedure is used to avoid polling the list too frequently, but the design of BUbiNG aims at keeping the todo queue nonempty and avoiding backoff altogether.

Once a fetching thread acquires a visit state, it tries to fetch the first URL of the visit state FIFO queue. If suitably configured, a fetching thread can also iterate the fetching process on more URLs for a fixed amount of time, so to exploit the "keepalive" feature of HTTP 1.1.

Each fetching thread has an associate *fetch data* instance in which the downloaded data are buffered. Fetch data include a transparent buffering method that keeps in memory a fixed amount of data and dumps on disk the remaining part. By sizing the fixed amount suitably, most requests can be completed without accessing the disk, but at the same time rare large requests can be handled without allocating additional memory.

After a URL has been fetched, the fetch data is put in the *results* queue so that one of the parsing threads will parse it. One the parsing is over, the parsing thread will signal back so the fetching thread will be able to start working on a new URL. Once a fetching thread has to work a new visit state, it puts the current visit state on a *done queue*, from which it will be dequeued by a suitable thread that will put it back on the workbench together with its associated entry.

Most of the time, a fetching thread is blocked on I/O, which makes it possible to run thousands of them in parallel. Indeed, the number of fetching threads determines the amount of parallelization BUbiNG can achieve while fetching data from the network, so it should be chosen as large as possible, compatibly with the amount of bandwidth available and with the memory used by fetch data.

3.4 Parsing threads

A *parsing thread* iteratively extracts from the results queue fetch data that have been previously enqueued by a fetching thread. Then, the content of the HTTP response is analyzed and possibly parsed. If the response contains an HTML page, the parser will produce a set of URLs that will be first checked against the URL cache, and then, if not already seen, either sent to another agent, or enqueued to the sieve (given that the maximum number of URLs per host has not been exceeded).

During the parsing phase, a parsing thread computes a signature of the content of the response. In the case of HTML pages, some heuristic is used to collapse near-duplicates (e.g., most HTML attributes are stripped). The signature is stored

[3]Every URL is made [6] by a scheme (also popularly called "protocol"), an authority (a host, possibly a port number, and possibly some user information) and a path to the resource, possibly followed by a query (that is separated from the path by a "?"). BUbiNG's data structures are built around the pair scheme+authority, but in this paper we will use the more common word "host" to refer to it.

in a Bloom filter [7] and it is used to avoid crawling several times the same page (or near-duplicate pages).[4] Finally, the content of the response is saved to the store.

The number of parsing threads should be equal to the number of available cores.

3.5 DNS threads

DNS threads are used to solve host names of new hosts: a DNS thread continuously dequeues from the list of newly discovered visit states and *resolves* its host name, adding it to a workbench entry (or creating a new one, if the IP itself is new), and putting it on the workbench.

The number of DNS threads is limited by the kind of DNS service the crawler relies upon. In our experience, it is essential to run a local recursive DNS server to avoid the bottleneck of an external server.

3.6 The workbench virtualizer

The workbench virtualizer is a sequence of k on-disk URL queues (at the beginning $k = 1$), called *virtual queues*; the last virtual queue is called *overflow queue*. This design is inspired by the BEAST module of IRLbot [22], albeit it is more geared towards maintaining the visit order as close as possible to a breadth-first visit, rather than using prioritization.

Conceptually, all URLs that have been extracted from the sieve but have not yet been fetched are enqueued in the workbench visit state they belong to, in the exact order in which they came out of the sieve. Since, however, we aim at crawling with an amount of memory that is *constant* in the number of discovered URLs, part of the queue must be written on disk. Each virtual queue contains a fraction of URLs from each visit state, in such a way that the overall URL order respects, *per host*, the original breadth-first order.

Virtual queues are consumed as the visit proceeds, following the natural per-host breadth-first order. As fetching threads download URLs, the workbench is partially freed and can be filled with URLs coming from the virtual queues. When all virtual queues preceding the overflow queue have been exhausted, the number of virtual queues is increased (usually doubled) and the content of the overflow queue is redistributed on the set of new queues based on an estimate of the future time at which each URL will be needed.

3.7 The distributor

The *distributor* is a high-priority thread that orchestrates the movement of URLs out of the sieve, and loads as necessary URLs from virtual queues into the workbench.

