Chapter 2
Scalability Challenges in Web Search Engines

Berkant Barla Cambazoglu and Ricardo Baeza-Yates

Abstract Continuous growth of the Web and user bases forces web search engine companies to make costly investments on very large compute infrastructures. The scalability of these infrastructures requires careful performance optimizations in every major component of the search engine. Herein, we try to provide a fairly comprehensive coverage of the literature on scalability challenges in large-scale web search engines. We present the identified challenges through an architectural classification, starting from a simple single-node search system and moving towards a hypothetical multi-site web search architecture. We also discuss a number of open research problems and provide recommendations to researchers in the field.

2.1 Introduction

Large-scale search engines are the primary means to access the content in the Web. As of February 2011, the indexed Web is estimated to contain *at least* 16.3 billion pages.[1] The actual size of the Web is estimated to be much larger due to the presence of dynamically generated pages. The main duty of web search engines is to fetch this vast amount of content and store it in an efficiently searchable form. Commercial search engines are estimated to process hundreds of millions of queries daily on their index of the Web.

[1]The size of the indexed Web (visited on February 1, 2010), http://www.worldwidewebsize.com/.

B.B. Cambazoglu (✉) · R. Baeza-Yates
Yahoo! Research, Diagonal 177, p9, 08018 Barcelona, Spain
e-mail: barla@yahoo-inc.com

R. Baeza-Yates
e-mail: rbaeza@acm.org

B.B. Cambazoglu
url: http://research.yahoo.com/Berkant_Barla_Cambazoglu

R. Baeza-Yates
url: http://www.baeza.cl

M. Melucci, R. Baeza-Yates (eds.), *Advanced Topics in Information Retrieval*,
The Information Retrieval Series 33,
DOI 10.1007/978-3-642-20946-8_2,

Given the very high rate of growth in the number of web pages and the number of issued queries, the scalability of large-scale search engines becomes a real challenge. Increasing the investment on hardware in order to cope with this growth has certain financial implications for search engine companies. Therefore, designing efficient and scalable search systems is crucial to reduce the running cost of web search engines.

In this chapter, we provide a survey of the scalability challenges in designing a large-scale web search engine. We specifically focus on software design and algorithmic aspects of scalability. For a hardware perspective, interested readers may refer to (Barroso et al. 2003; Barroso and Hölzle 2009).

In Sect. 2.2, we first provide a very brief overview of the main components in a web search engine. We then, in Sect. 2.3, summarize the quality and efficiency objectives that must be considered during the design of these components. A number of parameters that affect the scalability of search engines are discussed in Sect. 2.4. In Sect. 2.5, we provide a detailed survey of research issues in search engines, going into detail only on scalability-related issues. Finally, in Sect. 2.6, we discuss several open research problems in the field.

2.2 Components

A full-fledged web search engine contains three main components: a crawler, an indexer, and a query processor. In practice, commercial search engines include many other components as well (e.g., web graph builder, spam classifier, spelling corrector). However, we prefer to omit these supplementary components herein as they are too specific to a particular purpose or they do not have a significant impact on the scalability.

Figure 2.1 illustrates the functioning of the three components mentioned above. The main duty of the crawler is to locate the pages in the Web and download their content, which is stored on disk for further processing. This processing typically involves various parsing, extraction, and classification tasks. The indexer is responsible for extracting the textual content from the stored pages and building an inverted index to facilitate processing of user queries. It also extracts various features that will be used by the query processor in relevance estimations. The query processor

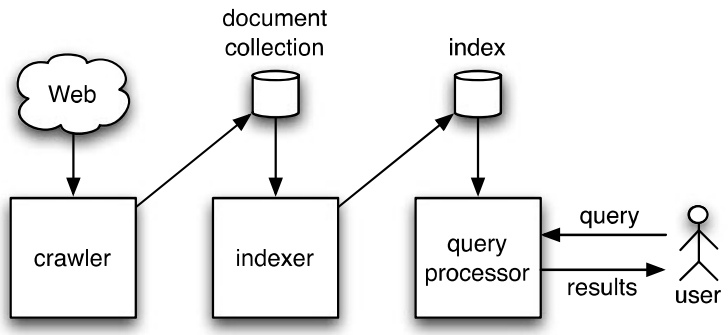


Fig. 2.1 A simplified view of the three major components in web search engines and their functioning

is responsible for receiving online user queries and evaluating them over the constructed index. It presents to the user a small set of best-matching results, typically links to ten pages selected from the indexed collection, together with their titles and short summaries (snippets). In general, the results are displayed in decreasing order of their estimated relevance to the query.

2.3 Objectives

All three components are designed to meet certain quality and efficiency objectives. Table 2.1 summarizes the main objectives we have identified. Despite their high number and variety, the ultimate goal in achieving all of these objectives is to maximize the satisfaction that users get from the service provided by the search engine.

