Challenges on Distributed Web Retrieval

Ricardo Baeza-Yates[1,2], Carlos Castillo[1], Flavio Junqueira[1], Vassilis Plachouras[1], Fabrizio Silvestri[3]

[1]Yahoo! Research Barcelona
Barcelona, Spain
{chato,fpj,vassilis}@yahoo-inc.com

[2]Yahoo! Research LA
Santiago, Chile
ricardo@baeza.cl

[3] ISTI - CNR
Pisa, Italy
f.silvestri@isti.cnr.it

Abstract

In the ocean of Web data, Web search engines are the primary way to access content. As the data is on the order of petabytes, current search engines are very large centralized systems based on replicated clusters. Web data, however, is always evolving. The number of Web sites continues to grow rapidly and there are currently more than 20 billion indexed pages. In the near future, centralized systems are likely to become ineffective against such a load, thus suggesting the need of fully distributed search engines. Such engines need to achieve the following goals: high quality answers, fast response time, high query throughput, and scalability. In this paper we survey and organize recent research results, outlining the main challenges of designing a distributed Web retrieval system.

1 Introduction

A standard Web search engine has two main parts [2]. The first part, conducted off-line, fetches Web pages (crawler) and builds an index with the text content (indexer). The second part, conducted online, processes a stream of queries (query processor). From the user's perspective, there are two main requirements: a short response time and a large Web collection available in the index. Given the large number of users having access to the Web, these requirements imply the necessity of a hardware and software infrastructure that can index a high volume of data and that can handle a high query throughput. Currently, such a system comprises a cluster of computers with sufficient capacity to hold the index and the associated data. To achieve the desired throughput, several clusters are necessary.

To get an idea of such a system, suppose that we have 20 billion (20×10^9) Web pages, which suggests at least 100 petabytes of text or an index of around 25 petabytes. For efficiency purposes, a large portion of this index must fit into RAM. Using computers with several gigabytes of main memory, we need approximately 3,000 of them in each cluster to hold the index. As for response time, large search engines answer queries in a fraction of a second. Suppose a cluster that can answer 1,000 queries per second, using caching and without considering other system overheads. Suppose now that we have to answer 173 million queries per day, which implies 20,000 per second one average. We then need to replicate the system at least 20 times, ignoring fault tolerance aspects or peaks in the query workload. That means we need at least 60,000 computers overall. Deploying such a system may cost over 100 million US dollars, not considering the cost of ownership (people, power, bandwidth, etc.). If we go through the same exercise for the Web in 2010, being conservative, we would need clusters of 50,000 computers and at least 1.5 million computers, which is unreasonable.

The main challenge is hence to design large-scale distributed systems that satisfy the user expectations, in which queries use resources efficiently, thereby reducing the cost per query. This is feasible as we can exploit the topology of the network, layers of caching, and high concurrency. However, designing a distributed system is difficult because it depends on several factors that are seldom independent. Designing such a system depends on so many considerations that one poor design choice can affect performance adversely or increase costs. For example, changes to the data structures that hold the index may impact response time.

In this paper we discuss the main issues with the design of a distributed Web retrieval system, including discussions on the state of the art. The paper is organized as follows. First, we present the main goals and high-level issues of a Web search engine (Section 2). The remaining sections are divided up by the main system modules: crawling (Section 3), indexing (Section 4) and querying (Section 5). Finally, we present some concluding remarks.

2 Main Concepts and Goals

The ultimate goal of a search engine is to answer queries well and fast using a large Web page collection, in an en-

1-4244-0803-2/07/$20.00 ©2007 IEEE.

Table 1. Main modules of a distributed Web retrieval system, and key issues for each module.

	Partitioning	Communication	Dependability (synchronization)	External factors
Crawling (Sec. 3)	URL assignment	Re-crawling	URL exchanges	Web growth, Content change, Network topology, Bandwidth, DNS, QoS of Web servers
Indexing (Sec. 4)	Document partitioning, Term partitioning	Re-indexing	Partial indexing, Updating, Merging	Web growth, Content change, Global statistics
Querying (Sec. 5)	Query routing, Collection selection, Load balancing	Replication, Caching	Rank aggregation, Personalization	Changing user needs, User base growth, DNS

vironment that is constantly changing. Such a goal implies that a search engine needs to cope with Web growth and change, as well as growth in the number of users and variable searching patterns (user model). For this reason, the system must be scalable. Scalability is the ability of the system to process an increasing workload as we add more resources to the system. The ability to expand is not the only important aspect. The system must also provide high capacity, where capacity is the maximum number of users that a system can sustain at any given time, given both response time and throughput goals. Finally, the system must not compromise quality of answers, as it is easy to output bad answers quickly. These main goals of scalability, capacity, and quality are shared by the modules of the system as detailed below.

- The crawling module downloads and collects relevant objects from the Web. A crawler must be scalable, must be tolerant to protocol and markup errors, and above all must not overload Web servers. A crawler should be distributed, efficient with respect to the use of the network, and prioritize high-quality objects.
- The indexing module has two tasks: partitioning the crawled data, and (actual) indexing. Partitioning consists in finding a good allocation schema for either documents or terms into the partition of each server. Indexing, as in traditional IR systems, consists in building the index structure. Parallel hardware platforms can be exploited to design and implement efficient algorithms for indexing documents.
- The query processing module processes queries in a scalable fashion, preserving properties such as low response time, high throughput, availability, and quality of results.

As shown in Table 1, there are four high-level issues that are common to all modules, all of them crucial for the scalability of the system: partitioning, dependability, communication, and external factors.

Partitioning deals with data scalability, and communication deals with processing scalability. A system is dependable if its operation is free of failures. Dependability is hence the property of a system that encompasses reliability, availability, safety, and security. The external factors are the external constraints on the system. We use these issues to subdivide each of the remaining sections.

3 Distributed Crawling

Implementing a Web crawler does not seem to be a very difficult issue as the basic procedure is simple to understand: the crawler receives a set of starting URLs as input, downloads the pages pointed to by those URLs, extracts new URLs from those pages, and continues this process recursively. In fact, many software packages (*e.g.*, `wget`) implement this functionality with a few hundred lines of code.

The operation of a large-scale Web crawler, however, may not be quite so straightforward because it consumes bandwidth and processing cycles of other systems. In fact, "running a crawler which connects to more than half a million servers (...) generates a fair amount of email and phone calls" [9]. Web crawlers can have a detrimental effect on the network if they are deployed without taking into account a set of operational guidelines to minimize their impact on Web servers [28].

The most important restriction for a Web crawler is to avoid overloading Web servers. *De facto* standards of operation state that a crawler should not open more than one connection at a time to each Web server, and should wait several seconds between repeated accesses [27]. To enable scaling to millions of servers, large-scale crawling requires distributing the load across a number of agents while still respecting these constraints.

A distributed crawler[1] is a Web crawler that operates simultaneous crawling agents [18]. Each crawling agent runs on a different computer, and in principle some agents can be on different geographical or network locations. On every crawling agent, several processes or execution threads running in parallel keep (typically) several hundred TCP connections open at the same time.

