UniCrawl: A Practical Geographically Distributed Web Crawler

Do Le Quoc, Christof Fetzer
Systems Engineering Group
Dresden University of Technology, Germany

Pierre Sutra, Valerio Schiavoni,
Étienne Rivière, Pascal Felber
University of Neuchâtel, Switzerland

***Abstract*—As the wealth of information available on the web keeps growing, being able to harvest massive amounts of data has become a major challenge. Web crawlers are the core components to retrieve such vast collections of publicly available data. The key limiting factor of any crawler architecture is however its large infrastructure cost. To reduce this cost, and in particular the high upfront investments, we present in this paper a geo-distributed crawler solution, UniCrawl. UniCrawl orchestrates several geographically distributed sites. Each site operates an independent crawler and relies on well-established techniques for fetching and parsing the content of the web. UniCrawl splits the crawled domain space across the sites and federates their storage and computing resources, while minimizing thee inter-site communication cost. To assess our design choices, we evaluate UniCrawl in a controlled environment using the ClueWeb12 dataset, and in the wild when deployed over several remote locations. We conducted several experiments over 3 sites spread across Germany. When compared to a centralized architecture with a crawler simply stretched over several locations, UniCrawl shows a performance improvement of 93.6% in terms of network bandwidth consumption, and a speedup factor of 1.75.**

Keywords*—*web crawler, geo-distributed system, cloud federation, storage, map-reduce.

I. Introduction

Over the last thirty years, publicly available digital data has grown exponentially, and it will continue to do so. Volumes are projected to reach around 40 zettabytes by 2020, or equivalently a few gigabytes per person [29].[1] This data contains very rich and valuable information, and its mining and exploitation will drive growth in all sectors. Consequently, many data-oriented companies have recently emerged and more are expected to come to the market in the coming years. Interest for extracting intelligence from data goes beyond traditional Internet applications, such as search engines and recommendation systems, to areas like business planning, marketing analysis, etc.

Most publicly available digital data on the web is in an unstructured, or loosely structured, format. Thus, some processing is necessary in order to extract meaningful information. The very first step of this processing, that is constructing the data collection, is usually achieved using a web crawler. A web crawler accesses remote servers on the Internet in order to fetch, process and store web pages. The crawler starts from some initial seed URLs, store the corresponding web pages for later processing, and then extracts links to other web pages. Those links are themselves subsequently crawled. This process can be repeated until a given depth, and pages are periodically re-fetched to discover new pages and to detect updated content.

Due to the size of the web, it is mandatory to make the crawling process parallel [23] on a large number of machines to achieve a reasonable collection time. This requirement implies provisioning large computing infrastructures. Existing commercial crawlers, such as Google or Bing rely on big data centers. However, this approach imposes heavy requirements, notably on the cost of the network infrastructure. Furthermore, the high upfront investment necessary to set up appropriate data centers can only be made by few large Internet companies, leaving smaller ones out of the market.

Although public crawl repositories exist, such as Common Crawl [3], using them for processing requires externalizing computation and data hosting to a commercial cloud provider. For a data-driven company, this also poses the problem of data availability in the mid-long term. Moreover, the lack of selectivity in the public crawling process may require post-processing large amounts of data, whereas only a small subset would be necessary.

A solution to the above problems is to distribute the crawling effort over several geographically distributed locations. The use of multiple sites can reduce both capital and operating expenses, for instance by allowing several small companies to mutualize their crawling infrastructures. In addition, such an approach leverages data locality as sites can crawl web servers that are geographically nearby. Finally, multiple sites can act as replicas to be resilient to local disasters; collected data is then safe even if a whole site disappears. On the other hand, geo-distributed crawling requires synchronization between the crawlers at the different sites, which imply communicating over a wide area network. A careful system design for a distributed crawler aims at reducing such communication costs, as the costs and delays are higher than for intra-site communication.

In this paper, we present UniCrawl, an efficient geo-distributed crawler that aims at minimizing inter-site communication costs. UniCrawl orchestrates multiple geographically distributed sites. Each site uses an independent crawler and relies on well-established techniques for fetching and parsing the content of the web. UniCrawl partitions the crawled domain space across the sites and federates their storage and computing resources. Our design is both practical and scalable. We assess this claim with a detailed evaluation of UniCrawl in a controlled environment using the ClueWeb12 dataset, as well as in geo-distributed setting using 3 distinct sites located in Germany.

[1] A zettabyte corresponds to one thousand exabytes, that is 10^{21} bytes.

Outline The rest of the paper is organized as follows. We survey related work in Section II. Section III introduces the crawler architecture, refining it from existing well-founded central designs. Details about the implementation internals are given in Section IV. Section V presents the experimental results, both in-vitro, and in-vivo over multiple geographical locations in Germany. We discuss our results and future work in Section VI. We conclude the paper in Section VII.

II. Related work

While the amount of information available on the web is certainly finite, the number of pages on which that information resides is too large to be easily managed. At core, this comes from the fact that the web is a huge, ever growing, and dynamic media. As identified by Olston and Najork [33], this poses several key challenges that every crawler (sometimes referred to as a *web spider*) needs to solve: (i) since the amount of information to parse is huge, a crawler must scale; (ii) a crawler should select which information to download first, and which information to refresh over time; (iii) a crawler should not be a burden for the web sites that host the content; and (iv) adversaries, e.g., spider traps, need to be avoided with care.