The main quality objectives for the crawler are to achieve high web coverage (Dasgupta et al. 2007), high page freshness (Cho and Garcia-Molina 2000, 2003), and high content quality (Cho et al. 1998; Najork and Wiener 2001). The crawler aims to locate and fetch as many pages as possible from the Web. In this way, it increases the likelihood that more pages useful to users will be indexed by the search engine. In the meantime, the crawler tries to keep the pages that are already discovered as fresh as possible by selectively refetching them, as an effort towards providing pages' up-to-date versions in the Web, rather than their stale versions in the repository. Finally, the crawler tries to prioritize fetching of pages in such a way that relatively more important pages are downloaded earlier (Cho et al. 1998) or are fetched more often, keeping them more fresh compared to less important pages (Cho and Garcia-Molina 2000).

Achieving the above-mentioned quality objectives requires sustaining high page download rates, which is the most important efficiency objective for the crawler. This is simply because, as the crawler downloads pages faster, it can cope better with the growth and evolution of the Web. Hence, it can achieve a higher web coverage

Table 2.1 The main quality and efficiency objectives in crawling, indexing, and query processing

Component	Quality objectives	Efficiency objectives
Crawling	High web coverage High page freshness High content quality	High download rate
Indexing	Rich features	Short deployment cycle High compactness Fast index updates
Query processing	High precision High recall Result diversity High snippet quality	Low average response time Bounded response time

and page freshness, perhaps with some positive impact on search result quality as well (Cambazoglu et al. 2009).

The most important quality objective for the indexer is to extract a rich set of features from the downloaded document collection and potentially from other sources of information. The extracted features are used by the query processor to estimate relevance of documents to queries. Therefore, it is vital to identify a fairly high number of features that are good indicators of relevance and to precompute them as accurately as possible. In current search engines, the commonly computed features include corpus statistics, e.g., term frequencies, document lengths (Baeza-Yates and Ribeiro-Neto 2010), various link analysis metrics (Page et al. 1999), click features (Joachims 2002), and features computed by means of query log analysis and session analysis (Boldi et al. 2008).

The efficiency objectives in indexing involve reducing the length of index deployment cycles, speeding up index update operations, and creating highly compact indexes. The first objective is meaningful for search engines where the index is constructed and deployed periodically. Reducing the length of the deployment cycle is important because the search queries are continued to be evaluated over an old copy of the index until the new index is deployed. This may harm the user experience as some search results may involve stale results (Lewandowskii 2008). The second objective is meaningful for search engines where the index updates are incremental and performed on the copy of the index that is actively used in query processing. In this case, optimizing these update operations becomes important as they share the same computing resources with the query processor (Büttcher and Clarke 2005). Independently of the strategy used to construct the index, the created web index needs to be compact, i.e., the memory requirement for storing the index should be low (Zhang et al. 2008). This is because a large fraction of the index has to be cached in main memory to prevent costly disk accesses during query processing.

The success of a search engine typically correlates with the fraction of relevant results in the generated search results (i.e., precision) and the fraction of relevant results that are returned to the user in all relevant results that are available in the index (i.e., recall) (Baeza-Yates and Ribeiro-Neto 2010). Achieving high precision and high recall are the two main quality objectives for the query processor as they are highly related to user satisfaction (in practice, precision is more important as users view only the first few result pages). Another quality objective that has recently become popular is to achieve high result diversity, where the goal is to minimize the overlap in the information provided to the user and to cover as many different intents of the query as possible in the presented search results (Agrawal et al. 2009; Rafiei et al. 2010). Finally, providing descriptive summaries of documents, i.e., high quality snippets (Varadarajan and Hristidis 2006), is important as this increases the likelihood of users to click on search results (Clarke et al. 2007).

The response time to a query is determined by the query processing time and the network latency between the user and the search site. The main efficiency objective for the query processor is to evaluate the queries in a fast manner (Cambazoglu and Aykanat 2006) and return the results to the user typically under one second.

This goal is perhaps one of the most challenging among all efficiency objectives mentioned before. In addition, it has been recently shown that high query response times may have a negative impact on user satisfaction and hence the revenues of the search engine (Schurman and Brutlag 2009). Therefore, a bounded response time is a must.

2.4 Parameters

There are a number of parameters that may affect the scalability of a search engine. These can be classified as internal and external parameters. Internal parameters are those on which the search engine has at least some control. External parameters are those that cannot be controlled by the search engine. Table 2.2 gives a summary of the parameters we identified.

