BUbiNG: Massive Crawling for the Masses

PAOLO BOLDI, ANDREA MARINO, MASSIMO SANTINI, and SEBASTIANO VIGNA,
Dipartimento di Informatica, Università degli Studi di Milano, Italy

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 systems, and, at the same time, scales linearly with the amount of resources available. This article aims at filling this gap, through the description of BUbiNG, our next-generation web crawler built upon the authors' experience with UbiCrawler [9] and on the last ten years of research on the topic. BUbiNG is an open-source Java fully distributed crawler; a single BUbiNG agent, using sizeable hardware, can crawl several thousand pages 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 to achieve very high throughput.

CCS Concepts: • **Information systems** → **Web crawling**; *Page and site ranking*; • **Computer systems organization** → *Peer-to-peer architectures*;

Additional Key Words and Phrases: Web crawling, distributed systems, centrality measures

ACM Reference format:
Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2018. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web* 12, 2, Article 12 (June 2018), 26 pages.
https://doi.org/10.1145/3160017

1 INTRODUCTION

A *web crawler* (sometimes also known as a *(ro)bot* or *spider*) is a tool that downloads systematically a large number of web pages starting from a seed. Web crawlers are, of course, used by search engines, but also by companies selling "Search-Engine Optimization" services, by archiving projects such as the Internet Archive, by 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 basic inner working of a crawler is surprisingly simple from a theoretical viewpoint: it is a form of graph traversal (e.g., a breadth-first visit). Starting from a given *seed* of URLs, the set of associated pages is downloaded, their content is parsed, and the resulting links are used iteratively to collect new pages.

The authors were supported by the EU GA 288956 under EU-FET Grant GA 288956 GA 288956 "NADINE".
Authors' addresses: P. Boldi, Dipartimento di Informatica, Universitá degli Studi di Milano, via Comelico 39, Milano, MI, 20135, Italy; email: paolo.boldi@unimi.it; A. Marino, Dipartimento di Informatica, Universitá degli Studi di Milano, via Comelico 39, Milano, MI, 20135, Italy; email: andreamarino.it@gmail.com; M. Santini, Dipartimento di Informatica, Universitá degli Studi di Milano, via Comelico 39, Milano, MI, 20135, Italy; email: massimo.santini@unimi.it; S. Vigna, Dipartimento di Informatica, Universitá degli Studi di Milano, via Comelico 39, Milano, MI, 20135, Italy; email: vigna@acm.org.

https://doi.org/10.1145/3160017

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 article, we describe the design and implementation of BUbiNG, our new web crawler built upon our experience with UbiCrawler [9] and on the last 10 years of research on the topic.[1] 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 Apache License 2.0.
— 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, threads that 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 the overall design and makes several data structures more efficient.
— It is very fast: on a 64-core, 64GB workstation, it can download hundreds of millions of pages at more than 10,000 pages per second, respecting politeness both by host and by IP, analyzing, compressing, and storing more than 160MB/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, which can further be combined with standard Boolean-algebra operators.
— It fully respects the robot exclusion protocol, a *de facto* standard that well-behaved crawlers are expected to obey.
— It guarantees that politeness constraints are satisfied both at the host and the IP level, i.e., that any two consecutive data requests to the same host (name) 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.
— It aims (in its default configuration) at a breadth-first visit in order to collect pages in a more predictible and principled manner. To reach this goal, it uses a best-effort approach to balance download speed, the restrictions imposed by politeness, and the speed differences between hosts. In particular, it guarantees that, host-wise, the visit is an exact breadth-first visit.

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

[1] A preliminary poster appeared in Ref. [10].

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 article.

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 MOTIVATION

There are four main reasons why we decided to design BUbiNG as we described above.

Principled sampling. Analyzing the properties of the web graph has proven to be an elusive goal. A recent large-scale study [30] has shown, once again, that many alleged properties of the web are actually due to crawling and parsing artifacts instead. By creating an open-source crawler that enforces a breadth-first visit strategy, altered by politeness constraints only, we aim at creating web snapshots providing more reproducible results. While breadth-first visits have their own artifacts (e.g., they can induce an apparent in-degree power law *even on regular graphs* [5]), they are a principled approach that has been widely studied and adopted. A more detailed analysis, like spam detection, topic selection, and so on, can be performed offline. A focused crawling activity can actually be *detrimental* to the study of the web, which should be sampled "as it is."

Coherent time frame. Developing a crawler with speed as a main goal might seem restrictive. Nonetheless, for the purpose of studying the web, speed is essential, as gathering large snapshots over a long period of time might introduce biases that would be very difficult to detect and undo.

Pushing hardware to the limit. BUbiNG is designed to exploit hardware to its limits by carefully removing bottlenecks and contention usually present in highly parallel distributed crawlers. As a consequence, it makes performing large-scale crawling possible, even with limited hardware resources.

Consistent crawling and analysis. BUbiNG comes along with a series of tools that make it possible to analyze the harvested data in a distributed fashion, also exploiting multicore parallelism. In particular, the construction of the web graph associated with a crawl uses the *same parser as the crawler*. In the past, a major problem in the analysis of web crawls turned out to be the inconsistency between the parsing as performed at crawl time and the parsing as performed at graph-construction time, which introduced artifacts such as spurious components (see the comments in Ref. [30]). By providing a complete framework that uses the same code both online and offline, we hope to increase the reliability and reproducibility of the analysis of web snapshots.

3 RELATED WORKS

Web crawlers have been developed since the very birth of the web. The first-generation crawlers date back to the early 90s: World Wide Web Worm [29], RBSE spider [21], MOMspider [24], and WebCrawler [37]. 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 Module crawler [15] of the Internet Archive (a non-profit corporation aiming to keep large archival-quality historical records of the world wide web) and the first generation of the Google crawler [13]. This generation of spiders was able to download efficiently tens of millions of pages. At the beginning of 2000, the scalability, extensibility, and distribution of crawlers became a key design point: this was the case of the Java crawler Mercator [35] (the distributed version of Ref. [25]), Polybot [38], IBM

WebFountain [20], and UbiCrawler [9]. 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 Ref. [27]. 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 about the main issues in the design of crawlers, we refer the reader to Refs [32] and [36].

3.1 Open-source Crawlers

Although web crawlers have been around for twenty years by now (since the spring of 1993, according to Ref. [36]), 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` or mngoGoSearch, to cite a few) do not (and are not designed to) scale to download more than few thousands or tens of thousands of pages. They can be useful to build an intranet search engine, but not for web-scale experiments.

Heritrix [2, 33] 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 [1] and it has been since actively developed. Heritrix (available under the Apache license), although it is of course multi-threaded, is a single-machine crawler, which is one of the main hindrances 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 the Internet Archive proposed, implemented (in Heritrix), and fostered a standard format for archiving web content, called WARC (Web ARChive), that is now an ISO standard [4] and that BUbiNG is also adopting for storing the downloaded pages.

Nutch [26] 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 a 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,[2] 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]. To do this, they used a Hadoop cluster of 100 machines [16], so their real throughput was of about 7.55 pages/s *per machine*. This poor performance 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[3] fashion. It should not be surprising that using a modern job-distribution framework like BUbiNG does increase the throughput by orders of magnitude.

[2]The new ClueWeb12 dataset was collected using Heritrix, instead: five instances of Heritrix, running on five Dell PowerEdge R410, were run for 3 months, collecting 1.2 billion pages. The average speed was of about 38.6 pages per second per machine.

[3]In theory, Hadoop may perform the prioritization, de-duplication, and distribution tasks *while* the crawler itself is running, but this choice would make the design very complex and we do not know of any implementation that chose to follow this approach.

4 ARCHITECTURE OVERVIEW

BUbiNG stands on a few architectural choices, which in some cases contrast the common folklore wisdom. We took our decisions after carefully comparing and 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 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* [31] 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 [9], 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 fetching threads enqueue buffers of fetched data and wait for a parsing thread to analyze them. Parsing threads poll the result list using an exponential back-off scheme, perform actions such as parsing and link extraction, and signal back to the fetching thread 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. 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.

In fact, every time a URL is discovered, it is checked first against a high-performance approximate LRU (Least Recently Used) cache (kept in core memory) containing 128-bit fingerprints: more than 90% of the URLs discovered are discarded at this stage. The cache avoids that frequently found URLs put the sieve under stress, and it has also another important goal: it avoids that frequently found URLs assigned to another agent are retransmitted many times.

