

Information and Communication Technologies (ICT) Programme

Project No: FP7-ICT- 248036

## COAST



---

### ***D3.1: Search Engine Architecture design and metrics for evaluation***

---

**Author(s):** X. Bai, B. B. Cambazoglu, F. Junqueira, I. Kelly, V. Leroy (Yahoo, BM), V. Cruz (TID)

**Status -Version:** Final

**Delivery Date (DOW):** 31 January 2011

**Actual Delivery Date:** 16 February 2011

**Distribution - Confidentiality:** Public

**Code:** COAST\_D3.1\_YAHOO\_FF\_20110216

**Abstract:** In this deliverable, we describe the COAST search engine architecture. First, we present the interactions between the distributed components of the search engines. Then, we go into the details of a specific search site. Finally, we present the metrics, data, and experimental setups that will be use to evaluate the efficiency of this architecture.



## Disclaimer

This document contains material, which is the copyright of certain COAST contractors, and may not be reproduced or copied without permission. All COAST consortium partners have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

The COAST Consortium consists of the following companies:

No	Participant name	Participant short name	Role	Country
1	ST Microelectronics	STM	Co-ordinator	Italy
2	Synelixis Solutions Ltd	Synelixis	Contractor	Greece
3	Yahoo Iberia SL	Yahoo	Contractor	Spain
4	NEC Europe Ltd	NEC	Contractor	UK
5	Telefonica Investigacion Y Desarrollo SA	TID	Contractor	Spain
6	Fraunhofer HHI	HHI	Contractor	Germany
7	Politecnico di Torino	Polito	Contractor	Italy
8	Technische Universitaet Berlin	TUB	Contractor	Germany
9	Fundacio Barcelona Media	BM	Contractor	Spain
10	University of California, Los Angeles	UCLA	Contractor	USA
11	Seoul National University	SNU	Contractor	S. Korea

The information in this document is provided “*as is*” and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.



This page has been intentionally left blank



# Table of contents

- 1. Introduction ..... 8**
- 2. Overview ..... 9**
- 3. Multi-site search engine ..... 11**
  - 3.1. Content acquisition ..... 11*
    - 3.1.1. Approaches ..... 11
    - 3.1.2. Distributed crawling..... 15
  - 3.2. Distributed indexing ..... 17*
    - 3.2.1. Selecting a master indexer ..... 17
    - 3.2.2. Replicating content..... 18
  - 3.3. Distributed query processing..... 19*
    - 3.3.1. Query forwarding..... 19
    - 3.3.2. Integrating popular tail content ..... 21
- 4. Inside a search site ..... 22**
  - 4.1. Crawling ..... 22*
    - 4.1.1. Content quality considerations..... 22
    - 4.1.2. Performance considerations..... 24
    - 4.1.3. Constraints on crawlers ..... 25
  - 4.2. Indexing..... 25*
    - 4.2.1. Inverted index..... 25
    - 4.2.2. Index maintenance ..... 26
    - 4.2.3. Index partitioning ..... 27
  - 4.3. Query Processing..... 27*
    - 4.3.1. Preprocessing..... 27
    - 4.3.2. Top-k processing ..... 28
    - 4.3.3. Query processing on a partitioned Index ..... 28
    - 4.3.4. Snippet generation ..... 29
  - 4.4. Caching..... 29*
    - 4.4.1. Posting list caching ..... 29
    - 4.4.2. Query result caching..... 29
    - 4.4.3. Other types of caching..... 29
  - 4.5. Coordination..... 30*
    - 4.5.1. Configuration ..... 30
    - 4.5.2. URL Exchange..... 31
    - 4.5.3. Page Exchange ..... 31
    - 4.5.4. Index statistics exchange ..... 32
    - 4.5.5. Threshold Exchange..... 32
    - 4.5.6. DPI sink ..... 33
    - 4.5.7. Content submission sink..... 33
- 5. Metrics and evaluation ..... 34**
  - 5.1. Crawling ..... 34*
    - 5.1.1. Active crawling..... 34
    - 5.1.2. Passive crawling..... 35
  - 5.2. Search performance ..... 37*
    - 5.2.1. Data set..... 37
    - 5.2.2. Initial placement of documents..... 37
    - 5.2.3. Document replication ..... 38



5.2.4. Query forwarding.....	38
5.2.5. Search latency.....	39
5.3. Search quality.....	39
5.3.1. Document collection.....	40
5.3.2. Scoring functions .....	40
<b>6. Summary .....</b>	<b>42</b>
<b>7. References .....</b>	<b>43</b>



### Abbreviations

ALTO	Application Level Traffic Optimizer
API	Application Programming Interface
COI	COAST Content Identifier
DNS	Domain Name Service
DPI	Deep Packet Inspection
GeoIP	Service mapping IP addresses to geographical information
IP	Internet Protocol
ISP	Internet Service Provider
MPD	Multimedia Presentation Description
REST	Representational State Transfer
RTT	Round-Trip Time
UGC	User Generated Content
URL	Unified Resource Locator
WSDL	Web Services Description Language



## Executive Summary

The COAST project aims at deploying a content-centric network on top of the existing network infrastructure. This enables a better use of network resources, by accessing Internet content without referring to a particular copy of the data. This new architecture also enables new features for search, which we explore in the project.

The COAST search engine follows a distributed approach: several search sites are deployed in various geographical locations and pair wise communicate to collaboratively provide a search service. This paradigm differs from current search engine architectures, in which each site is independent and provides search engine functionalities by itself. In COAST, each search site crawls and indexes content in its region. Consequently, content relevant only to users in a region is known to the search site of that region, while the search site in a different region does not contain it. Such a choice of documents for each site enables search sites to be more efficient when computing results originating from their respective regions, and provide a higher quality of service to the end users. Yet, if a user in a given region queries for a document of a different region, the former search site forwards the query to the latter site to obtain a relevant set of search results.

In this document, we present the design of each component of the COAST search engine, namely the crawler, the indexer, and the query processor. We first present the interactions between the distributed search sites. In particular, we show how the COAST search engine benefits from information from the COAST overlay to better identify the interests and needs of end users. We next present the internal architecture of a given search site, providing detail on the specific components. Finally, we propose metrics associated with experiments that will enable us to evaluate the COAST search engine architecture and assess its efficiency. The results of these experiments will appear in subsequent deliverables.



## 1. Introduction

Search engines are key services of the Web: they enable users to find relevant information despite the large amount of content available. Content is made available on the Web through Web pages, each Web page being identified by a URL. As the number of documents available over time has increased substantially in the past decade, it is difficult for users to recall previously accessed URLs or even find new URLs by a simple sequential inspection of documents found arbitrarily. Consequently, search engines become an indispensable service to navigate on the Web in an effective manner.

To use search engines, users submit queries as sequences of terms that describe the content they are interested in, and in return, a search engine generates a page of results containing a list of potentially relevant Web content, including hypertext links to access them. Search engines mitigate the problem of finding content among billions of pages, and currently are able to generate a set of results under a second. As the Web keeps increasing, the question of search engine scalability and efficiency becomes more critical, as they will condition the ability of users to access Web content efficiently in the near future. Scaling search engine infrastructure along with the Web becomes consequently necessary for future generations of search engines. Search engines to date have relied on techniques suitable to a single large data center, which crawls and indexes the Web. This data center is sometimes replicated in different locations, for performance reasons and disaster tolerance, but these copies remain independent and can answer any query without communicating with each other. Although such an approach does not require different locations to communicate, it is inefficient with respect to bandwidth utilization for crawling, storage utilization for indexing, and CPU cycles used for query processing.

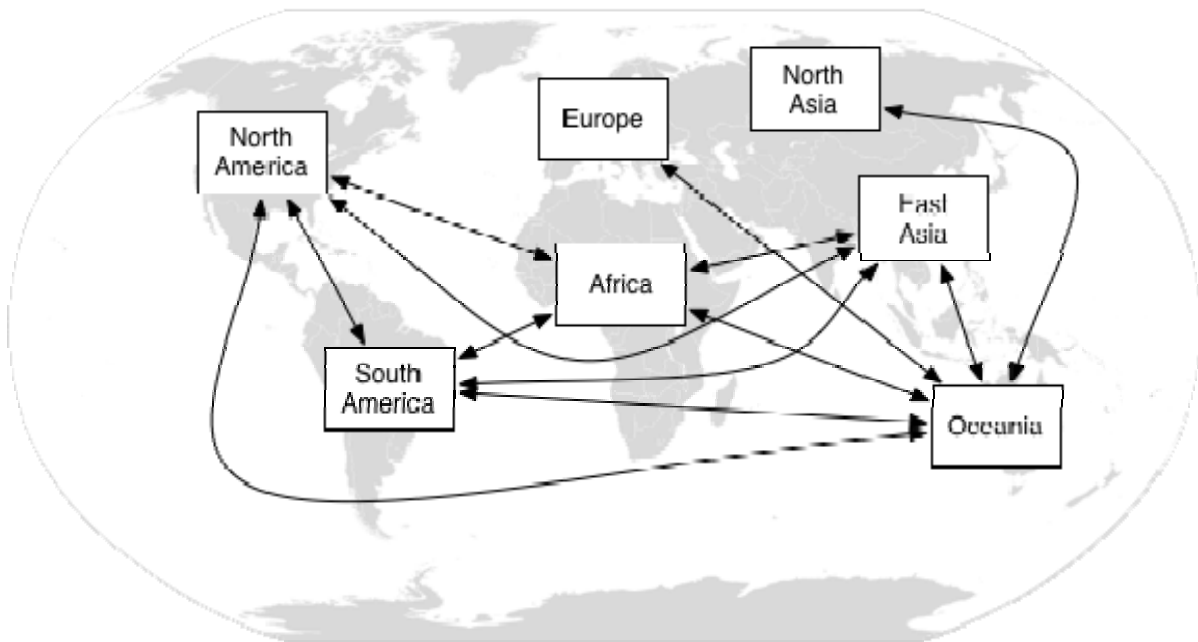
To aggravate the problem of the amount of content available even further, there is with COAST the potential of a significant increase on the amount of content available to users, since users are now able to directly publish content. Consequently, a search engine for COAST has to be able to serve search results for content both from the Web and from COAST. Such an increase on the amount of content has on the one hand the potential to improve the overall user experience while online, but aggravates the problem of search engine scalability and efficiency, making more imminent the necessity of solving the scalability issue.

In this report, we discuss the overall architecture of the COAST search engine, which is a core component of COAST [2]. At a high level, we spread the search engine functionality across multiple data centers, with the two main goals of meeting the performance expectations of users, while not requiring a commercial search engine to build larger data centers as the amount of content available increases. To add more capacity, we simply add more data centers, but not necessarily larger ones. We leverage functionality existing in COAST to improve the user experience, such as learning new content and user preferences through deep-packet inspection. Finally, we incorporate results from users in near real time, thus making user content published through COAST readily available for search purposes.



## 2. Overview

In COAST, we propose a distributed search engine architecture. Several search sites (small or large data centers) are deployed in different regions across the world, and communicate to collaboratively answer queries (Figure 1). Each data center serves a fraction of the Web that is most relevant to the users in its region. When a user accesses the search engine, the DNS service automatically connects the user to the search site in its region. Hence, the users benefits from a fast response time, as the search site only searches through documents of interest to the user. If the user, however, performs an unusual query for a given region, the search site forwards it to others to provide a higher-quality set of results.



**Figure 1: Deployment of the COAST search engine**

In general, a search engine consists of three main components: a crawler, an indexer, and a query processor. The crawler discovers the content of the Web to add it to the search engine's content collection. Typically, this content consists of text Web pages, but also images, videos, as well as Web services. The COAST search engine differentiates itself from traditional search engines by leveraging passive crawling techniques. This enables the crawler to discover content not accessible to traditional crawlers and that users are interested in. Furthermore, the crawler also benefits from the geographic distribution by optimizing the retrieval of content to better use network resources.

The COAST search engine comprises several search sites, each of them maintaining a different content collection. Each of these search sites is specialized for its location: it references content required by the users in its region. Hence, the crawler selects, for each new document discovered, the most relevant search site to send it to. This content goes through an indexer that generates an inverted index: a map, for each term, to Web content related to it. As a lot of content is popular in several regions, these indexes also replicate popular content originally indexed at other locations. Finally, the query processor uses the index to answer queries. In COAST, the search engine resolves these queries collaboratively, by contacting other search sites when this is necessary, but reduces the amount of forwarding by selectively indexing content in sites. Since indexes are specialized to match the interests of the users of a given search sites, most results can be computed locally. However, if the results are not satisfactory and another location could provide a better answer, then the query is automatically forwarded to generate a complete answer to the query.



We present the distributed architecture of the search engine in Section 3, and explain the details of each search site in Section 4. Finally, in Section 5, we present the metrics we will use to evaluate the performance of the COAST search engine.

We now introduce the terminology we will use in this document:

- The **COAST search engine** refers to the overall architecture we describe in this document. It provides search functionalities, from discovering new documents to answering user queries. Consequently, it encompasses crawling, indexing and query processing.
- A **search site** refers to a data center that hosts search services for the COAST search engine. In our case, the search engine is distributed among different search sites, which are geographically distributed and communicate to provide search services.
- The Web contains many different types of objects, such as text documents, images, videos, and even Web services. In this document, for the sake of generality, we refer to all those categories as **content**.
- **Crawling** consists in discovering and fetching content. Most of the existing crawlers are active, as they pull content from the Web and analyze it to discover new content. In this document, we introduce the notion of passive crawling. It is fundamentally different as the crawler receives input from the user; this is a push process.
- **Indexing** is the action of taking the collection of content obtained through crawling and building a data structure used for search purposes. At this step, the indexer extract features from the content in order to match it with keywords that the users will use in their search queries.
- When a user issues a query, the **query processor** leverages the information in the index to find relevant documents and rank them. The query processor generates a results page, which is then served to the user.



### 3. Multi-site search engine

The functionalities of the COAST search engine are distributed over several, geographically distant search sites. They involve acquisition of new content, construction of an index over the acquired content, and evaluation of issued user queries over the constructed index. In what follows, we separately detail these functionalities.

