LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D1.4
Title of Deliverable:	Collaborative network and cost aware crawling
Contractual Date of Delivery:	M36 – 9/30/2015
Actual Date of Delivery:	9/30/2015

Abstract

This deliverable presents the different techniques we use to efficiently crawl the publicly available data with respect to the network conditions and the operational cost of LEADS and other third parties present.

This task aims to study and do research on algorithms that are able to efficiently fetch the publicly available data in a fully distributed environment with respect to (1) network conditions in terms of bandwidth, and (2) operational costs of LEADS clouds and third parties involved. Techniques have been developed to dynamically adjust the crawling task partition among LEADS clouds based on network performance. Cost models of the underlying LEADS clouds and third parties servers involved have been taken into account in terms of computational cost and communication cost, and have been used to partition the crawling tasks among LEADS clouds.

List of Contributors

Name	Organization	E-mail
Gaurav Singh	BM-Y!	gauravonline20@gmail.com
Ioanna Tsalouchidou	BM-Y!	ioanna@yahoo-inc.com
Janette Lehmann	BM-Y!	janettel@yahoo-inc.com
Necati Bora Edizel	BM-Y!	edizel@yahoo-inc.com
David Garcia Soriano	BM-Y!	davidgs@yahoo-inc.com
Aslay Cidgem	BM-Y!	aslayci@yahoo-inc.com
B. Barla Cambazoglu	BM-Y!	barla@yahoo-inc.com
Hossein Vahabi	BM-Y!	puya@yahoo-inc.com
Aggelos Aggelidakis	TSI	aaggelidakis@softnet.tuc.gr
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	christof.fetzer@tu-dresden.de
André Martin	TUD	andre.martin@tu-dresden.de
Do Le Quoc	TUD	do@se.inf.tu-dresden.de
Jons-Tobias Wamhoff	TUD	jons@inf.tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch
Anita Sobe	UniNE	Anita.Sobe@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Hossein Vahabi	puya@yahoo-inc.com	30.09.2015
Approved by GA Member 1	Etienne Riviere	etienne.riviere@unine.ch	17.09.2015
Approved by GA Member 2	Christof Fetzer	christof.fetzer@gmail.com	17.09.2015

Contents

List of Figures

List of Tables

Executive summary

LEADS is a decentralized Data-as-a-Service (DaaS) framework that runs on an elastic collection of micro-clouds to gather, store and process data. Data collection in LEADS is performed in a fully distributed way through two mechanisms: crawling-based data collection, which is a geographically distributed version of the traditional web crawling, and user-aided data collection, which relies on LEADS users to actively report their data to LEADS.

The user-aided data collection aims to collect user generated data and data that are difficult to be collected by crawling such as dynamic web pages. It utilizes a user publishing interface that enables real-time publishing of user generated content. The focus of deliverable D1.3 was the application-oriented data pre-processing, which ensures that only high quality data are incorporated to LEADS. Up to M12, we had mainly focused on the design and implementation of the user-aided publishing interface. For the M13 - M24 period covered by document D1.3, we reported on our efforts for pre-processing of user published data as well as user behaviour based web crawling and particularly page refreshing.

In this deliverable we aim to present different techniques to be used to efficiently crawl the publicly available data with respect to the network conditions and operational cost of LEADS and third parties servers involved. We present techniques and results of research on how to efficiently fetch the publicly available data in a fully distributed environment with respect to (1) network conditions, (2) operational cost of LEADS, and (3) operational costs of third parties servers involved. Cost models of the underlying LEADS clouds and third parties servers involved have been taken into account in terms of computational cost and communication cost, and have been used to partition the crawling tasks among LEADS clouds and to reach the optimal crawling.

1. Introduction

Web crawlers are integral components of large-scale web search engines. A web crawler is responsible for discovering new web pages on the web, refreshing the content of already downloaded pages, and storing the content. During these operations, a commercial-grade web crawler can issue a very large number of page download requests to the servers in the web, which has the side effect of increasing the carbon footprint of web servers. Another challenge is how to store the crawled data. Storing large quantity of data coming from web is not trivial at all, especially in a distributed system such as LEADS framework.

In this deliverable we propose techniques to solve a few challenges that today crawlers are facing, namely: (1) Can a crawler behave smartly in asking web pages so that it minimizes the amount of energy consumption over the servers? (2) Can a crawler have a crawling policy that minimizes the cost of LEADS framework? (3) How to store in a distributed system the large amount of available data using graph-partitioning algorithms?

The rest of the deliverable is organized as follows. In Section 2, we present a new crawling technique that takes into account the cost of network, communication, and computation of the third parties servers. In Section 2, we take into account the crawling cost of network, communication, and computation of the LEADS framework. Section 4 presents a graph-partitioning algorithm to dynamically partition the downloaded pages. Appendix A briefly reports the progress of the interaction between Ensembles, the LEADS geo-distributed storage platform implemented in WP2, and the Nutch-based web crawler developed in WP1 (see D1.2).

2. Geo-Distributed Crawling: an Optimal Scheduling Policy

We focus here on the problem of green web crawling from a set of web servers, where the goal is to reduce the carbon footprint incurred by a large-scale web crawler. Our objective is to devise a page refresh policy that minimizes the total staleness of pages subject to a constraint on the amount of carbon emissions due to web page processing and download at the distant server. At each time slot, the web server and web page for download are selected based on a metric that captures large page staleness, high greenness of the generated energy at the server premises, and small page size. Beyond creating an environment-friendly web crawler, our work draws guidelines for the design of large-scale commercial web search engine companies, which need to comply with certain greenness regulations.

2.1 Introduction

In the last decade, the continuously growing and rapidly changing nature of the web has turned large-scale web search engines into irreplaceable tools for accessing the information in the web. In practice, the operations of a typical commercial web search engine can be grouped under three main components: web crawling, indexing, and query processing [BBB+11]. The web crawling component is responsible for traversing the hyperlink structure among the web pages to discover and retrieve the content in the web. Finally, the user queries are evaluated over this index and each user query is matched to a set of pages that are estimated to be relevant to the query.

Given the vast size of the web and the ever-increasing volume of user queries, continuous operation of the above- mentioned components requires maintaining very large data centres. In particular, query processing is an expensive operation as determining the best-matching pages for a query requires accessing and processing large amounts of index data that is distributed over a large number of computers.