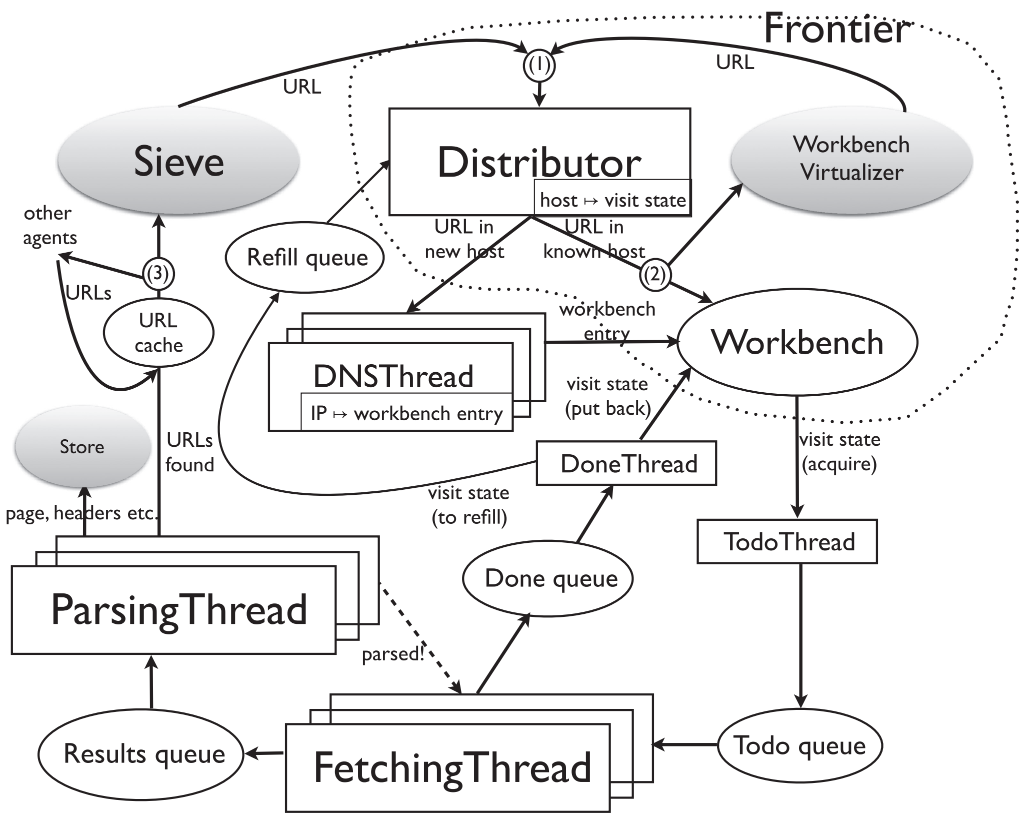

Fig. 1. Overview of the architecture of a BUbiNG agent. Ovals represent data structures, whereas rectangles represent threads (or sets of threads); we use a gray background for data structures that are partly on-disk, as explained in the text (the store can in fact be implemented in different ways, although it will typically be on-disk). The numbered circles are explained in the text.

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*,[4] 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 *visit states*, one for each host currently being crawled: each visit state contains a FIFO queue of the next URLs to be retrieved from the associated host, and some information about politeness. This information makes it possible for the workbench to check in constant time which hosts can be accessed for download without violating the politeness constraints. Note that to attain the goal of several thousand downloaded pages per second without violating politeness constraints, it is necessary to keep track of the visit states 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 *to-do queue* by a suitable thread. Fetching threads poll the to-do queue with an exponential backoff, fetch resources from the retrieved visit state[5] by accessing its URL queue and then put it back onto the workbench. Note that we expect that once a large crawl has started, the

[4]Note that "frontier" is also a name commonly used for the *set* of URLs that have been discovered, but not yet crawled. We use the same term for the data structure that manages them.
[5]Possibly multiple resources on a single TCP connection using the "keepalive" feature of HTTP 1.1.

to-do queue will never be empty, so fetching threads will never have to wait. Most of the design challenges of the frontier components are actually geared toward avoiding that fetching threads ever wait on an empty to-do queue.

The main active component of the frontier is the *distributor*: it is a high-priority thread that processes URLs coming out of the sieve (and that 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 onto the workbench.

Since, however, breadth-first visit queues grow exponentially, and the workbench can use only a fixed amount of in-core memory, it is necessary to *virtualize* a part of the workbench, 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 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 independently compressing pages and using concurrent primitives to pass compressed data to a flushing thread.

In the next sections, we review in more detail the components we just introduced. This time, we use a bottom-up strategy, detailing first lower-level data structures that can be described and understood separately, and then going up to the distributor.

4.1 The Sieve

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 once. Sieves of URLs (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 URLs to a disk file, guaranteeing that a URL is not returned multiple times requires *ad-hoc* data structures—a standard dictionary implementation 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 Ref. [25] (hence its name).[6] Each URL known to the sieve is stored as a 64-bit hash in a sorted disk file. Every time a new URL is enqueued, its hash is stored in an in-memory array, and the URL is saved in an auxiliary file. When the array is full, it is sorted (indirectly, so to keep track of the original order, too) and compared with the set of 64-bit hashes known to the sieve. The auxiliary file is then scanned, and previously unseen URLs are stored for later examination. All these operations require only sequential access to all files involved, and the sizing of the array is based on the amount of in-core memory available. Note that the output order is guaranteed to be the same of the input order (i.e., new URLs will be examined in the order of their first appearance).

[6]Observe that different hardware configurations (e.g., availability of large SSD disks) might make a different sieve implementation preferable.

A generalization of the idea of a sieve, with the additional possibility of associating values with the elements, is the Disk Repository with Update Management (DRUM) structure used by IRLBot and described in Ref. [27]. A DRUM provides additional operations to retrieve or update the values associated with the 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 that are an order of magnitude larger than what the Mercator sieve would allow using the same quantity of in-core memory. However, to do so, DRUM must sacrifice breadth-first order: due to the inherent randomization of the way keys are placed 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 Ref. [27] about the properties of DRUM is unavoidably bound to the single-agent approach of IRLBot: for example, the authors conclude that a URL cache is not useful to reduce the number of insertions in the DRUM, but the same cache reduces significantly network transmissions. Based on our experience, 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 adopt an explicit queue and a *Bloom filter* [8] to remember enqueued URLs. Albeit popular, this choice has no theoretical guarantees: 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 number of *discovered* URLs, and this number is essential in sizing the Bloom filter. If the discovered URLs are significantly more than expected, an unpredictable number of pages will 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.

We remark that 64-bit fingerprints can give rise to collisions with significant probability when crawling more than a few hundred billion URLs per agent (the number of agents has no impact on collisions). It is easy to increase the number of bits in the fingerprints, at the price of a proportionally higher core-memory usage.

Finally, in particular for larger fingerprints, it can be fruitful to compress the file storing sorted fingerprints using succinct data structures such as the Elias–Fano representation of monotone sequences [22].

4.2 The Workbench

The *workbench* is an in-memory data structure that contains the next URLs to be visited. 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. It is a significant improvement over IRLBot's two-queue approach [27], as it can detect in constant time whether a URL is ready for download without violating politeness limits.

First of all, URLs associated with a specific host[7] 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.

[7]Every URL is made [7] by a scheme (also popularly called "protocol"), an authority (a host, optionally a port number, and perhaps 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 article we will use the more common word "host" to refer to it.

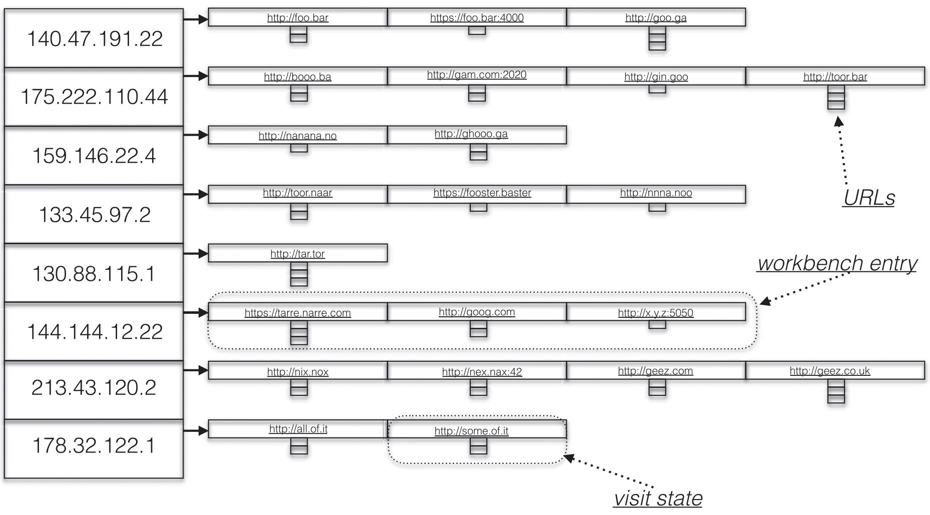

Fig. 2. The *workbench* is a priority queue (the vertical queue on the left), whose elements (the workbench entries) are associated with IP addresses. Each *workbench entry* is itself a priority queue (the horizontal queues appearing on the right), whose elements (the visit states) are associated with a host (more precisely: a scheme and an authority). Each *visit state* contains a standard FIFO queue (the small piles of blocks below each visit state), whose elements are the URLs to be visited for that host.