#### 3.1. Content acquisition

In the COAST search engine framework, the content is acquired in three different ways, through active crawling, passive crawling, and publishing. Active crawling [30] is the standard technique used in commercial search engines to discover and acquire new content. This approach has certain limitations, especially when dealing with the content in the hidden Web [31]. The COAST search engine framework innovates over active crawling by introducing the concept of passive crawling. In passive crawling, the content is passively discovered by the network itself and pushed to the search engine. This leads to resource savings on the search engine side as well as improvements in coverage and freshness of the content possessed by the search engine. The COAST search engine framework also supports publishing of content by external means, i.e., content producers can push the content to the search engine. In Section 3.1.1, we present these three content acquisition approaches, emphasizing their design in the COAST framework. In Section 3.1.2, we go into detail in our distributed (active) crawling architecture.

##### 3.1.1. Approaches

The COAST search engine relies on three different approaches to discover Web content. The first one, active crawling (Section 3.1.1.1), is the traditional process used by search engine, builds a graph of Web content to discover documents linked to each other. However, some contents are not linked to the rests of the Web, as they are isolated, or generated on the fly. Therefore, COAST uses passive crawling (Section 3.1.1.2), to learn about documents as users access to them. Finally, users can publish documents (Section 3.1.1.3) to explicitly ask the search engine to take them into account and provide them as search results.

###### 3.1.1.1 Active crawling

A typical active crawler [3][16][17][32][33] works as follows. The crawler is initially given a set of seed pages, which potentially contain many links to high-quality content. It then fetches these URLs from their Web servers through the HTTP protocol. The fetched content is stored in a content repository. The content is parsed and new URLs are extracted. These URLs are added into a queue of URLs to be fetched, referred to as the frontier of the crawler. The crawler continuously expands its frontier by discovering new URLs. In the mean time, the previously fetched content is refetched to guarantee its freshness.

Although implementing a basic crawler is a relatively simple task, implementing a robust and efficient crawler is quite challenging. A successful design should take into account many issues, such as DNS caching, multi-threading, complexity of data structures, hazards such as Web traps and link farms, duplicate content and mirror site detection, prioritization of URLs, and politeness to Web servers. We briefly discuss these issues in Section 4.1. We note that the COAST search engine does not innovate on sequential crawling. However, it presents some innovations to distributed crawling, which will be discussed in Section 3.1.2.



### 3.1.1.2 *Passive crawling*

An important drawback of active crawling is the discovery of the hidden Web content [31], i.e., a part of the Web that is not connected through hyperlinks and thus cannot be discovered by the traditional crawling approach. Such content is often dynamically generated and is not reachable to an active crawler that solely relies on link following. Fetching hidden Web content requires different techniques, which are typically not implemented within an active crawler. In addition to the hidden Web content, the Web content that is disconnected or poorly linked is unlikely to be discovered by an active crawler.

In practice, the crawlers simply do not have the resources to obtain all the content on the Web and keep it up to date. Hence, crawlers focus on the most important part of the Web, which is likely to answer most of the users' queries. However, there are some important pages that are not indexed because the crawlers never discover them [40]. Consequently, an active crawler misses the opportunity to incorporate such content into its repository, and the search engine cannot serve them.

The COAST search engine relies on a novel mechanism to discover content users are interested in. We refer to this mechanism as passive crawling, where the content is discovered by means external to the search engine. In practice, passive crawling can be implemented in different ways. For example, links to the content can be extracted from emails, instant messages, and browser toolbars. In an extreme scenario, URLs can be discovered and pushed to the crawler by ISPs.

Passive crawling offers significant advantages over active crawling. First, passive crawling has the potential to improve the coverage of a crawler, as certain URLs that are not accessible to the crawler are discoverable through such techniques. Second, passive crawling improves the accuracy of the importance estimations for content by providing some form of popularity information. Third, in a hybrid scenario where active and passive crawling are both employed, the active crawler's workload may be reduced, as content discovery is performed by agents that are external to the search engine.

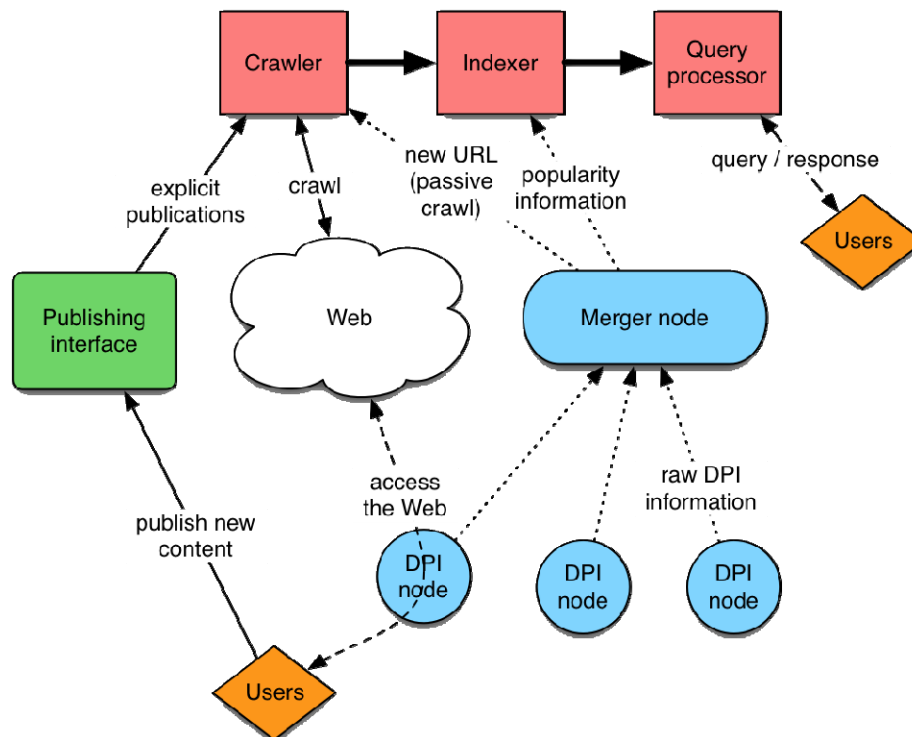
In the COAST framework, passive crawling is implemented through Deep Packet Inspection techniques that identify user traffic. Note that the development of the COAST passive crawling solution is part of work package 4. This document only presents an overview of its architecture and focuses on its interactions with the search engine. The reader may refer to [1] for more information about the DPI techniques used in COAST.

DPI nodes are disseminated on routers in the network and observe the user traffic (illustrated in Figure 2). They identify the HTTP requests that indicate a user is accessing a Web page and store the corresponding URL. The DPI nodes perform very simple operations, as the processing should not impact the traffic on the router. Hence, the operations are lightweight and fast. DPI nodes record data over short time periods, and at the end of each period they compress the data structures storing the lists of URLs. This information is then transmitted to a merger node that aggregates the data coming from several DPI nodes. Since merger nodes are not operating on routers, they are less limited in the operations they can perform. They compute statistics on the browsing patterns of the users and identify trends in user traffic.

Passive crawling observes the activity of the users in real-time and, as a consequence, identifies which content they are interested in. Identifying user interest is important for search engines. Commercial search engines log the clicks of the users on the results pages to refine the scoring of the Web pages. Similarly, the data obtained through passive crawling is used by the search engine to improve the quality of the results. First, the URLs discovered through passive crawling are transmitted to the active crawler. This approach enables the active crawler to fetch content that it had initially considered not important, or discovering parts of the deep Web. Fetching such content increases the coverage of the search engine. Second, statistics about the observation of URLs are



transmitted to the indexer. These observations are good indicators of popularity and interest and can be used as part of the scoring function, in addition to the click logs.



**Figure 2: Interactions between the elements of the COAST distributed search engine**

Finally, we will also explore the use of passive crawling to improve the freshness of the COAST search engine. In addition to recording the URLs accessed by the users, DPI nodes could also compute digests of the HTML content of the Web pages. This enables the passive crawler to determine whether the content of a Web page has been modified without any network cost: the user is accessing the Web page and the node simply reads the data as it goes through the router. By transmitting the digests of URLs along with the list of accessed URLs, the passive crawler could inform the active crawler about outdated pages. This would clearly improve the freshness of the documents collection of the crawler, as well as the efficiency of the bandwidth usage. The crawler would only retrieve a Web page once it has actually been modified, and not simply to check if it has been modified. While this functionality is promising, it is not yet clear if current DPI techniques enable the computation of pages digests.

### 3.1.1.3 User publishing

The popularization of inexpensive digital photo and video cameras has transformed regular users from mere content consumers also into producers of their own content or *prosumers*. More and more, users share their content on the Internet through specialized Web services and social networks. However, search engines cannot discover user generated content (UGC) unless it is explicitly linked through posting on public Web sites.

To make UGC more visible and accessible, COAST provides an explicit publishing mechanism that let users notify the COAST search engine about the availability of their content, thus taking advantage of the searching and caching features of COAST.

The COAST user publishing mechanism, which is also addressed in section 4.2.1 of Deliverable D2.2 [73], consists of a publishing front-end (see Figure 2) which receives information from the users about the published content. This element performs the following actions:



- Let the user enter metadata information about the content to help the search engine with the indexing process. This metadata may be stored in an associated file as described in [73].
- Generate the COAST Object Identifier (COI) by calculating the SHA1 or MD5 hash of the content.
- Interact with the search engine to include the content's COI into the crawler queue and provide the entered metadata.
- Inject the content into the COAST caches.

Once the publishing process completes, the search engine eventually indexes the content, which start appearing as a search result when relevant. Due to the COI attached to the content, it will be possible in COAST to locate its copies in the caches and select the better one for delivery when the content is requested.

The metadata a user provides may include the following items:

- Predefined fields such as title, description, author, and date;
- Tags to be freely defined by the user as a way to attach relevant keywords to the content.

This metadata information is provided to the search engine through the publishing interface but may also be saved into a file. This metadata file may be published along with the content to provide a standard mechanism to obtain content's metadata. While we may consider the metadata file as optional, some types of content will always have an associated metadata file. This is the case of MVC and SVC encoded video where the content is divided among several files. The metadata file, named MPD (Multimedia Presentation Description), contains references to all related files. In fact, in this case the COI associated to the content identifies the metadata file as it is always needed to access all the video files that integrate the content.

The publishing front-end may be conceived in several different ways:

- As part of the COAST platform, providing a publishing service to the users. This element would provide a Web interface allowing the users to upload the content and enter the metadata. The content would be stored in the COAST caches.
- As a software add-on to be installed on regular Web servers to enable COAST publishing features. It would provide all the COAST specific functionality: metadata collection, COI generation and interaction with the search engine. The content would be stored in the Web server and only cached in COAST when needed.

These implementations of the publishing functionality are not mutually exclusive but entail different business models regarding UGC publishing.

Another element that may play an important role in the publishing mechanism is the User Agent, the piece of software running on the user terminal. While a specific software application will not be required for publishing content into COAST, the User Agent may help the user, especially on those terminals with limited input capabilities, such as mobile devices or set-top boxes. Some of the functionalities that the User Agent may provide are:

- Generation of metadata: fields like author, date or location may be automatically inserted.
- Simplification of the interaction with the publishing front-end: the User Agent may provide a user interface integrated into the content generation functionality of the terminal for improving the user experience.
- Additional functionalities: such as management of published content or creation of albums or sets of related content.

In summary, the user publishing mechanism provided by COAST will enable users to share their own content easily and find other users' content without resorting to specialized services. UGC will



appear in search results and access to it will be done efficiently due to the COAST cache infrastructure.

### 3.1.2. Distributed crawling

In [3], the authors review many of the challenges met by Web crawlers and propose solutions to avoid bottlenecks and achieve efficient crawling. Nevertheless, there still are fundamental performance issues that centralized crawlers cannot overcome [4][5]. Crawling pages hosted on a distant server is intrinsically long, as the delay to obtain the document is at least equal to the round-trip-time (RTT) to the server and depends upon flow and congestion controls of TCP, which are sensitive to network distance. The COAST active crawler architecture is specially designed to overcome this limitation: several crawlers are distributed in the network and each crawler only retrieves documents at a short network distance (Figure 3).



Figure 3: Crawlers exchanging links to download close content

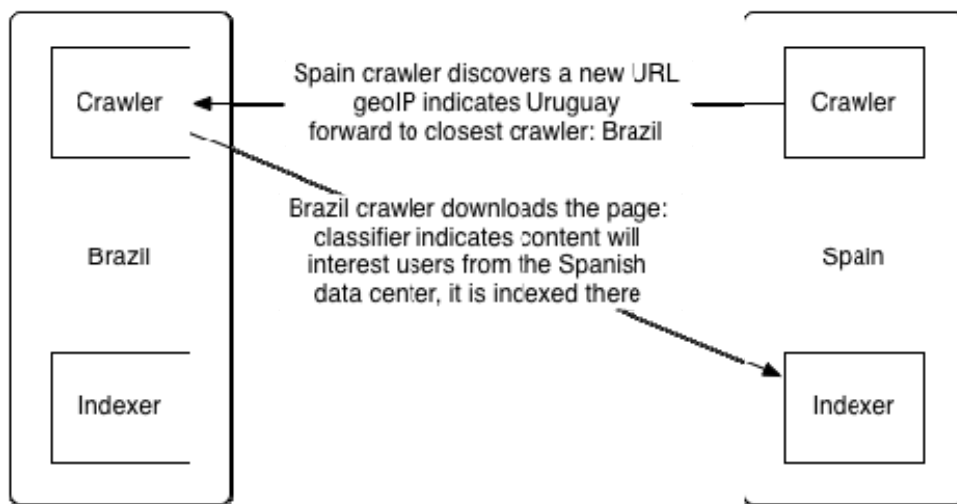
The COAST active crawler is composed of several crawlers that are placed at different geographical locations to evenly cover the entire Internet. Each of these crawlers is based on a state-of-the-art centralized crawler, such as the one described in [3], with the addition of interfaces that allow it to communicate with the other crawlers. Using multiple crawlers in parallel [4] is a usual way to saturate the available network bandwidth. However, distributing these crawlers in different locations has only recently been considered as a viable solution to increase the crawler efficiency [5]. The preliminary results presented in [5] suggest that specializing each crawling location to only retrieve local content could significantly improve the crawling capability of a search engine. Nevertheless, the authors also note that many implementation issues need to be solved to deploy such a solution. This is the challenge addressed by the COAST search engine. Theoretical models on Web partitioning for the sake of efficient distributed Web crawling are discussed in [43].