So far, a fairly large number of research works has investigated the techniques for reducing the query processing workload of a search engine, thus achieving reduction in energy consumption and savings in the electricity bill. This component is somewhat different than the indexing and query processing components in that it leads to energy consumption on the remote web servers whose pages are to be crawled. These devices consume energy while serving the requests of the web crawler, e.g., when fetching the requested pages from the disk, processing them in the CPU, and transferring pages over the local network devices.

The contributions of our deliverable are the following:

(1) We introduce the problem of reducing the carbon footprint of a large-scale web crawler and propose a related system model. Since we are interested in incremental web crawlers, which aim to preserve the freshness of the downloaded collection by re-crawling the discovered pages, we study a page refresh policy that minimizes the total staleness of web pages, while keeping the amount of carbon emissions low enough. At each time slot, the web server and web page for download are selected based on a metric that captures large page staleness, high greenness of the generated energy at the server premises, and small page size.

(2) For one server and one thread, the optimal web page download scheduling policy turns out to be a greedy one. At each time slot, the web server and web page for download are selected based on a metric that captures large page staleness, high greenness of the generated energy at the server premises, and small page size.

(3) We also conduct realistic experiments with real data on a large-scale web crawler obtained from a commercial web search engine. We propose different heuristics, along the lines of the optimal greedy policy, that use less information in order to study the trade-off between greenness and staleness of a web crawler. The problem is very relevant due to existing contracts in terms of the maximum incurred energy consumption at the servers, between the crawler and the remote servers, which need to be respected by the crawler during the download pro- cess. Our work shows that the carbon footprint generated by a web crawler during its external operations can be considerably reduced without compromising the freshness of pages, and draws guidelines for the design of large-scale commercial web search engine companies, which need to comply with certain greenness regulations.

The rest of this section is organized as follows. In Section 2.2, we describe our system model. In Section 2.3, the corresponding optimization problem is formulated, and the optimal policy is derived. In Sections 2.4, we describe our experimental setup and set forth numerical results obtained through simulations.

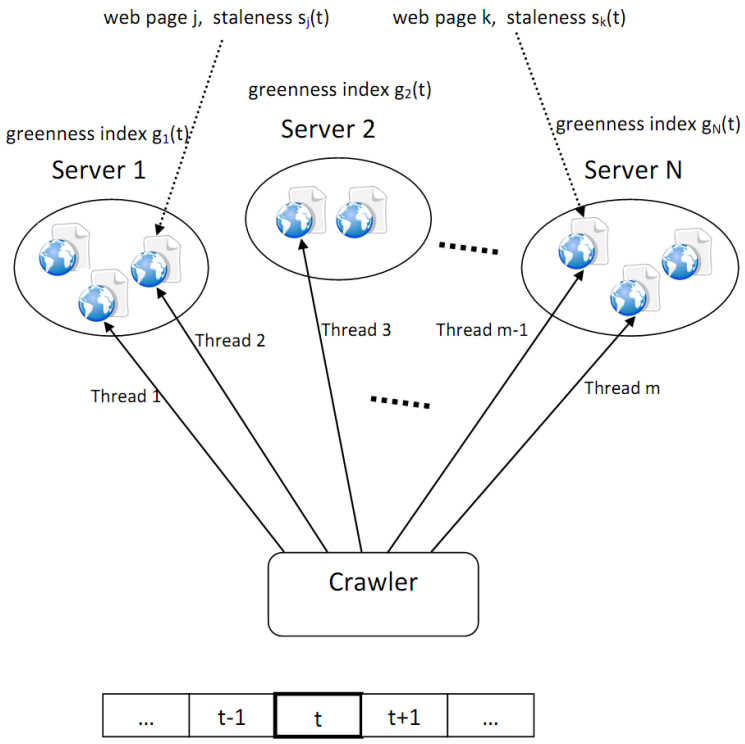


Figure 1: A depiction of the system we are considering with a crawler, N servers and the timeline split into slots.

2.2 The Considered Model

In this section we shall describe a model for our problem statement. In particular the aim of this section is to establish basic concepts that represents a crawler, webserver, web pages and what is for us a green policy.

Crawler

In large-scale search engines, clusters of computers perform web crawling where each computer runs multiple crawling threads, or even crawlers that are geographically distributed across the globe [BBB+08]. The crawler starts with a given set of seed URLs that are selected from hub web pages, which provide many links to important pages (content quality is an important objective for web crawlers [HGM+98]. The crawler first downloads the content of these pages to a local storage facility. To this end, it establishes a number m of threads that handle different HTTP connections to web servers hosting pages that are to be downloaded, and it retrieves the page contents provided by the servers.

In parallel, the downloaded pages are parsed, and potential new links to pages that have not been previously seen are discovered. Achieving a high coverage of the pages in the web is an important objective for a web crawler [RKU+07]. These are pages whose content is subject to change due to updates, leading to inconsistencies between the crawler's local copy of a page and the original copy in the web.

Web Servers and Web Pages

Figure 1 depicts the model that we assume to have here. Let us assume there exists a set *K* of web-servers, where the size of *K* is *N (|K| = N)*. Given a server $i \in K$, we consider *Wi* to be the set of pages hosted by server i. Also, we consider W to be the set of all the pages available in the considered scenario (in order words the union of the set of pages contained in all the considered servers).
We indicate with symbol j one single page, so $j \in Wi$ and we indicate the size of the page *j* with *Pj* that indicate the content size of page *j*.

Let us consider the following scenario, the crawler is asking to server *i* to retrieve and send to him the page *j*. Server *i* needs to retrieve, prepare and send through the network the page *j*. This process requires a certain consumption of energy by the third party server that needs to process the request. In particular it is necessary to fetch the page *j* from disk, or if it is a dynamic page to process the request using the CPU, then if it requires to access to database it is necessary to process the DB request and finally it is necessary to compress and transmit the final data. Figure 2 depicts a three-tier architecture where Data, Logic, and Presentation layer are considered independent.

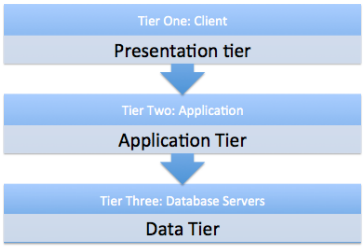


Figure 2: A depiction of the Three-Tier architecture in a client-server software architecture pattern, where presentation, logic and data tier are considered separated and maintained independently.