Visit states are further grouped into *workbench entries* based on their IP address; 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 associated to that IP address yet) or in an existing one.

A workbench entry contains a queue of visit states (associated with the same IP) prioritized by their `next-fetch` field, and 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 (see Figure 2). The two `next-fetch` fields are updated each time a fetching thread completes its access to a host by setting them to the current time plus the required host/IP politeness delays.

Note that due to our choice of priorities, there is a host that can be visited without violating host or IP politeness constraints if and only if the host associated with 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.

Therefore, 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. Both removals happen in logarithmic time (in the number of visit states). 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, access to the workbench is sandwiched between two lock-free queues: a *to-do queue* and a *done queue*. Those queues are managed by two high-priority threads: the *to-do thread* extracts visit states whose hosts can be visited without violating politeness constraints and moves them to the to-do queue, where they will be retrieved by a fetching thread; on the other side, the *done thread* picks the visit states after they have been used by a fetching thread and puts them back onto the workbench.

The purpose of this setup is to avoid contention by thousands of threads on a relatively slow structure (as extracting and inserting elements in the workbench takes logarithmic time in the number of hosts). Moreover, it makes the number of visit states that are ready for downloads easily measurable: it is just the size of the to-do queue. The downside is that, in principle, using very skewed per-host or per-IP politeness delays might cause the order of the to-do queue not to reflect the actual priority of the visit states contained therein; this phenomenon might push the global visit order further away from a breadth-first visit.

4.3 Fetching Threads

A *fetching thread* is a very simple thread that iteratively extracts visit states from the to-do queue. If the to-do queue is empty, a standard exponential back-off procedure is used to avoid polling the list too frequently, but the design of BUbiNG aims at keeping the to-do queue non-empty 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 associated *fetch data* instance in which the downloaded data are buffered. Fetch data instances include a transparent buffering method that keeps a fixed amount of data in memory 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 resource has been fetched, the fetch data is put in the *results* queue so that one of the parsing threads can parse it. Once this process is over, the parsing thread sends a signal back so that the fetching thread is able to start working on a new URL. Once a fetching thread has to work on a new visit state, it puts the current visit state in a *done queue*, from which it will be dequeued by a suitable thread that will then 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 fetched data.

4.4 Parsing Threads

A *parsing thread* iteratively extracts from the result queue the 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 same agent's sieve (circle numbered (3) in Figure 1).

During the parsing phase, a parsing thread computes a digest of the response content. The signature is stored in a Bloom filter [8] and it is used to avoid saving several times the same page (or near-duplicate pages). Finally, the content of the response is saved to the store.

Since two pages are considered (near-)duplicates whether they have the same signature, the digest computation is responsible for content-based duplicate detection. In the case of HTML pages, in order to collapse near-duplicates, some heuristic is used. In particular, a hash fingerprint is computed on a summarized content, which is obtained by stripping HTML attributes, and discarding digits and dates from the response content. This simple heuristic allows, for instance, to collapse pages that differ just for visitor counters or calendars. In a post-crawl phase, there are several more sophisticated approaches that can be applied, like *shingling* [14], *simhash* [18], *fuzzy fingerprinting* [17, 23], and others (e.g., Ref. [28]).

For the sake of description, we will call *duplicate* pages that are (near-)duplicates of some other page previously crawled according to the above definition, while we will call *archetypes* the set of pages that are not duplicates.

Since we are interested in archival-quality crawling, duplicate detection is by default restricted to be intra-site (the digest is initialized with the host name). Different hosts are allocated to different agents, so there is no need for inter-agent detection. Indeed, post-crawl experiments show that, even with relaxing duplicate detection to work inter-site, we would eliminate less than 10% of the pages in the crawls discussed in Section 6 (in fact, 6.5% for `gsh-2015`, 8.6% for `uk-2014`, and 3.3% for `eu-2015`).

4.5 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 address itself is new), and putting it on the workbench. In our experience, it is essential to run a local recursive DNS server to avoid the bottleneck caused by an external server.

Presently, in case a host resolves to mulitple IPs, we pick the first one returned by the DNS resolver. Since the DNS resolver class is entirely configurable, this can be decided by the user (round robin, random, etc.). The default implementation is based on the open-source DnsJava resolver.[8]

4.6 The Workbench Virtualizer

The workbench virtualizer maintains on disk a mapping from hosts to FIFO *virtual queues* of URLs. 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 queues 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. This action is performed by the same thread emptying the done queue (the queue containing the visit states after fetching): as it puts visit states back on the workbench, it selects visit states with URLs on disk but no more URLs on the workbench and puts them on a *refill queue* that will be later read by the distributor.

Initially, we experimented with virtualizers inspired by the BEAST (Budget Enforcement with Anti-Spam Tactics) module of IRLbot [27], although many crucial details of their implementation were missing (e.g., the treatment of HTTP and connection errors); moreover, due to the static once-for-all distribution of URLs among a number of physical on-disk queues, it was impossible to guarantee adherence to a breadth-first visit in the face of unpredictable network-related faults.

Our second implementation was based on the Berkeley DB, a key/value store that is also used by Heritrix. While extremely popular, Berkeley DB is a general-purpose storage system, and in particular, in Java, it has a very heavy load in terms of object creation and corresponding garbage

[8] http://www.xbill.org/dnsjava/.

collection. While providing in-principle services like URL-level prioritization (which was not one of our design goals), Berkeley DB was soon detected to be a serious bottleneck in the overall design.

We thus decided to develop an *ad-hoc* virtualizer oriented toward breadth-first visits. We borrowed from Berkeley DB the idea of writing data in log files that are periodically collected, but we decided to rely on memory mapping to lessen the I/O burden.

In our virtualizer, on-disk URL queues are stored in log files that are memory mapped and transparently thought of as a contiguous memory region. Each URL stored on disk is prefixed with a pointer to the position of the next URL for the same host. Whenever we append a new URL, we modify the pointer of the last stored URL for the same host accordingly. A small amount of metadata associated with each host (e.g., the head and tail of its queue) is stored in main memory.

As URLs are dequeued to fill the workbench, part of the log files become free. When the ratio between the used and allocated space goes below a threshold (e.g., 50%), a garbage-collection process is started. Due to the fact that URLs are always appended, there is no need to keep track of free space: we just scan the queues in order of first appearance in the log files and gather them at the start of the memory-mapped space. By keeping track (in a priority queue) of the position of the next URL to be collected in each queue, we can move items directly to their final position, updating the queue after each move. We stop when enough space has been freed, and delete the log files that are now entirely unused.

Note that most of the activity of our virtualizer is caused by appends and garbage collections (reads are a lower-impact activity that is necessarily bound by the network throughput). Both activities are highly localized (at the end of the currently used region in the case of appends and at the current collection point in the case of garbage collections), which makes a good use of the caching facilities of the operating system.

4.7 The Distributor

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

As the crawl proceeds, URLs get accumulated in visit states at different speeds, both because hosts have different responsiveness and because websites have different sizes and branching factors. Moreover, the workbench has a (configurable) limit size that cannot be exceeded, since one of the central design goals of BUbiNG is that the amount of main memory occupied cannot grow unboundedly in the number 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.

The *front* of a crawl, at any given time, is the number of visit states that are ready for download respecting the 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 ever has to wait.* To attain this goal, the distributor enlarges dynamically the *required front size*, which is an estimate of the number of hosts that must be visited in parallel to keep all fetching threads busy: each time a fetching thread has to wait, albeit the current front size is larger than the current required front size, the latter is increased. After a warm-up phase, the required front size stabilizes to a value that depends on the kind of hosts 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.