As depicted in Figure 1, the COAST active crawler is deployed in many different locations, all around the world, and each of these crawler sites can communicate with the others. Following the usual crawling paradigm, each crawler site retrieves Web content and discovers new URLs through hyperlinks. However, instead of directly adding these new URLs to its pool of pages to crawl, it first determines the location this content should be fetched from. The goal is to determine which crawler instance is the closest, in terms of IP distance, to the server hosting the Web page. RTT has been shown to be proportional to geographical distance [6]. Given that each crawler instance is aware of the architecture deployed and knows the locations of the other distributed crawlers, the only thing needed to determine which crawler is the closest is the geographical location of the server hosting the Web page. This information can be obtained quite accurately through GeoIP services.

The crawler first resolves the URL through the DNS service, which returns the IP address of the server hosting the Web site. Then, the GeoIP services take an IP address as an input and return a location (latitude and longitude) as an answer. Using the Haversine formula [22], one can then compute the geographical distance between any crawler and the server hosting a Web site.



Hence, the COAST active crawler will rely on a GeoIP database to obtain the latitude and the longitude of the server hosting the Web page and transfer the page to the distributed crawler closest to this location. This is illustrated by the first step of Figure 4: a crawler located in Spain discovers a new URL. It obtains its IP address thanks to the DNS service and then uses the GeoIP database to assign it to a location. Given that the server is located in South America, the Spain crawler decides to forward the link to the crawler in Brazil, as it is much closer.



**Figure 4: Interactions between crawlers in Brazil and Spain**

An alternative to using a simple GeoIP database is relying on an ALTO service. The ALTO service benefits from information provided by ISPs about the architecture of the network and the quality of connections. Therefore, the ALTO server is able to predict which crawler would be most likely to retrieve a Web page efficiently. Still, to be accurate, the ALTO server requires a lot of information that ISPs might be reluctant to provide. As a consequence, we will only consider the case of a GeoIP database in this document.

Figure 5 depicts the different components the distributed crawler uses to collaborate with other crawlers and exchange links. Each time a crawler obtains a Web page, it extracts the hypertext links present and, using the DNS service as well as the GeoIP data, it assigns each link to the closest crawler. Some of the links should be locally crawled, and are therefore added to the crawler's own queue. However, some links point to distant servers, and should be crawled by the other crawlers of the system, and are therefore added to the list of links to forward them. Similarly, the crawler receives links from these crawlers and adds them to its queue.

Downloading content from close locations improves the speed of retrieving content [5][38]. While this is the main focus of our design, we still foresee other notable improvements. Distributing the crawlers at different locations improves disaster tolerance (a whole data center becoming unavailable). Indeed, it is not likely that two very distant crawlers suffer from correlated failures. Furthermore, downloading content from a close location reduces the probability of network link failure or congestion between the crawler and the server. The network path is shorter and contains fewer routers, which overall increases the probability to download a Web page without errors and improves crawling performance. Finally, crawling locally reduces the overall network load. Crawlers download a huge amount of pages and cause an important load on the network infrastructure. As each crawler fetches local content, fewer routers are involved and the overall network load is smaller.

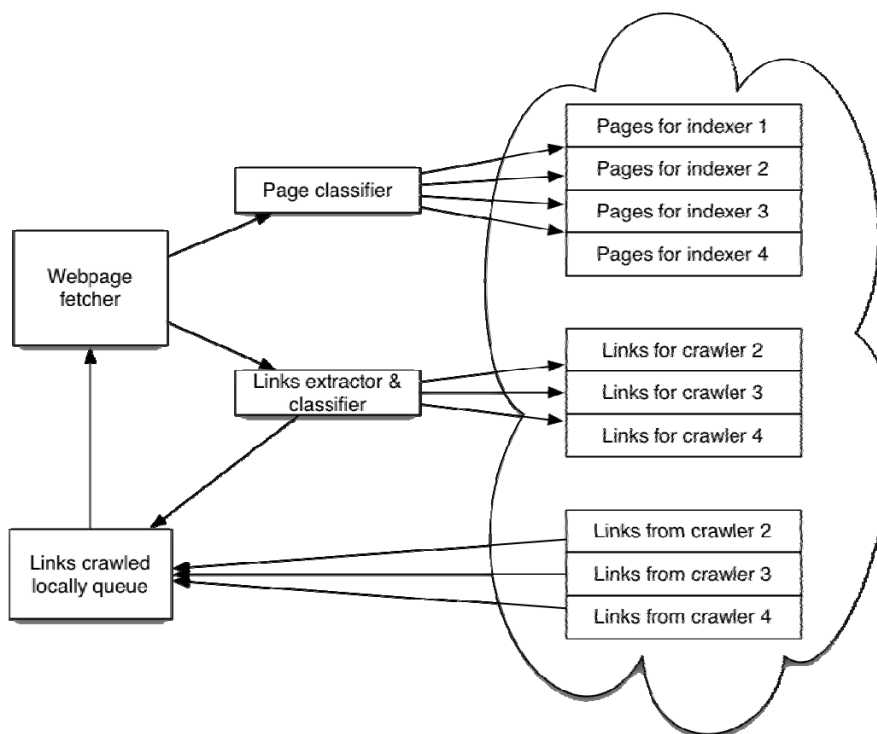


Figure 5: Links exchange and pages classification at crawler 1

### 3.2. Distributed indexing

The COAST crawlers discover documents by following hypertext links, monitoring the interests of the users, and receiving publications from the users. Once these documents are retrieved, they have to be processed by an indexer to generate the index that the query processor relies on to answer the users' queries. The index also contains a full version of the documents, as they are used to generate snippets, which are presented to the users on the results page.

As explained previously, the COAST search engine is a distributed system that relies on data centers placed at different locations around the world (Figure 1). In traditional search engines, each search site contains either identical copies of an index or approximate copies, which contains all the content retrieved by crawlers. This solution poses major scalability problems [38], which is why COAST relies on specialized search sites. The index generated at each location is specially targeted for the users in this particular region.

Hence, once a crawler has obtained a given content, it analyzes it to determine which search site is most suitable to index it, i.e. which users are most likely to be interested in it. Since many documents are very popular, they should be present in several indexes. We first describe how we address the problem of selecting one location to index a document. We call this location the master of the document. Then, we explain how other sites can replicate this content in their own index to be able to answer more queries.

#### 3.2.1. Selecting a master indexer

In COAST, each crawler can communicate with all the indexers to transfer content. To be efficient, each crawler batches the transfers of documents to compress them and reduce the communications cost. The main challenge addressed in COAST for distributed indexing, is to accurately select which indexer a given document should be sent to. While the choice of a crawler



to retrieve a document (Section 3.2.1) was solely based on the IP address of the server hosting the Web site, this decision process can rely on the content of the document to compute features.

To select an indexing location, we rely on similarity functions to determine which indexer is the best match for a content. We place documents at the location where their corresponding users are likely to request it through queries. To enable such a decision, each search site computes statistics on the query patterns of the users, and transmits these statistics to the crawlers. The crawlers can then use these statistics to compute, for each document, a score for each indexer. The document is sent to the indexer that obtains the highest score.

There are some important, practical constraints on the statistics transferred by the indexers. First, they must be updated regularly to follow the dynamics of the users' interests, and updates must be concise, as they are transmitted to all the crawlers. Second, the similarity function must be implementable in an efficient way to keep up with the throughput of the crawlers. An important observation, however, is that the assignment of the documents to indexers is not on the critical path of the crawler, and can be computed once the document has been fetched. Hence, it is possible to parallelize this task on several machines to increase overall throughput.

Figure 4 illustrates this process. The crawler in Brazil fetches some content originally discovered by the crawler in Spain, due to network proximity. The server hosting the Web page is in South America, and its content is in Spanish. In our example, the indexer in Brazil is mostly responsible for Portuguese content, while the indexer in Spain handles most of the Spanish content. Therefore, after computing similarities, the crawler assigns it to the indexer in Spain.

Figure 5 depicts the architecture of a given crawler. As we can see, the classification module is located just after fetching the page and is independent of the link classification problem. The document is assigned to an index and is transmitted.

The problem of placing documents in distributed indexes has been considered in [7]. The authors use machine-learning techniques to build a classifier. They show that the language and the region of a page have a high influence on where it is requested. However, the problems considered in [7] is slightly different from the selection of a master index: the classifier can decide to place a document at several locations, while we only want to find one good location and rely on a replication mechanism to copy the document in the other indexes.

### **3.2.2. Replicating content**

Each COAST search site is specialized to answer queries originating from users in its region. This specialization is crucial to ensure the scalability of the search engine: the index remains small, as it does not contain useless documents. This makes the search process faster and reduces the latency perceived by the users. In Section 3.2.1 we explained how each document is assigned to a master index. Yet, there are still many cases in which we would like a document to be present in several indexes. In fact, important pages are likely to draw the attention of users in different regions, and ideally each index is able to answer locally most of the queries.

A few distributed index strategies have already been explored. In most cases, such as the one presented in [8], the full document collection is split in two parts: the important documents and the local documents. The important content is present in all the indexes, while the local documents are only present in the index of one indexer, the master indexer in our case. This strategy is quite efficient. The importance of Web content follows a power law, which means that even a small set of important documents is sufficient to answer most of the users' queries. Still, the solution presented in [8] is based on a static analysis of the documents collection, and does not capture the dynamics of the popularity of Web pages. In COAST, we will implement a more fine-grained replication scheme. Different from previous work, each indexer will be responsible for selecting the documents it replicates, and these will be different at every location. For example, an indexer in Spain may



replicate many documents from an indexer in South America, as both will contain documents in Spanish and Portuguese, but the indexer in Japan will not replicate these documents if they are not requested in Asia. Basically, the replication strategy adopted in COAST is reactive. A search site computes statistics on the queries that are not answered locally, and once they reach a given threshold, the documents corresponding to the queries are replicated to the local index. Similarly, if some replicated documents are not popular anymore, they are evicted from the index. Note that this cannot happen for the documents the indexer is the master of; they always remain in the index.

As a consequence, each indexer will use two different inputs to build the index. The first one is the documents for which the search site is the master. These have been assigned by the crawlers and must be in the index, to guarantee an extensive coverage of the search engine. The second input consists of documents that have been frequently requested at a given search site, even though they were assigned to another location. These documents are optional and are added to the index to answer more queries locally, without consulting another index. In fact, there is an important trade-off between the amount of replicated content and the fraction of queries answered locally. If all content is replicated, then all queries can be answered locally, but such a design choice leads to poor scalability, as the index is very large. On the other hand, a small degree of replication induces a higher communication cost at query time since other indexes are consulted to compute an accurate answer. Note that the architecture we propose is modular and we will experiment with different replications strategies and analyze their efficiency.

### ***3.3. Distributed query processing***

The COAST search engine relies on a distributed architecture to enhance the user experience by reducing the response time, and achieve higher scalability by splitting the responsibility of documents between several indexes instead of replicating a single content collection. As explained in Section 3.2, each index references content selected through two different processes. Each indexer is the master of some content, which is assigned by the crawlers. This selection relies on similarity metrics to place content at a location where it is most likely to be requested by users. This step ensures that all the content is present in at least one index. The second step that determines the content of an index is replication. For each search site, documents local users request, but do not have this search site as their master, are replicated in the index of the site. This ensures that most queries are answered locally, while keeping the size of the index small. Yet, the COAST search engine is designed for high precision, and this is the main reason for the processing of queries to rely on a distributed algorithm. The search service consults multiple indexes across sites if it estimates that they are likely to contain relevant documents that are not in a given local index. This search process is detailed in Section 3.3.1. The COAST search engine, through passive crawling and content publishing, is particularly good at determining trends in the interests of the users. Section 3.3.2, we describe how live results can be integrated to the processing of queries to propose more results to the users.

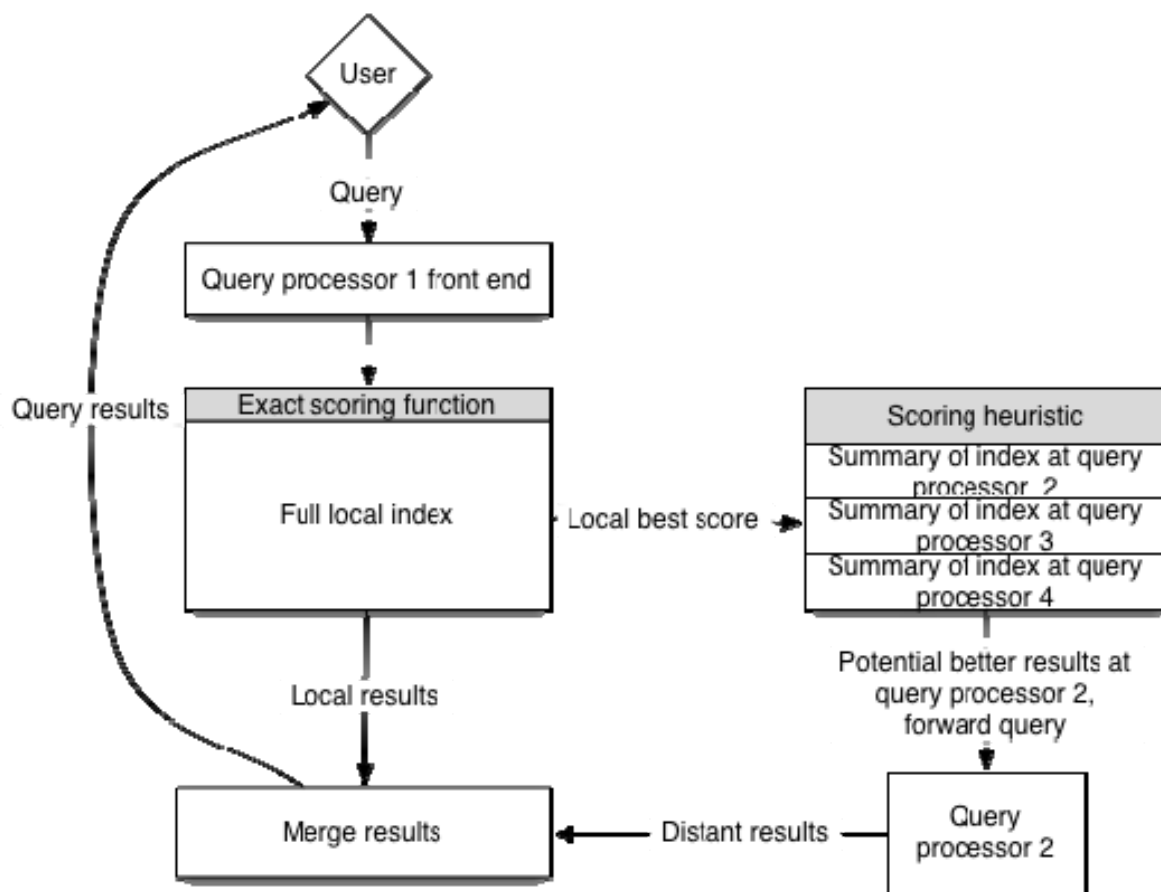
#### ***3.3.1. Query forwarding***

When a user sends a query to a given search site, the query processor uses the local index to compute search results. In the common case, all the necessary content is present in the index, so this step is sufficient to ensure that the service presents the best possible set of results to the users. The COAST search engine is distributed; hence each index only references a subset of the documents. While the crawlers and indexers are optimized to generate an index that is able to answer most of the queries locally, some queries have to be processed in a distributed way.