As the crawl proceeds, URLs get accumulated in workbench visit states at different speeds, both because hosts have different responsiveness and because websites have different sizes and branching factors. Moreover, the size occupied by the workbench has a (configurable) limit that cannot be exceeded, as one of the central design goals of BUbiNG is that the amount of central memory occupied cannot grow unboundedly in the size of the discovered URLs, but only in the number of hosts discovered. Thus, filling the workbench blindly with URLs coming out of the sieve would soon result in having in the workbench only URLs belonging to a limited number of hosts.

[4]In a post-crawl phase, there are several more sophisticated approaches that can be applied, like *shingling* [10], *simhash* [14], *fuzzy fingerprinting* [17, 13] and others, like [23].

The *front* of a crawl, at any given time, is the number of visit states that is ready for download by politeness constraints. The front size determines the overall throughput of the crawler—because of politeness, the number of distinct hosts currently being visited is the crucial datum that establishes how fast or slow the crawl is going to be.

One of the two forces driving the distributor is, indeed, that *the front should always be large enough so that no fetching thread has ever to wait.* To attain this goal, the distributor enlarges dynamically the *required front size*: each time a fetching thread has to wait, although the current front size is larger than the current required front size, the latter is increased. After a warmup phase, the required front size stabilizes to a value that depends on the kind of host visited and on the amount of resources available. At that point, it is impossible to have a faster crawl given the resources available, as all fetching threads are continuously downloading data. Increasing the number of fetching threads, of course, may cause an increase of the required front size.

The second force driving the distributor is the (somewhat informal) requirement that *we try to be as close to a breadth-first visit as possible.* Note that this force works in an opposite direction with respect to enlarging the front—URLs that are already in existing visit states should be in principle visited *before* any URL in the sieve, but enlarging the front requires dequeueing from the sieve to find new hosts.

The distributor is also responsible for filling the workbench with URLs coming either out of the sieve, or out of virtual queues (circle numbered (1) in Figure 1). Once again, staying close to a breadth-first visit requires loading URLs in virtual queues, but keeping the front large might require reading URLs from the sieve to discover new hosts.

The distributor balances these two forces by keeping an eye on the *limbo*—the set of visit states that currently have URLs in virtual queues, but few (or no) no URLs in memory:

- if the limbo is large, the distributor will try to read from the virtual queues, in the hope that the front (the number of hosts currently being visited) can increase at the expense of the limbo size;
- if the limbo is small, the distributor will rather read from the sieve, hoping to find new sites to make the front larger.

The limbo is considered to be large when it contains more than a small fraction (typically, 1%) of the visit states with some URLs in virtual queues. If there is no room in the workbench, or the front is already large enough, the distributor just waits. The overall behavior is depicted in Figure 2.

Note that if the distributor takes the decision to read from the virtual queues (i.e., to read the first URL of the first non-empty queue), the URL is always put in the workbench (and will be later fetched). On the other hand, if the workbench reads a URL from the sieve it can be either put in the workbench or written in a virtual queue, depending on the estimate of the future time at which the URL will be needed. If the distributor decides to write the URL on a virtual queue, another URL will have to be taken either from the sieve or from the virtual queues, and so on until the workbench is full again or until the front is large enough.

3.8 Configuration and Heuristics

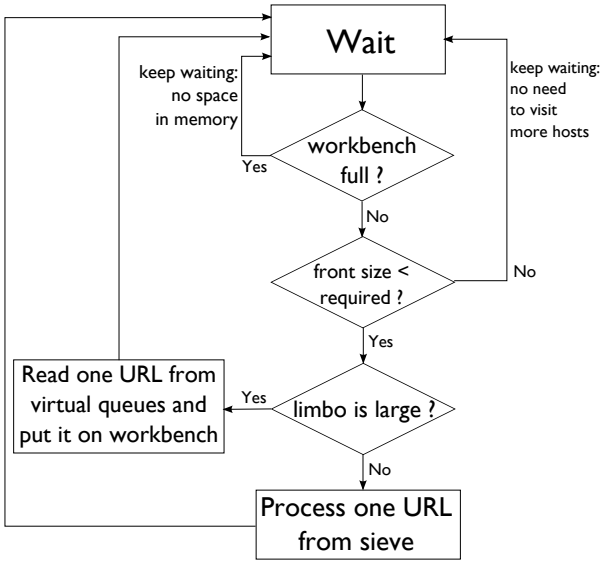

Figure 2: How the distributor interacts with the sieve, the workbench and the workbench virtualizer.