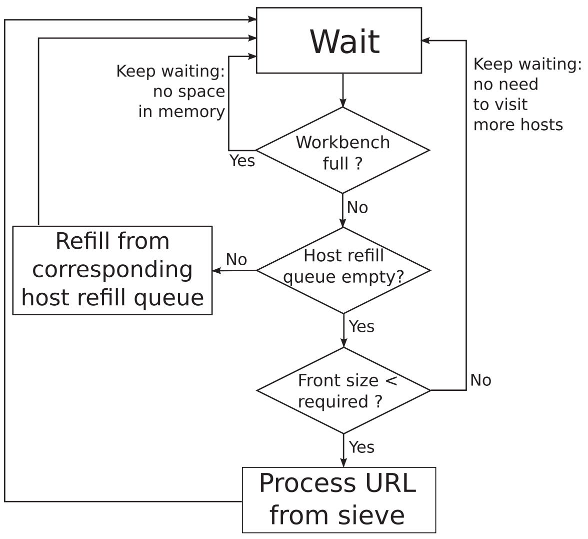
Wait
Keep waiting: no space in memory
Keep waiting: no need to visit more hosts
Workbench full ?
Yes
No
Refill from corresponding host refill queue
Host refill queue empty?
No
Yes
Front size < required ?
No
Yes
Process URL from sieve

Fig. 3. How the distributor interacts with the sieve, the workbench, and the workbench virtualizer.

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 more URLs 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 call for reading URLs from the sieve to discover new hosts.

The distributor privileges refilling the queues of the workbench using URLs from the virtualizer because this makes the visit closer to an exact breadth-first. However, if no refill has to be performed and the front is not large enough, the distributor will read from the sieve, hoping to find new hosts to make the front larger.

When the distributor reads a URL from the sieve, the URL can either be put in the workbench (circle numbered (2) in Figure 1) or written in a virtual queue, depending on whether there are already URLs on disk for the same host, and on the number of URLs per IP address that should be in the workbench to keep it full, but not overflowing, when the front is of the required size.

4.8 Configurability

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.

Different filters apply to different types of objects: a *prefetch* filter is one that can be applied to URLs (typically: to decide whether a URL should be scheduled for a later visit, or should be fetched); a *postfetch* filter is one that can be applied to fetched responses and decides whether to do something with a response (typically: whether to parse it, to store it, etc.).

Table 1. Comparison Between BUbiNG and the Main Existing Open-Source Crawlers

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	4
Heritrix (*in vitro*)	1	115	370	370	4.5	4.5
IRLBot	1	6,380	1,790	1,790	40	40
BUbiNG (iStella)	1	500	3,700	3,700	154	154
BUbiNG (*in vitro*)	4	1,000	40,600	10,150	640	160

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 [27], although the latter is not open source. Note that ClueWeb09 was gathered using a heavily customized version of Nutch.

4.9 URL Normalization

BURL (a short name for "BUbiNG URL") is the class responsible for parsing and normalizing URLs found in web pages. The topic of parsing and normalization is much more involved than one might expect—very recently, the failure in building a sensible web graph from the ClueWeb09 collection stemmed in part from the lack of suitable normalization of the URLs involved. BURL takes care of fine details such as escaping and de-escaping (when unnecessary) of non-special characters and case normalization of percent-escape.

4.10 Distributed Crawling

BUbiNG crawling activity can be distributed by running several agents over multiple machines. Similarly to UbiCrawler [9], all agents are identical instances of BUbiNG, without any explicit leadership: all data structures described above are part of each agent.

URL assignment to agents 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 local, each agent will be responsible for the large majority of URLs found in a typical HTML page [36]. Assignment of hosts to agents is by default performed using *consistent hashing* [9].

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 (User Datagram Protocol). More sophisticated communications between the agents rely on the TCP (Transmission Control Protocol)-based JMX (Java Management Extension) Java standard remote-control mechanism, which exposes most of the internal configuration parameters and statistics. Almost all crawler structures are indeed modifiable at runtime.

5 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 different 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 an HTTP proxy,[9] simulating network connections toward the web and generating fake HTML pages

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

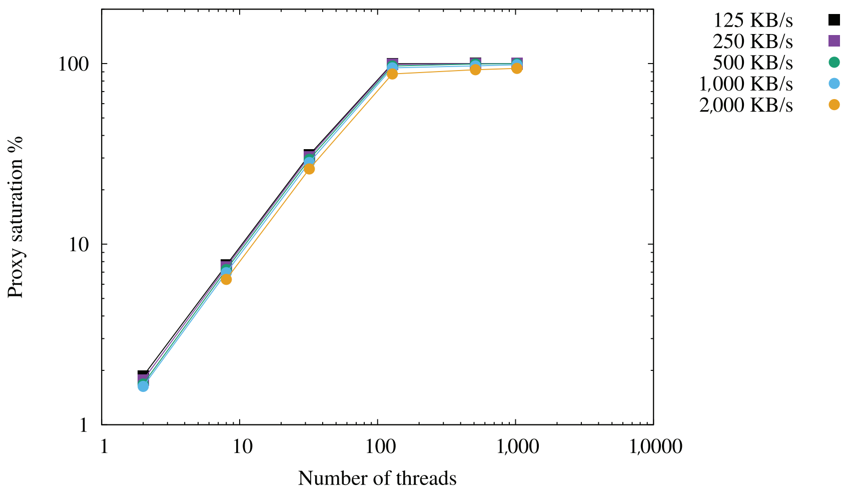

Fig. 4. The saturation of a 100-threads proxy that simulates different download speeds (per thread) using a different number of fetching threads. Note the increase in speed until the plateau, which is reached when the proxy throughput is saturated.

(with a configurable behavior that includes delays, protocol exceptions, etc.); and another batch of experiments was performed *in vivo*.

5.1 *In Vitro* Experiments: BUbiNG

To verify the robustness of BUbiNG when varying some basic parameters, such as the number of fetching threads or the IP delay, we decided to run some *in vitro* simulations on a group of four machines sporting 64 cores and 64GB of core memory. In all experiments, the number of parsing and DNS threads was fixed and set respectively to 64 and 10. The size of the workbench was set to 512MB, while the size of the sieve was set to 256MB. We always set the host politeness delay equal to the IP politeness delay. 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 yields a better usage of the network, and hence, a larger number of requests per second, until the bandwidth is saturated. The results of this experiment are shown in Figure 4 and have been obtained using a 100-thread proxy and a politeness delay of 8 seconds. Each thread emits data at the speed shown in the legend, and, as we remarked previously, the proxy generates also a fraction of very slow pages and network errors to simulate a realistic environment.

The behavior visible in the plot tells us that the increase in the number of fetching threads yields a linear increase in network utilization until the available (simulated) bandwidth is reached. At that point, we do not see any decrease in the throughput, witnessing the fact that our infrastructure does not cause any hindrance to the crawl.

Politeness. Our second *in vitro* experiment tests what happens when one increases the amount of politeness, as determined by the IP delay, depending on the amount of threads. We plot BUbiNG's throughput as the IP delay (hence the host delay) increases in Figure 5 (middle): to maintain the same throughput, the front size (i.e., the number of hosts being visited in parallel, shown in Figure 5, top) must increase, as expected. The front grows almost linearly with the number of threads until the proxy bandwidth is saturated. In the same figure (middle), we show that the average throughput is independent from the politeness (once again, once we saturate the proxy