In COAST, each query processor has access to its own local index and communicates with the other query processors, to collaboratively answer queries that they cannot resolve on their own. The goal



here is to perform as few communication steps as possible, as each communication with a distant search site adds delay to the query processing and increases the latency perceived by the user. Therefore, each site has to be able to locally estimate whether a query should be forwarded, and to which search sites it should be sent to. The COAST search engine relies on the threshold algorithm, already explored in [8] and presented in Figure 6. This algorithm relies on the properties of the scoring function of the search engine. The authors show that if each search site communicates the upper bounds on scores it can assign for each term in its index, then each search site can use heuristics to estimate whether another search site is likely to contribute interesting results to the query. This approach is very conservative and ensures that the search engine always returns the best possible results. Yet, it can also result in a large number of communications, as, for long queries, summing upper bounds largely overestimates the actual score documents obtain. To make it less conservative, the authors propose some variations of the threshold algorithm. They introduce a slackness parameter, to trade a small drop in search results quality for a shorter response time and fewer communication steps. In [75], the authors improve the threshold algorithm by computing thresholds on queries instead of simple terms. For a given query, the algorithm computes a subset of queries with known bounds that cover the new query. In practice, the thresholds computed for queries are tighter; hence fewer queries are unnecessarily forwarded.



**Figure 6: Distributed query processing using a threshold algorithm**

The efficiency of the trade-off algorithm lies in the fact that popular documents, which achieve high scores, are replicated in every index and therefore are not taken into account in the computation of thresholds. In Section 3.2.2, we argue that a dynamic selection of documents to be replicated could be beneficial to the search engine. As a consequence, the threshold algorithm needs to be adapted to take this drawback into account: there is no longer a set of documents that is known to be present in every index. The details of the algorithm used in COAST are part of a future deliverable.



We believe that personalized thresholds, based on which content is replicated, could be used to improve the performance of the threshold algorithms, as the upper-bound values propagated are tighter. We are yet to evaluate the scalability of this solution.

### ***3.3.2. Integrating popular tail content***

Passive crawling informs the COAST search engine about interest trends observed on the network. In addition to the discovery of new documents, DPI enables the search engine to have more information about the documents the users of a given region access. Search engines monitor the behavior of users on the result pages, and record the links they click. Still, click logs are very sparse, and for many unpopular documents, called tail content, the search engine doesn't have any information. The DPI mechanism is able to follow the activities of users outside the search portal, and can therefore track more activity. Such tail content can be explored to provide alternative results to users. With the addition of videos and images, search engines have introduced the principle of faceted search: the search page includes different kinds of media to offer more choice to the user.

It is difficult to compare such tail content with content obtained directly through active crawling in a sensible way. The different set of features available for documents in each set is very different: content discovered through active crawling is part of a graph of Web documents, which can be leveraged to compute a score, while content found through DPI are isolated. Consequently, we plan to present such tail content as a different set of results. Contrary to normal Web pages results, which have been studied and improved throughout the years, the tail content will feature pages for which the passive crawling mechanism has observed a spike of popularity recently, and in the area of the user.



## 4. Inside a search site

In this section, we present the software components of the search engine. We provide more detail compared to Section 3 by focusing on the architecture of a given search site, instead of considering the distributed system as a whole.

### 4.1. Crawling

In Section 4.1.1, we discuss the techniques that are employed by a Web crawler to increase the quality of its content repository. Several performance-related issues in Web crawling are given in Section 4.1.2. Finally, we discuss some external constraints imposed on crawlers, in Section 4.1.3. Figure 7 summarizes the different components of a Web crawler.

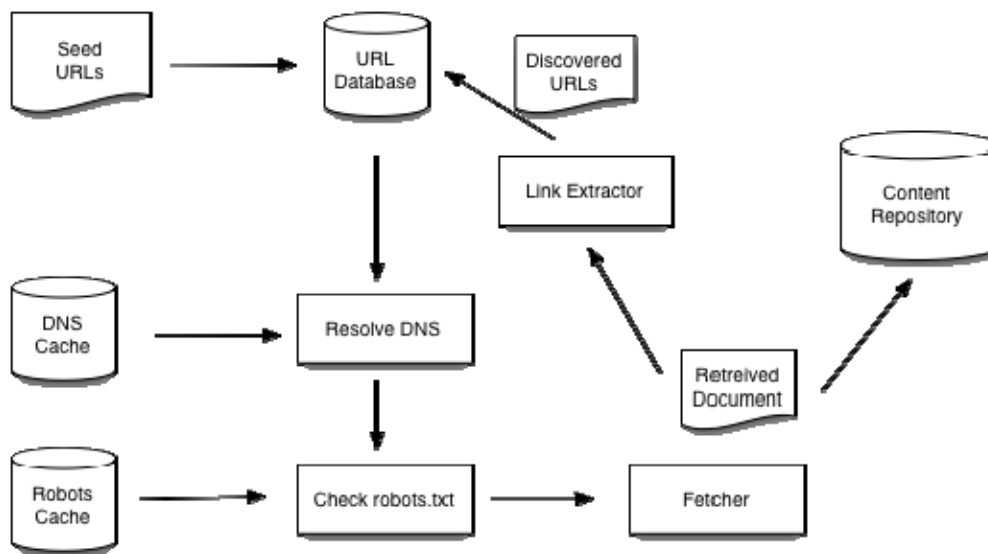


Figure 7: Architecture of a Web crawler

#### 4.1.1. Content quality considerations

##### 4.1.1.1 URL prioritization

The Internet contains many billions of pages and is constantly growing [68][74]. It would be impossible for a Web crawler to download the whole Internet. A Web crawler needs selection strategies to ensure that it downloads the most important content that will be best able to service the end user queries [38][42]. An important way of increasing the content quality is to prioritize the URLs so that more important or high-quality content is crawled earlier than the rest [12].

Previous techniques prioritize the URLs in the frontier according to a quality metric (e.g., the linkage between pages) and recrawl the stored content based on the likelihood of modification. More recent approaches prioritize URLs and refresh the content to directly increase the benefit on search result quality. In this impact-based crawling approach, pages that are expected to frequently appear in search results are crawled earlier or more often than the others. Similarly, Web sites can be prioritized for crawling so that sites that contribute more pages to search results are crawled deeper or more often.

##### 4.1.1.2 Web spam and mirror sites

Web spam [36] comprises techniques to boost artificially the importance of certain, potentially useless Web content so that they are displayed at higher ranks in search results. One form of such



spam is link farms [35], which are massive Web sites containing content linking to each other, with the goal of increasing their content popularity and importance values computed by the search engine. It is important for Web crawlers to early detect such intent so that both spam content is filtered and the crawler does not allocate a significant portion of its resources to download these spam pages. Mirror sites form a similar threat even though they are not maliciously created. Detecting mirror sites [37] is important for good utilization of the network and computational resources of the crawler.

#### **4.1.1.3 Recrawling**

Maintaining the freshness of a content collection is another challenge of crawling. Ideally, a crawler tracks every change to a given content. Since there is no mechanism to notify a crawler about content modifications, every page needs to be recrawled [39]. Unnecessary fetches constitute a waste of bandwidth and other crawling resources, as the content of the re-crawled pages may not have significantly changed. In some cases, when a document changes, the change may only consist of different ads or dynamic elements included from other sites, such as RSS feeds. These changes are uninteresting to a search engine, so re-crawling them is pointless. There has been work on techniques for discovering important changes of a document. In [14], the authors propose to rely on past behavior to estimate the optimal recrawl time for a given Web page.

Statistically learning the frequency at which a document changes might lead to better re-crawling. However gathering this information potentially leads to poor resource utilization, since it requires that a document be re-crawled multiple times initially. Cho outlines an algorithm that effectively estimates the rate of change of a page without prior information [15].

#### **4.1.1.4 Hidden Web**

While traditional crawlers are quite good at crawling the important content on the regular Web, it is impossible for them to access some parts of the Web that are generated dynamically. An example of this could be a library in which the user has to enter the ISBN of a book to return information on that book. It is difficult for a crawler to access this content as it is usually hidden behind HTML forms. Current crawlers gain access to these pages by dynamically generating inputs to these forms [29]. COAST augments this technique using information from DPI to guide the generation of these inputs.

#### **4.1.1.5 Web service discovery**

The Web is currently changing from a Web of pages to a Web of services. One way for discovering a specific Web service is to use standard search engine to perform keyword search, which in turn requires efficiently crawling and effectively indexing the Web services.

Most Web services are published either using WSDL (Web Service Description Language) documents or following a RESTful (Representational State Transfer) approach, e.g., Web APIs. WSDL is an XML-based language that provides a model for describing non-semantic Web services. It is also used to describe the operations that can be performed by a certain Web service as well as its location. Web APIs are HTML documents, same as other Web pages, which expose a functionality that can be invoked by adding a specific query string to the URL that calls a specific method in the background. Both forms of documents are accessible by following the links on a page as a Web crawler does. Yet COAST envisions focused crawling for publicly available Web services.

An approach for crawling WSDL documents, presented in [9], identifies seed URLs by checking whether a fetched XML resource is a valid WSDL description and whether it refers to publicly accessible endpoints. The URLs in the frontier can be scheduled according to the link graph and the keywords in the URL like "?wsdl", "ws" etc. For crawling the pages describing Web APIs, some data



mining approaches have been considered to classify new pages against a positive example set of Web API pages.

For indexing and searching Web services, current search engines partially match the search terms entered by the user with the Web service's name, location, business, or tModel [10] in the Web service description file to get the results. Yet, the use of these kinds of keywords is limited in WSDL specification. To improve the search quality, the first possibility [10] is to perform a broad matching process through expanding a user's query with the keywords used by other users for similar Web services. The second possibility [11] is to cluster Web services into similar functional groups according to their description files to reduce the search space to more relevant services.

COAST will identify Web services portals through their links to WSDL files and will distinguish them on the results page to highlight their presence for the end-users. We believe that this is more useful than pointing to WSDL files, as a user will be more interested in accessing the Web service through a proper interface, such as the ones found on their Web portal.

#### **4.1.1.6 Coast Media objects**

COAST media objects are treated differently of non-COAST objects by the search engine. A non-COAST object is typically a text document encapsulated in some sort of markup, such as HTML or DOCX or a "printed" document in formats such as PDF. It is trivial to extract the text content from these documents and use this text to index the document.

COAST media objects however, do not consist of simple text like this. COAST media objects are typically videos, images and music files. According to the media format specification [73], COAST objects must have a description as part of their metadata. For COAST media objects, we use this description for indexing. Extracting visual features from videos and images is an important research topic and have been explored elsewhere. Such feature extraction, however, is not in the scope of COAST, the interested reader may refer to [78] and [79] for an overview of this subject.

As the COAST metadata is stored outside of the COAST object itself, it can be fetched independent of the content. This is important as the size of the content itself could be of hundreds of megabytes, and fetching it wastes crawler resources if only the metadata is needed.

### **4.1.2. Performance considerations**

#### **4.1.2.1 Download throughput**

Crawlers typically make use of multi-threading [17], i.e., Web pages are fetched and processed by many task-wise identical threads, each handling a different HTTP connection. Typically, a crawler increases the number of threads until the available download bandwidth saturates or the overhead of context switches in the operating system beats the purpose of multi-threading.

High latency when downloading a page can result in a large number of outstanding requests on a system. This adds overhead to the operating system, which needs to maintain information on all open connections. COAST achieves lower latencies by assigning each Web page to the crawler located at the smallest network distance

#### **4.1.2.2 Data structures**

With multi-threading, disk accesses and memory form the main bottlenecks of crawling. Efficient implementation of data structures used by the crawler is therefore crucial. These data structures mainly involve the frontier queue of the crawl and the data structure that keeps previously seen URLs. The latter requires a random access for every discovered URL, which can be very expensive. In [3], the authors avoid this bottleneck by building a buffer of URLs to check and checks them against the URL database as a batch process.



### 4.1.2.3 Hazards

A few obstacles arise for crawlers due to malicious intent of Web site owners. An example of malicious intent is delay attacks, where Web servers introduce unnecessary delays in their responses to requests of the crawler. An important hazard to crawler is spider traps, which try to keep the crawler busy by dynamically creating infinitely many useless links. A crawling thread caught in a spider trap can significantly degrade utilization of system resources. There are few techniques to mitigate this problem. The work of [3] introduces a technique that uses reputation to ensure that the crawler will only crawl a trap site for a finite number of iterations.

### 4.1.3. Constraints on crawlers

#### 4.1.3.1 Politeness

Web crawlers can create significant load on the Web servers they are crawling. In fact, this is comparable to a Denial of Service attack for some Web servers: the high request rate on the server could even make the Web site unavailable for normal users. To avoid such negative impact, Web crawlers must implement a politeness policy to limit their crawl rate [41]. Most crawlers handle this constraint by enforcing that crawling processes do not make more than one request to any particular host at the same time [3][16][17]. Additionally, a delay can be added between requests to the same host.

#### 4.1.3.2 Robots exclusion protocol

The robots.txt file is a method used by Web masters to tell Web crawlers which pages they would like crawled and which they would like ignored [21]. The crawler must maintain a cache of these files. When crawling a page, the robots.txt for that page must be checked. This requires a random access, which can lower throughput. To mitigate this problem, the robots cache can be checked in a batch fashion, similar to the seen list.