Among the internal parameters, the amount of hardware is the most important for achieving scalability. Increasing the hardware processing and storage capacity greatly helps the efficiency objectives in both crawling, indexing, and query processing. However, adding more machines is not a long term solution for scalability due to high depreciation and maintenance costs. Besides hardware, an important parameter for crawling is the network bandwidth. Assuming that the backend text processing system is powerful enough and hence does not form a bottleneck, the network bandwidth available to the crawler determines the page download rate. For scaling the query processing component, it is vital to increase the hit rate of the result cache (Cambazoglu et al. 2010c). This significantly helps reducing the query traffic reaching the query processor. Another important parameter is the peak query processing throughput that can be sustained by the query processor (Chowdhury and Pass 2003). This is mainly determined by the amount of available hardware and the efficiency of the data structures used by the query processor.

There are three main external parameters that affect the scalability of the crawling component: web growth (Ntoulas et al. 2004), Web change (Fetterly et al. 2004), and malicious intent (Gyöngyi and Garcia-Molina 2005b). Finding and surfacing

Table 2.2 The parameters that affect the scalability of a search engine

Component	Internal parameters	External parameters
Crawling	Amount of hardware Available network bandwidth	Rate of web growth Rate of web change Amount of malicious intent
Indexing	Amount of hardware	Amount of spam content Amount of duplicate content
Query processing	Amount of hardware Cache hit rate Peak query processing throughput	Peak query traffic

the content in the Web becomes the most important challenge as it continues to grow (Lawrence and Giles 2000). In the mean time, modifications on web pages affect freshness of indexed collections and force the crawlers to refresh previously downloaded content by refetching their potentially new versions on the Web (Cho and Garcia-Molina 2000). Moreover, malicious intent of Internet users may have a negative impact on crawling performance. This mainly involves spider traps (Heydon and Najork 1999) and link farms (Gyöngyi and Garcia-Molina 2005a), both of which lead to useless work for the crawler. Similarly, spam or duplicate pages lead to useless work for the indexer as these pages need to be identified to exclude them from the index. Otherwise, both the quality of search results and the efficiency of the query processor is negatively affected by the presence of such pages. Finally, the distribution of the query traffic is an important parameter for scaling the query processing component. If the peak query traffic exceeds the maximum query processing throughput that can be sustained by the query processor, queries are dropped without any processing or processed in degradation mode, yielding partial, less relevant results (Cambazoglu et al. 2010c).

2.5 Scalability Issues

A large body of work exists on the scalability issues in search engines (see (Arasu et al. 2001) and (Baeza-Yates et al. 2007c) for related but older surveys). Herein, we provide a detailed survey on these issues in two different dimensions. First, we classify the issues according to the component associated with them, i.e., crawling, indexing, and query processing. Second, we consider the complexity of the search architecture. More specifically, we consider four search architectures at different granularities: a single-node system, a multi-node cluster, a multi-cluster site, and a multi-site engine, listed in increasing order of design complexity.

Figure 2.2 shows our classification of scalability issues. We note that all of the issues that fall under a simpler architecture are also issues in more complex architectures, but not the other way around. For example, index partitioning, which appears under multi-node cluster, is also an issue in multi-cluster and multi-site architectures, but not in single-node systems. In Fig. 2.3, we summarize the symbols used by the figures in the rest of the paper.

2.5.1 Single-Node System

This is a very simple architecture in which a single computer is dedicated to each component (Fig. 2.4). In this minimalist approach, the Web is crawled by a single computer (potentially, through multiple crawling threads (Heydon and Najork 1999)), and a common document repository is formed. This repository is converted into an index by another computer that runs an indexer (Zobel and Moffat 2006).