To introduce our energy based policy of crawling we assume a linear model of energy consumed to receive a web page *j*, which is, proportional to the size of the web page. Let us write it formally:

$$e_j = \alpha p_j + \beta, \text{ with } \alpha, \beta > 0,$$

Where e_j indicates the amount of energy consumed in order to serve the page j, while α, β are two constants, and p_j is the size of page j (as previously defined).

The assumptions are made to better describe the main characteristics of our optimal green crawling model that we will describe in the next section.

To take into account the refresh requirement of each web page we consider two additional factors: (1) page rank and (2) the likelihood of change of a particular web page. The page rank is clearly a measure of authoritativeness and it indicates the importance of a page. If a page is more important and authoritative than it will receive more in-links and therefore it should be refreshed more frequently due to its importance. Computing page rank is computationally expensive (*Ω(Number of links + Number of pages)*), but it can be done using the iterative method (power method). The change likelihood of the pages is determined in our case by considering the historical data. If a page has been changed many times during the last crawling process it means that it is highly probable that it is going to change in the future. In conclusion we assume high page rank and high likelihood of change to indicate a high refresh requirement. We define weights *Yj*, $\forall j \in Wi$ with $i \in K$, which indicate the different refresh requirements of web pages. Pages with high *Yj* should be refreshed more often. These weights can be estimated based on web crawler statistics and past crawling history.

Greenness

For each server i, we define a time-varying index *gi(t)* that indicates (on a given scale) the amount of carbon emissions per unit of consumed energy (Watthour). In fact, this index denotes the "greenness" of the energy consumed to run the server. For example, on a scale of [0,1], the carbon emissions (or else greenness) index of a very green energy source is close to zero (no carbon emissions-maximum greenness) and as the index increases the source becomes more "brown". For instance, a server may be powered entirely by clean nuclear or wind generated power or entirely by brown power generated from coal or lignite, or it can be powered by a mixture of these.

The carbon emissions index of a server varies with time, due to the time-varying output power generated by renewable sources or due to the fact that a high demand causes the local electricity company to increase the brown-ness. We assume that there is no a priori knowledge of *gi(t)* and that its value can be communicated from the server to the crawler only at the time that a decision needs to be taken.

Staleness

We assume that time is divided into slots. We assume that the time slot size is large enough to cover the download of the largest page, hence page downloads occur on a per time slot basis. This assumption is made to simplify the subsequent analysis and better expose the structure of the optimal policy. Nevertheless, it is not restrictive, since it still captures the different energy consumption incurred at remote servers.

At the beginning of each time slot, m threads are directed by the crawler towards m web pages that are selected for download. For each web page *j*, we need to define a measure of its staleness. Let *sj(t)* be the staleness of a web page *j* at the beginning of slot *t*. If this page is selected for download, then at most by the end of the slot the page download will finish and at the end of the slot its staleness will be 0. However, if this page is not selected, then its staleness will increase and at the end of the slot it will be *sj(t) + 1*. We observe that as long as the web page *j* is not selected, it becomes more and more stale, which means that the staleness of the page depends on the time elapsed from the end of the last download. It should be noted that the slot assumption above leads to a somewhat conservative consideration in terms of the computed staleness, in the sense that it leads to larger staleness increase for web pages that are not scheduled for download. However, even if this assumption is relaxed, the structure of the optimal policy presented below is not expected to change.

2.3 The Problem Statement

We are interested in the problem of scheduling the web page download threads from the crawler to the web servers with the objective to keep the web pages as much as possible fresh and the carbon emissions due to page download requests low enough. The decision at each time *t* is to pick a server $i \in K$ and a web page $j \in Wi$ to download in such a way that the total staleness of the web pages is minimized and the amount of carbon emissions is kept below a given threshold.

On the one hand, we would like to choose web pages with large staleness. On the other hand, we would like to schedule downloads of web pages from servers that have low index *gi(t)*, that is high greenness of energy. Also, out of all pages it is not clear whether we would like to schedule for download the ones with the smaller size or the ones with larger size. Smaller size web pages consume less energy. However, larger web pages should also be downloaded at some point in order to reduce their staleness. Moreover, there may be pages with high refresh requirements γj , which should be given high priority in the download process. The joint consideration of all parameters above and the conflicting objectives of keeping the web pages fresh and the carbon emissions at the remote servers low, makes the thread scheduling problem non-trivial and calls for a thorough investigation.

Single web server scheduling problem

First, we investigate the simple case of one server that hosts a set *W* of web pages. As we mentioned above the download time is one slot $\forall j \in W$, and we assume that during this slot *g(t)* remains stable. Also, at each time *t, m = 1* thread is sent to fetch one web page. We define the variable *x(t) = (xj (t) : j ∈ W)*, where *x (t) = 1* if at time t the web page $j \in W$ $j\sum$ is downloaded, else *xj (t) = 0*. Clearly, $j \in W$ *xj (t) = 1*, $\forall t$, since at each time *t*, only one thread to a web page is allowed to be active to the server. The time evolution of the staleness of a web page j can now be written as *sj(t + 1) = (sj (t) + 1)(1 − xj (t))*. Consider an interval *[0,T]* of interest. Our goal is to find a policy *x∗ = (x(t) : t = 1,...,T)* that minimizes the total (over time and over pages) staleness of the server, i.e.,

$$\min_{x*} \sum_{t=0}^{T} \sum_{j \in W} S_j(t)$$

subject to the constraint that the total amount of carbon emissions due to page download requests does not exceed a given threshold *G*, i.e.,

$$\text{s.t. } \min_{x*} \sum_{t=0}^{T} \sum_{j \in W} x_j(t)\, e_j g(t) \leq G$$

where *G* is set by the agreement between the crawler and the remote server.

The exact details on how to solve the previous equation are going beyond the scope of this document, and it can be found in Appendix 2. However to continue with our consideration, let us assume to have a parameter *λ* (Lagrange multiplier $\lambda \in R+$) denoting the significance of greenness for the server. Its value, which is set in consultation with the crawler, depends on the capability of using green energy at the local server premises. If there is no such capability, then *λ = 0* and the crawler's goal is to just minimize the total staleness. On the other hand, if there is high potential for energy from renewable sources, then the value of *λ* is high. In that case, besides minimizing page staleness, the crawler also wishes to keep the carbon footprint as low as possible, as suggested by the values of *λ* and *G* . We observe that as *λ* increases, the amount of carbon emissions becomes more and more important for the server and the web page download scheduling is largely determined by its value.