Partitioning A parallel crawling system requires a policy for assigning the URLs that are discovered by each agent, as the agent that discovers a new URL may not be the one in charge of downloading it. All crawling agents have to agree upon such a policy at the beginning of the computation.

To avoid downloading more than one page from each server simultaneously, the same agent is responsible for all the content of a set of Web servers in most distributed crawling systems. This also enables exploiting the locality of links, that is, the fact that most of the links on the Web point to other pages in the same server makes it unnecessary to transfer those URLs to a different agent.

An effective assignment function balances the load across the agents such that each crawling agent gets approximately the same work load [7]. In addition, it should be dynamic with respect to agents joining and leaving the system. Another important feature of such an assignment function is the reduction on the load of servers as we add more agents to the pool. Such a feature enables a scalable crawling system.

A trivial, but reasonable assignment policy is to use hashing to transform server names into a number that corresponds to the index of the corresponding crawling agent. Such a policy, however, does not consider the number of documents on servers. Hence, the resulting partition may not balance the load properly across crawling agents.

Dependability One of the issues with a policy for distributing the work of a crawler is how to re-distribute the work load when a crawling agent leaves the pool of agents (voluntarily or due to a failure). A solution could be re-hashing all the servers to re-distribute them to agents, although this increases message complexity for the communication among agents. The authors of [7] propose to use consistent hashing, which replicates the hashing buckets. With consistent hashing, new agents enter the crawling system without re-hashing all the server names. Under such assignment function, we guarantee that no agent downloads the same page more than once, unless a crawling agent terminates unexpectedly without informing others. In this case, it is then necessary to re-allocate the URLs of the faulty agent to others.

[1] Also named *parallel crawler* in the literature.

Web crawlers require large storage capacity to operate, and failure rates that may be negligible for individual hard disks have to be taken into account when using large storage systems. In a Web-scale data collection, disk failures will occur frequently enough and the crawling system must be tolerant to such failures.

Communication (synchronization) Crawling agents must exchange URLs, and to reduce the overhead of communication, these agents exchange them in batches, *i.e.*, several URLs at a time. Additionally, crawling agents can have as part of their input the most cited URLs in the collection. They can, for example, extract this information from a previous crawl [18]. This information enables a significant reduction on the communication complexity due to the power-law distribution of the in-degree of pages. In this way, agents do not need to exchange URLs found very frequently.

Given that an agent crawls several Web servers, it is possible to reduce communication costs even further by having all the servers assigned to the same agent topologically "close" in the Web graph and sharing many links among them.

A different issue is the communication between the crawler and the Web servers. The Web crawler is continuously *polling* for changes in the Web pages, and this is inefficient. A way around this problem is to use the HTTP `If-modified-since` header to reduce, but not to eliminate, the overhead due to this polling. This could be improved if the Web server informs the crawler of the modification dates and modification frequencies for its local pages. There have been several proposals in this direction [24, 8, 15], and recently three of the largest search engines agreed on a standard for this type of server-crawler cooperation (*http://www.sitemaps.org/*).

External factors DNS is frequently a bottleneck for the operation of a Web crawler (*e.g.*, because it has to submit a large number of DNS queries [25, 43]). This is particularly important because the crawler does not control the DNS servers it probes. A common solution is to cache DNS lookup results.

Another important consideration is that servers on the Web are often slow, and some go off-line intermittently or present other transient failures. A distributed Web crawler must be tolerant to transient failures and slow links to be able to cover the Web to a large extent [16], where coverage is the percentage of pages obtained by crawling among all pages available on the Web.

The network topology can also be a bottleneck. To solve this problem, we can carefully distribute Web crawlers across distinct geographic locations [21]. This optimization

problem has many variables, including network costs at different locations and the cost of sending data back to the search engine. In this scenario, part of the indexing (or at least, part of the parsing) should also be distributed before creating a central collection.

In practice, the Web is an open environment, in which the crawler design cannot assume that every server will comply strictly with the standards and protocols of the Web. There are many different implementations of the HTTP protocol, and some of them do not adhere to the protocol correctly (*e.g.*, they do not honor `Accept` or `Range` HTTP headers). At the same time, there are many pages on the Web for which the HTML code was either written by hand or generated by software that does not adhere to the HTML specification correctly, so it is very important that the HTML parser is tolerant to all sort of errors in the crawled pages.

4 Distributed Indexing

Indexing in IR is the process of building an *index* over a collection of documents. Typically, an *inverted index* is the reference structure for storing indexes in IR systems [2]. A vanilla implementation consists of a couple of data structures, namely the *Lexicon* and the *Posting Lists*. The former stores all the distinct terms contained in the documents. The latter is an array of lists storing term occurrences within the document collection. Each element of a list, a *posting*, contains in its minimal form the identifier of the document containing the terms. Current inverted index implementations often keep more information, such as the number of occurrences of the term in each document, the position, the context in which it appears (*e.g.*, the title, the URL, in bold). Depending on how the index is organized, it may also contain information on how to efficiently access the index (*e.g.*, skip-lists).

In principle, we could represent a collection of documents with a binary matrix ($D \times T$), where rows represent documents and columns represent terms. Each element (i, j) is "1" if the document i contains term j, and it is "0" otherwise. Under this model, building an index is thus simply equivalent to computing the transpose of the $D \times T$ matrix. The corresponding $T \times D$ matrix is then processed to build the corresponding inverted index. Due to the large number of elements in this matrix for real Web collections (millions of terms, billions of documents), this method is often not practical.

Indexing can also be considered as a "*sort*" operation on a set of records representing term occurrences. Records represent distinct occurrences of each term in each distinct document. Sorting efficiently these records using a good balance of memory and disk usage, is a very challenging operation. Recently it has been shown that *sort-based* approaches [50], or *single-pass* algorithms [32], are efficient in several scenarios, where indexing of a large amount of data is performed with limited resources.

In the case of Web Search Engines, data repositories are extremely large. Efficiently indexing on large scale distributed systems adds another level of complexity to the problem of sorting, which is challenging enough by itself. A distributed index is a distributed data structure that servers use to match queries to documents. Such a data structure potentially increases the concurrency and the scalability of the system as multiple servers are able to handle more operations in parallel. Considering the $T \times D$ matrix, a distributed index is a "*sliced*", partitioned version of this matrix. Distributed indexing is the process of building the index in parallel using the servers available in a large-scale distributed system, such as a cluster of PCs.

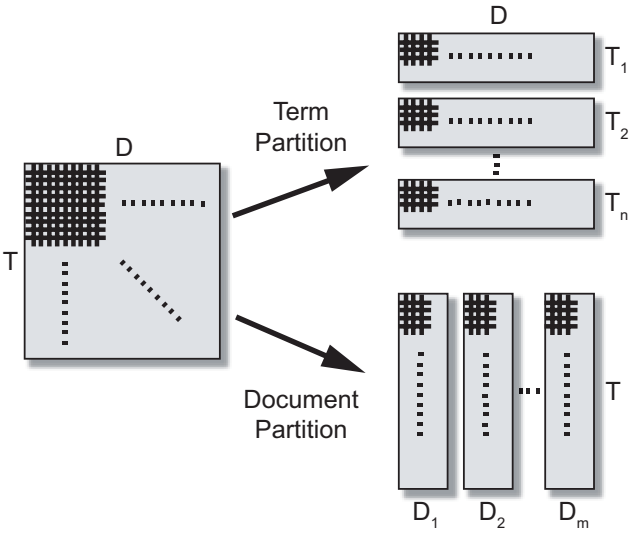

Figure 1. The two different types of partitioning of the term-document matrix.