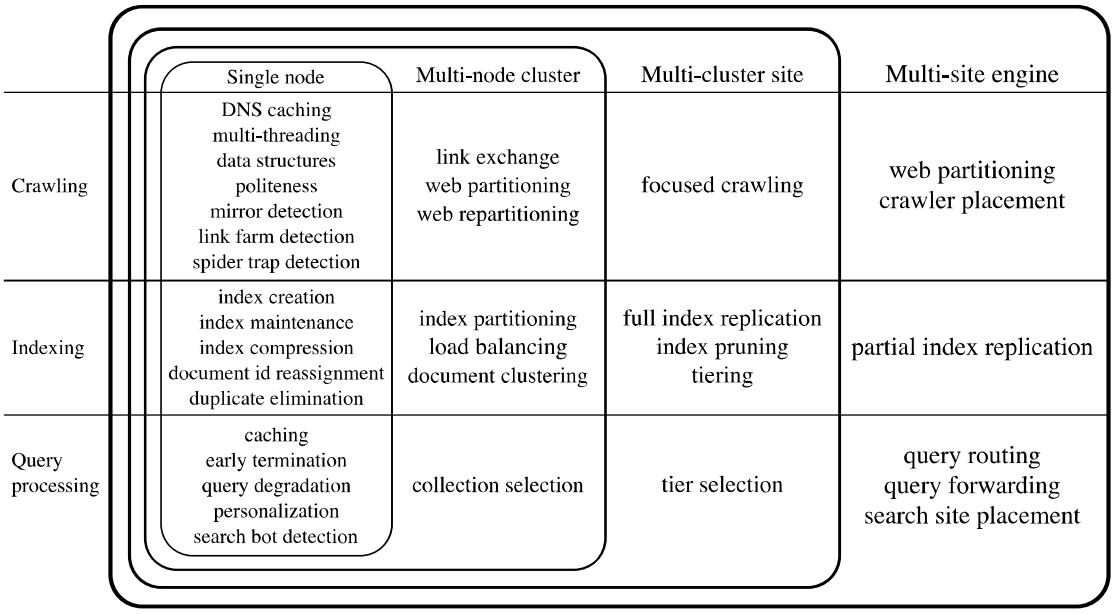


Fig. 2.2 Issues that have an impact on scalability

Fig. 2.3 The legend for the symbols used in the rest of the figures

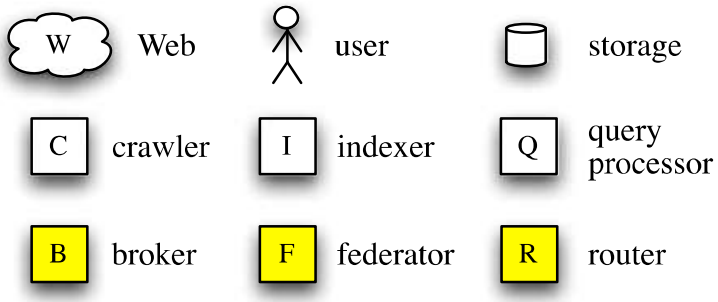
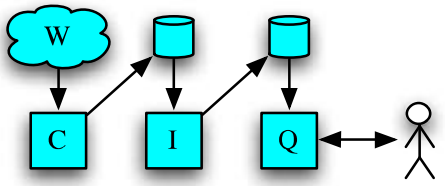


Fig. 2.4 A search architecture made up of a crawler, an indexer and a query processor, each running on a single computer



The query processor evaluates a query sequentially over the constructed web index (Cambazoglu and Aykanat 2006). In this architecture, users issue their queries directly to the query processing node and receive the search results from this node.

2.5.1.1 Single-Node Crawling

A standard sequential crawler works as follows. It starts with a given set of seed URLs and iteratively fetches these URLs from the Web by establishing HTTP connections with their web servers. Downloaded pages are stored in a repository (see

(Hirai et al. 2000) for repository management issues). In the mean time, they are parsed and the extracted new links are added into the crawler's frontier, i.e., the set of URLs that are discovered but not yet downloaded. Typically, the frontier is implemented as a queue. Hash values of discovered URLs are stored in a large hash table so that the existence of a URL in the repository or the frontier can be easily checked. Different techniques for efficient in-memory caching of seen URLs is discussed by Broder et al. (2003b). The crawler also maintains a repository of the encountered `robots.txt` files and a cache for DNS entries of hosts. DNS caching and multi-threading are vital to reduce the overhead of DNS resolution, which needs to be performed at every HTTP request (Heydon and Najork 1999).

For the single-node crawling scenario, the network bandwidth is hardly a bottleneck. Therefore, the crawler makes use of multi-threading, i.e., Web pages are fetched and processed by many task-wise identical threads, each handling a different HTTP connection (Heydon and Najork 1999). The number of threads can be increased until the available download bandwidth saturates or the overhead of context switches in the operating system beats the purpose of multi-threading. Under multi-threading, disk accesses and memory form the main bottlenecks in crawling. Hence, efficient implementation of data structures used by the crawler is crucial. A very good discussion of data structure issues, together with a comparison of previous implementations can be found in (Lee et al. 2008).

Another important issue is to decide on the download order of URLs. Older techniques prioritize the URLs in the frontier according to a quality metric (e.g., the linkage between pages) (Cho et al. 1998; Najork and Wiener 2001) and recrawl the stored content based on the likelihood of modification (Cho and Garcia-Molina 2003; Edwards et al. 2001). More recent approaches prioritize URLs (Pandey and Olston 2008) and refresh the content (Fetterly et al. 2009; Pandey and Olston 2005; Wolf et al. 2002) to directly increase the benefit to search result quality. In this impact-based crawling approach, the 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 the sites that contribute more pages to search results are crawled deeper or more often. A recent work also takes into account the information longevity of pages (Olston and Pandey 2008).