There exists a large number of web crawler designs and implementations. Mercator [28] is an extensible web crawler written in Java. The authors give a blueprint web crawler design and several experimental results where they cover 891 million pages in a period of 17 days. Polybot [34] is a distributed system, consisting of a crawl manager, multiple downloading processes, and a DNS resolver. IBM WebFountain [26] is an industrial-strength design that relies on a central controller to operate several multi-threaded crawling agents. Ubicrawl [17] uses consistent hashing to partition URLs across crawling agents, leading to graceful performance degradation in the event of failures. It was shown to be able to download about 10 million pages per day using five crawling machines at a single site.

The IRLbot system [31] is a single-process web crawler able to scale to large web data collections with small performance degradation. Lee et al. [31] describe a crawl that ran on a quad-CPU machine over a two months period, and downloaded nearly 6.4 billion web pages. The Internet Archive uses the Heritrix crawler [6] whose design is similar to Mercator. Recently, the Lemur project employed Heritrix to gather the ClueWeb12 data set [2] which we use for our in-vitro experiments. Nutch is a popular open-source web crawler that relies on the map-reduce paradigm [25]. It was notably used by Baldoni et al. [15] to analyze online communities (e.g., news, blogs, or Google groups). We also used the Nutch code base to implement UniCrawl.

All of the previous architectures we described operate several crawling agents. Cho and Garcia-Molina [23] are among the first to study the principles of distributed crawler design. They advocate the use of a set of independent agents and propose several classifying parameters. The assignment of URLs to agents can be either static or dynamic. Agents might operate either in firewall, cross-over, or exchange mode when communicating each other. Our design is in line with the conclusion of this work: we favor the exchange of newly discovered URLs and their static partitioning among agents.

After fetching a page, the crawler needs to check whether discovered URLs were previously crawled or not. As a consequence, caching is a cornerstone mechanism of every web spider. Broder et al. [20] study how to efficiently cache URLs for the purpose of crawling the web. They ran several simulations over a trace containing 26 billions of URLs. Their main conclusion is that a cache of roughly 50,000 entries can achieve a hit rate of almost 80% (even when relying on a random eviction policy).

All the pages on the web do not have the same importance. Due to the scale of the Internet, important pages should be fetched first and refreshed more often. Page importance is thus a core property that guides the discovery and refreshing of pages [24]. Ranking algorithms determine what is the most significant information at the *frontier* of the crawl, i.e., the set of pages whose existence is known but that are not fetched yet. A large literature exists on ranking algorithms starting from the seminal work of Google founders on PageRank [19]. Following the Nutch architecture, UniCrawl may adapt various ranking strategies at each site. During our experiments, we make use of the Opic algorithm of Abiteboul et al. [10]. This algorithm requires a single pass over existing fetched data per crawling round, and is thus arguably more scalable than a full PageRank computation [12, 31].

Baeza-Yates and Castillo [11] study the depth at which a crawl should stop. They introduce three stochastic models of web surfing, fit their models with information extracted from web server logs, and measure the amount of information a surfer retrieves at each level. Their conclusions are that most surfers stopped before depth 5 due to the fact that 80% of the best pages are in these first levels.

Several researchers recently studied the complex challenges posed by geographically distributed web retrieval. Baeza-Yates et al. [14] focus on the feasibility of geo-distributed search engines. This question is also addressed in a series of paper by Cambazoglu et al. [21, 22]. In particular, these studies assess analytically the performance and interest of a web search architectures distributed over multiple sites. They also show that this question is of practical importance. We can quote from their text: *"it is critical to have a system design that can cope with the growth of the web, and that is not constrained by the physical limitations of a single data center"*. Exposto et al. [27] show that, with an appropriate multi-criteria partitioning, it is very beneficial to split the crawling process across multiple geo-distributed agents. Baeza-Yates et al. [12] compare analytically several crawling strategies to crawl geo-distributed web sites. They show that without historical information, strategies such as largest web site first and Opic perform well (i.e., they gather the most important pages first). Baeza-Yates et al. [13] identify several open problems regarding geo-distributed web retrieval: (crawling) how to properly prioritize the crawling frontier and exchange URLs between distributed agents; (indexing) how to partition the index and balance the load across sites; and (querying) routing and data replication matters as the cross-site network bandwidth is a scarce resource. We explain in the remainder of this paper how UniCrawl answers those challenges, both theoretically with its design, and practically with our evaluation.

III. Distributed Crawler Architecture

The general algorithmic sequence used by a web crawler is quite simple. It can be seen as the continuous execution of the following four phases. In the *generate* phase, the crawler extends the crawl frontier by determining the next set of URLs that it needs to fetch. These URLs are downloaded from web hosts in the *fetch* phase. The crawler analyzes the content retrieved during the *parse* phase and refreshes appropriately the crawl database in the *update* phase.