According to the way servers partition the $T \times D$ matrix, we can have two different types of distributed indexes (Figure 1). We can perform a horizontal partitioning of the matrix. This approach, widely known as *document partitioning*, means partitioning the document collection into several smaller sub-collections, and building an inverted index on each of them.

Its counterpart approach is referred to as *term partitioning* and consists of performing a vertical partitioning of the $T \times D$ matrix. In practice, we first index the complete collection, and then partition the lexicon and the corresponding array of posting lists. The disadvantage of term partitioning is having to build initially the entire global index. This does not scale well, and it is not useful in actual large scale Web search engines. There are, however, some advantages of this approach in the query processing phase. Webber *et al.* show that term partitioning results in lower utilization of re-

sources [49]. More specifically, it significantly reduces the number of disk accesses and the volume of data exchanged. Although document partitioning is still better in terms of throughput, they show that it is possible to achieve even higher values.

The major issue for throughput, in fact, is an uneven distribution of the load across the servers. Figure 2 (originally from [49]) illustrates the average busy load for each of the 8 servers of a document partitioned system (left) and a pipelined term partitioned system (right). The dashed line in each of the two plots corresponds to the average busy load on all the servers. In the case of the term partitioned system (using a pipeline architecture), there is an evident lack of balance in the distribution on the load of the servers, which has a negative impact on the system throughput. To overcome this issue, one could try to use "smart" partitioning techniques that would take into account estimates of the index access patterns to distribute the query load evenly.

Note that load balance is an issue when servers run on homogeneous hardware. When this is the case, the capacity of the busiest server limits the total capacity of the system. If load is not evenly balanced, but the servers run on heterogeneous hardware, then load balance is not a problem if the load on each server is proportional to the speed of its hardware.

Partitioning How to partition the index is the first decision one should make when designing a distributed index. In addition to the scheme used, the partition of the index should enable efficient query routing and resolution. Moreover, being able to reduce the number of machines involved, together with the ability of balancing the load, enables an increase in the system's ability to handle a higher query workload.

A simple way of creating partitions is to select elements (terms or documents) at random. For document partitioning, a different, more structured approach is to use k-means clustering to partition a collection according to topics [30, 33].

Although document and term partitioning have been widely studied, it is still unclear on the circumstances under which each one is suitable. Moreover, it is unclear which are good methods to evaluate the quality of the partitioning. To date, the TREC evaluation framework has served this purpose. In large-scale search engines, however, the evaluation of retrieved results is difficult.

Partitioning potentially enhances the performance of a distributed search engine in terms of capacity as follows. In the case of document partitioning, instead of using all the resources available in a system to evaluate a query, we select only a subset of the machines in the search cluster that ideally contains relevant results. The selected subset would contain a portion of the relevant document set. However, the ability of retrieving the largest possible portion of relevant documents is a very challenging problem usually known as *collection selection* or *query routing*. In the case of term partitioning, effective collection selection is not a hard problem as the solution is straightforward and consists in selecting the server that holds the information on the particular terms of the query. Upon receiving a query, we will forward it only to the servers responsible for maintaining the subset of terms in the query.

The scale and complexity of Web search engines, as well as the volume of queries submitted every day by users, make query logs a critical source of information to optimize precision of results and efficiency of different parts of search engines. Features such as the query distribution, the arrival time of each query, the results that users click on, are a few possible examples of information extracted from query logs. The important question to consider now is: can we use, exploit, or transform this information to enable partitioning the document collection and routing queries more efficiently and effectively in distributed Web search engines? In the past few years, using query logs to partition the document collection and query routing has been the focus of some research projects [38, 45].

For a term partitioned IR system, the major goal is to partition the index such that:

- The number of contacted servers is minimal;
- Load is equally spread across all available servers.

For a term partitioned system, Moffat *et al*. [36] show that it is possible to balance the load by exploiting information on the frequencies of terms occurring in the queries and postings list replication. Briefly, they abstract the problem of partitioning the vocabulary in a term partitioned system as a *bin-packing problem*, where each bin represents a partition, and each term represents an object to put in the bin. Each term has a weight which is proportional to its frequency of occurrence in a query log, and the corresponding length of its posting list. This work shows that the performance of a term partitioned system benefits from this strategy since it is able to distribute the load on each server more evenly. Experimental results show that the document partitioned system achieves higher throughput than the term partitioned system, even when considering the performance benefits due to the even distribution of load. Similarly, Lucchese *et al*. [34] build upon the previous bin-packing approach by designing a weighted function for terms and partitions able to model the query load on each server. In the original bin-packing problem we simply aim at balancing the weights assigned to the bins. In this case, however, the *objective function* depends both on the single weights assigned to terms (our objects), and on the co-occurrence of terms in queries. The main goal of this function is to as-

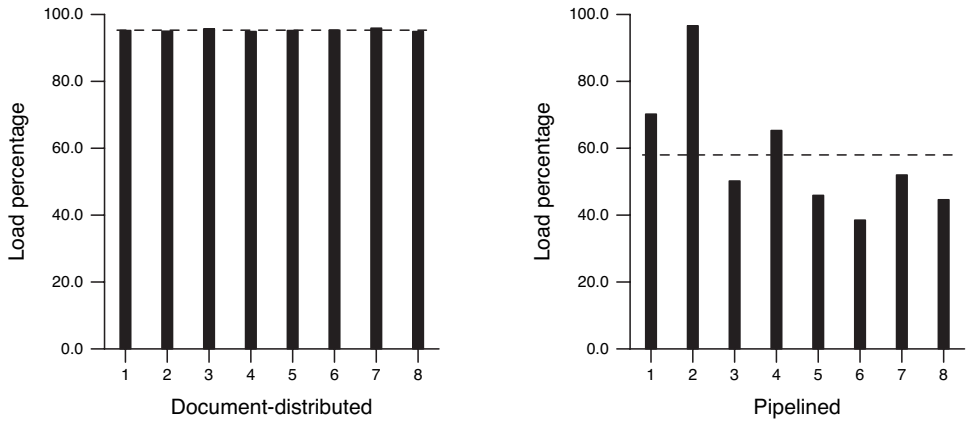

Figure 2. Distribution of the average load per processor in a document partitioned, and a pipelined term partitioned IR systems [49].

sign co-occurring terms in queries to the same index partition. This is important to reduce both the number of servers queried, and the communication overhead on each server. As this approach for partitioning IR systems requires building a central index, it is not clear how one can build a scalable system out of it.