To make BUbiNG capable of a versatile set of tasks and behaviors, every crawling phase (fetching, parsing, following the URLs of a page, scheduling new URLs, storing pages) is controlled by a *filter*, a Boolean predicate that determines whether a given resource should be accepted or not. Filters can be configured both at startup and at runtime allowing for a very fine-grained control.

The type of objects a filter considers is called the *base type* of the filter. In most cases, the base type is going to be a URL or a fetched page. More precisely, a *prefetch* filter is one that has a BUbiNG URL as its base type (typically: to decide whether a URL should be scheduled for later visit, or should be fetched); a *postfetch* filter is one that has a fetched response as base type and decides whether to do something with that response (typically: to parse it, to store it, etc.).

Even if it is relatively easy to write a filter, BUbiNG contains a number of filters ready to be used. The prefetch filters include, for instance filters that accept only URLs whose host ends with a certain string, or URLs whose path ends with one of a given set of suffixes. The postfetch filters include, for instance, filters accepting certain contain types, or streams that appear to be binary.

Filters can be composed by means of Boolean operators with a short-circuit semantics. Additionally, we provide a parser that makes it possible creating filters by reflection. An example of a textual description of a composed filter is:

```
(HostEndsWith(foo.bar) and not
    ForbiddenHost(http://xxx.yyy/list-of-hosts))
or NoMoreSlashThan(10)
```

One filter, in particular, accepts only URLs whose path does not contain too many *duplicate segments*. Indeed, it is not uncommon to find URLs generated by badly configured servers that look like `http://.../foo/bar/foo/bar/...`. Our filter will not accept URLs containing a sequence of consecutive segments appearing more times than a given threshold. The implementation uses ideas from [20] to simulate a suffix-tree visit on a suffix array, and the approach of [33], for the linear-time detection of tandem arrays using suffix trees: the resulting code is one order of magnitude faster than regular expressions. We observe that, for the same purpose of avoiding bad URLs (of different kinds), it would be interesting to add a filter implementing the DUSTER (Different URL's with Similar Text) technique [5].

3.9 Distributed crawling

BUbiNG crawling activity can be distributed by running several agents over multiple machines. All agents are identical instances of BUbiNG, without any explicit leadership, similarly to UbiCrawler [8]: all data structures described above are part of each agent.

URL assignment to agent is entirely configurable. By default, BUbiNG uses just the host to assign a URL to an agent, which avoids that two different agents can crawl the same host at the same time. Moreover, since most hyperlinks are relative, each agent will be himself responsible for the large majority of URLs found in a typical HTML page [29]. Assignment of hosts to agent is performed using *consistent hashing* [8].

Communication of URLs between agents is handled by the message-passing methods of the JGroups Java library; in particular, to make communication lightweight URLs are by default distributed using UDP. More sophisticated communications between the agents rely on the TCP-based JMX Java standard remote-control mechanism, which exposes most of the internal configuration parameters and statistics. Most of the crawler structures are indeed modifiable at runtime, including, for instance, the number of parsing, fetching and DNS threads.

4. EXPERIMENTS

Testing a crawler is a delicate, intricate, arduous task: on one hand, every real-world experiment is obviously influenced by the hardware at one's disposal (in particular, by the available bandwidth). Moreover, real-world tests are difficult to repeat many times with varying parameters: you will either end up disturbing the same sites over and over again, or choosing to visit every time a different portion of the web, with the risk of introducing artifacts in the evaluation. Given these considerations, we ran two kinds of experiments: one batch was performed *in vitro* with a HTTP proxy[5] simulating network connections towards the web and generating fake HTML pages (with a configurable behavior that includes delays, protocol exceptions etc.), and another group of experiments were performed *in vivo*.

4.1 In vitro experiments

To verify the robustness of BUbiNG when varying some basic parameters, such as the number of fetching threads or the IP delay, we have run some *in vitro* simulations on a group of four machines sporting 64 cores and 64 GB of core memory. In all experiments, the number of parsing and DNS threads has been fixed and set respectively to 64 and 10. The size of the workbench has been set to 512MB, while the size of the sieve has been set to 256MB. Every *in vitro* experiment was run for 90 minutes.