Ideally, a crawler is expected not to overload the same web server (or even subnetworks) with repetitive connection requests in a short time span, i.e., it should be polite in its interaction with web servers (Eichmann 1995). In practice, web crawlers limit the number of concurrent connections they establish to a server or set an upper-bound on the time between two successive requests in order to avoid complaints from web site owners or even long-term blocking of the crawler. As noted by Lee et al. (2008), which suggests using per-server time delays instead of per-host delays, achieving politeness may have an important impact on the performance of a crawler due to complications in data structures.

A number of obstacles arise for crawlers due to malicious intent of web site owners. An example of malicious intent is the delay attacks, where web servers introduce unnecessary delays in their responses to requests of the crawler. Perhaps, the most important hazard is spider traps, which try to keep the crawler busy by dynamically

creating infinitely many useless links (Heydon and Najork 1999). A crawling thread caught in a spider trap can significantly degrade utilization of system resources. Another important hazard is link farms (Gyöngyi and Garcia-Molina 2005a), which are composed of spam sites linking to each other with the hope of increasing the popularity values assigned to them by search engines for ranking purposes. Although the main objective of link farms is search engine optimization by means of link spam, they also harm crawlers since a crawler may allocate a significant portion of its resources to download these spam pages. A well known technique to detect link spam, TrustRank, is proposed by Gyöngyi et al. (2004). Mirror sites form a similar threat even though they are not maliciously created (Bharat and Broder 1999; Bharat et al. 2000; Cho et al. 2000). Early detection of link farms and mirror sites is important for a good utilization of the network and computational resources of the crawler. A good discussion of other hazards is available in Heydon and Najork (1999).

2.5.1.2 Single-Node Indexing

A sequential indexer is responsible for creating an efficiently searchable representation of the collection. To this end, each document in the crawled collection is passed through a preprocessing pipeline, composed of a number of software modules. At each module in the pipeline, various tasks are performed. Typical tasks involve HTML parsing, link extraction, text extraction, spam filtering, soft 404 detection, text classification, entity recognition, and feature extraction.

After preprocessing, each document is represented by the terms appearing in it and also by some additional terms (e.g., the anchor text extracted from the links pointing to the document). The indexing component takes these cleansed forms of documents and converts them into an inverted index (Harman et al. 1992; Zobel and Moffat 2006), which consists of an inverted list for each term in the vocabulary of the collection and an index that keeps the starts of these lists. The inverted list of a term keeps a set of postings that represent the documents containing the term. A posting stores useful information, such as a document id and the frequency of the term in the document. This information is later used by the query processor in estimating relevance scores. In addition, for each term-document pair, a position list is maintained to record the positions that the term appears in the document. The position information is needed for computing relevance scores based on proximity of query terms in documents (Büttcher et al. 2006b; Rasolofo and Savoy 2003). Similarly, for each term in the document, the section information, i.e., different parts of the document that the term appears (e.g., title, header, body, footer), is also stored as this is used by the ranking system.

In principle, creating an inverted index is similar to computing the transpose of a sparse matrix. There are different approaches in research literature for index creation (Fox and Lee 1991; Harman and Candela 1990; Moffat and Bell 1995; Witten et al. 1999). One possible indexing strategy is based on in-memory inversion (Fox and Lee 1991; Moffat and Bell 1995). This strategy requires making two passes over the

document collection. In the first phase, the storage requirement of every inverted list is determined and a skeleton is created for the index. In the second phase, the actual index content is computed in-memory. It is also possible to keep and update the skeleton on the disk. For large collections, making two passes over the collection is too costly and a one-phase indexing approach performs better (Heinz and Zobel 2003; Moffat and Bell 1995). In this approach, parts of the index are computed and flushed on the disk periodically as the memory becomes full. A disk-based merge algorithm is then used to combine these indexes into a single index.

To keep the index fresh, modifications over the document collection, i.e., document insertions, deletions, and updates, should be reflected to the index. In general, there are three ways to maintain an inverted index fresh (Lester et al. 2004). The simplest option is to rebuild the index from scratch, periodically, using one of the techniques mentioned above. This technique ignores all modifications over the collection and is the preferred technique for mostly static collections. Another option is to incrementally update the current index as new documents arrive (Cutting and Pedersen 1990; Shieh and Chung 2005). It is also possible to accumulate the updates in memory and perform them in batches (Clarke et al. 1994; Tomasic et al. 1994). Incremental indexing requires keeping additional unused space in inverted lists and the efficiency of update operations is important. This is the technique preferred in indexing time-sensitive collections (e.g., a crawl of news sites). A final option is to grow a separate index and periodically merge this index into the main index (Lester et al. 2008). A hybrid strategy that combines incremental updates with merging is proposed by Büttcher et al. (2006a).