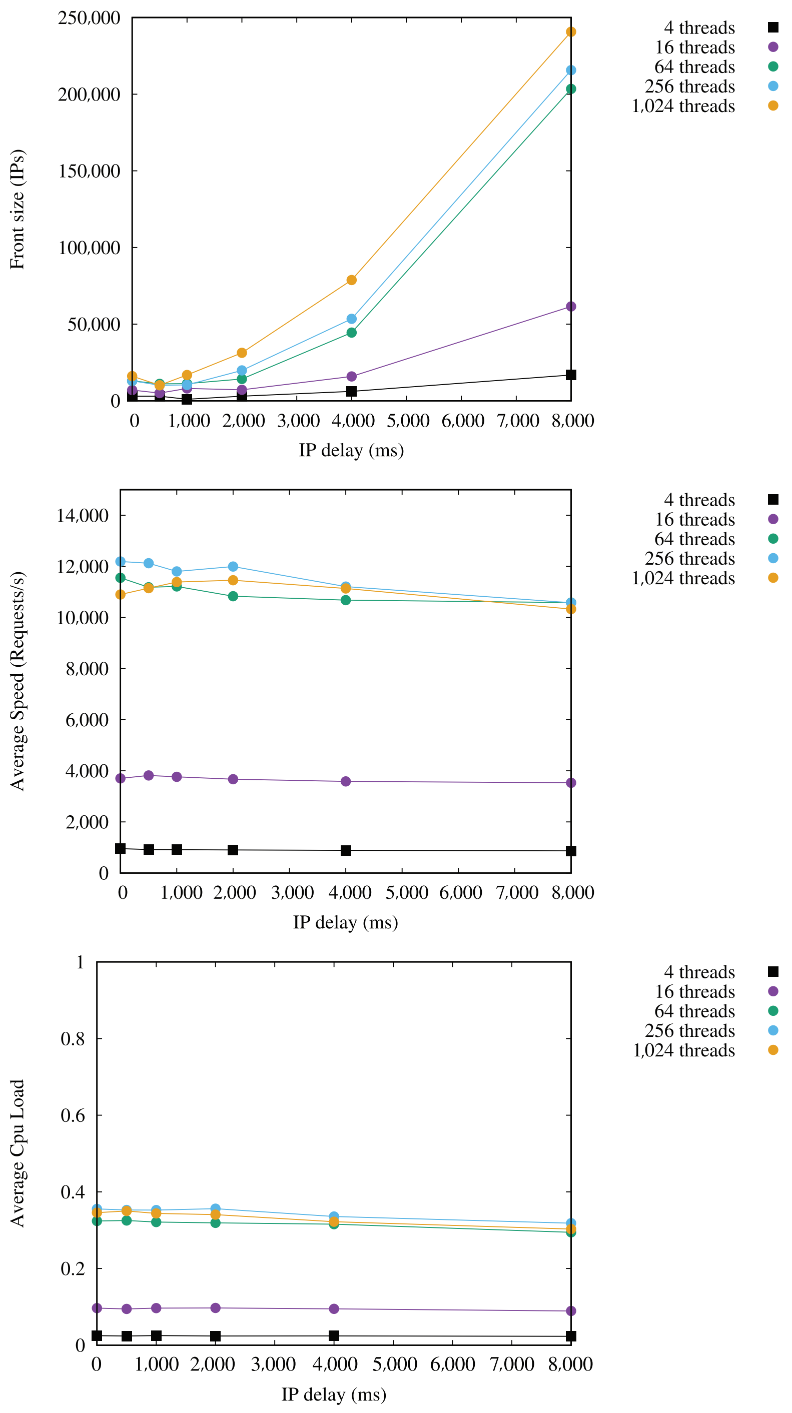

Fig. 5. The average size of the front, the average number of requests per second, and the average CPU load with respect to the IP delay (the host delay is set to eight times the IP delay). Note that the front adapts to the growth of the IP delay, and that the number of fetching threads has little influence once we saturate the proxy bandwidth.

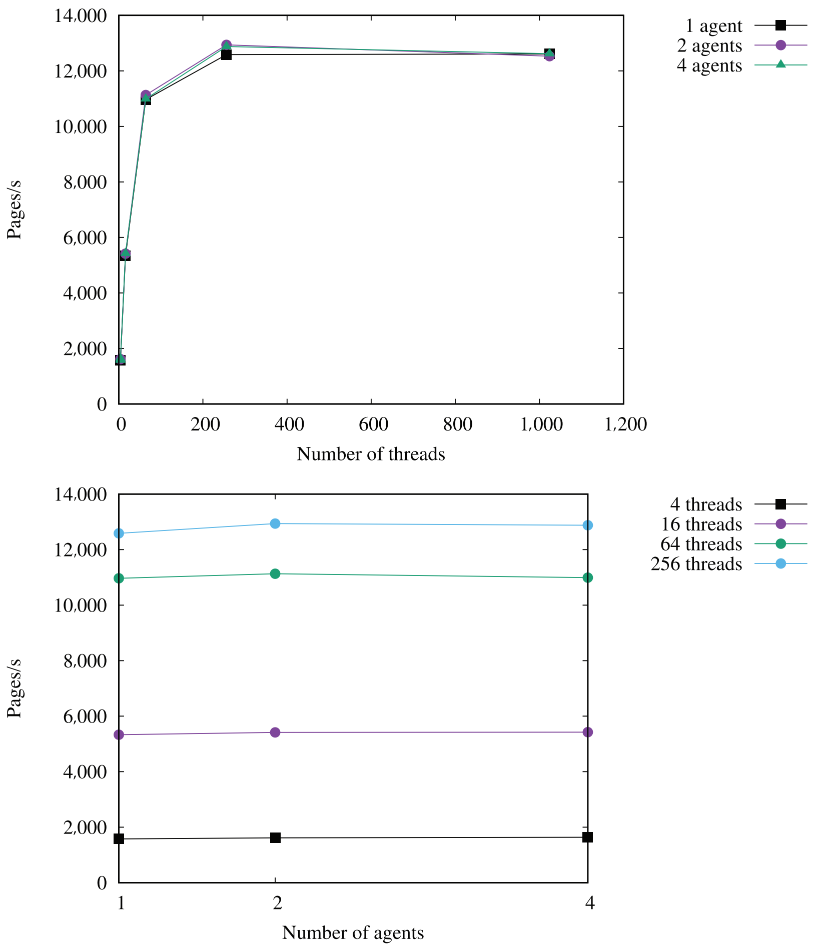

Fig. 6. The average number of pages per second per agent using many agents, with varying number of fetching threads per agent.

bandwidth, the throughput becomes stable with respect to the number of threads), and the same is true of the CPU load (Figure 5, bottom). This is a consequence of BUbiNG dynamically modifying the number of hosts in the front.

Multiple agents. A similar experiment was run with multiple crawling agents (1, 2, 4) still experimenting with a varying number of fetching threads per agent. The results are shown in Figure 6. The average speed is not influenced by the number of agents (upper plot), but only by the number of threads.

Testing for bottlenecks: No I/O. Finally, we wanted to test whether our lock-free architecture was actually able to sustain a very high parallelism. To do so, we ran a no-I/O test on a 40-core workstation. The purpose of the test was to stress the computation and contention bottlenecks in absence of any interference from I/O: thus, input from the network was generated internally using the same logic of our proxy, and while data was fully processed (e.g., compressed), no actual storage was performed. After 100 million pages, the average speed was 16,000 pages/s (peak

22,500) up to 6,000 threads. We detected the first small decrease in speed (15,300 pages/s, peak 20,500) at 8,000 threads, which we believe is to be expected due to increased context switch and Java garbage collection. With this level of parallelism, our lock-free architecture is about 30% faster in terms of downloaded pages (with respect to a version of BUbiNG in which threads access directly the workbench). The gap widens as the threads increase and the politeness policy gets more strict, as keeping all threads busy requires enlarging the front, and thus, the workbench: a larger workbench implies logarithmically slower operations, and thus, more contention. Of course, if the number of threads is very small the lock-free structure is not useful, and in fact, the overhead of the "sandwich" can slightly slow down the crawler.

5.2 *In Vitro* Experiments: Heritrix

To provide a comparison of BUbiNG with another crawler in a completely equivalent setting, we ran a raw-speed test using Heritrix 3.2.0 on the same hardware as in the BUbiNG raw-speed experiment, always using a proxy with the same setup. We configured Heritrix to use the same amount of memory, 20% of which was reserved for the Berkeley DB cache. We used 1,000 threads, locked the politeness interval to 10 seconds regardless of the download time (by default, Heritrix uses an adaptive scheme), and enabled content-based duplicate detection.[10] The results obtained will be presented and discussed in Section 5.4.

5.3 *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 1Gb/s geographic link, so, in fact, we were not initially able to perform any test in which the network was not capping the crawler. Finally, iStella, an Italian commercial search engine, provided us with a 48-core, 512GB RAM with a 2Gb/s link. The results are extremely satisfactory: in the iStella experiment we were able to keep a steady download speed of 1.2Gb/s using a single BUbiNG agent crawling the `.it` domain. The overall CPU load was about 85%.

5.4 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 throughput than crawling just HTML, since HTML pages are often rendered dynamically, sometimes causing a significant delay, whereas most other types are served statically. The crawling policy also has a huge influence on the throughput: prioritizing by indegree (as IRLBot does [27]) or alternative importance measure shifts most of the crawl on sites hosted on powerful servers with large-bandwidth connection. We remind that BUbiNG aims at archival-quality crawling, to which such a significant departure from the natural (breadth-first) crawl order would be extremely detrimental.

Ideally, crawlers should be compared on a crawl with a 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 Table 1, we gather some evidence of the excellent performance of BUbiNG. Part of the data is from the literature, and part has been generated during our experiments.