Fetching threads. The first thing we wanted to test was that increasing the number of fetching threads produces a

[5]The proxy software is distributed along with the rest of BUbiNG.

Crawler	Machines	Resources (Millions)	Resources/s		Speed in MB/s	
			overall	per agent	overall	per agent
Nutch (ClueWeb09)	100 (Hadoop)	1 200	430	4.3	10	0.1
Heritrix (ClueWeb12)	5	2 300	300	60	19	3.9
IRLBot	1	6 380	1 790	1 790	40	40
BUbiNG (Milano)	3	650	2 200	735	96	32
BUbiNG (Pisa)	1	100	2 500	2 500	71	71
BUbiNG (Pisa)	4	37	5 400	1 350	168	42
BUbiNG (iStella)	1	115	3 700	3 700	135	135
BUbiNG (*in vitro*)	4	1 000	36 600	9 150	584	146

Table 1: Comparison between BUbiNG and the main existing open-source crawlers. Resources are HTML pages for ClueWeb09 and IRLBot, but include other data types (e.g., images) for ClueWeb12. For reference, we also report the throughput of IRLbot [22], although the latter is not open source. Note that ClueWeb09 was gathered using a heavily customized version of Nutch.

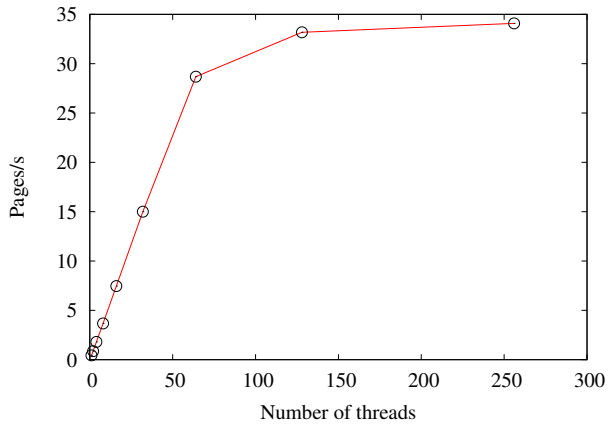

Figure 3: The average number of pages per second with respect to the number of threads using a simulated slow connection. Note the linear increase in speed until the plateau, due to the limited (300) number of threads of the simulator.

better usage of the network, and hence a larger number of requests, until the bandwidth is saturated. The results of this experiment are shown in Figure 3 and have been obtained by having the proxy simulate a network that saturates quickly, using no politeness delay. The behavior visible in the plot tells us that the increase in the number of fetching threads produces a linear increase in the number of requests until the available (simulated) bandwidth is reached; after that, the number of requests stabilizes to a plateau. Also this part of the plot tells us something: after saturating the bandwidth, we do not see any decrease in the throughput, witnessing the fact that our infrastructure does not cause any hindrance to the crawl.

Politeness. The experiment described so far uses a small number of fetching threads, because the purpose was to show what happens before saturation. Now we show what happens under a heavy load. Our second *in vitro* experiment keeps the number of fetching threads fixed but increases the amount of politeness, as determined by the IP delay. The IP (respectively host) delay is a lower bound of the time between two successive requests to the same IP (respectively host). In our simulations we varied the IP delay and always set the host delay to be eight times the IP delay. We plot BUbiNG's throughput as the IP delay (hence the host delay) increases in Figure 4 (top): to maintain the same throughput, the front size (i.e., the number of hosts being visited in parallel) must increase, as expected. Moreover, this is independent on the number of threads (of course, until the network is saturated). In the same Figure we show that the average throughput is essentially independent from the politeness (and from the number of fetching threads) and the same is true of the CPU load. Even if this could seem surprising, this is the natural consequence of the following two observations:

- even with a small amount of fetching threads, BUbiNG always tries to fill the bandwidth and to maximize the computational resources;
- even varying the IP and host delay, BUbiNG modifies the number of hosts under visit in order to tune the interleaving between their processing.