## 4.2. Indexing

### 4.2.1. Inverted index

Efficient processing of queries requires converting the crawled document collection into an inverted index [23]. This process is illustrated in Figure 8.

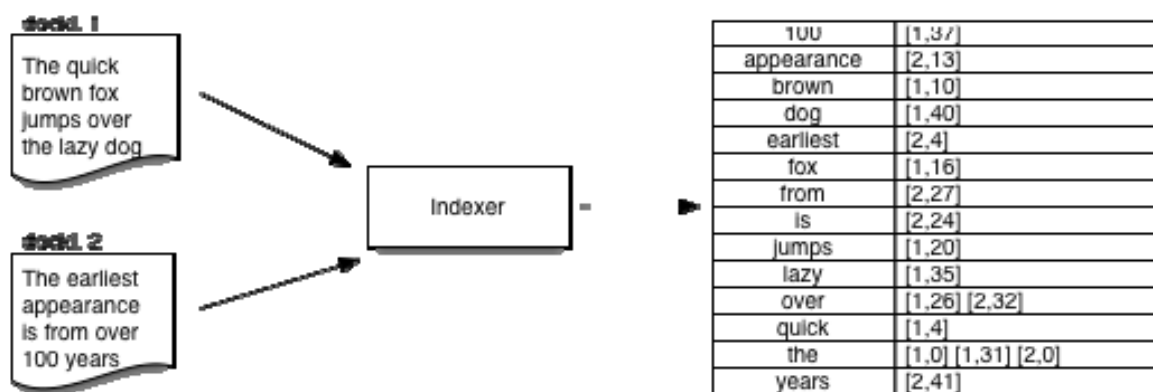


Figure 8: Indexer transforming documents into an inverted index

An index consists of an inverted list for each term in the vocabulary of the collection and an index that keeps pointers to the beginning of these lists. The inverted list of a term keeps a set of postings



that represent the documents containing the term. A posting typically stores a document id and the frequency of the term in the document. Optionally, a position list can be maintained to record the positions that the term appears in the document.

In practice, the index is small enough to fit entirely in memory. Depending on the availability of RAM, some inverted lists may be stored on the disk, in compressed form [44][45]. Encoding speed is not very important, as index generation is an offline task.

Skip lists [46] are another popular way to improve the processing of posting lists. Descriptors are placed at intervals in the posting list, which make it possible to see which doc ids a chunk may possibly contain without decompressing it (Figure 9). However, with the asymmetric increase in CPU speeds in comparison to disk speeds, some of the performance gain of using skip lists has dissipated.

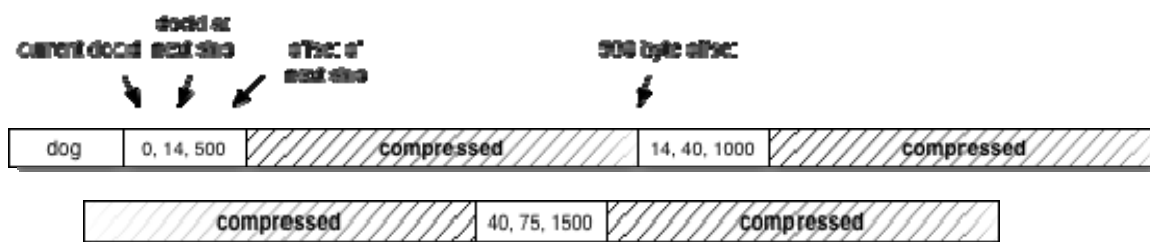


Figure 9: Structure of a skip list

### 4.2.2. Index maintenance

An index can be constructed in one or two passes. The two-pass strategy [47] works as follows. In the first phase, the size of each inverted list is found and a skeleton is created for the index. In the second phase, the content of the postings are filled. The skeleton can be maintained in memory or on disk. If the document collection is large, the single-pass approach may perform better [48]. In this approach, parts of the index are computed and flushed on the disk periodically as the memory becomes full. These parts are then merged into a single index.

Constructing an index is not sufficient, as the search engine continuously crawls the Web and finds new content. The index must be updated to reflect this new content [49]. Regeneration of the whole index is one option. However, this is quite expensive, and a substantial amount of processing is potentially repeated. Another option is to incrementally update the index [50]. This technique is preferred when indexing time-sensitive collections. The last option is to create an auxiliary index [51]. New documents are added to the auxiliary index. When a query is served from the index, it is simultaneously looked up in the main index and the auxiliary index (Figure 10). The final result is an intersection of the highest scoring documents from both sets. Eventually, the auxiliary index becomes too large and we merge it with the main index.

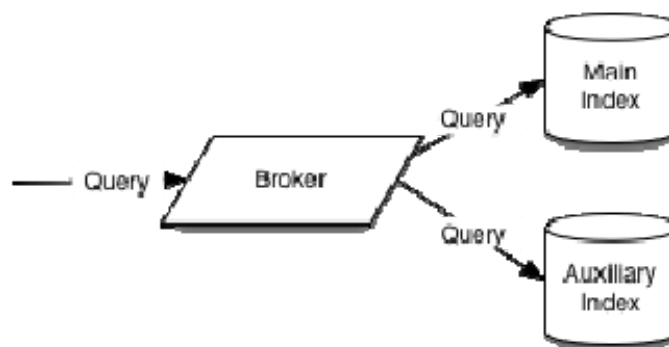


Figure 10: Incremental indexing through an auxiliary index



### 4.2.3. Index partitioning

An index for the Web does not fit into one machine. It needs to be partitioned among several machines. Typically, the partitioning is based on terms or documents

#### 4.2.3.1 Term-based partitioning

Term-based index partitioning [52] involves placing the posting lists for specific terms on separate machines. For example, one machine could have all terms beginning with 'A', another those terms beginning with 'B' and so on, as illustrated Figure 11.

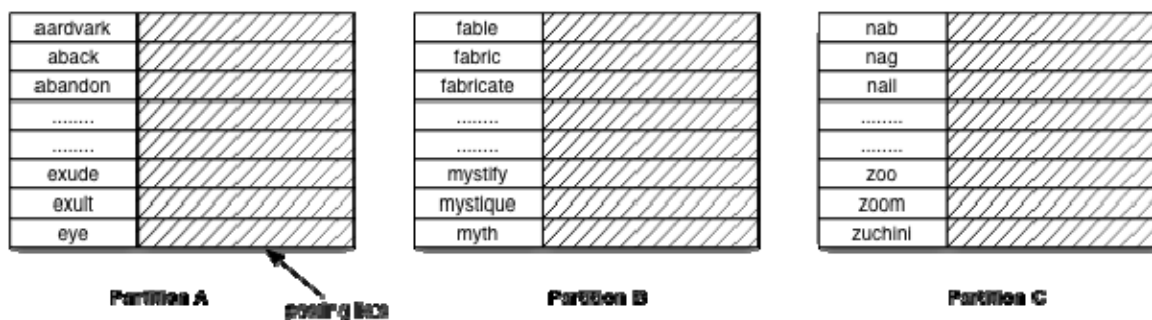


Figure 11: Term-based index partitioning

Term partitioning schemes tend to suffer from load balancing [53]. Moreover, the failure of one machine in the site can mean the loss of a number of terms from the index, which severely degrades the relevance of results from the index if a query containing one of the lost terms is evaluated.

#### 4.2.3.2 Document-based partitioning

Document-based partitioning [54] divides all the documents in the collection across several machines (Figure 12). Each machine acts as if it is its own independent index. Document partitioning allows for good load balancing. Document partitioning is more resilient to faults compared to term partitioning. If a single machine crashes, only the documents on that machine are unavailable, and the query processor will still be able to serve relevant results.



Figure 12: Document-based index partitioning

## 4.3. Query Processing

### 4.3.1. Preprocessing

Each query goes through a number of preprocessing steps before it is evaluated and matched to documents. The purpose of preprocessing is to help improving the quality of matching. Most commonly used preprocessing steps include stopword elimination, case-folding, stemming, spell



correction, and phrase extraction. After preprocessing, the user query is transformed into an internal representation through query rewriting techniques, which may change the semantics of the query. This transformed query is processed on search nodes to retrieve the best-matching documents. More details about query preprocessing can be found in [76].

### 4.3.2. Top-k processing

The query is basically processed in two steps: matching and scoring.

#### 4.3.2.1 Matching

First, the query is matched to some potentially relevant documents by traversing the posting lists associated with the query terms. Matching can be in conjunctive or disjunctive mode. In the conjunctive mode, a document is considered to be a match for the query only if it contains all query terms (Figure 13). In the disjunctive mode, it is sufficient to have only one query term present in the document.

Due to the decompression overhead of posting lists, the matching operation can be costly. To speed up matching, early termination techniques [55][56][57] are employed. This way, in certain cases, the matching operation can be terminated before all postings associated with the query terms are visited. Some early exit algorithms guarantee the correctness of results, with respect to computation over the full lists, while some others yield only approximate results

#### 4.3.2.2 Ranking

Once the matching documents are determined, they are ranked in decreasing order of estimated relevance scores (this can be interleaved with matching as well). The relevance scores are estimated using statistical techniques such as BM25 [77]. Optionally, more sophisticated machine-learned ranking techniques [58] can be applied to a subset of the documents (those that are highly ranked by BM25). Finally, the top scored *k* (typically 10 in Web search) documents are returned to the user.

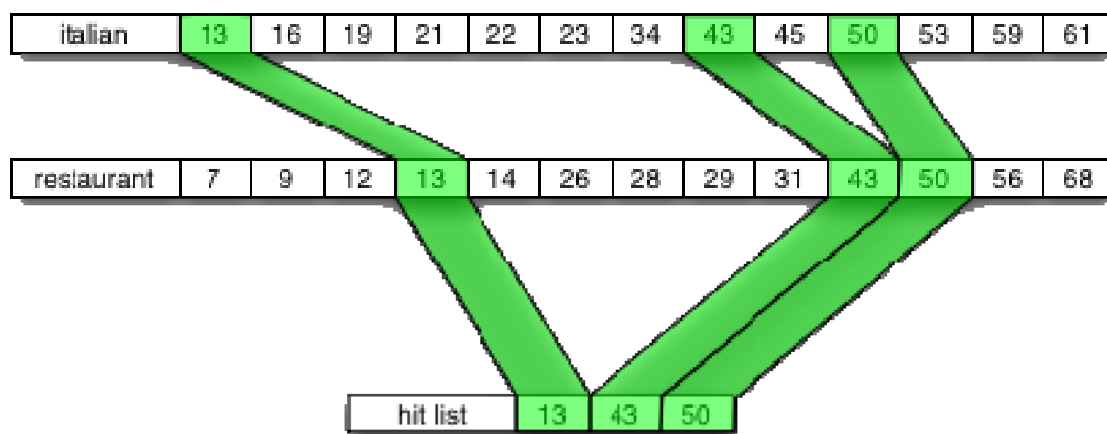


Figure 13: Conjunctive query processing using posting list intersection

### 4.3.3. Query processing on a partitioned Index

The way queries are processed on a search cluster with multiple nodes depends on the way the index is partitioned across the nodes. If document-based partitioning [59][60] is employed, the query is issued by a broker to all search nodes, which individually compute their best-matching *k* documents over their local indexes. These documents are returned to the broker, which then



merges them into a global top- $k$  result set. If term-based partitioning [52][61] is employed, the query is issued to only the nodes that contain at least one posting list associated with the query terms. In this case, the nodes return their entire answer set to the broker to guarantee the correctness of results. In practice, document-based partitioning is preferable to term-based partitioning since it achieves both better load balancing and lower query processing times. Most search engines employ a document-based partitioning strategy [61] as it also provides better fault tolerance in case of node failures.

#### **4.3.4. Snippet generation**

When top- $k$  search results are presented to the users, the URLs are accompanied with descriptive summaries of the content they point to. These summaries are known as snippets. Generating high-quality snippets that describe the content well is important, since snippets constitute an important way to increase the engagement of users with the result page. Snippet generation is one of the final steps in result page generation. A snippet is composed of several short pieces of text, extracted from the content. The extracted text typically contains the terms appearing in the query. An efficient snippet generation technique is proposed in [34].

### **4.4. Caching**

Users expect search results to return under a second. At the same time, the amount of data indexed and searched is massive and continuously growing. Keeping the whole index in memory is impractical, and to achieve low latency in query evaluation, search engines employ caching. There are two primary types of caching used in a search engine, term caching and query caching. An additional benefit of caching is the elimination of a significant fraction of redundant work. Since the distribution of queries typically follows a power-law, caching implies a more efficient utilization of compute resources.

#### **4.4.1. Posting list caching**

To decrease the overhead of fetching posting lists from the disk, the posting lists associated with the most frequently or recently accessed terms are stored in memory[62][63]. This cache can be accompanied with an intersection cache, which stores precomputed intersections of postings. This cache speeds up the matching phase, as fewer postings need to be traversed [64].

#### **4.4.2. Query result caching**

A vital component that significantly reduces the query processing workload of backend search servers is the query result cache [63][65]. This cache maintains the results of frequently or recently processed queries. In the query results are cached, they constitute a hit and are served immediately without further processing. A successful cache decision takes into account issues related to admission [66], prefetching [67], and eviction [26]. Another important issue is to maintain the freshness of cached entries [68][69].

#### **4.4.3. Other types of caching**

Search engines may employ other types of caching as well. This includes caching of document content, caching of scores, and caching of node ids that contribute documents to the top- $k$  result set [70][71][72].



## 4.5. Coordination

The following sections describe the services necessary for the distinct sites in the multisite search engine to work together. Section 4.5.1 describes the configuration services that hold all the sites together and informs them of the service parameters for each other. The subsequent sections then outline the purpose and mechanism of each service.

### 4.5.1. Configuration

Search sites in COAST send data and requests to each other. To enable such coordination across sites, we assume a distributed configuration service (Figure 14).

Each site's configuration stores the host and port of each of the services running on that site. This information is stored for the page sink, the URL sink, the threshold database and the index statistics database. The purpose of these services is explained in the subsequent sections.

Individual sites update this configuration periodically on their local server, for example, when a new threshold database becomes available. The update is sent to all the other services in the configuration service immediately.