First of all, we report performance data for Nutch and Heritrix from the recent crawls made for the ClueWeb project (ClueWeb09 and ClueWeb12). The figures are those available in Ref. [16] along with those found in Ref. [3] and http://boston.lti.cs.cmu.edu/crawler/crawlerstats.html. Notice that

[10]We thank Gordon Mohr, one of the authors of Heritrix, for suggesting how to configure it for a large workstation.

the data we have about those collections are sometimes slightly contradictory (we report the best figures). The comparison with the ClueWeb09 crawl is somewhat unfair (the hardware used for that dataset was "retired search-engine hardware"), whereas the comparison with ClueWeb12 is more unbiased, as the hardware used was more recent. We report the throughput declared by IRLBot [27], too, albeit the latter is not open source and the downloaded data is not publicly available.

Then, we report experimental *in vitro* data about Heritrix and BUbiNG obtained, as explained in the previous section, using the same hardware, a similar setup, and a HTTP proxy generating web pages.[11] These figures are the ones that can be compared more appropriately. Finally, we report the data of the iStella experiment.

The results of the comparison show quite clearly that the speed of BUbiNG is several times that of IRLBot and one to two orders of magnitude larger than that of Heritrix or Nutch.

All in all, our experiments show that BUbiNG's adaptive design provides a very high throughput, in particular, when a strong politeness is desired: 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 THREE DATASETS

As a stimulating glimpse into the capabilities of BUbiNG to collect interesting datasets, we describe the main features of three snapshots collected with different criteria. All snapshots contain about one billion unique pages (the actual crawls are significantly larger, due to duplicates).

— `uk-2014`: a snapshot of the `.uk` domain, taken with a limit of 10,000 pages per host starting from the BBC (British Broadcasting Corporation) website.
— `eu-2015`: a "deep" snapshot of the national domains of the European Union, taken with a limit of 10,000,000 pages per host starting from `europa.eu`.
— `gsh-2015`: a general "shallow" worldwide snapshot, taken with a limit of 100 pages per host, always starting from `europa.eu`.

The `uk-2014` snapshot follows the tradition of our laboratory of taking snapshots of the `.uk` domain for linguistic uniformity, and to obtain a regional snapshot. The second and third snapshot aims at exploring the difference in the degree distribution and in-website centrality in two very different kinds of data-gathering activities. In the first case, the limit on the pages per host is so large that, in fact, it was never reached; it is a quite faithful "snowball sampling" due to the breadth-first nature of BUbiNG's visits. In the second case, we aim at maximizing the number of collected hosts by downloading very few pages per host. One of the questions we are trying to answer using the latter two snapshots is: how much is the indegree distribution dependent on the cardinality of sites (root pages have an indegree usually at least as large as the site size), and how much is it dependent on inter-site connections?

The main data, and some useful statistics about the three datasets, are shown in Table 2. Among these, we have the average number of links per page (average outdegree) and the average number of links per page whose destination is on a different host (average external outdegree). Moreover, concerning the graph induced by the pages of our crawls, we also report the average distance, the harmonic diameter (e.g., the harmonic mean of all the distances), and the percentage of reachable pairs of pages in this graph (e.g., pairs of nodes (x, y) for which there exists a directed path from x to y).

[11]Note that, with the purpose of stress testing the crawler internals, our HTTP proxy generates fairly short pages. This feature explains the wildly different ratio between MB/s and resources/s when looking at *in vitro* and *in vivo* experiments.

Table 2. Basic Data

	uk-2014	gsh-2015	eu-2015
Overall	1,477,881,641	1,265,847,463	1,301,211,841
Archetypes	787,830,045	1,001,310,571	1,070,557,254
Avg. content length	56,039	32,526	57,027
Avg. outdegree	105.86	96.34	142.60
Avg. external outdegree	25.53	33.68	25.34
Avg. distance	20.61	12.32	12.45
Harmonic diameter	24.63	14.91	14.18
Reachable pairs	67.27%	80.29%	85.14%

Table 3. Comparison of HTTP Statuses with Those from Ref. [6], Reporting IRLBot Data [27] and Mercator Data from Ref. [34]

	uk-2014		gsh-2015		eu-2015			
	All	Arch.	All	Arch.	All	Arch.	IRLBot	Mercator
2XX	85.56%	81.11%	86.41%	86.39%	90.34%	87.4%	86.79%	88.50%
3XX	11.02%	11.6%	10.53%	12.57%	7.18%	11.62%	8.61%	3.31%
4XX	2.74%	6.31%	2.58%	0.88%	2.24%	0.84%	4.11%	6.46%
5XX	0.67%	0.98%	0.48%	0.16%	0.24%	0.14%	0.35%	—
Other	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	0.12%	1.73%

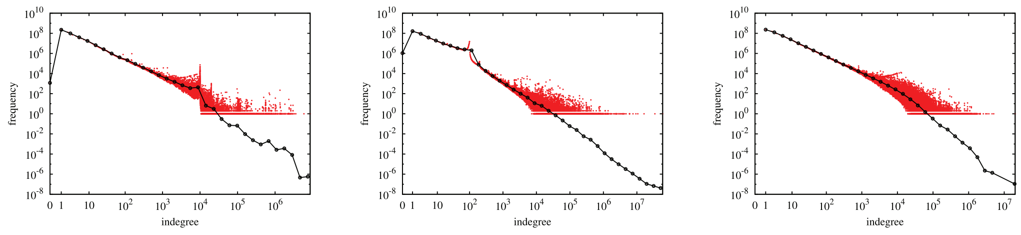

Fig. 7. Indegree plots for uk-2014, gsh-2015, and eu-2015 (degree/frequency plots with Fibonacci binning).

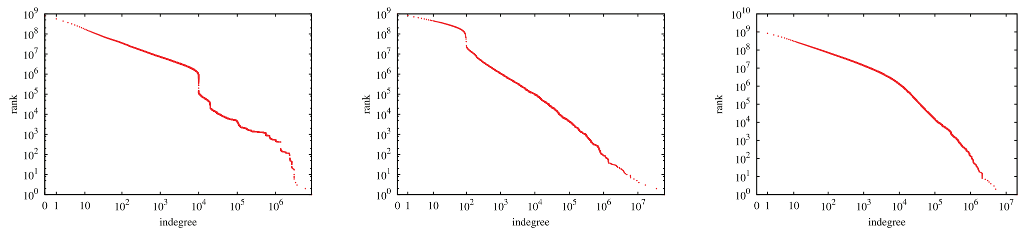

Fig. 8. Indegree plots for uk-2014, gsh-2015, and eu-2015 (cumulative degree/rank plots).

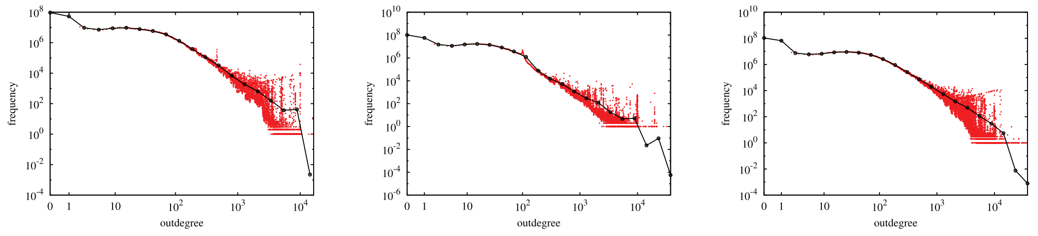

Fig. 9. Outdegree plots for uk-2014, gsh-2015, and eu-2015 (degree/frequency plots with Fibonacci binning).

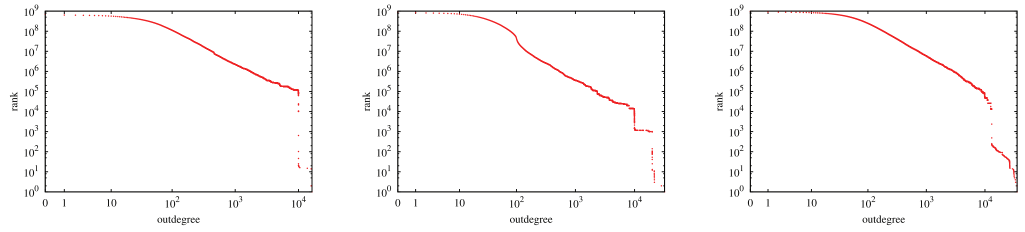

Fig. 10. Outdegree plots for uk-2014, gsh-2015, and eu-2015 (cumulative degree/rank plots).