For efficiency reasons, during query processing, the inverted index is tried to be kept in the main memory in a compressed form. When processing queries over the index, the relevant portions of the index are decompressed and used. The objective in index compression (Anh and Moffat 2004, 2006a; Moffat and Stuiver 2000; Scholer et al. 2002; Shieh et al. 2003) is to create the most compact index, with the least decompression overhead. In the research literature, different techniques are available for compressing different parts of the inverted index (e.g., document ids (Zhang et al. 2008), term frequencies (Yan et al. 2009b), and term positions (Yan et al. 2009a)). For an overview of inverted index compression techniques (Witten et al. 1999) can be referred to. An experimental evaluation of recent index compression techniques can be found in (Zhang et al. 2008).

The compressibility of document ids can be improved by reassigning document ids in such a way that the gaps between the ids are reduced (Blandford and Blelloch 2002; Blanco and Barreiro 2006; Silvestri 2007; Silvestri et al. 2004). In (Yan et al. 2009b), new algorithms are proposed for compressing term frequencies, under reassigned document ids. A scalable document id reassignment technique is given in (Ding et al. 2010).

A final issue in indexing is to eliminate exact duplicates and near duplicate documents. This helps reducing the index size and saves computing power. Detection of exact duplicates is an efficient and accurate task as comparison of document hashes is sufficient to identify duplicate documents. Near duplicate detection (Chowdhury et al. 2002; Cooper et al. 2002; Henzinger 2006), on the other hand, is a relatively

more costly and inaccurate task as it typically involves computing a number of shingles (Broder et al. 1997) for every document and their comparison.

2.5.1.3 Single-Node Query Processing

After a user query is issued to the search node, it goes through a series of preprocessing steps. These steps involve basic cleansing procedures, such as case-folding, stemming and stopword elimination, as well as more complicated procedures, such as spell correction, phrase recognition, entity recognition, localization, query intent analysis, and various other processing techniques. After these steps, the user query is transformed into an internal representation, which may be substantially different from the original user query.

The internal representation of the query is looked up in a result cache to determine if the related search results are already computed for a previous submission of the query. In case of a hit, the results are immediately served by the cache. In the research literature, both static (Baeza-Yates and Saint-Jean 2003; Baeza-Yates et al. 2007b; Markatos 2001; Ozcan et al. 2008) and dynamic (Baeza-Yates et al. 2007b; Markatos 2001; Saraiva et al. 2001) caching of query results is considered. Static and dynamic caching is combined in (Fagni et al. 2006). So far, most works have focused on admission (Baeza-Yates et al. 2007a), eviction (Markatos 2001; Saraiva et al. 2001), and prefetching (Lempel and Moran 2003) policies trying to increase the hit rates of result caches. Some works proposed strategies that take into account the predicted processing costs of queries when making caching and eviction decisions (Altingovde et al. 2009; Gan and Suel 2009). A novel incremental result caching technique is proposed by Puppin et al. (2010). Two recent works argue that the main issue in current result cache architectures is freshness (Blanco et al. 2010; Cambazoglu et al. 2010c). In (Blanco et al. 2010), to achieve freshness, cache entries are invalidated as the index is updated due to modifications on the document collection. In (Cambazoglu et al. 2010c), the result cache is tried to be kept fresh by expiring cache entries via a simple time-to-live mechanism and proactively refreshing the entries during the idle cycles of the backend search system.

Queries whose results are not found in the cache are processed over the index. To speed up processing of queries, various optimization techniques are employed. An important technique is posting list caching, where most frequently or recently accessed posting lists are tried to be kept in the main memory (Baeza-Yates et al. 2007b; Jónsson et al. 1998; Tomasic and Garcia-Molina 1993). In (Zhang et al. 2008), the impact of inverted index compression on the performance of posting list caches is studied. The idea of caching intersections of frequently accessed posting lists and using these intersections for efficient query processing is proposed by Long and Suel (2005).

Perhaps, one of the most important optimization techniques in query processing is early termination, where processing of a query is tried to be terminated before all postings that are associated with the query are traversed. A large number of early termination optimizations are proposed for query processing algorithms that process

queries in term order (Anh et al. 2001; Anh and Moffat 2006b; Buckley and Lewit 1985; Harman and Candela 1990; Moffat and Zobel 1996; Persin 1994; Wong and Lee 1993) or document order (Brown 1995; Broder et al. 2003a; Strohman et al. 2005; Turtle and Flood 1995). Some of these algorithms guarantee the correctness of results with respect to evaluation over the full index while some others do not. Interestingly, most algorithms proposed so far assume disjunctive mode of query processing, whereas search engines typically process their queries in conjunctive mode.

There are also other works that propose similar optimizations for term-proximity-aware scoring computations (Schenkel et al. 2007; Tonellotto et al. 2010). In large-scale search engines, a selected set of documents that are top-ranked over the index are further processed via more complex ranking techniques, such as machine-learned ranking. A recent work proposes early termination techniques for such an architecture (Cambazoglu et al. 2010a).