For document partitioned systems, there has not been much work on the problem of assigning documents to partitions. The majority of the proposed approaches in the literature adopt a simple approach, where documents are randomly partitioned, and each query uses all the servers. Distributing documents randomly across servers, however, does not guarantee an even load balance [1].

The major drawback of document partitioned systems is that servers execute several operations unnecessarily when querying sub-collections, which may contain only few or no relevant documents. Thus far, there are few papers discussing how to partition a collection of documents to make each collection "well separated" from the others. Well separated, in this context, means that under such partition, each query maps to a distinct collection containing the largest number of relevant documents possible. To construct such a mapping, one can use, for example, query logs. Puppin *et al*. [38] use query logs and represent each document with all the queries that return that document as an answer. This representation enables the use of a co-clustering algorithm to build clusters of queries and clusters of documents. The result of this co-clustering step is then used to both partition the collection using document clusters, and to build a collection selection function using both query and document clusters. Their results show that using this technique, they are able to outperform the state-of-the-art model, namely CORI [14], that is currently the best known collection selection function for textual documents. CORI is only based on information contained within the collection, whereas the technique by Puppin *et al*. has the advantage of partitioning based on a model built upon usage information that is probably valid for future queries. Another interesting result of this paper is that not only query logs enable an effective partitioning of a collection, but also that it is possible to identify a subset of documents that future queries are unlikely to recall. In the paper they show that this subset comprises 53% of the documents.

This partitioning scheme, however, considers neither content nor linkage information. Thus, this research direction sounds promising, although there are still important open questions. For instance, collection selection might introduce in the document partitioning scheme the problem of load unbalance. Designing the collection partitioning algorithm not only to reduce the number of servers involved in the evaluation of a query, but also to balance the overall load is still an open issue.

To this point, we discussed issues and challenges related to the partitioning of the index. Building an index in a distributed fashion is also an interesting and challenging problem. So far, very few papers suggest approaches to build an index in a distributed fashion. For example, a possible approach to create an index in a distributed fashion is to organize the servers in a pipeline [35]. Alternatively, Dean *et al*. [20] propose a traditional parallel computing paradigm (map-reduce) and show how to build an index using a large cluster of computers. This is the area in which research has not been particularly active recently, perhaps because existing techniques already achieve good results in practice. To the best of our knowledge, testing these techniques extensively on large document collections has not been performed, and it is an important problem to verify that the results match the expectations.

Dependability Distributed indexing by itself is not a critical process. The proper functioning of a search engine, however, depends upon the existence of the index structures that enable query resolution. For example, if enough index servers fail and it is not possible to access index data to resolve a query, then the service as a whole has also failed, although other parts of the system may be working properly. Another issue with dependability is the update of the index. In systems in which it is crucial to have the latest results for queries and content changes very often, it is important to guarantee that the index data available at a given moment reflects all the changes in a timely fashion.

There are some dependability issues with partitioning schemes. In term partitioned systems if a server of the system fails, then it is impossible to recover the content of that server unless it is replicated. If this is not the case, then a possible inefficient way to recover is to rebuild the entire index. Another possibility would be to make the partitions partially overlapping such that if a server fails, at least the others will be able to answer queries. Document partitioned systems are more robust with respect to servers failures. Suppose that a server fails. The system might still be able to answer queries without using all the sub-collections, possibly without losing too much effectiveness. The dependability issues are not really well studied in the field of distributed systems for IR. An accurate analysis using real system traces may clarify these points.

Communication (synchronization) In a large search engine, there are hundreds of thousands to millions of queries a day. Logging these actions and using these query logs effectively is challenging because the volume of data is extremely high. Thus, moving this data from server to server is rarely a possibility due to bandwidth limitations.

Dealing with such a problem is important because the user model with which the engine is currently operating may not correspond to the reality. If we discover, for example, that the distribution of queries has changed, then the system should adapt to the new one and partition the index again to reflect the up-to-date distribution of queries. With respect to the index, a simple and straightforward approach is to halt a part of the index, substitute it and re-initiate. This constraint, however, is not a problem for the correct behavior of the underlying search engine, although it reduces capacity temporarily.

User model becoming inaccurate is an issue on its own. Since user behavior changes over time, one should be able to update the model accordingly. A simple approach is to schedule updates of the model at fixed time intervals. The question now is how frequently we need to update it. Recall that a higher update frequency implies a higher network traffic and a lower query processing capacity. Ideally, the system adapts to variations of the underlying model whenever they occur.

The indexing process is subject to distributed merge operations. A practical approach for achieving this goal is a primitive map-reduce [20]. Such a primitive, however, requires a significant amount of bandwidth if different sites execute operations independently. In this case, a good mechanism for such merge operations must consider both the communication and computational aspects.

In practical distributed Web search engines, indexes are usually rebuilt from scratch after each update of the underlying document collection. It might not be the case for certain special document collections, such as news articles, and blogs, where updates are so frequent that there is usually some kind of online index maintenance strategy. This dynamic index structure constrains the capacity and the response time of the system since the update operation usually requires locking the index using a mutex thus possibly jeopardizing the whole system performance. A very interesting problem is the one of understanding whether it is possible to safely lock the index without experiencing too much loss of performance. This is even more problematic in the case of term partitioned distributed IR systems. Terms that require frequent updates might be spread across different servers, thus amplifying the lockout effect.

External factors In distributed IR systems there are several bottlenecks to deal with. Depending on how one decides to partition the index there may be some serious degradation due to different factors. In a document partitioned IR system, for instance, it might be necessary to compute values for some global parameters such as the collection frequency or the inverse document frequency of a term. There are two possible approaches. One can compute the final global parameter values by aggregating all the local statistics available after the indexing phase. Often, it is possible to avoid the final aggregation. At this point, the problem of computing global statistics moves to the system broker, which is responsible for both dispatching queries to query processing servers and merging the results.

To compute such statistics, the broker usually resolves queries using a two-round protocol. In the first round the broker requests local statistics from each server, in the second round it requests results from each server, piggybacking the global statistics onto the second message containing the query. The question at this point is: given a "smart" partitioning strategy using local instead of global statistics, what is the impact on the final engine effectiveness? Answering this question is very difficult. In a real world search engine, in fact, it is difficult to define what is a correct answer for a query, thus it is difficult to understand whether using only local statistics makes a difference. A possible way to measure this effect is comparing the result set computed on the global statistics with the result set computed using only

local statistics. Furthermore, note that if we make use of a collection selection strategy, using the global statistics is not feasible.

5 Distributed Query Processing

Processing queries in a distributed fashion consists in determining which resources to allocate from a distributed system when processing a particular query. In a distributed search system, the pool of available resources comprises components having one of the following roles: coordinator, cache, or query processor. A coordinator receives queries from client computers, and makes decisions on how to route these queries to different parts of the system, where other components can evaluate it appropriately.[2] The query processors hold index or document information, which are used to retrieve and prepare the presentation of results, respectively.

Network communication is an essential part of such distributed systems as several parties need to communicate to make progress. As the variation in latency can be substantial depending on the type of network and physical proximity, involving several servers in the processing of a single query therefore might be expensive. To mitigate this problem, cache servers hold results for the most frequent or popular queries, and coordinators use cached results to reply to a client. In the case that communication is expensive, cache servers reduce query latency and load on servers by making query resolution as simple as contacting one single cache server.