Raw speed. Finally, we wanted to test the raw speed of a cluster of BUbiNG agents. We thus ran four agents using a larger workbench (2 GB) and 1000 fetching threads, IP delay 500 ms and host delay 4 s. We ran the agents until we gathered one billion pages, averaging 36 600 pages per second on the whole cluster. We also ran the same test on a single machine, obtaining essentially the same per-machine speed, showing that BUbiNG scales linearly with the number of agents in the cluster.

4.2 In vivo experiments

We performed a number of experiments *in vivo* at different sites. The main problem we had to face is that a single BUbiNG agent on sizable hardware can saturate a 1 Gb/s geographic link, so, in fact, we were not able to perform any test in which the network was not capping the crawler. Due to resource constraints, we decided to perform medium-size experiments on a variety of architectures and network connections. In the final version of the paper, we will report data for longer-running experiments.

A first, longer experiment was performed at our university (Milano): three BUbiNG agents using the same hardware of the *in vitro* experiments gathered 650 million pages from domains of the EU, but the connection was capped at 250 Mb/s. A second set of short-running experiments was

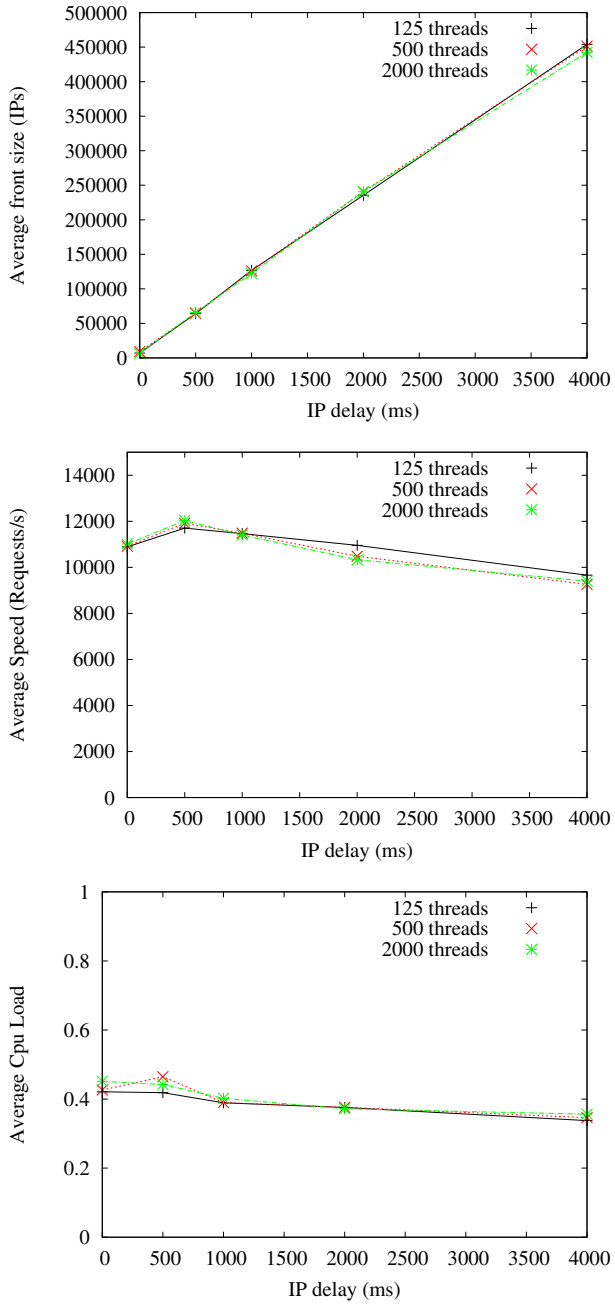

Figure 4: The average size of the front, the average number of requests per second, and the average CPU load with respect to the IP delay (the host delay is set to eight times the IP delay). Note that the front adapts linearly to the growth of the IP delay, and, due to the essentially unlimited bandwidth of the simulator, the number of fetching threads is irrelevant.

performed at Università di Pisa, using slower hardware (24-core, 24 GB RAM) capped at 1 Gb/s. Finally, iStella, an Italian commercial search engine provided us with a 48-core, 512 GB RAM machine capped at 1 Gb/s.

The results confirm the knowledge we have gathered with our *in vitro* experiment: in the iStella experiment we were able to saturate the 1 Gb/s link using a single BUbiNG agent. The two experiment at Pisa show a single, small-sized agent being able to download 2 500 pages per second, using about 2/3 of the available bandwidth, and three agents, saturating the 1 Gb/s link, downloading 5 400 pages per second. Finally, the long-running experiment at our university, albeit slow in comparison, shows the steadiness of BUbiNG after a large number of pages have been downloaded (see Figure 5).