Each site monitors the configuration for all the other sites, and takes the appropriate action based on which configuration item is changed. For example, if the URL sink details are changed, the crawler will be notified to send URLs to the new host. If the threshold database entry changes, the query processor will be notified, so that it can download the new database and take it into account while processing queries.

Each site also has a liveness configuration item. The site creates its liveness item on joining the configuration service. If the site leaves the configuration service, this liveness item will disappear. This mechanism allows sites to monitor which other sites are available. If a site is not available, due to a network partition or natural disaster etc, the remaining sites will take this into account and not try to forward any queries to it.

When the site comes back online, it will sync to the latest configuration from the service and download the newest threshold and index statistic databases if they differ from those it had before it went offline.

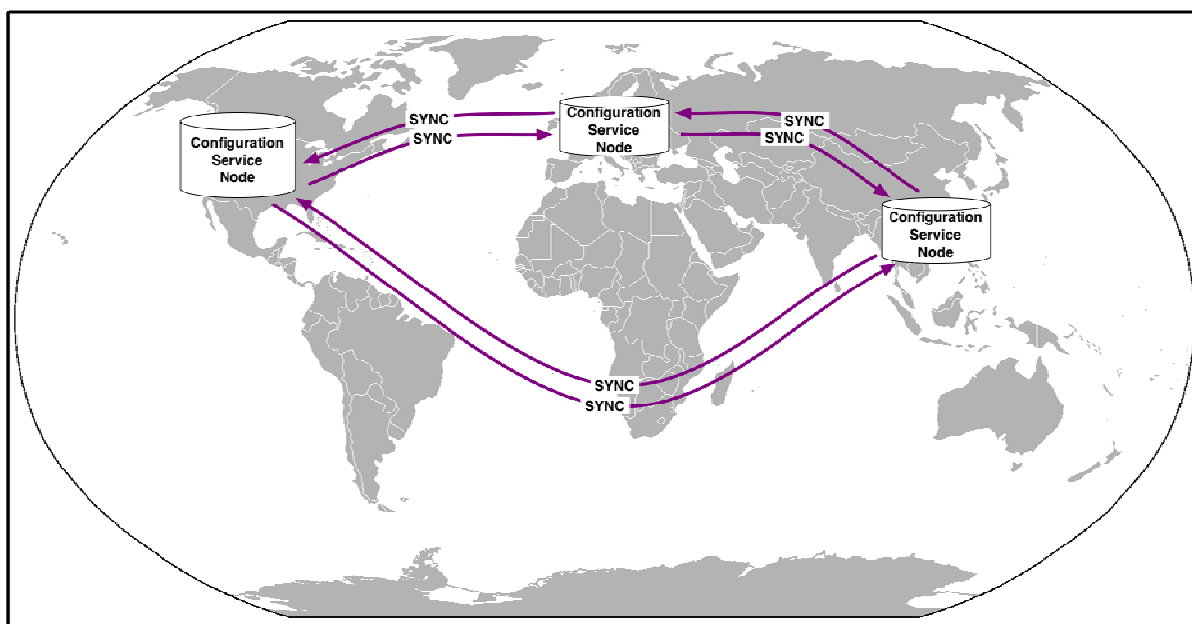


Figure 14: Coordination of the distributed services



The configuration service itself is statically configured in COAST, i.e., each server in the service needs to be manually told about the other servers. Adding dynamic configuration is a future possibility. However, for COAST there will be only a handful of sites, so manual configuration is not a significant burden.

**4.5.2. URL Exchange**

Each site has a sink where URLs that have to be crawled by that site are sent. The sink includes the URLs in a database, to be picked up by an asynchronous process and put into a URL queue for the crawler. This is illustrated Figure 15.

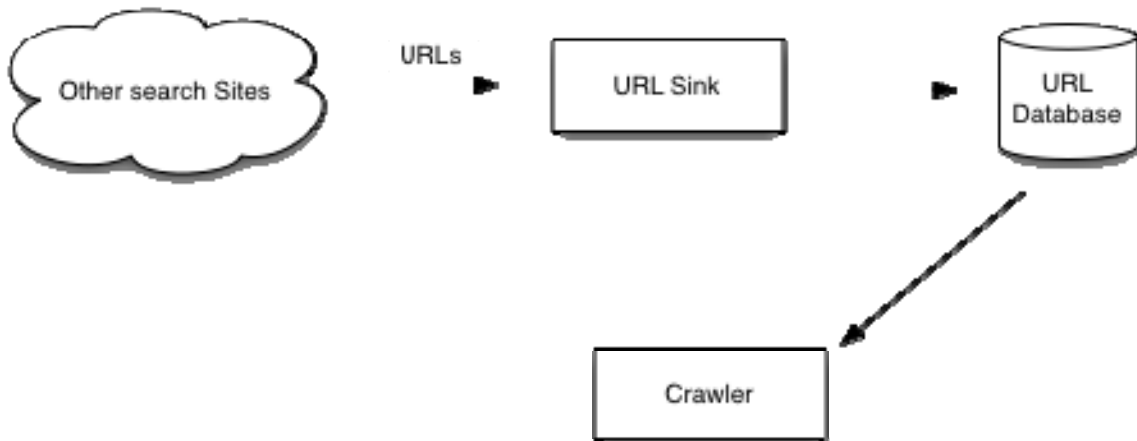


Figure 15: URL sink for the URL exchange procedure

**4.5.3. Page Exchange**

Each site has a sink into which pages which have to be indexed by the site are uploaded. The sink includes the pages in a database that an asynchronous process picks to feed them into the indexer (Figure 16).

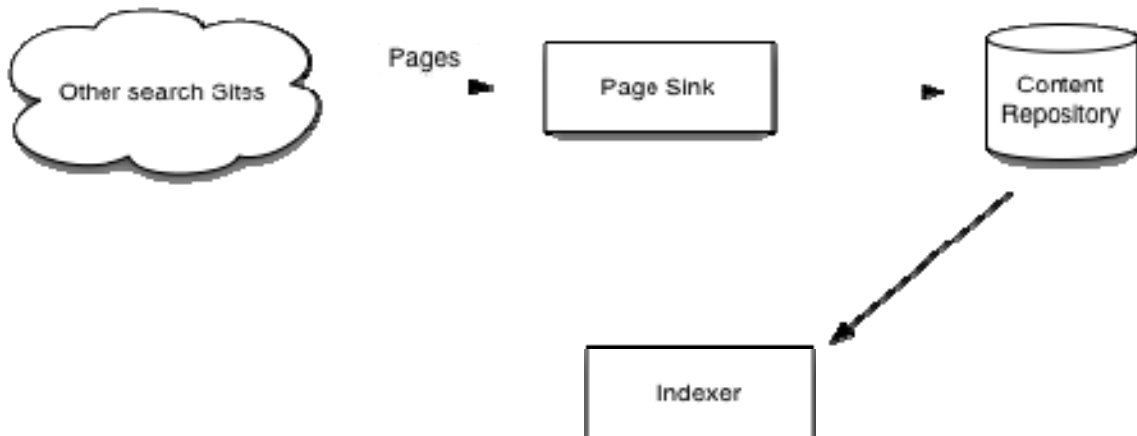


Figure 16: Page sink for the page exchange procedure



#### 4.5.4. Index statistics exchange

Statistics for page classifier (Section 3.2.1) are exchanged periodically. To send its newest page classification index to other sites, a site makes the index available over HTTP and notifies the configuration service that the latest classification index is available at the HTTP URL that it has made available. Other sites download the index once they see the change in the configuration (Figure 17).

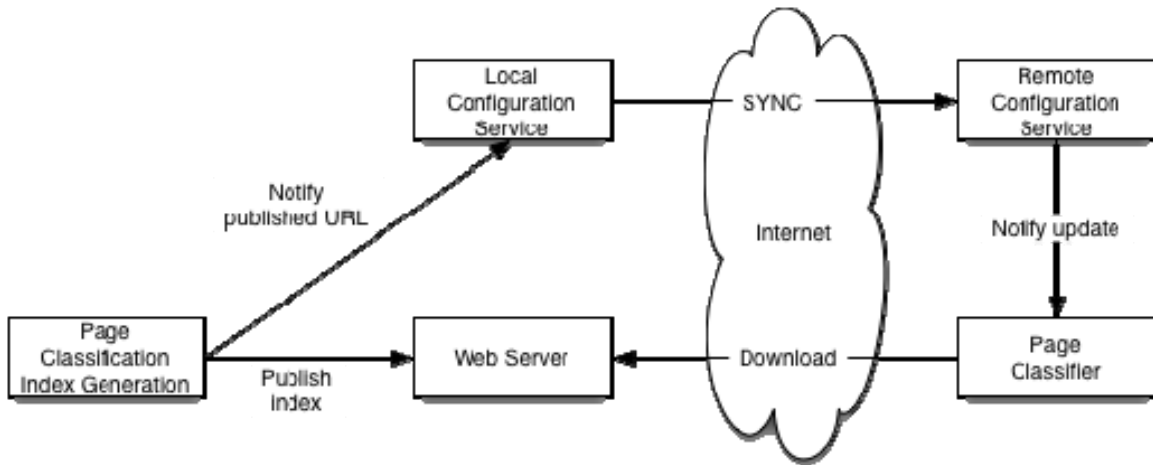


Figure 17: Page classification index exchange procedure

#### 4.5.5. Threshold Exchange

Threshold exchange works similarly to page classification index exchange. To send the newest threshold database to other sites, a site makes it available over HTTP and updates the configuration service with the URL of the newly available database.

Every time a document is indexed, the threshold database may potentially be updated. Therefore, there is a limit on the rate at which databases are exchanged, based on the number of thresholds that have changed and the time since the previous exchange (Figure 18).

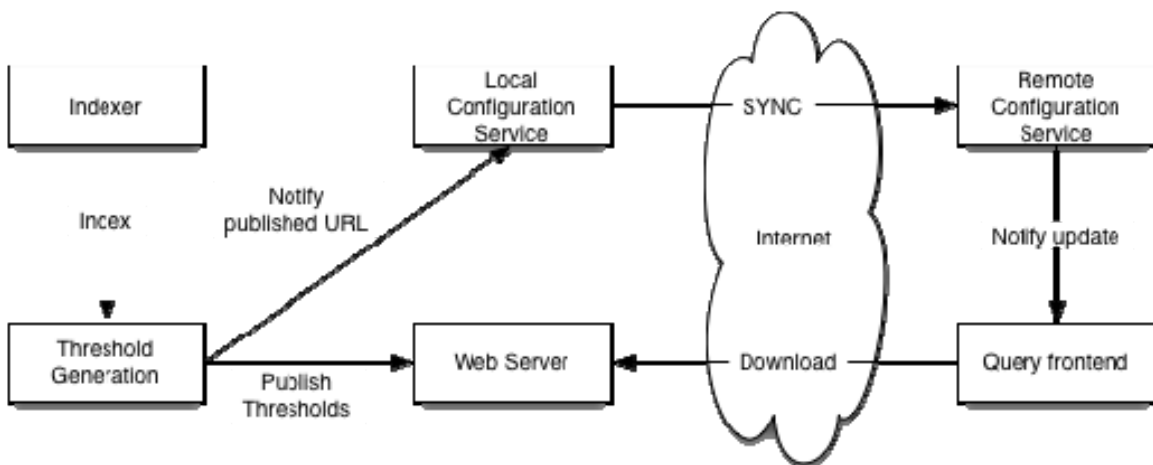


Figure 18: Threshold exchange procedure



#### 4.5.6. DPI sink

Each site has an interface to receive DPI information. The interface receives the information and places it directly into a database to be processed by an offline process as shown Figure 19.

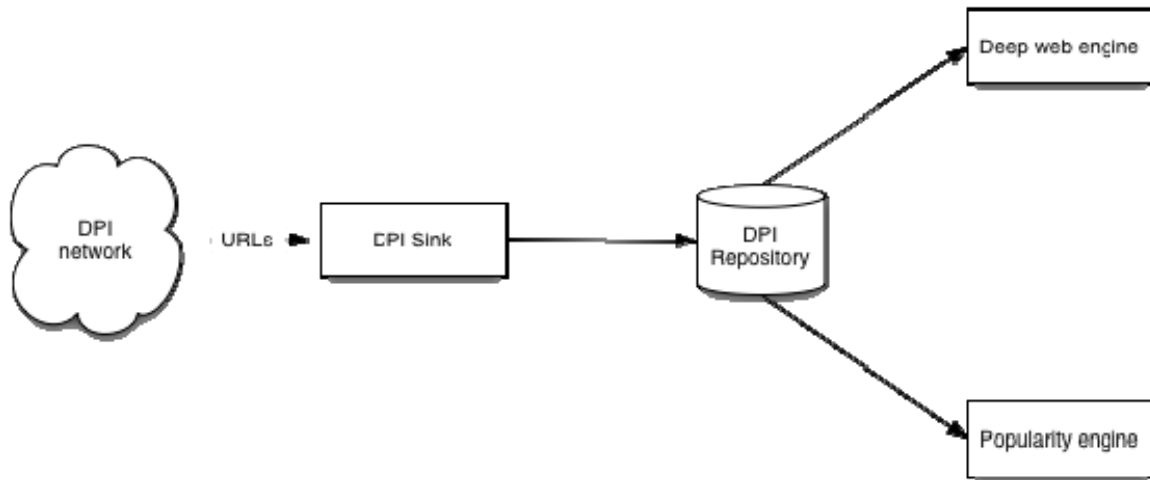


Figure 19: Passive crawling input for crawling

#### 4.5.7. Content submission sink

There is a user interface for users to submit COAST content to be indexed. The user interface places the metadata about the COAST content into a database, which is then picked up by an asynchronous process, which indexes the content and make it available for search.



## 5. Metrics and evaluation

In Sections 3 and 4, we describe the architecture of the COAST search engine. This architecture is distributed in several search sites, and fundamentally differs from the existing designs. In this section, we present the experiments we will perform to evaluate the efficiency of our approach. We first focus on the new crawling mechanisms (Section 5.1), and then evaluate the search engine in terms of performance (Section 5.2) and quality of results (Section 5.3).

### 5.1. Crawling

#### 5.1.1. Active crawling

The COAST distributed crawler is designed to leverage network proximity to increase crawling performance by retrieving documents faster. To quantify the gains achieved, we plan to simulate a crawl on real Web traces and compute the crawling cost in different crawler setups.