An important assumption is that one or more servers implement each of these components. This assumption is particularly important for large-scale systems: systems with a high request volume and a large amount of data. Designing components in such a way that we can add more physical servers to increase the overall system capacity is fundamental for such large-scale systems as it makes the system scalable. In fact, separating parts of the system into component roles is already an attempt to promote scalability as a single monolithic system cannot scale to the necessary size of a Web search engine. As these servers can be in different physical locations and in different geographical regions, we call a *site* to each group of collocated servers.

Figure 3 depicts an instance of our multi-site distributed system model comprising the components described above. Figure 4 describes the role of each of the components in Figure 3. There are three sites located in different regions. Each site comprises a number of coordinators, caches, and query processors. A query from a client is directed to the closest site, which is site A (1). The coordinator of site A routes the query to site B (2) for reasons such as load balancing, or collection selection. Site B resolves the query on its query processors and returns the results to the client (3).

[2]The coordinators can be the brokers of a distributed document or term partitioned system, or site-level brokers that route queries to different sites. Thus, we use a more general term instead of calling them brokers.

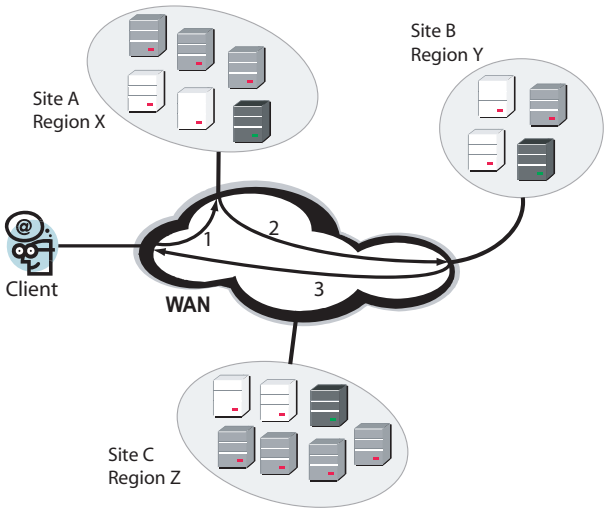

Figure 3. Instance of a distributed query processing system. The role of each server is described in Figure 4.

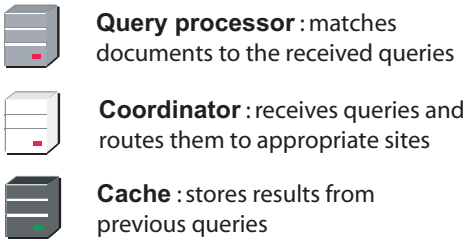

Figure 4. The role of each server in the distributed query processing framework.

We classify distributed query processing systems according to four attributes:

- **Number of components:** For each role, a system can have multiple components. Having multiple coordinators has the potential of improving the user experience by improving response time and availability. There is a similar argument for multiple cache components: they have the potential of boosting both response time and availability, reducing the server load on the query processors. Availability for caches refers not only to the failure of cache components, but also to failures of query processors. If a query processor is temporarily unavailable, then the cache participants can serve cached results during the period of the outage. Finally, multiple query processors enable a more dependable

system, due to geographical and resource diversity, as well as a more scalable solution;

- **Connectivity:** All components are either connected to the same local-area network or geographically spread and connected through a wide-area network;

- **Distinction of roles:** Components of a query processing system can have one or multiple roles. Typically, the components (coordinators, caches, query processors) implement the server side, which processes queries, whereas clients only submit queries. This implements the traditional *client/server* model, as there is a clear distinction between clients that only submit queries and servers that only process queries. Alternatively, participants can be both clients and servers such as in *peer-to-peer* (P2P) systems [5, 19, 48]. In such systems, all peers have all the roles we mention above;

- **Interaction:** In a *federated* system, independent entities form a single system. As an example, an organization spanning different countries can have independent systems that together form the whole of the organization's system. Federated systems might also comprise sites of different organizations, and some formal agreement constrains the operations of each site to a particular behavior. The interaction in the case of federations is simpler as it is reasonable to assume that there is some entity that oversees the system. Components can then trust each other, have access to all the information necessary from any other component, and assume that all other components behave in the best interest of the system. In *open* systems[3], however, this may not be the case [44]. Sites from different organizations cooperate as opposed to forming a unity and therefore may act from self-interest by, for example, changing priorities on query resolution that affect the performance of the evaluation of any particular query.

The number of components is important because it determines the amount of resources available for processing queries. Depending on how these components are connected (local-area network vs. wide-area network), the choices on the allocation of components change as different choices lead to different performance values. In fact, minimizing the amount of resources per query is in general an important goal because, in using fewer resources for each query, the total capacity of the system increases. In client/server systems, the amount of resources available on the server side determines the total capacity of the system. Thus, the total amount of resources available for processing queries does not increase with the number of clients. In peer-to-peer systems, however, any new participant is both a new client and a new server. Consequently, the total amount of resources available for processing queries increases with the number of clients, assuming that free-riding is not prevalent. On federated systems, independent systems combine to form a single system, and consequently it is not necessary to consider issues such as trust among parties and correct behavior. On open systems, partnerships have the potential to improve the overall quality of the service the system provides to clients. In such systems, however, parties may allocate resources in a self-interested fashion, thereby having a negative impact on the results a particular party obtains.

Typically, the following two architectures appear in the literature when discussing distributed information retrieval: peer-to-peer and federated systems. Peer-to-peer systems often assume a large number of components geographically spread, and each peer builds its own version of the index and is capable of resolving queries. Federated systems often work in a client/server fashion, although there is no major constraint that prevents such a system from being peer-to-peer. In such a system, however, unknown participants cannot subscribe and participate as another peer. If any participant can register to join such a system, then it becomes an open system.

We now discuss more specifically the challenges in implementing systems for distributed query processing considering the four attributes above. In this section, we focus on the behavior of system components instead of discussing specific mechanisms that implement them.

Partitioning As the Web grows, the capacity of query processors of a Web search engine has to grow as well in order to match the demand for high query throughput, and low latency. It is unlikely, however, that the growth in the size of a single query processor can match the growth of the Web, even if a large number of servers implement such a processor due to, for example, physical and administrative constraints (*e.g.*, size of a single data center, power, cooling) [3]. Thus, the distributed resolution of queries using different query processors is a viable approach as it enables a more scalable solution, but it also imposes new challenges. One such challenge is the routing of queries to appropriate query processors, in order to utilize more efficiently the available resources and provide more precise results. Factors affecting the query routing can be, for example, the geographical proximity, the topic, or the language of a query. Geographical proximity aims to reduce the network latencies and to utilize the resources closest to the user submitting the query. A possible implementation of such a feature is DNS redirection: according to the IP address of the client, the DNS service routes the query to the appropriate coordinator, usually the closest in network distance [47]. As another example, the DNS service can use geographical location to determine where to route queries to. As there

[3]Open systems are also called non-cooperative in the literature.

is fluctuation in submitted queries from a particular geographic region during a day [4], it is also possible to offload a server from a busy area by re-routing some queries to query processors in less busy areas.