6.1 Degree Distribution

The indegree and outdegree distributions are shown in Figures 7, 8, 9, and 10. We provide both a degree-frequency plot decorated with *Fibonacci binning* [39], and a degree-rank plot[12] to highlight with more precision the tail behavior.

From Table 2, we can see that pages at low depth tend to have less outlinks, but more external links than inner pages. The content is similarly smaller (content lives deeper in the structure of websites). Not surprisingly, moreover, pages of the shallow snapshot are closer to one another.

The most striking feature of the indegree distribution is an answer to our question: *the tail of the indegree distribution is, by and large, shaped by the number of intra-host inlinks of root pages*. This is very visible in the uk-2014 snapshot, where limiting the host size at 10,000 causes a sharp step in the degree-rank plot; and the same happens at 100 for gsh-2015. But what is maybe even more interesting is that the visible curvature of eu-2015 is almost absent from gsh-2015. Thus, if the latter (being mainly shaped by inter-host links) has some chance of being a power-law, as proposed by the class of "richer get richer" models, the former has none. Its curvature clearly shows that the indegree distribution is not a power-law (a phenomenon already noted in the analysis of the Common Crawl 2012 dataset [30]): fitting it with the method by Clauset, Shalizi, and Newman [19] gives a p-value $< 10^{-5}$ (and the same happens for the top-level domain graph).

6.2 Centrality

Tables 4, 5, and 6 report centrality data about our three snapshots. Since the page-level graph gives rise to extremely noisy results, we computed the *host graph* and the *top-level domain* graph. In the first graph, a node is a host, and there is an arc from host x to host y if some page of x points to some page of y. The second graph is built similarly, but now a node is a *set of hosts* sharing the same *top-level domain* (TLD). The TLD of a URL is determined from its host using the *Public*

[12]Degree-rank plots are the numerosity-based discrete analogous of the complementary cumulative distribution function of degrees. They give a much clearer picture than frequency dot plots when the data points are scattered and highly variable.

Table 4. Most Relevant Hosts and TLD of uk-2014 by Different Centrality Measures

Indegree		PageRank		Harmonic centrality	
Host Graph					
postcodeof.co.uk	726,240	postcodeof.co.uk	0.02988	postcodeof.co.uk	1,318,833.61
namefun.co.uk	661,692	namefun.co.uk	0.02053	www.google.co.uk	1,162,789.19
www.slovakiatrade.co.uk	128,291	www.slovakiatrade.co.uk	0.00543	www.nisra.gov.uk	1,130,915.01
catalog.slovakiatrade.co.uk	103,991	catalog.slovakiatrade.co.uk	0.00462	www.ons.gov.uk	1,072,752.04
www.quiltersguild.org.uk	93,573	london.postcodeof.co.uk	0.00462	www.bbc.co.uk	1,067,880.20
quiltersguild.org.uk	93,476	www.spanishtrade.co.uk	0.00400	namefun.co.uk	1,057,516.35
www.spanishtrade.co.uk	87,591	catalog.spanishtrade.co.uk	0.00376	www.ordnancesurvey.co.uk	1,043,468.95
catalog.spanishtrade.co.uk	87,562	www.germanytrade.co.uk	0.00346	www.gro-scotland.gov.uk	1,025,953.27
www.germanytrade.co.uk	73,852	www.italiantrade.co.uk	0.00323	www.ico.gov.uk	1,004,464.11
catalog.germanytrade.co.uk	73,850	catalog.germanytrade.co.uk	0.00322	www.nhs.uk	1,003,575.06
TLD Graph					
bbc.co.uk	60,829	google.co.uk	0.00301	bbc.co.uk	422,486.00
google.co.uk	54,262	123-reg-expired.co.uk	0.00167	google.co.uk	419,942.55
www.nhs.uk	22,683	bbc.co.uk	0.00150	direct.gov.uk	375,068.62
direct.gov.uk	20,579	ico.gov.uk	0.00138	parliament.uk	371,941.43
nationaltrust.org.uk	20,523	freeparking.co.uk	0.00093	www.nhs.uk	370,448.34
hse.gov.uk	13,083	ico.org.uk	0.00088	ico.gov.uk	368,878.14
timesonline.co.uk	11,987	website-law.co.uk	0.00087	nationaltrust.org.uk	367,367.47
amazon.co.uk	11,900	hibu.co.uk	0.00085	telegraph.co.uk	364,763.80
parliament.uk	11,622	1and1.co.uk	0.00073	hmrc.gov.uk	364,530.15
telegraph.co.uk	11,467	tripadvisor.co.uk	0.00062	hse.gov.uk	361,314.39

Table 5. Most Relevant Hosts and TLD of eu-2015 by Different Centrality Measures

Indegree		PageRank		Harmonic centrality	
		Host Graph			
www.toplist.cz	174,433	www.myblog.de	0.001227	youtu.be	2,368,004.25
www.radio.de	139,290	www.domainname.de	0.001215	ec.europa.eu	2,280,836.77
www.radio.fr	138,877	www.toplist.cz	0.001135	europa.eu	2,170,916.37
www.radio.at	138,871	www.estranky.cz	0.000874	www.bbc.co.uk	2,098,542.10
www.radio.it	138,847	www.beepworld.de	0.000821	www.spiegel.de	2,082,363.21
www.radio.pt	138,845	www.active24.cz	0.000666	www.google.de	2,061,916.72
www.radio.pl	138,843	www.lovdata.no	0.000519	www.europarl.europa.eu	2,050,110.04
www.radio.se	138,840	www.mplay.nl	0.000490	news.bbc.co.uk	2,046,325.37
www.radio.es	138,839	zl.lv	0.000479	curia.europa.eu	2,038,532.77
www.radio.dk	138,838	www.mapy.cz	0.000472	eur-lex.europa.eu	2,011,251.37
		TLD Graph			
europa.eu	74,129	domainname.de	0.001751	europa.eu	1,325,894.51
e-recht24.de	59,175	toplist.cz	0.000700	youtu.be	1,307,427.57
youtu.be	47,747	e-recht24.de	0.000688	google.de	1,196,817.20
toplist.cz	46,797	mapy.cz	0.000663	bbc.co.uk	1,194,338.96
google.de	40,041	youronlinechoices.eu	0.000656	spiegel.de	1,174,629.32
mapy.cz	38,310	europa.eu	0.000640	free.fr	1,164,237.86
google.it	35,504	google.it	0.000444	bund.de	1,158,448.65
phoca.cz	30,339	youtu.be	0.000437	mpg.de	1,155,542.20
webnode.cz	28,506	google.de	0.000420	admin.ch	1,153,424.50
free.fr	27,420	ideal.nl	0.000386	ox.ac.uk	1,135,822.35

Suffix List published by the Mozilla Foundation,[13] and it is defined as one dot level above that of the public suffix of the host: for example, a.com for b.a.com (as .com is on the public suffix list) and c.co.uk for a.b.c.co.uk (as .co.uk is on the public suffix list).[14]

For each graph, we display the top 10 nodes by indegree, PageRank (with constant preference vector and $\alpha = 0.85$), and by *harmonic centrality* [12], the harmonic mean of all distance toward a node. PageRank was computed with the highest possible precision in IEEE format using the LAW library, whereas harmonic centrality was approximated using HyperBall [11].

Besides the obvious shift of importance (UK government sites for uk-2014, government/news sites in eu-2015, and large US companies in gsh-2015), we can confirm the results of Ref. [30]: on these kinds of graphs, harmonic centrality is much more precise and less prone to spam than indegree or PageRank. In the host graphs, almost all results of indegree and most results of PageRank are spam or service sites, whereas harmonic centrality identifies sites of interest (in particular, in uk-2014 and eu-2015). At the TLD level, noise decreases significantly, but the difference in behavior is still striking, with PageRank and indegree still displaying several service sites, hosting providers and domain sellers as top results.

7 COMPARISON WITH PREVIOUS CRAWLS

In Table 3 , we compare the statistics of the HTTP statuses found during the crawling process. We use as a comparison Table I from Ref. [6], which report both data about the IRLBot crawl [27] (6.3

[13] http://publicsuffix.org/list/.
[14] Top-level domains have been called *pay-level domains* in Ref. [30].