##### 5.1.1.1 Data set

The Yahoo! search engine crawler generates a data structure, called the Web graph, a representation of the Web graph that Yahoo! crawls. We will use this data to obtain a subset of the Web and experiment different crawling strategies. Given that crawling is a long process that requires a significant amount of bandwidth and puts extra load on Web servers, relying on traces to simulate large crawls will provide us with a larger and more realistic source of information than experiments based on small-scale Web crawls that are difficult to reproduce. Therefore, we will obtain a dataset containing millions of pages and analyze both the geographical location of the servers hosting these pages, and the link structure between these pages. Yahoo! has access to an accurate GeolIP database, which will provide us with the necessary information to locate each server hosting a Web page in the sampled dataset. As a consequence, our crawling dataset will consist of a few millions Web pages, each of which is assigned geographical coordinates and is linked to other Web pages.

##### 5.1.1.2 Experimental setup

The goal of this experiment is to assess the efficiency of several distributed active crawlers, as opposed to one centralized crawler. Hence, we will devise several crawler configurations. We rely on the locations of existing Yahoo! data centers to determine the location of possible crawlers. We Such a choice for the location of crawlers make our results more realistic, since we are exploring crawler deployments that rely on the existing infrastructure. We expect a reuse of existing infrastructure to reduce the costs of setting up a search service, since building new data-centers for the only purpose of deploying one distributed crawler instance is not realistic.

Once the position of the crawlers is determined, we can compute the assignment of Web pages to crawlers, as well as estimate the download time. The time it takes to retrieve a Web page can be estimated from the geographical distance between the crawler accessing the page and the server hosting it. The strong correlation between geographic distance and network distance has already been studied in previous work, such as [6]. Furthermore, [27] models the behavior of TCP transfers depending on the network distance. HTTP transfers, used to obtain Web pages, are built on top of TCP. As a consequence, we obtain the network distance from the geographic coordinates and the download time from the network distance.

Finally, we can also compute the transfers of information required between crawlers. Each time a Web page is downloaded, a crawler extracts the hypertext links present in the page and contacts



the crawler responsible for the page it leads to. Once the pages are assigned to crawlers, it is easy to transform the graph of Web pages into a graph of crawlers and count the edges between them.

### 5.1.1.3 Metrics

As explained in Section 5.1.1.2, the crawling experiment mostly consists of two different metrics: the time it takes to download the documents (**Total Download Time**), and the amount of data transferred between crawlers (**Inter Crawler Communications**). The goal is to crawl the documents as fast as possible, while minimizing the data transmitted between crawlers.

$$TDT = \sum_{c \in \text{crawlers}} \sum_{p \in \text{assignPages}(c)} \text{downloadTime}(c,p)$$

$$ICC = \sum_{c \in \text{crawlers}} \sum_{d \in \text{crawlers}, d \neq c} \text{nbLinks}(c,d)$$

Given that network distance has a strong impact on download time, we expect the total download time to decrease as the number of crawlers increases. Crawlers are on average closer to the Web servers. However, deploying more crawlers induces more communication across crawling sites, since a hypertext link has a higher probability to lead to a page that should be fetched by a different crawler. We will propose different configurations and evaluate the different trade-offs. The last metric we will evaluate is load balance: the load distribution (**Load Balance**) across the crawlers should be as balanced as possible.

$$LB = \frac{\max(\text{load}(c), c \in \text{crawlers})}{\min(\text{load}(c), c \in \text{crawlers})}$$

## 5.1.2. Passive crawling

The deployment of the passive crawling solution is part of the work package 4. Therefore, our evaluation will only focus on its interactions with the search engine. We will evaluate how passive crawling can efficiently improve the coverage of the crawler by discovering new pages.

The passive crawling aims to improve the URL discovery process in the COAST distributed Web crawler. To quantify the impact of passive crawling on both the content system and the runtime system of a search engine, we consider different forms of user feedback as source of passive crawling and conduct a large-scale experimental evaluation on real datasets from the Web.

### 5.1.2.1 Dataset

In our experiments, we make use of a snapshot of the full Yahoo! Web crawl, referred to as a Web graph. This data structure is used to represent the URLs available at the Yahoo! Web crawler, which are obtained without any passive crawling technique.

To simulate the user feedback that is to be involved in the passive crawling, we consider two different datasets: the toolbar logs and the DPI traffic. A toolbar is software deployed on users' Web browsers. It can track some of the actions that the users perform on the browser, such as typing a URL and clicking a link, and report them to the toolbar log of the search engine. The URLs accessed by the users and recorded by the toolbar can thus be considered as a rich source of information to involve in the passive crawling. The toolbar log we use is collected through the Yahoo! Toolbar and it contains for each URL its access time by the user. The toolbar log on each day used in our experiments represents a fraction of the Yahoo! Toolbar daily workload.

In addition, to quantify the impact of passive crawling on the search result quality, we plan to use a large-scale query log from Yahoo! Search engine. This query log is necessary to evaluate how many pages discovered through passive crawling may have the query results improved.



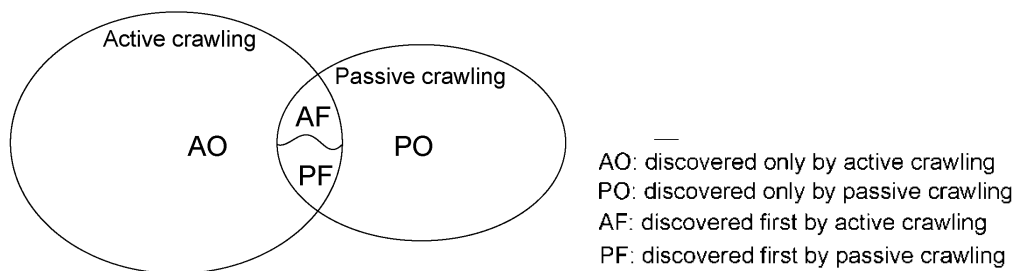
### 5.1.2.2 Setup

The goal of our experiments is to assess the impact of passive crawling on the URL discovery. As a consequence, we do not rely on any distributed architecture and we assume that all the URLs accessed by the users, recorded either via the toolbar log or the DPI traffic, are directly provided to the crawler containing the full Web crawl.

### 5.1.2.3 Metrics

We seek to evaluate the impact of passive crawling from two different aspects: the impact on the content system of a search engine, i.e. coverage, and the impact on the runtime system of a search engine, i.e. result quality.

For the impact on the content system, we are first interested in the quantity of URLs accessed by the users that are complementary to the current Web crawl obtained solely through active crawling. To this end, we compare the URLs on the Web map and those from the toolbar log or the DPI traffic to measure the fraction of the URLs that are only discovered by the users or discovered earlier by the users. A larger fraction of URLs discovered only by users or earlier by users implies a more significant impact on the coverage of the crawler. Figure 20 illustrates the nature of the URLs according to their discovery time. Our evaluation focuses on the URLs from the PO and PF sets since only these URLs potentially have positive impacts on the active crawling. We also quantify the time difference between the crawler and the user for the URLs discovered earlier by the users (PF). The larger the difference, the more advantageous the passive crawling is.



**Figure 20: URL sets based on the discovery time of URLs through active and passive crawling**

To measure the quality of the URLs and determine whether they are indeed useful, we use two different metrics. For the URLs that are only discovered through passive crawling (PO), we propose a user study to classify the usefulness of URLs based on their content. For the URLs that are discovered first through passive crawling (PF), we use both a link-metric whose value depends on the in-links of a URL ( $M_{link}(u)$ ) and a content-metric whose value depends on the content of a URL extracted from the Web graph ( $M_{content}(u)$ ). The values of both metrics are computed as follows with our sample and increasing values indicate higher importance of the URLs.

$$AvgM_{link} = \frac{\sum_{u \in urls} M_{link}(u)}{|urls|}$$

$$AvgM_{content} = \frac{\sum_{u \in urls} M_{content}(u)}{|urls|}$$

For the impact on the runtime systems, since a URL can appear in the query result only after it is discovered and indexed, we measure the number of queries whose top- $k$  results contain at least one URL discovered first by the users (PF). A larger number of URLs being discovered first implies a higher volume of query results being affected by passive crawling.



## **5.2. Search performance**

In this section, we evaluate the system performance of query processing. This includes the performance perceived by the user, but also the maintenance cost of the search engine. We discuss distinct experiments to evaluate the different mechanisms of the COAST search engine and assess their efficiency.

### **5.2.1. Data set**

To evaluate the search engine, we need to obtain a large collection of documents to index, as well as queries to constitute a workload. To this end, we will rely on the datasets already presented in Section 5.1.

When users access the COAST search engine, they are directed to the closest search site. Consequently, each search site is responsible for answering queries originating from a given region. To model this behavior, we plan to rely on query logs originating from the Yahoo! search engine. This data contains, for different countries, the query of the user as well as the results clicked. Depending on the number of users and the size of the country, we can decide to either assign one country to a given search site, or several neighboring countries.

### **5.2.2. Initial placement of documents**

After retrieving a document, a crawler has to decide which indexer has to incorporate it. This decision is important as this indexer will become the master for the document, and it has to keep this document in its index. If the indexer of a region incorporates documents that its users are not interested in, then the document causes the index of this region to inflate unnecessarily. Furthermore, the other search sites, where users actually are interested in the document, have to forward their queries, and eventually replicate the document if it becomes popular enough. This generates network traffic and increases the latency of user queries. It is therefore important to select indexer sites for documents precisely to enable efficient query processing and to avoid a poor utilization of system resources.

#### **5.2.2.1 Experimental setup**

We use our Web sample as a documents collection and index it in a search engine. Using query logs, we determine which of these documents are relevant in each different region, corresponding to different search sites. We use part of these documents as a training set, to create some background knowledge about each site, and the other part of these documents as a testing set. This means that we will assume that these documents are new and have to be incorporated by some indexer. We will implement different features in a classifier to assign each document to the search site it is most similar to, and evaluate the quality of the assignment using the ground truth obtained from the queries.

#### **5.2.2.2 Metrics**

The metric traditionally used for classification problems is **Precision** [28]. Basically, it represents the proportion of good classification decisions. In our case, it may be difficult to express the quality of our classifier simply through precision. We assume several search sites for our system and, in the case of popular documents, more than one region could be interested in a given document. In our case, the best assignment decision could be sending it to the search site where it is most popular. However, the score obtained for a classification decision should not be binary, as sending it to the second search site in terms of popularity ranking also is a quite good decision. Therefore, we will also compare the location decided by our classifier with the full ranking of popularities, and use the distribution of ordinal numbers as an evaluation.



$$P = \frac{\sum_{i \in \text{indexers}} |d \in \text{documents}, d \in \text{assignDoc}(i) \wedge i = \text{mostPopularLocation}(d)|}{|\text{documents}|}$$

### 5.2.3. Document replication

- Replicating documents is a challenging problem, since it has important consequences. A search site decides to replicate a document in its index when it determines that it is popular in its region. As a consequence, the search site does not need to forward queries concerning this document anymore. Replicating the document saves bandwidth and improves the query response time for the user. However, adding a document to an index increases the size of the posting lists, and makes it longer to process them. Consequently, replicating too many documents leads to an overall increase of processing time. Another consequence of increasing the size of the index is fitting a smaller fraction into main memory. Less data in memory implies more disk accesses when queries are processed and penalizes response time. In general, replicating popular documents across indexes improves performance, as the search site is able to compute a local answer, but replicating too many documents has a negative impact on the evaluation time of all the queries.

#### 5.2.3.1 Experimental setup

We will deploy a multi-site search engine and run queries to trigger the replication process. This experiment is a continuation of the measurements performed on the placement of documents, presented Section 5.2.2. As explained previously, the crawler sends a given document to the search site where it is most likely to be useful. Yet, users from other regions may also be interested in it. By running queries on these search sites, we will stress the replication mechanism and monitor its decisions and their impact. We will fix a maximum amount of space that can be used to replication without harming the global query performance, and experiment different replication strategies to see which one uses the replication budget best.

#### 5.2.3.2 Metrics

The document replication mechanism is introduced to improve query performance by answering them locally. Consequently, an important metric for these experiments is query latency. Similarly to [8] and [75], we will measure the proportion of queries answered locally with respect to the queries forwarded to other search sites (**Query Locality**).

$$QL = \frac{|q \in \text{queries} \wedge \text{local}(q)|}{|\text{queries}|}$$

We are also interested in the dynamics of the replication policy. The replication mechanism overall must be able to react quickly enough, both to replicate new popular documents, and to eliminate documents that are no longer popular to free space. This same mechanism, however, must not cause system instabilities by flapping documents rapidly across sites. The dynamics of replication can easily be observed by studying the evolution of the query locality over time.

### 5.2.4. Query forwarding

The documents are split across different indexers, and replicated in regions where users display interest for a given document. Hence, query processors benefit from highly optimized indexes that are designed to answer a large proportion of queries locally, and that yet are small enough to compute results quickly. To preserve the quality of the results, a query sometimes has to be forwarded to another query processor that benefits from a different index and know about other documents. Forwarding a query is expensive, as it consumes bandwidth, forces another query processor to compute results, and increases the overall latency of a query.



#### 5.2.4.1 Experimental setup

Once again, we will use our data to generate indexes at different locations and simulate queries. The focus of this experiment will be on the forwarding decision. Search sites communicate with each other to exchange summaries of their content that are used in heuristics to determine if a query should be forwarded. As already explained, we will rely on the threshold algorithm. Still, there are different policies ([8], [75]) to build and exchange the content summaries of the search sites. In this experiment, we will evaluate several information exchange patterns and heuristic to determine an efficient algorithm for query forwarding.

#### 5.2.4.2 Metrics

For each query, and each distant search site, the query processor has to make a decision: either forward the query, or only process it locally. Forwarding a query is useful if the distant site can contribute to the set of results by providing relevant results. The metric we consider, as in Section 5.2.2, is precision.

$$P = \frac{|q \in \text{queries}, \text{forwarded}(q) \wedge \text{result}(q) \neq \text{localresult}(q)|}{|q \in \text{querie}, \text{forwarded}(q)|}$$

The threshold algorithm always generates the best results for the user, so it is not possible not to forward a query when it would have been needed. Choices made by the threshold algorithm, however, are often conservative.