5. COMPARISON

When comparing crawlers, many measures are possible, and depending on the task at hand, different measures might be suitable. For instance, crawling all types of data (CSS, images, etc.) usually yields a significantly higher throughout than crawling just HTML, as HTML pages are often rendered dynamically, sometimes causing a significant delay, whereas most other types are served statically. The crawling policy has also a huge influence on the throughput: prioritizing by indegree (as IRLBot does [22]) or alternative importance measure shifts most of the crawl on sites hosted on powerful servers with large-bandwidth connection. Ideally, crawler should be compared on a crawl with given number of pages in breadth-first fashion from a fixed seed, but some crawlers are not available to the public, which makes this goal unattainable.

In this section, as a tradeoff, we briefly give some very simple comparison with recent crawls made for the ClueWeb project: ClueWeb09 and ClueWeb12. The data used in this comparison are those available in [12] along with those found at `http://lemurproject.org/clueweb09/` and `http://boston.lti.cs.cmu.edu/crawler/crawlerstats.html`: notice that the data we have about those collections are sometimes slightly contradictory. We report the throughput declared by IRLBot [22], too, albeit the latter is not open source.

The results of the comparison are shown in Table 1: they show quite clearly that the speed of BUbiNG is several times that of IRLBot and one to two orders of magnitude greater than that of Heritrix or Nutch. While the comparison with the ClueWeb09 crawl is somewhat unfair (the hardware was "retired search-engine hardware"), it shows the inherent slowness of batch, Hadoop-based crawlers. The comparison with ClueWeb12 is more interesting, as the hardware used was recent and very similar to the one used in the Milano and in the *in vitro* experiment, sporting 64 GB of core memory.

All in all, our experiments show that BUbiNG's design provides a very high throughput: indeed, from our comparison, the highest throughput. The fact that the throughput can be scaled linearly just by adding agents makes it by far the fastest crawling system publicly available.

6. CONCLUSIONS

In this paper we have presented BUbiNG, our next-generation distributed open-source Java crawler. BUbiNG is order of magnitudes faster than existing open-source crawlers, scales

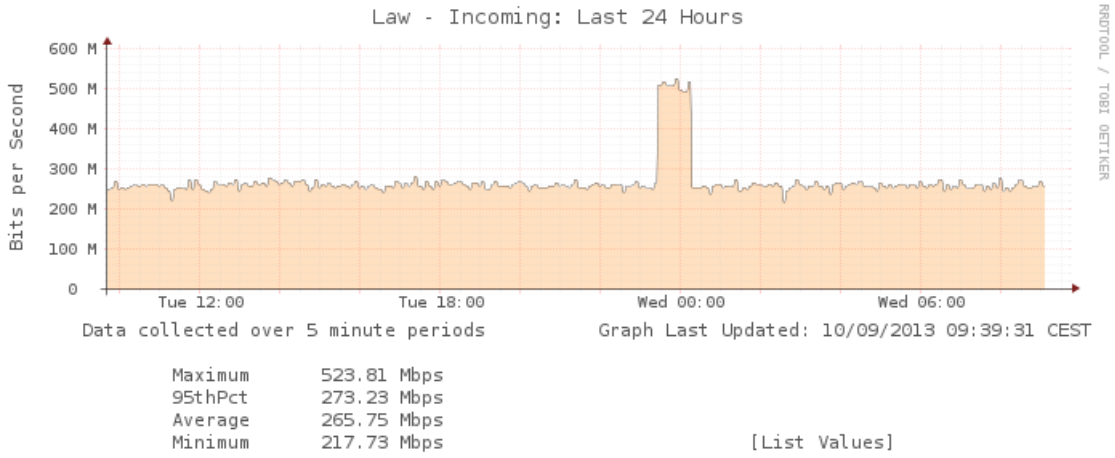

Figure 5: Network usage as reported by FlowViewer for the last 24 hours of the Milano experiment. Note that constant network usage. The peak around midnight is due to an internal data transfer.

linearly with the number of agents, and will provide the scientific community with a reliable tool to gather large data sets: this is the reason why, in the first place, the development of BUbiNG was financed in the framework of the EU-FET grant NADINE (New Algorithms for DIrected NEtworks).