Another important issue is query degradation (Cambazoglu et al. 2010c). Due to tight constraints on response times, queries are processed with certain budgets. If the query processing time exceeds the allowed budget (e.g., because the query requires processing too many postings or a maximum processing time has been reached), the computation may need to be terminated and the search results computed so far are returned to the user. Under a heavy query traffic that exceeds the capacity of the query processor, queries are processed in query degradation mode, in which case search results are only partially computed. A related issue here is to detect and filter the traffic generated by search bots (Yu et al. 2010).

Most commercial search engines employ some form of personalization (Liu et al. 2002; Pitkow et al. 2002; Tan et al. 2006; Teevan et al. 2005; Sun et al. 2005). The objective in personalization is to augment the general search results to suit individual users' tastes. Although the main impact of personalization is on search quality, it can also affect the performance of the search system. This is because personalization of search results imply a huge decrease in result cache hit rates. Consequently, the query traffic processed over the index significantly increases, hindering the scalability. A side issue is efficient computation of personalized page importance vectors (Jeh and Widom 2003).

Once the documents best matching the query are determined, their snippets are generated, typically using the terms adjacent in the document to the query terms. These snippets are presented to the user together with links to the documents. An efficient snippet generation technique is proposed by Turpin et al. (2007).

2.5.2 Multi-Node Cluster

Obviously, it is not possible to achieve any of the objectives mentioned in Sect. 2.3 by means of a simple, low-cost search system, such as the one described in Sect. 2.5.1. Assuming more financial investment is made, execution of the three

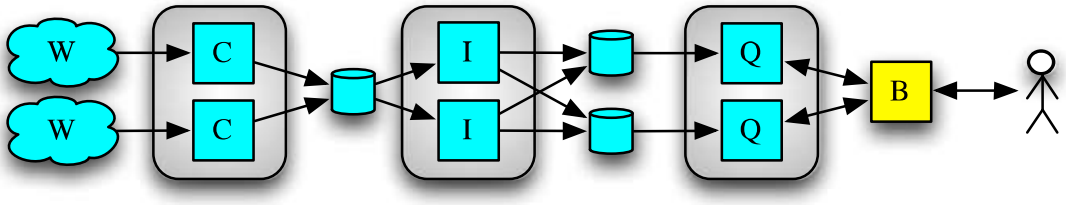


Fig. 2.5 An architecture with a separate multi-node cluster for crawling, indexing, and query processing

components can be parallelized over a multi-node cluster (Fig. 2.5), typically composed of off-the-shelf commodity computers (Barroso et al. 2003). In this architecture, parallel crawling offers increased page download rates as web pages can be concurrently crawled by many computers (assuming the network bandwidth is not saturated). In the mean time, index creation and deployment becomes a much faster process due to the availability of many indexing computers. Finally, query response times are reduced as the query processing task is parallelized over multiple computers. In this multi-node cluster architecture, a user query is issued to a broker node, which dispatches the query to the search nodes. This node is also responsible for merging the results retrieved from the search nodes and returning a final result set to the user.

2.5.2.1 Multi-Node Crawling

Given a robust crawler that runs on a single node, it is relatively easy to build a parallel crawler that runs on multiple nodes (Boldi et al. 2004; Heydon and Najork 1999; Shkapenyuk and Suel 2002; Zeinalipour-Yazti and Dikaiakos 2002). In practice, crawlers on different cluster nodes can run in a pretty much embarrassingly parallel mode and can crawl the Web independently. However, this approach results in high redundancy as the same web pages may be downloaded by different crawling nodes. A simple yet effective solution is to partition the Web across different crawling nodes. This approach guarantees that each page will be downloaded once, but another problem remains: because each crawling node has access to only its local data structures, it cannot decide, without coordinating with the other nodes, whether a newly found link is already fetched or not. In addition, having only local information implies that the crawling order cannot be optimized in a global manner.

In (Cho and Garcia-Molina 2002), three possible solutions are proposed to the above-mentioned coordination problem. The first solution is a naïve one, which simply ignores the newly discovered link if the current node is not responsible for fetching it. Obviously, this approach reduces the coverage of the crawler as some pages are never discovered. The second solution delays the download of non-local links until all local links are exhausted. This approach achieves full coverage, but the redundant crawling problem remains as the same page may be fetched by different nodes. The third and best solution is to communicate the discovered non-local links

to the nodes that are responsible for fetching them. Even though the total communication volume is not very high, communicating individual links right after they are discovered incurs a significant overhead on the system as link exchanges require traversing the entire TCP/IP stack at the sender and receiver nodes (Cho and Garcia-Molina 2002). As a solution, links can be accumulated an exchanged in batch mode. Delaying exchange of links is shown not to cause any loss in the quality of downloaded collections (Cho and Garcia-Molina 2002).