However, the size of the web and its rate of change (estimated at 7% per week [18]) introduce some serious system design issues. This section presents how the design of UniCrawl addresses those problems, describing first the internals of a site, then cross-site operations.

A. Single site Design

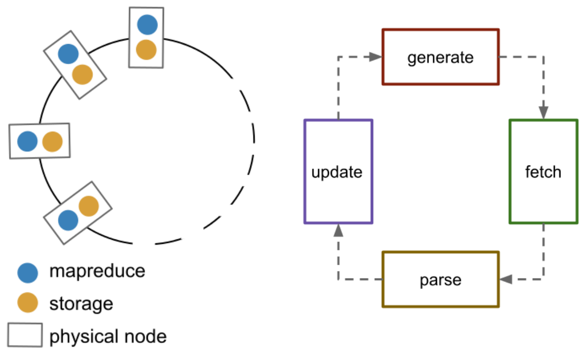
At each site, we build UniCrawl upon the well-established architecture of Apache Nutch [9], more specifically its version 2.x. We contribute to Nutch the necessary improvements and novel features that we shall cover next.

Figure 1 depicts an overview of the architecture. UniCrawl makes use of the map-reduce paradigm and persists the content of the crawl in a distributed key-value store. This design choice, inherited from Nutch, ensures that the system is able to process large amount of data. In what follows, we recall how map-reduce works, detail the storage internals, then explain how we combine the two to implement the various phases of a crawling round.

1) Map-reduce: Map-reduce is a paradigm for processing large data sets in parallel on a cluster of workers. Schematically, map-reduce executes three steps, each step operating over a set of key-value pairs. In the *map* step, each worker applies a $map()$ function to its local *split* of the pairs, and *spills* the output to some temporary storage. During the *shuffle* step, workers distributes pairs stored in temporary storage among them, ensuring that all data corresponding to the same key is located on a single worker. Then, workers in the reduce step process output data grouped per key, applying to it a $reduce()$ function.

2) Site storage: Every web spider needs to persist a large amount of information over time: the fetched pages themselves, but also other data structures that maintain the state of the crawling process. To that end, a crawler uses a *crawl database*. In UniCrawl, the crawl database of a site is implemented as a single distributed map structure. This map contains for each page its URL, content, and outlinks (i.e., the URLs of pages linked from it). In addition, it also contains several additional metadata fields used during the crawl. In particular, the crawl database stores the page status (generated, fetched, moved, etc.) and its *score* attributed by the ranking algorithm. Notice that the score of two pages are always comparable, and that this order defines the *ranking* of a page.

To implement the crawl database, UniCrawl makes use of Infinispan [32], a distributed key-value store that supports the following features: *(Routing)* Nodes are organized in a ring. Infinispan uses a one-hop routing design, i.e., every node knows all the other nodes. *(Elasticity)* Infinispan is elastic, meaning that storage nodes can be added or removed on the fly. Upon joining, a node chooses a random identifier along the ring and fetches the ring structure from some other Infinispan node. It then informs its neighbors that it is joining. *(Storage)* Infinispan uses consistent hashing [30] to assign blocks to nodes with a replication factor ρ: a data block with a key l is stored at the ρ nodes whose identifiers follow l on the ring. *(Reliability)* Infinispan is built upon the JGroups communication library [16]. This library use failure detectors to maintain a consistent view of the system. The repair mechanisms of consistent hashing are triggered upon a lack of response of a storage node within a timeout. *(Interface)* Infinispan offers to the end-user a concurrent map interface that provides the classical put, get, $remove$ and $putIfAbsent$ operations. (*Consistency*) Infinispan implements strongly consistent operation on the distributed map using a primary-backup replication scheme. *(Querying)* Every Infinispan node maintains a configurable index of the data it stores. This index is purely local, and based on Apache Lucene [8]. It allows to execute SQL-like queries over the indexed content of the node.



(a) Site architecture (b) Crawling phases

Fig. 1. Overview of UniCrawl at a site.

3) Crawling phases: In Figure 1, we present the different crawling phases of UniCrawl. UniCrawl executes one map-reduce job per phase. Schematically, a job first fetches data from Infinispan with an appropriate query, executes some computation upon it, then stores the corresponding results during the reduce phase.

Following the map-reduce paradigm, for each phase, we execute one instance of the input query per storage node. In addition, we rely on pagination to further increase parallelism. To that end, we compute tentatively the amount of results at each storage node, then split the original query according to some pagination level. This leverages the fact that multiple mappers can process in parallel the data available from the same local Infinispan node.

We now detail each of the phases which compose a crawling round.

(*Generate*) The goal of the generate phase is to select a set of pages to process during the round. To that end, the map step computes the subset of pages in the crawl database which were not previously fetched during the previous rounds. These pages are grouped by host in the reduce phase, and each reducer outputs the k top ranked pages. In total, r reducers generates rk pages, and thus rk defines

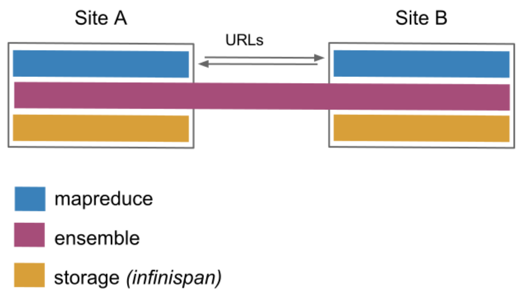


Fig. 2. Multisite architecture of UniCrawl.

the crawl *width* at a site. In UniCrawl, the query at each storage node retrieves only the rk top ranked pages. This improvement, not present in the original Nutch design, ensures that at each round the complexity of the generate phase is bounded by the crawl width.