Generally, query routing depends upon the distribution of documents across query processors, and consequently the partition of the index. One way to partition the index across the query processors is to consider the topics of documents [39]. For example, a query processor that holds an index with documents on a particular topic, may process queries related to that topic more effectively. Routing the queries according to their topic involves identifying the topics of both Web pages and queries. Matching queries to topics is a problem of collection selection [14, 44]. The partitions are ranked according to how likely they are to answer the query. Then, a number of top-ranked partitions can actually process the query. A challenge in such a partitioning of the index is that changes in the topic distribution of documents or queries might have a negative effect on the performance of the distributed retrieval system. As shown in [13], simulations of distributed query processing architectures indicate that changes in the topic distribution of queries can adversely impact performance, resulting in either the resources not being exploited to their full extent or allocation of fewer resources to popular topics. A possible solution to this challenge is the automatic reconfiguration of the index partition, considering information from the query logs of a search engine.

Partitioning the index according to the language of queries is also a suitable approach. Identifying the languages in a document can be performed automatically by comparing n-gram language models for each of the target languages and the document [17], or by comparing the probabilities that the most frequent words of a particular language occur in the document [23]. Similar techniques enable the identification of the languages in queries, even though the amount of text per query and additional contextual metadata is very limited, and such process may introduce errors. Another challenge in routing queries using language is the presence of multilingual Web pages. For example, Web pages describing technical content can have a number of English terms, even though the primary language is a different one. In addition, queries can be multilingual, involving terms in different languages.

As mentioned in the discussion of the previous section about the partitioning of documents or terms within a query processor, a particular partitioning scheme may introduce a workload unbalance during query processing. A partitioning scheme of documents based on the topics or the languages of documents potentially introduces a similar unbalance in workload across query processors, although provisioning the sites accordingly is a viable solution when it is possible to forecast the workload.

Dependability Faults can render a system or parts of a system unavailable. This is particularly undesirable for mission-critical systems, such as the query processing component of commercial search engines. In particular, availability is often the property that mostly affects such systems because it impacts the main source of income of such companies. In a distributed system with many components, however, we can leverage the plurality of resources to cope with faults in different ways.

To provide some evidence that achieving high availability is not necessarily straightforward, we repeat in Figure 5 a graph that originally appeared in the work by Junqueira and Marzullo [26]. This graph summarizes availability figures obtained for the sites of the BIRN Grid system [6]. BIRN had 16 sites connected to the Internet during the measurement period (January through August 2004). Each site comprises a number of servers, and we say that a site is unavailable if it is not possible to reach any of the servers of this site, either because of a network partition or because all servers have failed. Each bar on the graph corresponds to the average number of sites that had monthly availability under the corresponding value on the x-axis for the period of measurement. For example, the first bar to the left shows that out of the 16 sites participating in this system, on average 10 experience at least one outage (less than 100% of availability) in a given month, which is significant compared to the total number of sites. This illustrates that sites can be often unavailable in multi-site systems.

Although BIRN is a system with different goals, its design builds upon the idea of multiple independent sites forming a single system, which share similarities with distributed IR systems. In particular, this observation on site unavailability applies to federated IR systems comprising a number of query processors (equivalent to sites) spread across a wide-area network.

A classical way of coping with faults is replication. In a distributed information retrieval system, there are different aspects to replicate: network communication, functionality, and data. To replicate network communication, we replicate the number of links, making sites multi-homed. This redundancy on network communication reduces the probability of a partition preventing clients and servers from communicating in a client/server system. This is less critical in a peer-to-peer system with a large number of clients geographically spread because by design such systems already present significant diversity with respect to network connectivity.

There are two possible levels of replication for functionality and data: in a single site and across sites. In a single site, if either functionality or data is not replicated, then a single fault can render a service unavailable. For example, if a cluster has a single load balancer, and this load balancer crashes, then this cluster stops operating upon such a fail-

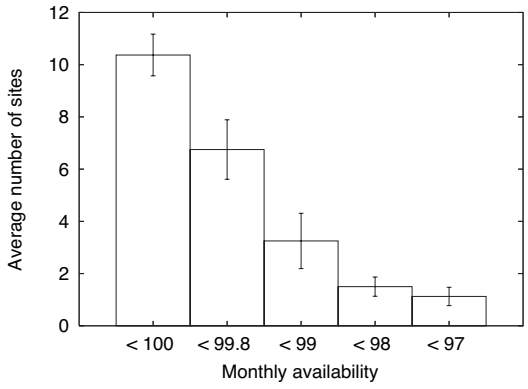

Figure 5. Site unavailability in the BIRN Grid system [26].

ure. Using multiple sites increases the likelihood that there is always some query processor available when resolving a query. This is particularly necessary when site failures are frequent. An open question in this context is how to select locations to host sites and the degree of replication necessary to meet a particular availability goal.

The observation regarding the utilization of multiple sites is valid for all the roles of participants we describe above. In particular, query processors have a crucial role as the system cannot fulfill client requests without the processing capacity and the data they store. Also, due to the large amount of data (*e.g.*, indexes, documents) they handle, it is challenging to determine good replication schemes for query processors. By replicating data across different query processors, we increase the probability that some query processor is available containing the data necessary to process a particular query. Having all query processors storing the same data, *i.e.*, being full replicas of each other, achieves the best availability level possible. This is likely to impose a significant and unnecessary overhead, also reducing the total storage capacity. Thus, an open question is how to replicate data in such a way that the system achieves adequate levels of availability with minimal storage overhead.

Although high availability is a very important goal for such online systems, it is not the only one. Consistency is also often critical. In particular, when we consider features such as personalization, every user has its own state space containing variables that indicate its preferences, and potentially upon every query there is an update to such a user state. In such cases, it is necessary to guarantee that the state is consistent in every update, and that the user state is never lost. There are techniques from distributed algorithms, such as state-machine replication [42, 29] and primary-backup [11, 52], to implement such fault-tolerant services. The main challenge is to apply such techniques on large-scale systems. Traditional fault-tolerant systems assume a small number of processes. An exception is the Chubby lock service, which serves thousands of clients concurrently and tolerates failures of Chubby servers [12]. Depending on the requirements of the application, it is also possible to relax the strong consistency constraint, and use techniques that enable stale results thus implementing weaker consistency constraints [41].