Table 6. Most Relevant Hosts and TLD of gsh-2015 by Different Centrality Measures

Indegree		PageRank		Harmonic centrality	
Host Graph					
gmpg.org	2,423,978	wordpress.org	0.00885	www.google.com	18,398,649.60
www.google.com	1,787,380	www.google.com	0.00535	gmpg.org	17,167,143.30
fonts.googleapis.com	1,715,958	fonts.googleapis.com	0.00359	fonts.googleapis.com	17,043,381.45
wordpress.org	1,389,348	gmpg.org	0.00325	wordpress.org	16,326,086.35
maps.google.com	959,919	go.microsoft.com	0.00317	play.google.com	16,317,377.30
www.miibeian.gov.cn	955,938	sedo.com	0.00192	plus.google.com	16,300,882.95
www.adobe.com	670,180	developers.google.com	0.00167	maps.google.com	16,105,556.40
go.microsoft.com	642,896	maps.google.com	0.00163	www.adobe.com	16,053,489.60
www.googletagmanager.com	499,395	support.microsoft.com	0.00146	support.google.com	15,443,219.60
www.blogger.com	464,911	www.adobe.com	0.00138	instagram.com	15,262,622.80
TLD Graph					
google.com	2,174,980	google.com	0.01011	google.com	10,135,724.15
gmpg.org	2,072,302	fonts.googleapis.com	0.00628	gmpg.org	9,271,735.90
wordpress.org	1,409,846	gmpg.org	0.00611	wordpress.org	8,936,105.80
fonts.googleapis.com	1,066,178	sedo.com	0.00369	fonts.googleapis.com	8,689,428.35
adobe.com	770,597	adobe.com	0.00307	adobe.com	8,611,284.30
microsoft.com	594,962	wordpress.org	0.00301	microsoft.com	8,491,543.60
blogger.com	448,131	microsoft.com	0.00277	wordpress.com	8,248,496.12
wordpress.com	430,419	blogger.com	0.00121	yahoo.com	8,176,168.72
yahoo.com	315,723	networkadvertising.org	0.00120	creativecommons.org	7,985,426.37
statcounter.com	313,978	61.237.254.50	0.00105	mozilla.org	7,960,620.27

billion pages) and data about a Mercator crawl [34] (819 million pages). The Mercator data should be compared with the columns labelled "Arch.," which are based on archetypes only, and thus do not comprise pages with duplicated content. The IRLBot data, coming from a crawler that does not perform near-duplicate detection, cannot, in principle, be compared directly to either column, as detecting near-duplicates alters the crawling process; but, as it is easy to see, the statistics are all very close. The main change we can observe is the constant increase of redirections (3XX). It would have been interesting to compare interesting structural properties of the IRLBot and Mercator datasets (harmonic diameter, etc.) with our crawls, but neither crawl is publicly available.

8 CONCLUSIONS

In this article, we have presented BUbiNG, a new distributed open-source Java crawler. BUbiNG is orders of magnitudes faster than existing open-source crawlers, scales linearly with the number of agents, and will provide the scientific community with a reliable tool to gather large data sets.

The main novel ideas in the design of BUbiNG are:

- a pervasive usage of modern lock-free data structures to avoid contention among I/O-bound fetching threads;
- a new data structure, the *workbench*, that is able to provide, in constant time, the next URL to be fetched respecting politeness both at the host and IP level;
- a simple but effective *virtualizer*—a memory-mapped, on-disk store of FIFO queues of URLs that do not fit into memory.

BUbiNG pushes software components to their limits by using massive parallelism (typically, several thousand fetching threads); the result is a beneficial fallout on all related projects, as witnessed by several enhancements and bug reports to important software libraries like the Jericho HTML parser and the Apache Software Foundation HTTP client, in particular, in the area of object creation and lock contention. In some cases, like a recent regression bug in the ASF client (JIRA issue 1461), it was exactly BUbiNG's high parallelism that made it possible to diagnose the regression.

Future work on BUbiNG includes integration with spam-detection software and proper handling of spider traps (especially, but not only, those consisting of infinite non-cyclic HTTP-redirects); we also plan to implement policies for IP/host politeness throttling based on download times and site branching speed, and to integrate BUbiNG with different stores like HBase, HyperTable, and similar distributed storage systems. As briefly mentioned, it is easy to let BUbiNG follow a different priority order than breadth first, provided that the priority is *per host* and *per agent*; the latter restriction can be removed at a moderate inter-agent communication cost. Prioritization at the level of URLs requires deeper changes in the inner structure of visit states and may be implemented using, for example, the Berkeley DB as a virtualizer: this idea will be a subject of future investigations.

Another interesting direction is the integration with recently developed libraries, which provides *fibers*, a user-space, lightweight alternative to threads that might further increase the amount of parallelism available using our synchronous I/O design.

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 experiments performed at the Università di Pisa. Finally, we thank Domenico Dato and Renato Soru for providing the hardware and bandwidth for the iStella experiments.

REFERENCES

[1] Internet Archive website. 1996. Homepage. Retrieved May 16, 2018 from http://archive.org/web/web.php.
[2] Heritrix Web Site. 2003. Homepage. Retrieved May 16, 2018 from https://webarchive.jira.com/wiki/display/Heritrix/.
[3] The ClueWeb09 Dataset. 2009. Homepage. Retrieved May 16, 2018 from http://lemurproject.org/clueweb09/.
[4] ISO 28500:2009, Information and documentation—WARC file format. Retrieved May 16, 2018 from https://www.iso.org/standard/44717.html.
[5] Dimitris Achlioptas, Aaron Clauset, David Kempe, and Cristopher Moore. 2009. On the bias of traceroute sampling: Or, power-law degree distributions in regular graphs. *Journal ACM* 56, 4 (2009), 21:1–21:28.
[6] Sarker Tanzir Ahmed, Clint Sparkman, Hsin-Tsang Lee, and Dmitri Loguinov. 2015. Around the web in six weeks: Documenting a large-scale crawl. In *Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1598–1606.
[7] Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. 2005. Uniform Resource Identifier (URI): Generic Syntax. Retrieved May 16, 2018 from http://www.ietf.org/rfc/rfc3986.txt.
[8] Burton H. Bloom. 1970. Space-time trade-offs in hash coding with allowable errors. *Comm. ACM* 13, 7 (1970), 422–426.
[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.
[10] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive crawling for the masses. In *WWW'14 Companion*. 227–228.
[11] Paolo Boldi and Sebastiano Vigna. 2013. In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond. In *Proc. of 2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW 2'13)*. IEEE.
[12] Paolo Boldi and Sebastiano Vigna. 2014. Axioms for centrality. *Internet Math.* 10, 3–4 (2014), 222–262.
[13] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117.

[14] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. 1997. Syntactic clustering of the web. In *Selected Papers from the 6th International Conference on World Wide Web*. Elsevier Science Publishers Ltd., Essex, UK, 1157–1166.

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

[16] Jamie Callan. 2012. The Lemur Project and its ClueWeb12 Dataset. Invited talk at the SIGIR 2012 Workshop on Open-Source Information Retrieval. (2012).

[17] Soumen Chakrabarti. 2003. *Mining the Web—Discovering Knowledge from Hypertext Data*. Morgan Kaufmann. I–XVIII, 1–345 pages.

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

[19] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. 2009. Power-law distributions in empirical data. *SIAM Rev.* 51, 4 (2009), 661–703.

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

[21] D. Eichmann. 1994. The RBSE spider: Balancing effective search against web load. In *Proceedings of the 1st World Wide Web Conference*.

[22] Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.* 21, 2 (1974), 246–260.

[23] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L. Wiener. 2003. A large-scale study of the evolution of web pages. In *Proceedings of the 12th Conference on World Wide Web*. ACM Press.

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

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

[26] Rohit Khare, Doug Cutting, Kragen Sitaker, and Adam Rifkin. 2004. Nutch: A flexible and scalable open-source web search engine. CommerceNet Labs Technical Report 04-04.

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

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

[29] Oliver A. McBryan. 1994. GENVL and WWWW: Tools for taming the web. In *Proceedings of the 1st World Wide Web Conference*. 79–90.

[30] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web—Analyzed on different aggregation levels. *The Journal of Web Science* 1, 1 (2015), 33–47.

[31] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*. ACM, 267–275.

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

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

[34] Mark Najork and Allan Heydon. 2001. *High-Performance Web Crawling*. Technical Report 173. Compaq Systems Research Center.

[35] Marc Najork and Allan Heydon. 2002. High-performance web crawling. In *Handbook of Massive Data Sets*, James Abello, Panos M. Pardalos, and Mauricio G. C. Resende (Eds.). Kluwer Academic Publishers, 25–45.

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

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

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

[39] Sebastiano Vigna. 2013. Fibonacci binning. *CoRR* abs/1312.3749 (2013).

Received January 2016; revised October 2017; accepted November 2017