(*Fetch*) During the fetch phase, the map step first groups by host the pages that were generated in the previous phase. In the reduce step, each worker receives a set of hosts together with the list of web pages it needs to retrieve from those hosts. A reducer is multi-threaded, yet in order to maintain politeness it uses a single queue per host. This queue is subject to a configurable wait policy. Moreover for each host, the reducers follow the instructions given in */robots.txt*.

(*Parse*) Once the pages are fetched, they are analyzed during the parse phase. This phase consists solely of a map step. During this step, the mapper extracts from the fetched pages stored locally the set of outlinks it contains and add them to the crawl database.

(*Update*) The goal of the update phase is to refresh the scores of pages that belong to the frontier in order to prioritize them. To that end, we use the OPIC algorithm of Abiteboul et al. [10]. This scoring method is less expensive than a full PageRank computation and suitable for on-line crawling. Moreover, as shown by Baeza-Yates et al. [12], it quickly converges toward pages of importance. At the beginning of the update phase, the map step retrieves both the score and outlinks of each fetched page. Then, it splits the score evenly across outlinks, and outputs for each outlink its URL together with the shard of the score. In the reduce step of the update phase, each reducer sums the scores obtained by pages located at the frontier. Pages that are among the top $l > k$ ranked ones are then added to the crawl database. Depending on the crawl width, part or all of these pages will then be generated at the beginning of the next crawling round. In Nutch, the update phase process all the pages in the crawl database. On the contrary, UniCrawl executes the map step only on the fetched pages and adds solely rl new pages to the crawl database.

B. Multi-site Operations

Figure 2 depicts the architecture of UniCrawl when we deploy it across multiple geographical locations. In a typical set-up, UniCrawl is composed of several sites interconnected with a wide-area network, and each site is equipped with a few dozen of machines communicating through a low-latency network.

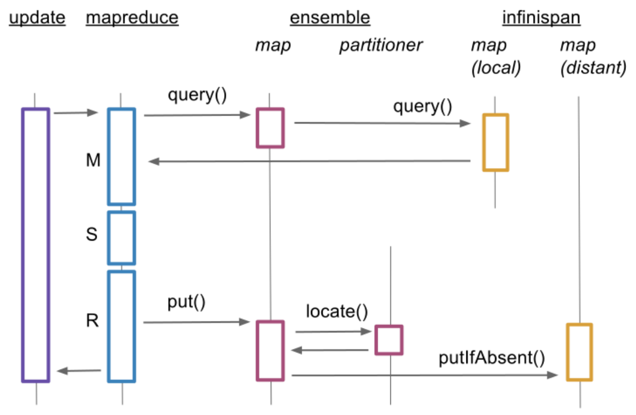


Fig. 3. Lifetime of the update phase.

Several key ideas allow UniCrawl to be practical in this setting: (i) Each site is independent and crawls the web autonomously; (ii) We unite all the site data stores. At each site, the map-reduce jobs of UniCrawl access transparently the federated storage which can provide dependability through geo-replication; and (iii) Sites exchanges dynamically the URLs they discover over the course of the crawl.

In the sections that follow, we cover the multi-site operations of UniCrawl in detail. Furthermore, we discuss the crawl quality in regard to the amount of communication between sites.

1) Federating the storage: One of the key design concerns of UniCrawl is to bring small modifications to the site code base in order be usable over multiple geographical locations. To achieve this, we rely on ENSEMBLE, a storage layer able to federate transparently multiple INFINISPAN deployments.

ENSEMBLE offers the same concurrent map interface to the map-reduce jobs of UniCrawl as INFINISPAN. An ENSEMBLE map is built upon a set of INFINISPAN maps and it can be either *replicated* or *distributed*. If the map is replicated, all the underlying INFINISPAN maps replicate its content. On the other hand, if the ENSEMBLE map is distributed, each INFINISPAN map stores a distinct part of the content. In this case, a partitioner defines how the content is split between the INFINISPAN maps.

A distributed map can operate in *normal* or *frontier* mode. In normal mode, put and get operations access the exact locations of the content, possibly on another site than the one where the call was made. When using the frontier mode, put operations behave correctly and may be remote, but get operations and queries are local to the site that call them. We use this later mode to implement the crawl database. The next section covers with more details how we proceed.

2) Collaboration between sites : Following the approach advocated by Cho and Garcia-Molina [23], UniCrawl exchanges newly discovered URLs over time. This exchange occurs at the end of the update phase. We depict the lifetime of this phase in Figure 3.