It is also possible to improve fault tolerance even further by using caches. Upon query processor failures, the system returns cached results. Thus, a system design can consider a caching system as either an alternative or complementary to replication. An important question is how to design such a cache system to be effective in coping with failures. Of course, a good design has also to consider the primary goals of a cache system, which are reducing the average response time, load on the servers operating on the query processors, and bandwidth utilization. These three goals translate into a higher hit ratio. Interestingly, a higher hit ratio potentially also improves fault tolerance. Different from the goal of reducing average latency, the availability of results to respond to a particular query is important when coping with faults. For example, a possible architecture for caching is one with several cache components communicating using messages through a wide-area network. Such an architecture does not necessarily improve query processing latency because message latency among cache components is high. In [51], Wolman *et al.* argue that cooperative Web caching does not necessarily improve the overall request latency, mainly because the wide-area communication eclipses the benefit of a larger user population. For availability, such an architecture increases the number of results that the system can use to respond to user queries, thus making the system more available. In fact, an important argument in favor of distributed cooperative caching to improve the availability of large-scale, distributed information retrieval systems is that on wide-area systems the network connectivity often depends upon providers, and routing failures happen frequently enough [37].

Communication (synchronization) Distributing the tasks of an information retrieval system enables a number of desirable features, as we have seen previously. A major drawback that arises from the distribution of tasks across a number of servers is that these servers have to communicate. Network communication can be a bottleneck as bandwidth is often a scarce resource, particularly in wide-area systems. Furthermore, the physical distance between servers also increases significantly the latency for the delivery of any particular message. While in local-area networks message latency is on the order of hundreds of microseconds, in wide-area networks, it can be as large as hundreds of milliseconds.

Mechanisms that implement a distributed information retrieval system have to consider such constraints. As a

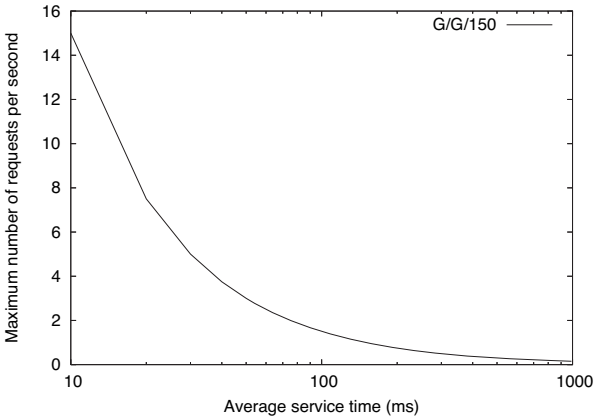

Figure 6. Maximum capacity of a front-end server using a $G/G/150$ model.

simple example, suppose we model a front-end server as a queueing system $G/G/c$ [4], where the c servers in the model correspond to the threads that serve requests on, for example, a Web server. If the response of each thread to a request depends upon the communication of this thread with other parts of the system, then bandwidth and message latency contribute to the time that the thread takes to answer such a request. Assuming that the $c = 150$ (a typical value for the maximum number of clients on Apache servers), Figure 6 shows an upper bound on the capacity of the system for different average service rate (for a given point (x, y), if x is the average service time, then the capacity has to be less than y, otherwise the service queue grows to infinity). From the graph, the maximum capacity drops sharply as the average service time of each thread increases: it drops from 15 to 2 as the average service time goes from 10*ms* to 100*ms*. This simple exercise illustrates the importance of considering the impact of network communication when designing mechanisms.

Distributed query-processing architectures then need to consider the overheads imposed by the communication and merging of information from different components of the system. A term partitioned system using pipelining routes partially resolved queries among servers [49, 36]. When position information is used for proximity or phrase search, however, the communication overhead between servers increases greatly because it includes both the position of terms and the partially resolved query. In such a case, the position information needs to be compressed efficiently, possibly encoding differently the positions of words that are likely to appear in queries.

In the case of a document partitioned system, query processors send the query results to the coordinator, which merges and detects the top ranked results to present to the user. The coordinator may become a bottleneck while merging the results from a great number of query processors. In such a case, it is possible to use a hierarchy of coordinators to mitigate this problem [13]. Furthermore, the response time in a document partitioned system depends on the response time of its slowest component. This constraint is not necessarily due to the distribution of documents, but it depends on the disk caching mechanism, the amount of memory, and the number of servers [1]. Thus, it is necessary to develop additional models that consider such characteristics of distributed query processing systems.

When multiple query processors participate in the resolution of a query, the communication latency can be significant. One way to mitigate this problem is to adopt an incremental query processing approach, where the faster query processors provide an initial set of results. Other remote query processors provide additional results with a higher latency and users continuously obtain new results. Incremental query processing has implications on the merging process of results because more relevant results may appear later due to latencies. A paradigm shift in the way that clients use search is also possible with incremental query processing [10]. As an example, we envision applications that from context infer a query and return results instead of having users directly searching using a search engine interface.

When query processing involves personalization of results, additional information from a user profile is necessary at search time, in order to adapt the search results according to the interests of the user. Query processing architectures do not consider such information as an integral part of their model [3]. An additional challenge related to personalization of Web search engines is that each user profile represents a state, which must be the latest state and be consistent across replicas. Alternatively, a system can implement personalization as a thin layer on the client-side. This last approach is attractive because it deals with privacy issues related to centrally storing information about users and their behavior. It also restricts the user to always using the same terminal.

External factors The design of search engines includes users (or clients) in different ways. For example, to evaluate the precision of a search engine, it is possible to engineer a relevance model. Similarly, the design and analysis of caching policies require information on users, or a user model [22, 31]. User behavior, however, is an external factor, which cannot be controlled by the search engine. Any substantial change in the search behavior of users can have an impact on the precision or efficiency of a search engine. For example, the topics the users search for have slowly changed in the past [46], and a reconfiguration of the search

[4] A $G/G/c$ queue models a system in which the distributions of request interarrival and service times are arbitrary, and there are c servers to serve requests [40].

engine resources might be necessary to maintain a good performance. A change in user behavior can also affect the performance of caching policies. Hence, if user behavior changes frequently enough, then it is necessary to provide mechanisms that enable either automatic reconfiguration of the system or simple replacement of modules. The challenge would then be to determine online when users change their behavior significantly.

6 Concluding Remarks

We summarize below all the main challenges outlined in the previous sections:

- **Crawling**: One of the main open problems is how to efficiently and dynamically assign the URLs to download among crawling agents, considering multiple constraints: minimize rate of requests to Web servers, locate crawling agents appropriately on the network, and exchange URLs effectively. Other open problems are how to efficiently prioritize the crawling frontier under a dynamic scenario (that is, on an evolving Web), and how to crawl the Web more efficiently with the help of Web servers.

- **Indexing**: The main open problems are: (1) For document partitioning, it is important to find an effective way of partitioning the collection that will let the query answering phase work by querying not all partitions, but only the smallest possible subset of the partitions. The chosen subset should be able to provide a high number of relevant documents; and (2) For both approaches to partitioning, determine an effective way of balancing the load among the different index servers. Both in term partitioning and in document partitioning (using query routing), it is possible to have an unbalanced load. It is very important to find a good strategy to distribute the data in order to balance the load as much as possible.