#### 5.2.5. Search latency

The total response time of a query, as seen by a user, depends on the latency from the user to the query processor and the processing time. The COAST distributed search engine architecture is designed to reduce the latency between users and query processors. Furthermore, as the indexes are smaller, the processing time is also reduced, as long as the evaluation remains local. If the query has to be forwarded, then additional network and processing delay is added to the response time. Query response time has a major impact on the satisfaction of the user, and this section describes the experiments designed to evaluate it in the context of COAST.

##### 5.2.5.1 Experimental setup

We use our data and workload to run queries on the search engine, and measure the response time. In particular, we are interested in varying the number of search sites. With more search sites, the latency to end-users is smaller. Yet, for a given total budget, each site will have smaller indexes and therefore more queries will be forwarded. This experiment will investigate the trade-offs between few large sites and many small ones.

##### 5.2.5.2 Metrics

In this experiment, we simply measure the response time observed by the user when it submits queries. While it is important to obtain a small average response time, we will also study its distribution, and favor a low standard deviation.

### 5.3. Search quality

The COAST search engine relies on a novel crawling infrastructure to retrieve documents. Active crawling is efficient due to the distribution of the crawlers, and passive crawling provides new information on the interest patterns of the users. Both architectural choices can significantly improve the quality of the results served by the search engine. The following experiments quantify



these improvements and establish the precise impact of each component of the COAST architecture.

### 5.3.1. Document collection

To serve relevant results, a search engine first has to acquire a collection of documents. This collection of documents is obtained through crawling. We design this experiment to highlight how the crawling decisions taken in COAST affect both the number of documents known by the search engine and their freshness.

#### 5.3.1.1 Dataset

This experiment evaluates the impact of crawling on the quality of the results. Consequently, we will use the data presented in Section 5.1. However, in this experiment, we focus on the temporal information as we are interested on how fast the active and passive crawlers can discover new interesting documents as well as modifications in already known pages. It is important to remember that crawlers do not fetch all documents, as many of them are not interesting for the given site. One important challenge is to determine whether the combination of active and passive crawling can capture all the interesting Web pages and react to their changes.

#### 5.3.1.2 Experimental setup

We rely on query traces and click logs to identify Web pages that are interesting for the users. Then, we identify how they were discovered, either through active or passive crawling. Using time information, we can compare the performance of the two crawling processes in terms of reactivity.

#### 5.3.1.3 Metrics

The main novelty behind the COAST crawling process is the addition of passive crawling. Therefore, our main metric to evaluate the quality of the documents collection of the search engine is focused on passive crawling. In particular, we will compute the contribution of the passive crawler to the documents returned to the users (**Passive Crawling Contribution**).

$$PCC = \frac{|d \in documents \wedge passiveCrawling(d)|}{|documents|}$$

We are also interested in the refresh rate of the crawlers, i.e. the average age of documents in the index. Hence, we compute the proportion of stale information in the index (**Stale Documents**).

$$\square \quad SD = \frac{|d \in documents \wedge stale(d)|}{|documents|}$$

### 5.3.2. Scoring functions

The scoring function  $\square$  assigns rates to documents with respect to a query. Given that, for many popular queries, search engines return millions of documents, the scoring function has a huge impact in determining which of these documents will be presented first to the users. The COAST search engine benefits from passive crawling information about the popularity of documents. This experiment aims at determining whether this knowledge can be leveraged to improve the scoring function of the search engine.

#### 5.3.2.1 Dataset

We rely on the documents collection (Section 5.1) and query logs (Section 5.2) to run query traces and compare the results of our search engine with the choices of the users.



### 5.3.2.2 *Experimental setup*

We elaborate different scoring functions involving the popularity of documents to determine if we can improve the user satisfaction by better ranking documents that they consider relevant. In particular, we focus on tail content, as it is not well covered by click logs.

### 5.3.2.3 *Metrics*

When they are presented with a results page, most of the users select a link that is on the first page. This shows the importance of **Top- $k$  Precision**: the ability of the search engine to place a relevant document among the  $k$  first ones. We will use this precision metric to evaluate the quality of our scoring function and the impact of passive crawling popularity information.

$$TKP = \frac{|q \in queries \wedge rank(result) \leq k|}{|queries|}$$

□



## 6. Summary

This report presents the architecture of the COAST search engine. This architecture comprises several search sites spread across different geographical locations. Contrary to current search engines, these search sites are not copies of each other: they are specialized to reflect the interests of the users in their region. Consequently, the search sites can answer most of the users' queries locally. However, if the search site cannot produce a satisfactory result set, it forwards the query to other search sites that benefit from more knowledge about this given topic.

In this document, we highlight the challenges posed by this distributed architecture, and propose different mechanisms to efficiently solve them. Finally, we describe the experiments that we will perform to evaluate the quality of our solutions and assess their efficiency. Future reports will put more emphasis on each of the three different components of the search engine (crawler, indexer and query processor), and present concrete experimental results.



## 7. References

- [1] COAST Consortium. D4.1: Evaluation of Algorithms (2011)
- [2] COAST Consortium . D2.1: Service Requirements Specification (2010)
- [3] Lee et al. IRLbot: scaling to 6 billion pages and beyond. TWEB (2009) vol. 3 (3) pp. 1-34
- [4] Cho and Garcia-Molina. Parallel crawlers. Proc. of WWW (2002) pp. 124-135
- [5] Cambazoglu et al. On the feasibility of geographically distributed Web crawling. Proc. of INFOSCALE (2008)
- [6] Huffaker et al. Distance metrics in the Internet. Proc. of ITS (2002)
- [7] Brefeld et al. Document assignment in multi-site search engines. Proc. of WSDM (2011)
- [8] Baeza-Yates et al. On the feasibility of multi-site Web search engines. Proc. of CIKM '09 (2009) pp. 425-434
- [9] Steinmetz et al. Web Service Search on Large Scale. Proc. of ISOC (2009) pp. 437-444
- [10] Nayak. Data mining in Web services discovery and monitoring. International Journal of Web Services Research (2010)
- [11] Elgazzar et al. Clustering WSDL Documents to Bootstrap the Discovery of Web Services. Proc. of ICWS (2010)
- [12] Cho et al. Efficient crawling through URL ordering. Computer Networks and ISDN Systems (1998) vol. 30 (1-7) pp. 161-172
- [13] Diligenti et al. Focused crawling using context graphs. Proc. of VLDB (2000) pp. 527-534
- [14] Olston and Pandey. Recrawl scheduling based on information longevity. Proc. of WWW (2008) pp. 437-446
- [15] Cho. Effective page refresh policies for Web crawlers. ACM Transactions on Database Systems (2003) vol. 28 (4)
- [16] Boldi et al. UbiCrawler: a scalable fully distributed Web crawler. SPE (2004) vol. 34 (8) pp. 711-726
- [17] Heydon and Najork. Mercator: a scalable, extensible Web crawler. World Wide Web (1999) vol. 2 (4) pp. 219-229
- [18] Sun et al. A large-scale study of robots.txt. Proc. of WWW (2007)
- [19] Guo et al. Board Forum Crawling: A Web Crawling Method for Web Forum. Proc. of WI (2006) pp. 745 - 748
- [20] Baeza-Yates and Castillo. Crawling the infinite Web: five levels are enough. Algorithms and Models for the Web-Graph (2004) pp. 156-167
- [21] <http://www.robotstxt.org/orig.html>. A Standard for Robot Exclusion
- [22] [http://en.wikipedia.org/wiki/Haversine\\_formula](http://en.wikipedia.org/wiki/Haversine_formula). Haversine formula
- [23] Zobel and Moffat. Inverted files for text search engines. ACM Computing Surveys (2006) vol. 38 (2)
- [24] Yan and Ding. Inverted index compression and query processing with optimized document ordering. Proc. of WWW (2009) pp. 401-410
- [25] Baeza-Yates et al. The impact of caching on search engines. Proc. of SIGIR '07 (2007) pp. 183-190



- [26] Gan and Suel. Improved techniques for result caching in Web search engines. Proc. of WWW (2009) pp. 431-440
- [27] Cardwell et al. Modeling TCP latency. Proc. of INFOCOM (2002)
- [28] Brin and Page. The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems (1998) vol. 30 (1-7)
- [29] Madhavan et al. Google's Deep Web crawl. Proc. of VLDB (2008)
- [30] Olston and Najork. Web Crawling. Foundations and Trends in Information Retrieval (2010) vol. 4 (3) pp. 175-246
- [31] Raghavan and Garcia-Molina. Crawling the Hidden Web. Proc. of VLDB (2001)
- [32] Shkapenyuk and Suel. Design and Implementation of a High-Performance Distributed Web Crawler. Proc. of ICDE (2002)
- [33] Zeinalipour-Yazti and Dikaiakos. Design and Implementation of a Distributed Crawler and Filtering Processor. Proc. of NGITS (2002)
- [34] Turpin et al. Fast generation of result snippets in Web search. Proc. of SIGIR (2007)
- [35] Gyongyi and Garcia-Molina. Link spam alliances. Proc. of VLDB (2005) pp. 517-528
- [36] Gyongyi and Garcia-Molina. Web spam taxonomy. Proc of AirWeb (2005)
- [37] Bharat et al. A comparison of techniques to find mirrored hosts on the WWW. Journal of the American Society for Information Science (2000) vol. 51 (12)
- [38] Cambazoglu et al. Quantifying performance and quality gains in distributed Web search engines. Proc. of SIGIR (2009) pp. 411-418
- [39] Cho and Garcia-Molina. Effective page refresh policies for Web crawlers. ACM Transactions on Database Systems (2003)
- [40] Dasgupta et al. The discoverability of the Web. Proc. of WWW (2007)
- [41] Eichmann. Ethical Web agents. Computer Networks and ISDN Systems (1995) vol. 28 (1)
- [42] Fetterly et al. The impact of crawl policy on Web search effectiveness. Proc. of SIGIR (2009)
- [43] Exposto et al. Efficient partitioning strategies for distributed Web crawling. Proc. of ICOIN (2007) pp. 544-553
- [44] Moffat and Stuiver. Binary interpolative coding for effective index compression. Information Retrieval (2000) vol. 3 (1)
- [45] Anh and Moffat. Improved Word-Aligned Binary Compression for Text Indexing. IEEE Transactions on Knowledge and Data Engineering (2006) vol. 18 (6) pp. 857-861
- [46] Chierichetti et al. On placing skips optimally in expectation. Proc. of WSDM (2008) pp. 15-24
- [47] Fox and Lee. FAST-INV: A fast algorithm for building large inverted files. (1991)
- [48] Heinz and Zobel. Efficient single-pass index construction for text databases. Journal of the American Society for Information Science and Technology (2003) vol. 54 (8) pp. 713-729
- [49] Lester et al. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. ACSC '04: Proc. of ACE (2004) vol. 26
- [50] Cutting and Pedersen. Optimization for dynamic inverted index maintenance. Proc. of SIGIR (1989)



- [51] Lester et al. Efficient online index construction for text databases. ACM Transactions on Database Systems (2008) vol. 33 (3)
- [52] Lucchese et al. Mining query logs to optimize index partitioning in parallel Web search engines. Proc. of InfoScale (2007)
- [53] Moffat et al. A pipelined architecture for distributed text query evaluation. Information Retrieval (2007) vol. 10 (3) pp. 205-231
- [54] Melink et al. Building a distributed full-text index for the Web. Transactions on Information Systems (2001) vol. 19 (3)
- [55] Turtle and Flood. Query evaluation: strategies and optimizations. Information Processing & Management (1995) vol. 31 (6)
- [56] Broder et al. Efficient query evaluation using a two-level retrieval process. Proc. of CIKM (2003)
- [57] Anh and Moffat. Pruned query evaluation using pre-computed impacts. Proc. of SIGIR (2006)
- [58] Cambazoglu et al. Early exit optimizations for additive machine learned ranking systems. Proc. of WSDM (2010) pp. 411-420
- [59] Cahoon et al. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. Transactions on Information Systems (2000) vol. 18 (1)
- [60] Hawking. Scalable Text Retrieval for Large Digital Libraries. Proc. of ECDL (1997)
- [61] Barroso et al. Web search for a planet: the Google cluster architecture. IEEE Micro (2003) vol. 23 (2)
- [62] Tomasic and Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. SIGMOD Record (1993) vol. 22 (2) pp. 129-138
- [63] Baeza-Yates et al. The impact of caching on search engines. Proc. of SIGIR (2007) pp. 183-190
- [64] Long and Suel. Three-level caching for efficient query processing in large Web search engines. Proc. of WWW (2005) pp. 257-266
- [65] Markatos. On caching search engine query results. ComCom (2001) vol. 24 (2) pp. 137-143
- [66] Baeza-Yates et al. Admission policies for caches of search engine results. Proc. of SPIRE (2007) pp. 74-85
- [67] Lempel and Moran. Predictive caching and prefetching of query results in search engines. Proc. of WWW (2003) pp. 19-28
- [68] Cambazoglu et al. A refreshing perspective of search engine caching. Proc. of WWW (2010) pp. 181-190
- [69] Blanco et al. Caching search engine results over incremental indices. Proc. of SIGIR (2010) pp. 82-89
- [70] Marin et al. New caching techniques for Web search engines. Proc. of HPDC (2010) pp. 215-226
- [71] Ozcan et al. A five-level static cache architecture for Web search engines. Information Processing & Management (2011)
- [72] Li et al. A hybrid cache and prefetch mechanism for scientific literature search engines. Proc. of ICWE (2007) pp. 121-136
- [73] COAST Consortium. D2.2: End-to-End Future Content Network Specification (2010)



- [74] <http://www.worldwidewebsite.com>. The size of the World Wide Web
- [75] Cambazoglu et al. Query forwarding in geographically distributed search engines. Proc. of SIGIR (2010) pp. 90-97
- [76] de Araújo Neto Ribeiro and Baeza-Yates. Modern information retrieval. (2009)
- [77] Zaragoza et al. Microsoft Cambridge at TREC-13: Web and HARD tracks. Proceedings of TREC (2004)
- [78] Datta et al. Image retrieval: Ideas, influences, and trends of the new age. Computing Surveys (2008) vol. 40 (2)
- [79] Lew et al. Content-based multimedia information retrieval: State of the art and challenges. Transactions on Multimedia Computing, Communications, and Applications (2006) vol. 2 (1)