In detail, we implement the crawl database as a distributed ENSEMBLE map that span all the sites. This map operates in frontier mode with a replication factor of one. As a consequence of this setting, (i) in the reduce step of the update phase, the reducers write the kn top ranked pages across all sites, while (ii) all the keys accessed by the generate, fetch and parse phases are local to the site. This allows sites to operate independently. Moreover, it reduces communication to the bare minimum of newly discovered URLs.

We can tune the partitioner which assigns each key in the ENSEMBLE map to an underlying INFINISPAN site map. In our current setting, we use two approaches to that end: consistent hashing and distance-based. The first solution is similar to the initial proposal of Boldi et al. [17]. The distance-based partitioner is more involved but reduces the distance between a web server and its corresponding fetching site, thus lowering the network cost associated to it. This partitioner relies on an ENSEMBLE map D replicated at all sites which associates a domain to its geographical coordinates. For some page p having URL u, when a $put(u, p)$ operation occurs on the ENSEMBLE map implementing the crawl database, the partitioner first extracts the domain d of u, then it executes a $get(d)$ operation on D. If the coordinates do not exist, the partitioner retrieves them using the IP address of the domain. Once the coordinates of d are present in D, the partitioner computes the closest geo-graphical site and returns the associated INFINISPAN map.

Notice that in UniCrawl, sites operates independently and execute their update phases at different times. As a consequence, a site that finds a new URL uses a $putIfAbsent$ operation when it accesses a distant site storage. This avoid race condition in case this URL was already crawled. Furthermore, during the update phase, each reducer (i) outputs at most $\frac{l}{m}$ URLs where m is the number of participating sites, and (ii) stops after it has retrieved l URLs local to its site. The former modification avoids to overwhelm a site, whereas the latter preserves them from starvation.

3) Crawl quality and cost: The quality of the crawling operation is not only measured by means of pure web-graph exploration but also by the rounds it takes to discover the most interesting pages. This is important because web crawling and web searching are two algorithms that are executed concurrently. Due to the massive size of the web, state-of-the-art crawlers focus on finding the most relevant portion of the internet as quickly as possible [10]. As we pointed out previously, they use to that end a scoring algorithm that estimates the importance of each page. The crawler prioritizes its crawling effort on the URLs with the highest scores.

Recall that parameter k determines the crawl width, and r the total amount of reducers during the generate phase. The generate phase outputs the rk most ranked pages in its frontier. This rank is attributed to pages from the OPIC algorithm computation that occurs during the update phase. As a consequence, when UniCrawl operates a single site, it fetches the rk most ranked pages in the frontier, and with good probability these pages are the most important ones (according to their PageRank [10, 12]).

In a geo-distributed setting, sites executing UniCrawl exchange URLs between them at the end of the update phase. In case a site A retrieves an URL u assigned to some site B, it calls the $putIfAbsent$ primitive (see Figure 3). This primitive has no effect if u is already stored at site B. As a consequence, the score at site B might not take into account all the contributions of pages stored at site A. Thus, UniCrawl may not always prioritize the most important pages in a geo-distributed setting. On the other hand, such an approach reduces the volume of data exchanged between sites.

When UniCrawl spans several locations, the key cost parameter is the amount of communication exchanged between sites. Section V assesses that UniCrawl reduces this cost to the bare minimum. In Section VI, we discuss a variation of UniCrawl whose crawl quality is instead optimal.

IV. IMPLEMENTATION

We implemented UniCrawl in Java, starting from the code base of Nutch version 2.3. Nutch relies on the Apache Hadoop framework [5]. To leverage the recent improvements in the map-reduce components of Hadoop [35], we modified Nutch to use the latest version (2.5.3).

Nutch makes use of Apache Gora [4], an open-source framework that provides an in-memory and persistent data model for big data. Gora offers generic hooks to substitute any Cloud storage system to HDFS in the Hadoop map-reduce eco-system. We implemented a Gora interface for ENSEMBLE that allows UniCrawl to execute geo-distributed operations. Our geo-distributed storage ENSEMBLE was itself developed as a module of the industrial-grade key-value store INFINISPAN [32].

In total, our contribution accounts for about a dozen thousands lines of code (LOC) split as follows: 9.4 kLOC for Ensemble, 1.1 kLOC for Gora and a 2.3 kLOC patch for Nutch.

A. Merging Phases

In Yarn, each new map-reduce job creation is expensive as it requires to start a dedicated Java virtual machine, and deploy the appropriate jars.[2] To lower this cost, we merge the fetch and parse phases in our UniCrawl implementation. This means that whenever a reducer fetches a new page, it parses its content and extract the out-links. These links are then directly inserted in the crawl database together with the fetched page.

B. Caching

To avoid sending out an URL multiple times across sites, we use a distributed caching solution. In more details, this cache is a bounded ENSEMBLE map C local to each site and replicated at all nodes in a site. During the update phase, when a reducer selects a URL in the frontier that is associated to a remote site, it first check locally with C if this URL was previously sent. If this is the case, the reducer simply skips the call to $putIfAbsent$. Since C is replicated at all nodes, every map-reduce node is co-located with an INFINISPAN node, and C is in memory, this inclusion test costs less than a millisecond.