Since parameter *g(t)* is not known a priori but can only be communicated to the crawler scheduling engine at the time of decision *t*, we focus on studying the online version of the problem. In this case, we observe that the objective that we are studying can be decomposed to separate terms to be optimized with respect to the scheduling decision only at time *t*, which means that the crawler's decision at slot t is independent of the decisions made at the other slots.

Formally, the optimal policy at each slot *t* involves greedy decision making and is the following: at each time slot t, the crawler chooses *x(t)* such that it maximize the following function:

$$\sum_{j \in W} \left(s_j(t) - \lambda e_j g(t) \right) x_j(t)$$

subject to,

$$\sum_{j \in w} x_j(t) = 1$$

for the selected web page *j*(t)*, it holds:

$$j^*(t) = \arg \max_{\text{j} \in \text{W}} (s_j(t) - \lambda e_j g(t))$$

which means that at each time *t*, given *g(t)*, the crawler decides based on the values of *sj(t)* and *ej*, for $j \in W$. Observe that in the special case where all pages need the same energy to be downloaded, i.e. *ej = e*, $\forall j \in W$, the crawler chooses the web page *j* with the maximum staleness *sj(t)*. On the other hand, if all pages have the same staleness at time *t*, i.e. *sj(t) = s*, $\forall j \in W$, then the crawler chooses the page *j* with the minimum required energy *ej*, i.e. the one with the minimum size *p*.

Multiple web server scheduling problem

In this paragraph we describe briefly how the previous problem definition and model can be extended to the case of multiple servers. In particular we try to give a general overview of the model used, the mathematical details are going beyond the scope of this document. In order to extend our problem for the case of multiple servers we assume there exists a set *K* of *N* web servers in our system, where each server has its own index of greenness *gi(t)*. Again, we assume that at each time *t*, only one thread to a web page is allowed to be active at a time to any server. Therefore we can define formally the problem that we are considering:

$$\sum_{i \in k} \sum_{j \in W_i} x_j(t) = 1 \; \forall t$$

Again, our goal is to find a policy $x^* = (x(t): t = 1, \ldots, T)$ that minimizes the total staleness (over time and over pages) of all servers, i.e.

$$\min_x * \sum_{t=0}^{T} \sum_{i=1}^{N} \sum_{j \in W_i} S_j(t)$$

subject to the constraint that the total amount of carbon emissions of each server *i* does not exceed a given threshold *G* i, i.e., such that:

$$\sum_{t=0}^{T} \sum_{j \in W_i} x_j(t) e_j g_i(t) \leq G_i, \forall i \in K$$

Following a similar approach as previously shown (note that we omit the mathematical detail and the proof), the optimal scheduling policy is the following: at each time t, the crawler chooses the web server *i* and the web page $j \in Wi$ that maximize,

$$S_j(t) - \lambda_i e_j g_i(t).$$

We see that at each time *t*, the crawler makes its decisions based on the values of *sj(t)*, *ej*, *λi* and *gi(t)* for *i=1,...,N* and $j \in Wi$. To help reader understand better the previous model we identify two special cases:

(1) If at time *t*, *sj(t)* is constant, it means that all the pages have the same staleness value for each server *i* and page *j*, than the crawler choses the serve and the page with the minimum product *λi * ej * gi (t)*. If it is also *λi* =constant and $ej = e, \forall i \in K, \forall j \in Wi$, then the crawler chooses randomly a web page from the server with the minimum index *gi(t)*.

(2) If *λi* = constant and $ej = e, \forall i \in K, \forall j \in Wi$, then the crawler chooses the server $i \in K$ and the web page $j \in Wi$ with the minimum *sj(t) − gi(t)*. If also at time *t*, *gi(t) = g(t)*, $\forall i \in K$, then the crawler chooses out of all the pages the web page *j* with the maximum staleness *sj(t)*.

2.4 Results

Dataset

To perform experimentation we use a collection of web pages sampled from a large web crawl performed by a commercial web search engine in 2011. Hence, our collection is large and also represents high-quality content that is of importance to a web search engine.
We processed the URLs of pages to obtain the names of web servers (i.e., a combination of protocol, host, and port information extracted from the URL). In our simulations, we use this information to

assign the web server to a country and hence estimate a greenness value for the server depending on the time zone of its country and the time of a page download request.

Staleness and Greenness computation from our data

First, we assume that each server can be powered by a mixture of solar (green) energy (during the day) and energy provided by the grid (brown) (during the night).
Based on the time-zone, latitude and longitude of each server, we find the day length as well the sunrise and sunset times at the server location. Since the carbon emissions index varies between 0 and 1, we assume that in the middle of the day it takes its minimum value 0 (which means maximum greenness) and during the night it takes its maximum value 1 (i.e. E.g., if for a server i located in a country in the northern hemisphere, the sun in winter rises at 07:30 a.m. and sets at 17:00 p.m., the carbon emissions index is 1 at 07:30 a.m., 0 at 12:15 p.m., again 1 at 17:00 p.m. and 1 during the night. For the pair of points (07:30,1), (12:15,0), we get a1 = −0.21 and b1 = 2.57 while for the pair (12:15,0), (17:00,1) we get a2 = 0.21 and b2 = −2.57.

We assume the downloading time of a web page *j* to be *Kj* slots, where $Kj = l + pj/b$, with l that indicates the network latency between the crawler and the server, *b* is the bandwidth among them and *pj* the size of the page. The nature of the data does not permit us to estimate the staleness of each page, therefore we compute the staleness of the server *i*, *Si(t)*. We increase the total staleness of server *i*, at time *t0*, and which selects to download a page *j*, by *kj* (the number of slots necessary to download the page) at time *t0+kj*, in other words: *Si(t0+kj) = Si(t0)+ kj * [number of pages in server i]*. Now, at time *t0+kj+1*, after downloading one page, we decrease the staleness by a factor proportional to *([number of pages in server i] - 1) / [number of pages in server i]*.

We fix *l=100ms* and *b=1Mbps*. First, we study the performance of the optimal policy as a function of parameter λ. Here, we use synthetic data as a way to study the performance at the level of web pages. We use a set of 1000 pages and normalized values between 0 and 1 for the required energy to download each page. We assume that staleness is measured in minutes (min) and that the amount of carbon emissions at each time *t*, *ejg(t)*, is measured in grams (g). Since, the crawler at each time decides based on the values of $(sj(t) - \lambda ejg(t))$, the measurement unit of λ is minutes/grams (min/g).