- **Querying**: Network bandwidth is a scarce resource in distributed systems, and network latency significantly increases response time for queries. Thus, when resolving queries in a distributed fashion, it is crucial to both minimize the number of necessary servers and efficiently determine which servers to contact. This problem is the one of query routing. Of course, query routing depends upon the distribution of the document collection across servers, and cannot be treated separately. Due to the intrinsic unreliability of computer systems, it is also crucial to be able to cope with faults in the system using, for example, replication. Because traditional replication techniques potentially reduce the total capacity of the system or increase the latency of any particular operation, it is not clear what schemes enable high availability and preserve the other properties. As response time and high throughput are among our goals, devising an efficient cache system is important. Although there has been extensive work on caching techniques, the difficulty for distributed Web retrieval is to design a scheme that is effective (high hit ratio) and at the same time overcomes the network constraints we point out above.

Although we have discussed crawling, indexing and query processing separately, another important consideration is the interaction among these parts. Considering this interaction makes the design of the system even more complex. A valuable tool would be an analytical model of such a system that, given parameters such as data volume and query throughput, can characterize a particular system in terms of response time, index size, hardware, network bandwidth, and maintenance cost. Such a model would enable system designers to reason about the different aspects of the system and to architect highly-efficient distributed Web retrieval systems.

Acknowledgment

We would like to thank Karen Whitehouse and Vanessa Murdock for valuable comments on preliminary versions of this paper.

References

[1] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Information Processing & Management*, 2006.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, May 1999.

[3] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, Mar./Apr. 2003.

[4] S. M. Beitzel, E. C. Jensen, A. Chowdhury, D. Grossman, and O. Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 321–328, New York, NY, USA, 2004. ACM Press.

[5] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. P2p content search: Give the Web back to the people. International Workshop on Peer-to-Peer Systems (IPTPS), February 2006.

[6] BIRN. The BIRN Grid System. `http://www.nbirn.net`, November 2006.

[7] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: a scalable fully distributed web crawler. *Software, Practice and Experience*, 34(8):711–726, 2004.

[8] O. Brandman, J. Cho, H. Garcia-Molina, and N. Shivakumar. Crawler-friendly web servers. In *Proceedings of the Workshop on Performance and Architecture of Web Servers (PAWS)*, Santa Clara, California, USA, 2000.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, April 1998.

[10] A. Z. Broder. The future of web search: From information retrieval to information supply. In *NGITS*, page 362, 2006.

[11] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Optimal primary-backup protocols. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG)*, pages 362–378, Haifa, Israel, November 1992. Springer-Verlag.

[12] M. Burrows. The chubby lock service for loosely-coupled distributed systems. to appear OSDI 2006, 2006.

[13] F. Cacheda, V. Carneiro, V. Plachouras, and I. Ounis. Performance analysis of distributed information retrieval architectures using an improved network simulation model. *Information Processing and Management*, 43(1):204–224, 2007.

[14] J. Callan. Distributed information retrieval. In W. B. Croft, editor, *Advances in Information Retrieval. Recent Research from the Center for Intelligent Information Retrieval*, volume 7 of *The Kluwer International Series on Information Retrieval*, chapter 5, pages 127–150. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[15] C. Castillo. Cooperation schemes between a web server and a web search engine. In *Proceedings of Latin American Conference on World Wide Web (LA-WEB)*, pages 212–213, Santiago, Chile, 2003. IEEE CS Press.

[16] C. Castillo. *Effective Web Crawling*. PhD thesis, University of Chile, 2004.

[17] W. B. Cavnar and J. M. Trenkle. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, US, 1994.

[18] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the eleventh international conference on World Wide Web*, pages 124–135, Honolulu, Hawaii, USA, 2002. ACM Press.

[19] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems. Technical Report 2003-75, Stanford University, 2003.

[20] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, USA, December 2004.

[21] J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino. Geographical partition for distributed web crawling. In *GIR '05: Proceedings of the 2005 workshop on Geographic information retrieval*, pages 55–60, Bremen, Germany, 2005. ACM Press.

[22] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[23] G. Grefenstette. Comparing two language identification schemes. In *Proceedings of the 3rd international conference on Statistical Analysis of Textual Data (JADT 1995)*, 1995.

[24] V. Gupta and R. H. Campbell. Internet search engine freshness by web server help. In *Proceedings of the Symposium on Internet Applications (SAINT)*, pages 113–119, San Diego, California, USA, 2001.

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

[26] F. Junqueira and K. Marzullo. Coterie availability in sites. In *Proceedings of the Internation Conference on Distributed Computing (DISC)*, number 3724 in LNCS, pages 3–17, Krakow, Poland, September 2005. Springer Verlag.

[27] M. Koster. Guidelines for robots writers. http://www.robotstxt.org/wc/guidelines.html, 1993.

[28] M. Koster. Robots in the web: threat or treat ? *ConneXions*, 9(4), April 1995.

[29] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, December 2001.

[30] L. S. Larkey, M. E. Connell, and J. Callan. Collection selection and results merging with topically organized u.s. patents and trec data. In *CIKM '00: Proceedings of the ninth international conference on Information and knowledge management*, pages 282–289, New York, NY, USA, 2000. ACM Press.

[31] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 19–28, New York, NY, USA, 2003. ACM Press.

[32] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM Press.

[33] X. Liu and W. B. Croft. Cluster-based retrieval using language models. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 186–193, New York, NY, USA, 2004. ACM Press.

[34] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. Submitted to SIAM Data Mining Conference 2007, 2007.

[35] S. Melink, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, 2001.

[36] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM Press.

[37] V. Paxson. End-to-end routing behavior in the Internet. *ACM SICOMM Computer Communication Review*, 35(5):43–56, October 2006.

[38] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *INFOSCALE 2006: Proceedings of the first International Conference on Scalable Information Systems*, 2006.

[39] K. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, pages 289–302, 2002.

[40] S. Ross. *Introduction to probability models*. Harcourt Academic Press, 2000.

[41] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.

[42] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[43] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, California, February 2002. IEEE CS Press.

[44] M. Shokouhi, J. Zobel, F. Scholer, and S. Tahaghoghi. Capturing collection size for distributed non-cooperative retrieval. In *Proceedings of the Annual ACM SIGIR Conference*, Seattle, WA, USA, August 2006. ACM Press.

[45] M. Shokouhi, J. Zobel, S. M. Tahaghoghi, and F. Scholer. Using query logs to establish vocabularies in distributed information retrieval. *Information Processing and Management*, 43(1), January 2007.

[46] A. Spink, B. J. Jansen, D. Wolfram, and T. Saracevic. From e-sex to e-commerce: Web search changes. *Computer*, 35(3):107–109, 2002.

[47] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai (travelocity-based detouring). In *Proceedings of the ACM SIGCOMM Conference*, pages 435–446, Pisa, Italy, September 2006.

[48] C. Tang, Z. Xu, and M. Mahalingam. psearch:Information retrieval in structured overlays. In *Proceedings of ACM Workshop on Hot Topics in Networks (HotNets)*, pages 89–94, Princeton, New Jersey, USA, October 2002.

[49] W. Webber, A. Moffat, J. Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 2006. published online October 5, 2006.

[50] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, May 1999.

[51] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. *ACM Operating Systems Review*, 34(5):16–31, December 1999.

[52] X. Zhang, F. Junqueira, M. Hiltunen, K. Marzullo, and R. Schlichting. Replicating non-deterministic services on grid environments. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Haifa, Israel, November 2006.