V. EVALUATION

In this section, we evaluate the performance of UniCrawl along several key metrics such as the page processing rate, the memory usage and the network traffic across sites. We split this evaluation in two parts. First, we evaluate our approach in-vitro, by running UniCrawl against the ClueWeb12 benchmark in an emulated multi-site architecture and crawling from a local

[2] Distributed caching of various files is hopefully possible.

repository. Then, we report several experimental results where we deploy UniCrawl at multiple locations in Germany and access actual web sites.

A. *In-vitro validation*

This set of experiments evaluates the performance of UniCrawl in a controlled environment. To that end, we first assess the scalability of our approach in the case of a single site. Then, we present several experimental results that were conducted in an emulated multi-site architecture.

Our experiments take place on a cluster of virtualized 8-core Xeon 2.5 Ghz machines with 8GB of memory. Nodes are running Ubuntu 14.04 64bits, and are connected with a virtualized 1 Gbps switched network. Network performance, as measured by *ping* and *netperf*, is 0.3ms for the round-trip delay and a bandwidth of 117 MB/s. We set-up Yarn to use 4 GB of the total node memory; the other half being used by INFINISPAN. A mapper (or a reducer) can use up to 1 GB of memory. Each node is equipped with a virtual hard-drive whose read/write (uncached) performance, as measured with `hdparm` and `dd`, is 97/91 MB/s.

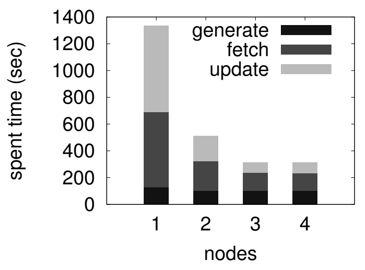
Our in-vitro evaluation uses the ClueWeb12 B13 dataset [2]. This dataset contains around 52 millions documents, which were gathered by the Lemur project [7] in February 2012. In total, this dataset weights 1.95 TB and we serve it via a dedicated Python server. During all our experiments, we start by injecting around 1 million URLs in the crawl database, then we execute several crawling rounds with a width of 50,000 pages.

1) Single site performance: Figure 4(a) present the average length of each crawl phase at a site, when we deploy UniCrawl on 1 to 4 nodes. As indicated in Section IV-A, UniCrawl merges the fetch and parse phases. Consequently, we only report the average time a fetch phase takes to execute.

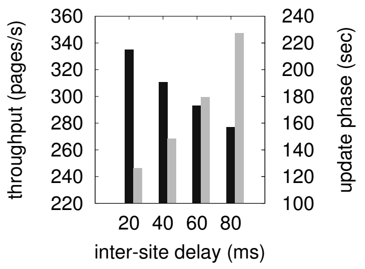
In Figure 4(a), we can observe that the total time of a round decreases from 1,337 to 312 seconds when UniCrawl scales out from 1 to 4 nodes. In term of pages per second, this translates into a 3.79 multiplicative factor, 4 nodes being able to process around 160 pages/s. In this figure, we also observe that the time required to execute the fetch phase with 3 and 4 nodes is close. This comes from the fact that our Python server hosting the ClueWeb12 dataset saturates at around 380 pages/s. Besides, with a crawl width of 50,000 pages, the time to execute both the generate phases does not change much, even with 4 nodes. This cost corresponds essentially to the time the map-reduce framework takes to propagate the jobs across workers and start the Java VMs.

2) Emulating multiple sites: Next, we evaluate how UniCrawl behaves in a multi-site environment. To that end, we emulate the long-distance routing between geo-distributed locations with the help of the Linux traffic shaping tools. Our experiment makes use of 3 sites, each site containing 3 UniCrawl nodes. We fix to 6 the number of reducers per site. Besides, to avoid saturating the Python server hosting the ClueWeb12 benchmark, we also set the number of fetcher threads per reducer to 20.

In Figure 4(b), we vary the ping distance between any two locations, from 20 to 80 ms. When the ping distance equals 20ms, this corresponds to a deployment at the scale of an European country. In the case of 80 ms, we emulate a cross-continent deployment. Figure 4(b) reports on the left y axis (in black) the average throughput of UniCrawl in number of pages per second, aggregated over all sites. The right y axis (in gray) reports the average length of the update phase during a crawling round.



(a) Single site scalability



(b) Impact of geo-distribution

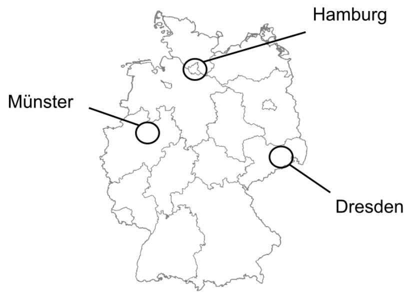
Fig. 4. Performances in-vitro

As expected, we observe in Figure 4(b) that the more distant the sites are, the slower UniCrawl is. Peeking from 334 pages/s when the sites are geo-graphically close, performance deteriorates to 278 pages/s when the ping distance equals 80ms. Moreover, we notice in Figure 4(b) that UniCrawl performance is linearly correlated with the time it takes to execute an update phase. Such a correlation is expected from the design of UniCrawl, as only the update phase necessitates some collaboration between the sites.