Figure 3 and Figure 4 show the performance of the optimal policy in terms of total (over time and over pages) staleness and total carbon emissions, respectively. It can be seen that as the value of λ increases, the total staleness increases in a non-linear, convex manner, whereas the total carbon emissions decrease almost linearly. It is obvious that there is a trade-off between web page staleness and server greenness. These results stem from the fact that as λ increases, the potential of using green energy increases as well and the server wants more and more to take advantage of this potential. Thus, although the crawler wants to minimize the staleness of its downloaded pages, he is hindered by the server's desire to keep its carbon emissions low.

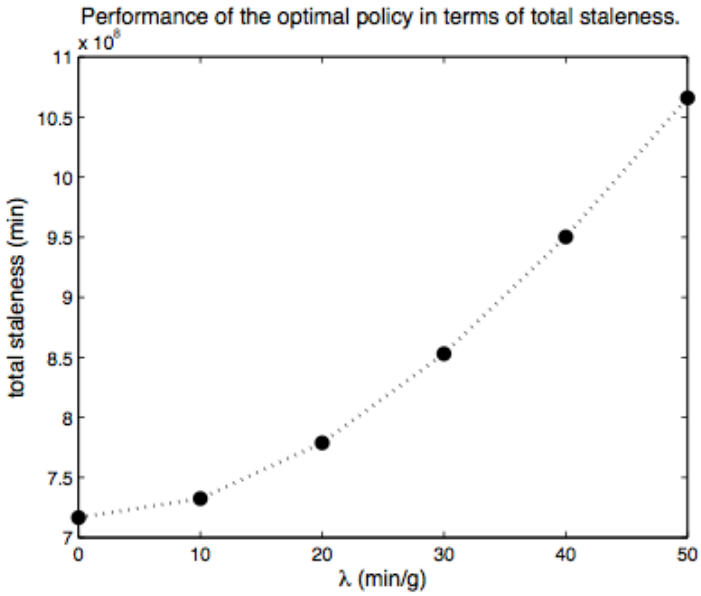


Figure 3: The performance of the optimal policy. We consider the total staleness with respect to lambda.

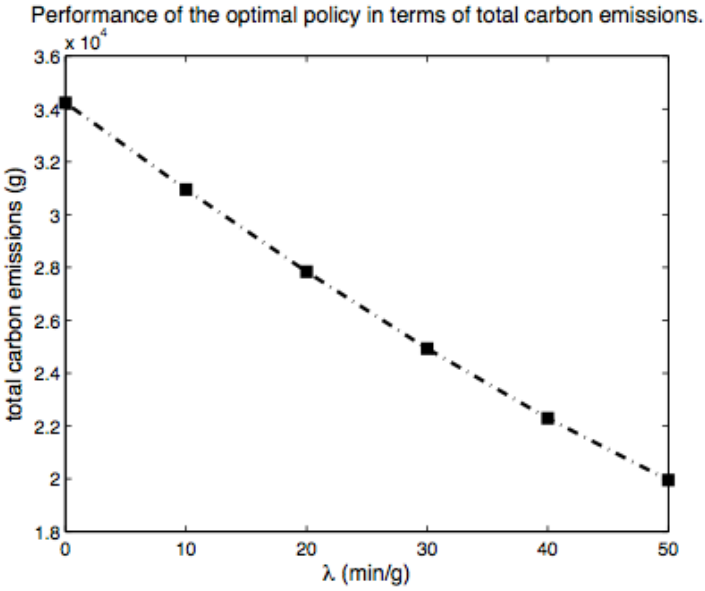


Figure 4: The performance of the optimal policy. We consider the total carbon emission with respect to lambda.

3. Geo-Distributed Crawling: Infrastructure Cost

The key-limiting factor of crawler architecture is its large infrastructure cost. To reduce this cost and adapt to the micro-clouds federation architecture of LEADS, we developed in WP1 UniCrawl, a geo-distributed crawler solution.

UniCrawl orchestrates the geographically distributed micro-clouds present in LEADS. Each micro-cloud 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 micro-clouds and federates their storage and computing resources, while minimizing the inter-micro-cloud communication cost.

We conducted several experiments over 3 micro-clouds spread across Germany provided by our partner Cloud&Heat. When compared to a centralized architecture with a crawler simply stretched over these locations, UniCrawl shows a performance improvement of 93.6% in terms of network bandwidth consumption, and a speedup factor of 1.75.

This section gives an overview of Unicrawl, and details some of our performance results. The interested reader may refer to our white paper [UNICR5] for additional details.

3.1 Distributed Crawler Architecture

The general algorithmic sequence used by a web crawler 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 analyses 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 introduce some serious system design issues. This section presents how the design of UniCrawl addresses those problems, describing first the internals of a micro-cloud, then cross-micro-cloud operations.

A. Single micro-cloud Design

At each micro-cloud, we build UniCrawl upon the well-established architecture of Apache Nutch [UNICR2], more specifically its version 2.x. We contribute to Nutch the necessary improvements and novel features that we shall cover next. Figure 5 depicts an overview of the architecture. UniCrawl makes use of the MapReduce 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.

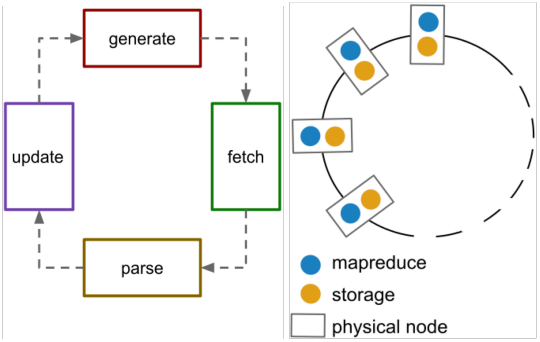


Figure 5: Overview of UniCrawl at a micro-cloud.

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 micro-cloud 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, 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 neighbours that it is joining. *(Storage)* Infinispan uses consistent hashing 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. This library uses 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. It allows to execute SQL-like queries over the indexed content of the node.

B. Multi-micro-cloud Operations

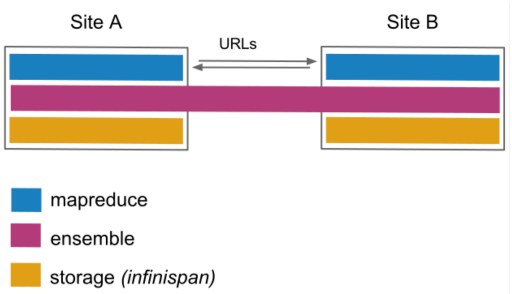


Figure 6: Multi-micro-cloud architecture of Unicrawl.