Future work on BUbiNG includes integration with spam-detection software, policies for IP/host politeness throttling based on download times and site branching speed, and integration with different stores like HBase, HyperTable and similar distributed storage systems.

Acknowledgments. We thank our university for providing bandwidth for our experiments (and being patient with bugged releases). We thank Giuseppe Attardi, Antonio Cisternino and Maurizio Davini for providing the hardware, and the GARR Consortium for providing the bandwidth for the Pisa experiments. Finally, we thank Domenico Dato and Renato Soru for providing the hardware and bandwidth for the iStella experiment.

7. REFERENCES

[1] Heritrix web site. `https://webarchive.jira.com/wiki/display/Heritrix/Heritrix`.

[2] Internet archive website. `http://archive.org/web/web.php`.

[3] The clueweb09 dataset. `http://lemurproject.org/clueweb09/`, 2009.

[4] Iso 28500:2009, information and documentation - warc file format. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=44717`, 2009.

[5] Ziv Bar-Yossef, Idit Keidar, and Uri Schonfeld. Do not crawl in the dust: different urls with similar text. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 111–120, New York, NY, USA, 2007. ACM.

[6] et al. Berners-Lee. Uniform resource identifier (uri): Generic syntax. `http://www.ietf.org/rfc/rfc3986.txt`, 2005.

[7] Burton H. Bloom. Space-time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, 1970.

[8] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

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

[10] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[11] M. Burner. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques*, 2(5), 1997.

[12] Jamie Callan. The lemur project and its clueweb12 dataset. Invited talk at the SIGIR 2012 Workshop on Open-Source Information Retrieval.

[13] Soumen Chakrabarti. *Mining the web - discovering knowledge from hypertext data*. Morgan Kaufmann, 2003.

[14] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[15] Jenny Edwards, Kevin McCurley, and John Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 106–113, New York, NY, USA, 2001. ACM.

[16] D. Eichmann. The RBSE spider: balancing effective search against web load. In *Proceedings of the first World Wide Web Conference*, Geneva, Switzerland, May 1994.

[17] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L. Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the Twelfth Conference on World Wide Web*, Budapest, Hungary, 2003. ACM Press.

[18] R. Fielding. Maintaining distributed hypertext infostructures: Welcome to momspider. In *Proceedings of the 1st International Conference on the World Wide Web*, 1994.

[19] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, April 1999.

[20] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[21] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin. Nutch: A flexible and scalable open-source web search engine. *Oregon State University*, 2004.

[22] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: Scaling to 6 billion pages and beyond. *ACM Trans. Web*, 3(3):8:1–8:34, July 2009.

[23] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 141–150, New York, NY, USA, 2007. ACM.

[24] Oliver A. McBryan. GENVL and WWWW: Tools for taming the web. In *Proceedings of the first World Wide Web Conference*, pages 79–90, 1994.

[25] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275. ACM, 1996.

[26] Seyed M Mirtaheri, Mustafa Emre Dincturk, Salman Hooshmand, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. A brief history of web crawlers. In *CASCON*, 2013.

[27] Gordon Mohr, Michele Kimpton, Micheal Stack, and Igor Ranitovic. Introduction to Heritrix, an archival quality web crawler. In *Proceedings of the 4th International Web Archiving Workshop (IWAW'04)*, September 2004.

[28] Marc Najork and Allan Heydon. High-performance web crawling. In James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, editors, *Handbook of massive data sets*, pages 25–45. Kluwer Academic Publishers, 2002.

[29] Christopher Olston and Marc Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[30] Brian Pinkerton. Finding what people want: Experiences with the WebCrawler. In Anonymous, editor, *Proceedings of the 2nd International World Wide Web*, volume 18(6) of *Online & CDROM review: the international journal of*, Medford, NJ, USA, 1994. Learned Information.

[31] Denis Shestakov. Current challenges in web crawling. In *ICWE*, pages 518–521, 2013.

[32] Vladislav Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *In Proc. of the Int. Conf. on Data Engineering*, pages 357–368, 2002.

[33] Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theor. Comput. Sci.*, 270(1-2):843–856, 2002.