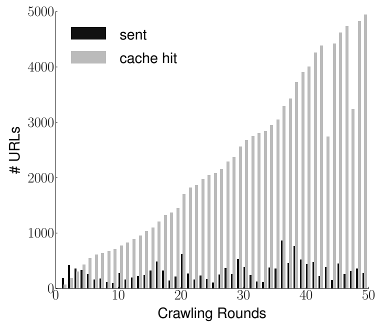
B. *UniCrawl in the wild*

To further assess the design of UniCrawl, we measure its performance in various real-world scenarios. We use several clouds provided by the Cloud & Heat company [1], and deploy UniCrawl at different locations in Germany. A cloud operates 3 instances (VMs) of medium size (4 GB RAM, 4 virtual CPU and 120 GB disk), connected via a gigabit ethernet and running Ubuntu Linux 14.04 64 bits. Figure 5(a) indicates the clouds' locations.

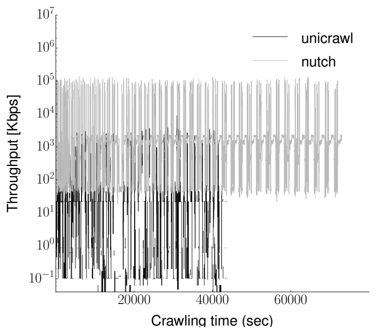
In all the experiments that follow, we start our crawl from a seed list of 30 US universities. We fix to 6 the number of reducers per site and use the default Nutch value of 10 fetcher threads per reducer. Our evaluation consists then in a sequence of 50 crawling rounds. Below, we first comment on the benefits of our caching mechanism. Then, we compare UniCrawl to an out-of-the-box Nutch deployment. We close our evaluation with an assessment of the scalability of our design.



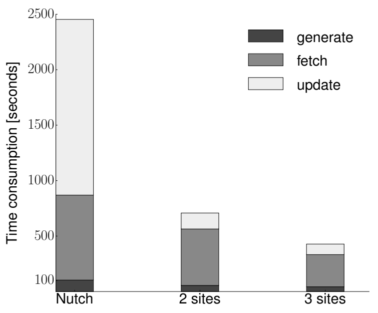
(a) Locations of the clouds



(b) Exchange rate at Hamburg



(c) Network traffic



(d) Scalability

Fig. 5. Evaluation in a geo-distributed setting

1) URL Exchange: In Figure 5(b), we evaluate the URLs exchange rate at Hamburg, when UniCrawl spans both this location and Münster. We observe that on average the Hamburg site finds 2,981 duplicate URLs per round, with an average hit rate of 92%. After 50 crawling rounds, this translates into a memory usage of around 3 MB for the URLs cache. This performance, inline with the simulation results of Broder et al. [20], shows the benefits of the caching mechanism.

2) Comparison with Nutch: Our second experiment aims at assessing the advantages of our design in comparison to a naive deployment of Nutch with a Cassandra[3] backend over multiple geographical locations. To that end, we deploy UniCrawl and Nutch at both Hamburg and Münster, and measure the traffic consumption between sites. Figure 5(c) details our results in a semi-log scale. Notice that in this experiment we fix the crawl width in such a way that both systems harvest in total a similar amount of pages (respectively 184K and 190K for UniCrawl and Nutch).

We observe in Figure 5(c) that UniCrawl is around 1.75 times faster than the Nutch deployment. Such a gap comes from the fact that Nutch executes the map-reduce steps of the crawling phase over the two locations, which implies a large penalty in term of processing time. In addition, this figure tells us that our approach reduces the inter-site traffic by 93.6%. This last point is of importance as WAN bandwidth is notoriously expensive. For instance, the very same Nutch deployment across two Amazon clouds would require around 3000 GB/month and costs 30 USD/month.[4] On the other hand, UniCrawl requires only around 192 GB/month (corresponding to 1.92 USD/month).

3) Scalability: Our last experiment consists in scaling-up UniCrawl over multiple locations. Figure 5(d) reports the average time of each phase when we deploy our system on two and three locations. We also show a comparison with a naive Nutch deployment at two sites. In all experiments, we fix the global crawl width to a constant factor of 400.

In this figure, we first observe that the key issue with the naive Nutch deployment comes from the update phase. Indeed, this phase is an order of magnitude slower than with a deployment of UniCrawl over the two same locations. We can explain this difference by the fact that the map-reduce steps are geo-distributed and that the update phase is the most demanding, both in terms of storage and processing power. Figure 5(d) also tells us that the total time to execute a crawling round improves when we scale UniCrawl from two to three sites. This improvement comes from the fact that the global crawl width is constant in both cases. In this setting, UniCrawl displays a scale-up factor of 1.42 from two to three sites.

VI. Discussion

Recall that in UniCrawl the decision of fetching some page p is taken at several locations. First, at some site A the URL of p is detected as the outlink of some crawled page. Second at some site B (which can also be site A), the crawler decides to fetch page p when p belongs to the highly ranked pages in the frontier. Section III-B3 points out that the score of p at site B might not reflect all the contributions stored at the other sites. Below, we sketch a variation of UniCrawl ensuring that the score at the crawling location represents this time the global contribution:

[3]http://cassandra.apache.org

[4]At the time of this writing, and according to http://calculator.s3.amazonaws.com/index.html.