Figure 6 depicts the architecture of UniCrawl when we deploy it across multiple geographical locations. In a typical set-up, UniCrawl is composed of several micro-clouds interconnected with a wide-area network, and each micro-cloud is equipped with a few dozens of machines communicating through a low-latency network.

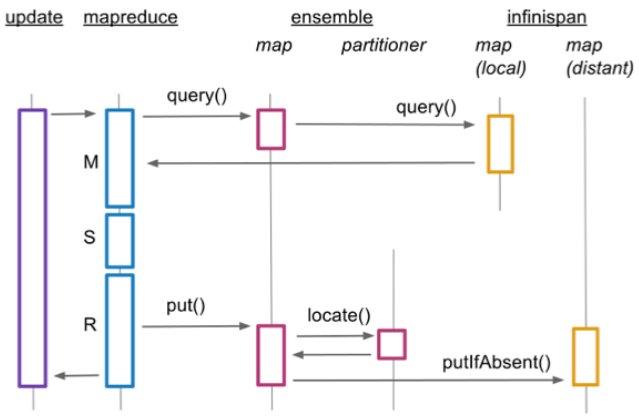


Figure 7: Lifetime of the update phase.

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

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

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

Ensemble offers the same concurrent map interface to the MapReduce 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 micro-cloud 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 micro-cloud 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 micro-clouds: 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 7.

In detail, we implement the crawl database as a distributed Ensemble map that spans all the micro-clouds. 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 micro-clouds, while *(ii)* all the keys accessed by the generate, fetch and parse phases are local to

the micro-cloud. This allows micro-clouds 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 micro-cloud 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. [UNICR3]. The distance-based partitioner is more involved but reduces the distance between a web server and its corresponding fetching micro-cloud, thus lowering the network cost associated to it. This partitioner relies on an Ensemble map *D* replicated at all micro-clouds 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 micro-cloud and returns the associated Infinispan map.

Notice that in UniCrawl, micro-clouds operate independently and execute their update phases at different times. As a consequence, a micro-cloud that finds a new URL uses a *putIfAbsent* operation when it accesses distant micro-cloud storage. This avoids race condition in case the URL was already crawled. Furthermore, during the update phase, each reducer *(i)* outputs at most l/m URLs, where *m* is the number of participating micro-clouds and l the overall size of the frontier, and *(ii)* it stops after it has retrieved *l* URLs local to its micro-cloud. The former modification avoids to overwhelm a micro-cloud, whereas the latter preserves them from starvation. We also makes use of a caching mechanism at each site, to avoid resending a URL (see [UNICR5] for further details).

Our latest version of Unicrawl supports the retrieval, storage, and querying of multiple versions of a web page. To support this feature, we implemented two modifications to the base crawling algorithms we described previously. First, the generate phase now retrieves all the existing pages from the database, and checks that they have to be re-crawled or not. If this is the case a novel version of the page is created. Second, during the update phase, links that are coming from outdated versions of a page are discarded, and thus not taken into account during the scoring computation.

3.2 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 micro-clouds. We report several experimental results where we deploy UniCrawl at multiple locations in Germany and access actual web micro-clouds.

We use several clouds provided by our LEADS partner Cloud & Heat [UNICR1], and we deployed UniCrawl at multiple geographical locations in Germany. At each location, 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 8 indicates the clouds' locations.

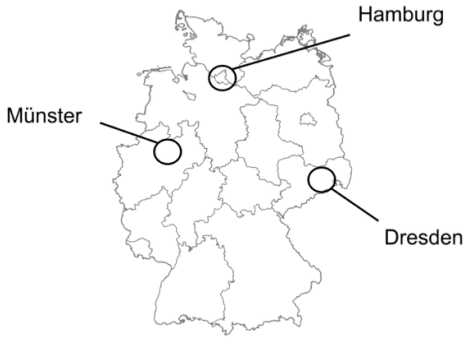
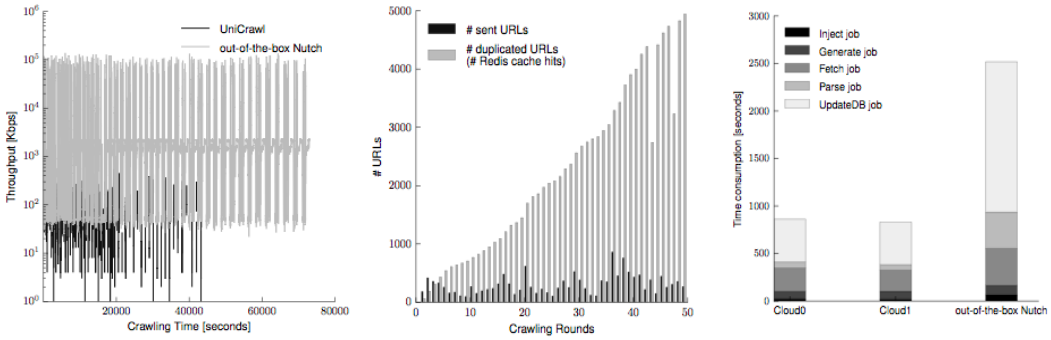


Figure 8: Location of clouds.

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 micro-cloud 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. Exchange rate at Hamburg b. Network traffic c. Scalability

Figure 9: Evaluation in a geo-distributed setting.

1) URL Exchange: In Figure 9(a), we evaluate the URLs exchange rate at Hamburg, when UniCrawl spans both this location and Münster. We observe that on average the Hamburg micro-cloud 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. [UNICR4], 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 Cassandra3 backend over multiple geographical locations. To that end, we deploy UniCrawl and Nutch at both Hamburg and Münster, and measure the traffic consumption between micro-clouds. Figure 5(b) 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 9(b) that UniCrawl is around 1.75 times faster than the Nutch deployment. Such a gap comes from the fact that Nutch executes the MapReduce 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-micro-cloud 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. 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(c) 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 micro-clouds. 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 MapReduce steps are geo-distributed and that the update phase is the most demanding, both in terms of storage and processing power. Figure 9(c) also tells us that the total time to execute a crawling round improves when we scale UniCrawl from two to three micro-clouds. 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 micro-clouds.