(**Construction**) We add a link database L, implemented as a distributed ENSEMBLE cache over all the sites. This database consists in triples of the form (u, i, s). During the reduce step of the update phase, for each new non-local URL u having an inlink i with score s, we add a triple (u, i, s) to L. Then, we add an additional map-reduce round after the update phase. In this round, the map step fetches all the tuples in L and grouped them by URL. For each such URL, the reduce step retrieves the original score from the crawl database then updates it accordingly with the aggregated score.

Notice that even if we clean-up L after each update phase, its size is linear in the amount of non-local discovered URLs. Without a large map-reduce cluster at each site, such a size tends to be intractable after several crawling rounds. As a consequence, and in our experience, we believe that this approach is inherently expensive in terms of storage and computation time. Nevertheless, we left a detailed evaluation and comparison with our current design as future work.

VII. Conclusion

In this paper, we propose a new crawling system for multiple geographically distributed sites, UniCrawl. At each site, we base UniCrawl on a well-founded design consisting in a sequence of a map-reduce jobs executed atop an industrial-grade distributed key-value store. Then, we extend this design to a geo-distributed environment with ENSEMBLE, a novel layer that federates the storage available at each site.

UniCrawl is both practical and scalable. We assess this claim with a detailed evaluation in a controlled environment using the ClueWeb12 dataset, as well as in a geo-distributed setting with 3 remote sites located in Germany. In comparison with a baseline technique where a central crawler is simply stretched over several locations, UniCrawl shows a performance improvement of 93.6% in terms of network bandwidth consumption, as well as a speedup factor of 1.75.

Acknowledgment

This work was supported by the LEADS project funded by the European Commission under the Seventh Framework Program (grant 318809). We are grateful to the Cloud & Heat company for providing us with the experimental platform, and in particular to Wojciech Barczynski for his help in this process.

References

[1] Cloud & Heat. http://cloudandheat.com.

[2] ClueWeb12 DataSet. http://www.lemurproject.org/clueweb12.php.

[3] CommonCrawl. http://commoncrawl.org.

[4] Apache Gora. http://gora.apache.org.

[5] Apache Hadoop. http://hadoop.apache.org.

[6] The Internet Archive Heritrix. http://crawler.archive.org.

[7] The Lemur Project. http://www.lemurproject.org.

[8] Apache Lucene. http://lucene.apache.org.

[9] Apache Nutch. http://nutch.apache.org.

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

[11] R. Baeza-Yates and C. Castillo. Crawling the infinite web: Five levels are enough. In S. Leonardi, editor, *Algorithms and Models for the Web-Graph*, volume 3243 of *Lecture Notes in Computer Science*, pages 156–167. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23427-2.

[12] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez. Crawling a country: Better strategies than breadth-first for web page ordering. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW, 2005.

[13] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *IEEE 23rd International Conference on Data Engineering*, ICDE, 2007.

[14] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Plachouras, and L. Telloli. On the feasibility of multi-site web search engines. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM, 2009.

[15] R. Baldoni, F. DAmore, M. Mecella, and D. Ucci. A software architecture for progressive scanning of on-line communities. In *IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2014.

[16] B. Ban. JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org, 2007.

[17] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004. ISSN 0038-0644.

[18] B. E. Brewington and G. Cybenko. How dynamic is the web? *Comput. Netw.*, 33(1-6):257–276, June 2000. ISSN 1389-1286.

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

[20] A. Z. Broder, M. Najork, and J. L. Wiener. Efficient URL caching for world wide web crawling. In *Proceedings of the 12th International Conference on World Wide Web*, WWW, 2003.

[21] B. B. Cambazoglu, V. Plachouras, F. Junqueira, and L. Telloli. On the feasibility of geographically distributed web crawling. In *Proceedings of the 3rd International Conference on Scalable Information Systems*, page 31. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[22] B. B. Cambazoglu, E. Varol, E. Kayaaslan, C. Aykanat, and R. Baeza-Yates. Query forwarding in geographically distributed search engines. In *Proceedings of the 33rd international ACM SIGIR Conference on Research and Development in Information Retrieval*, 2010.

[23] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the 11th International Conference on World Wide Web*, WWW, 2002.

[24] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 161–172, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[25] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[26] J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th International Conference on World Wide Web*, WWW, 2001.

[27] J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino. Geographical partition for distributed web crawling. In *Proceedings of the 2005 Workshop on Geographic Information Retrieval*, GIR, 2005.

[28] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, Apr. 1999. ISSN 1386-145X.

[29] IDC. EMC Digital Universe with Research & Analysis. http://www.emc.com/leadership/digital-universe/2014iview/index.htm.

[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC, 1997.

[31] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: Scaling to 6 billion pages and beyond. *ACM Trans. Web*, 3(3):8:1–8:34, July 2009. ISSN 1559-1131.

[32] F. Marchioni and M. Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[33] C. Olston and M. Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010. ISSN 1554-0669.

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

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC, 2013.