4. Geo-Distributed Graph Partitioning

4.1 Introduction & Model

In this section we discuss techniques to dynamically adjust the partitioning of web crawling among the LEADS micro-clouds. Such a partition maps the web graph nodes (i.e., the URLs) to a fixed set of sites. Recall that each site crawls its own component of the partition (i.e., retrieves the web pages pointed by the URLs). When a site finds a URL belonging to another partition, it sends the URL to the site in charge of this partition. Hence, our goal is to minimize cross-site communication (i.e., the number of links between different components in the partition) usage while (approximately) spreading the load across all the sites. Moreover, we would like to maintain this partition efficiently as new webpages and links are added.

We view the web as a (dynamic) graph, where vertices correspond to webpages and edges to links between them; each component corresponds to a site. A simple method that is sometimes used is based on simple hashing: assign each new vertex to a randomly chosen component out of k (chosen, for example, by applying a random hash function to the site's URL). Observe that, while this method is sometimes used in practice because of its simplicity, it performs very badly in terms of minimizing the number of crossing edges. Indeed, the probability that any given edge will be cut by a random partition into k sites is $1\text{-}1/k$, meaning that if k is large nearly all the edges will be cut, even if there is a solution with no crossing edges at all. Hence better algorithms are needed to keep the number of crossing edges low.

Below we study algorithms and heuristics to minimize the number of crossing edges while putting an upper bound in the size of the components (sites). We start by considering the static case, in which a graph is built from an initial set of web content, and then proceed to analyse how to handle dynamic updates to the partition as new sites are crawled.

Balanced graph partitioning

Let *G* be a graph with *n* vertices and *m* edges. Given two parameters $k \in N$ and $v \in [1, \infty]$, the *balanced graph partitioning* problem attempts to find a partition such that no component contains more than *[v*n/k]* of the graph vertices, and minimizes the number of edges connecting different components. It has wide-ranging applications in VLSI design, image segmentation, parallel computing, data mining, social network analysis, etc., and has been extensively studied from the point of view of algorithmic complexity and practical heuristics [BMS+13].

The special case where *k*=2 is the well-known *minimum bisection* problem, which is a classical NP-hard problem [GJS76]. While this precludes exact computation of the minimum bisection on reasonably-sized graphs, there are algorithms to efficiently find solutions with provable quality guarantees. The first polynomial-time approximation algorithm with a sub-polynomial approximation guarantee is due to Feige and Krauthgamer [KF06], who present a procedure to find a bisection whose cost (number of crossing edges) is within ratio $O(\log^{1.5} n)$ of optimal; the best approximation factor known is $O(\log n)$ [FF15]. It is also known that minimum bisection cannot be approximated arbitrarily well in polynomial time, unless NP admits randomized sub-exponential-time algorithms [Kho06], which is widely believed to be unlikely. Interestingly, these hardness results continue to hold even when the input is restricted to 3-regular graphs [BK02].

On the other hand, when k is not part of the input (i.e., not a constant) and $v=1$, no polynomial-time can achieve a finite approximation ratio unless $P=NP$, as shown in [AR06]. It is for this reason that, if we are to find efficient algorithms for balanced graph partitioning, we need to forego the requirement for perfectly balanced solutions $(v=1)$ and allow some slack in the size of the components (determined by the parameter v). Simon and Teng [ST97] showed that if $v=2$, balanced graph partitioning can be reduced to finding separators via a greedy recursive procedure, yielding an $O\left(\log k\sqrt{\log n}\right)$ approximation guarantee; see also [LMT90]. (Their papers only claim an $O\left(\log k \log n\right)$) approximation based on Leighton and Rao [LR99], but the improved factor can be obtained through the breakthrough result of Arora et al [ARV09]. Even et al [ERS99] present an algorithm with approximation factor $O(\log n)$ for any $v \geq 2$, later improved to $O(\sqrt{\log n \log k})$ by Krauthgamer et al. [KF06].

For $1<v<2$, Andreev and Räcke [AR06] gave an algorithm with approximation factor $O(\frac{\log n^{1.5}}{(v-1)^2,})$ via the use of spreading metric relaxations. The result was improved by Feldmann and Foschini, who present a dynamic programming-based PTAS for trees and derive an $O(\log n)$ approximation for general graphs via approximation by tree metrics [Räc08]. Note that their approximation factor is independent of v, but the running time is exponential in $v-1$.

These theoretical results rely on the use of linear and/or semi-definite programming solvers, which limit their applicability to very large graphs. Due to the practical importance of the problem, many fast heuristics have been devised. In particular, a multi-level graph partitioning heuristic called METIS (based on [KK98]) has proven very successful in practice, even though it does not provide any guarantee on the quality of the solution found.

Streaming graph partitioning

Very recently, researchers have taken on the problem of designing streaming algorithms for partitioning where the decision of placement of each vertex needs to be performed “on the fly”. We consider *incidence streams*, in which a new vertex comes accompanied with all its edges to previous vertices. The algorithm processes a vertex and, depending on its neighbourhood in the graph and the current decision, assigns the new vertex to a given component. For our purposes we need to maintain the consistency property that placement decisions are final: a vertex cannot reallocated after it has been assigned to one of the k components. Unfortunately, if vertices arrive in adversarial or even random order, this consistency constraint makes it impossible to approximate balanced graph partitioning within a sub-linear factor [Sta12]. Therefore we need to make assumptions on the order in

which vertices are received.

To fix notation, let $P^t = \cup_{i=1}^{k} P^t(i)$ refer to the partition at time *t*, with components $\{P^t(i)\}_{i \in [k]}$. Let v denote the vertex arriving at time *t*, *Γ(v)* the set of neighbours of *v*, and *C=vn⁄k* the maximum size of a component.

Stanton [Sta12] considers a broad range of heuristics and evaluates their performance on random stream order, BFS order, and DFS order (in the last two, the starting vertices are chosen at random from the graph). She proposes several easy-to-evaluate functions based on the number of connections and triangles between the new vertex and each element of the partition so far, and the size of the components. According to her experimental findings, the best-performing heuristic is the *linear-weighted deterministic greedy*, which places v to the component whose index *i* satisfies:

$$i = max_{i \in [k]} \{|P^t(i) \cap \Gamma(v)|(1 - \frac{|P^t(i)|}{C}) \}$$

Tsourakakis et al [TGRV14] introduce a new streaming method inspired by modularity maximization. They relax the objective function by adding a term that grows when the components are imbalanced; this term is parameterized by a constant *γ≥1*. They present a greedy algorithm that looks among those components that will not violate the maximum size constraint and picks the index i that satisfies

$$i = \max_{\substack{i \in [k] \\ |P(i)<C|}} \left\{ |P^t(i) \cap \Gamma(v)| - \frac{\gamma\, m\, k^{\gamma-1}}{2\, n^{\gamma}} \right\} \left((P^t(i) + 1) \right)^{\gamma} - P^t(i)\hat{}\gamma) \}$$

Based on experimental results, they suggest to choose *γ=3⁄2* and a balance factor *ν=1.1* .

4.2 Experiments

We report an experimental evaluation of the two streaming algorithms for graph partitioning outlined above, on graphs obtained from a commercial web search engine crawler. We use graphs of 5 different sizes derived from crawls with random starting points and different stopping times. We use a range of partition sizes and report the fraction of edges cut by the following three methods:

- METIS: state-of-the art offline partitioning method.
- Linear-weighted deterministic greedy, LDG (from [Sta12]) for streaming partitioning.

FENNEL (from [TGRV14]) for streaming partitioning, with $\gamma=3/2$.**Table 1** gives information about the number of vertices and edges of the datasets used.

Graph	N	M
G1	968	7680
G2	7721	23154
G3	39763	198790
G4	71005	709950

Table 1: Number of vertices and edges of the datasets used.

The table below reports the fraction of edges cut by each of these methods for each graph/partition size pair. We report average results over 10 random orderings. For all methods we use a load parameter ν=1.1 (recall that this means that we allow some components to be of size 10% larger than a perfectly balanced partition would allow).

Graph	K	METIS	LDG	FENNEL
G1	2	0.36	0.49	0.38
G1	4	0.54	0.74	0.50
G1	8	0.65	0.86	0.70
G1	16	0.72	0.93	0.75
G2	2	0.24	0.50	0.31
G2	4	0.38	0.75	0.48
G2	8	0.47	0.87	0.58
G2	16	0.53	0.92	0.64
G3	2	0.30	0.50	0.35
G3	4	0.46	0.75	0.54
G3	8	0.57	0.87	0.65
G3	16	0.64	0.92	0.72
G4	2	0.37	0.49	0.39
G4	4	0.56	0.74	0.60
G4	8	0.67	0.87	0.72
G4	16	0.74	0.93	0.78

Table 2: The table reports the fraction of edges cut by each of these methods for each graph/partition size pair.

From this we conclude that FENNEL outperforms the linear-weighted deterministic greedy algorithm on web graphs (of interest to LEADS), when vertices arrive in random order. (In fact, for the graphs we used, the performance of LDG is not significantly better than that of random partitioning.) Moreover, FENNEL produces partitions of quality comparable to those of the offline partitioning method METIS (typically within 10%), suggesting that it may not necessary to perform an initial offline partitioning based on a sample of the graph prior to use the streaming algorithm, as the gain in the number of edges cut are small.

5. Conclusion

Web crawlers are used to crawl publicly available web data. In this deliverable we have presented different optimal solution to solve the LEADS web crawling challenges in a distributed environment.

In particular we presented techniques to efficiently crawl by taking into account the operational cost of LEADS, and third parties servers involved. Cost models of the underlying LEADS clouds and third parties servers involved have been taken into account in terms of computational cost and communication cost, and have been used to partition the crawling tasks among LEADS clouds and to reach optimal crawling.

We introduced the problem of green web crawling from a set of web servers, with the goal to reduce the carbon footprint incurred by a large-scale web crawler, while keeping the freshness of downloaded pages high enough. We proved that the optimal solution, for a single server and a single scheduling thread, is a greedy page download scheduling policy that at each time decides based on a metric that captures large page staleness, high greenness of the generated energy at the server premises, and small page size.

Furthermore we have presented how to deal with the large amount of data in a distributed environment using graph partitioning algorithms.

6. References

[AR06] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput.Syst.*, 39(6):929–939, 2006.

[ARV09] Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.

[BBB+11] Berkant Barla Cambazoglu and R. Baeza-Yates, "Scalability Challenges in Web Search Engines", Advanced Topics in Information Retrieval, vol.33, pp.27-50, 2011.

[BBB+08] Berkant Barla Cambazoglu, V. Plachouras, F. Junqueira and Telloli, "On the feasibility of geographically distributed web crawling", Proc. 3rd International Conference on Scalable Information Systems, 2008.

[BK02] Piotr Berman and Marek Karpinski. Approximation hardness of bounded degree MIN-CSP and MIN-BISECTION. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, pages 623–632, 2002.

[BMS+13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[ENRS99] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, 1999.

[FF15] Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.

[HGM+98] J. Cho, H. Garcia-Molina, and L. Page, "Efficient Crawling Through URL Ordering", *Computer Networks and ISDN Systems*, vol.30, no.(1- 7), pp.161-172, 1998.

[KF06] Robert Krauthgamer and Uriel Feige. A polylogarithmic approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.

[Kho06] Subhash Khot. Ruling out PTAS for graph min-bisection, dense k -subgraph, and bipartite clique. *SIAM J. Comput.*, 36(4):1025–1071, 2006.

[KK98] George Karypis and Vipin Kumar. A feast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.

[LMT90] T. Leighton, F. Makedon and S. Tragoudas. Approximation algorithms for VLSI partition problems. In *Proceedings of the IEEE International Symposium on Circuits and Systems,* 1990.

[LR99] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[RKU+07] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins, "The Discoverability of the web", Proc. 16th Int'l Conf. on World Wide Web, 2007.

[Räc08] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 255–264, 2008.

[ST97] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM J. Scientific Computing*, 18(5):1436–1445, 1997.

[Sta12] Isabelle Stanton. Streaming balanced graph partitioning for random graphs. *CoRR*, abs/1212.1121, 2012.

[TGRV14] Charalampos E. Tsourakakis, Christods Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342, 2014.

[UNICR1] Cloud & Heat. http://cloudandheat.com.

[UNICR2] Apache Nutch. http://nutch.apache.org.

[UNICR3] A. 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.

[UNICR4] 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.

[UNICR5] Do Le Quoc, Christof Fetzer, Pierre Sutra, Valerio Schiavoni, Etienne Rivière and Pascal Felber, UniCrawl: A Practical Geographically Distributed Web Crawler, in Proceedings of the 8th IEEE International Conference on Cloud Computing, CLOUD, 2015