Partitioning of the Web, i.e., assignment of URLs to crawling nodes, can be performed in a straightforward manner by assigning fixed ranges of hash values to the nodes (Heydon and Najork 1999). Given a uniform hashing function, download workloads of nodes are expected to be balanced. In practice, it is better to assign to crawling nodes the entire web sites rather than individual web pages. This helps reducing the overhead of link exchanges. More importantly, this enables better politeness mechanisms as the load incurred on a web server can be tracked within the same crawling node. Based on a similar observation, in (Chung and Clarke 2002), a topical assignment approach is taken to improve duplicate content detection and page quality computations.

If the URL assignment depends on the structure of the Web (e.g., the graph partitioning approach in (Cambazoglu et al. 2004)), the URL space may need to be repartitioned among the crawling nodes. In this case, the URL assignment is periodically recomputed to maintain a load balance across crawling nodes. Achieving fault tolerance is another reason for reassigning URLs to crawling nodes. As an example, the URL space of a failed crawling node may have to be partitioned among the active nodes or a subset of existing URLs may have to be assigned to a newly added crawling node. In (Boldi et al. 2004), consistent hashing is proposed as a feasible solution to the URL reassignment problem.

2.5.2.2 Multi-Node Indexing

Since the created index will be stored in a multi-node search cluster, it is partitioned into a number of disjoint subindexes. In general, an inverted index can be partitioned based on the documents or terms. In document-based partitioning, each subindex contains the postings of a disjoint set of documents. This technique is also referred to as local index partitioning because each subindex can be locally built by a separate indexing node. In this technique, the assignment of documents to subindexes is typically obtained by creating a mapping between hashes of document ids and indexing nodes. Document-based partitioning leads to well-balanced partitions in terms of storage. However, it requires collecting and replicating on all nodes the global collection statistics (Melnik et al. 2001).

In term-based partitioning, which is also referred to as global index partitioning, each part contains a set of posting lists associated with a disjoint set of terms. Due to the large variation in sizes of posting lists and their access frequencies, the main goal is to achieve a reasonable load balance during parallel query processing. In

the research literature, there are works focusing on load balancing in term-based-partitioned indexes, by taking into account sizes and access patterns of posting lists (Jeong and Omiecinski 1995; Lucchese et al. 2007).

Both term- and document-based indexes can be created by means of a parallel indexing system. In (Melnik et al. 2001), parallel index creation and statistics gathering strategies are proposed to create a document-based-partitioned index. A few papers discuss strategies for creation of term-based-partitioned indexes in parallel (Ribeiro-Neto et al. 1998, 1999). Because of the time complexity of indexing and updating a global term-based-partitioned index, commercial search engines use document partitioning and some of them rely on the MapReduce framework (Dean and Ghemawat 2008) to parallelize the indexing process.

In contrast to the aforementioned non-topical partitioning approaches, it is also possible to create subindexes specific to certain topics by clustering documents and assigning each document cluster to a different search node. As before, each subindex acts as a separate entity that can be individually queried. In the literature, most approaches adopt the k-means clustering algorithm to cluster documents (Kulkarni and Callan 2010; Larkey et al. 2000; Liu and Croft 2004). In (Kulkarni and Callan 2010), three document clustering algorithms are evaluated, and k-means clustering is found to be superior to random and URL-based clustering, in terms of both search efficiency and effectiveness. A novel query-driven clustering technique is presented by Puppin et al. (2010). In this technique, a document is represented by the terms appearing in queries that have requested the document in the past. The co-clustering of queries and documents is then used to select the appropriate collections for queries.

2.5.2.3 Multi-Node Query Processing

Query processing in a multi-node search cluster depends on the way the index is partitioned. In case of document-based partitioning (Cahoon et al. 2000; Hawking 1997), the broker issues the query to all search nodes in the cluster. Results are concurrently computed and returned to the broker. Assuming that the number of results requested by the user is k, it suffices to receive only the top k results from each node. As global collection statistics are made available to all search nodes, the relevance scores assigned to the documents are compatible. Hence, merging of the results at the broker is a trivial operation. In (Badue et al. 2007), it is shown that even though document-based partitioning leads to well-balanced partitions, processing times of a query on different nodes may be imbalanced due to the effect of disk caching.

In case of term-based partitioning (Lucchese et al. 2007; Moffat et al. 2007), the query is issued to only the search nodes that contain the posting lists associated with query terms. The contacted nodes compute result sets, where document scores are partial, by using their posting lists that are related to the query. These result sets are then transferred to the broker, which merges them into a global result set. In this case, entire result sets have to be transferred to the broker. A novel pipelined query processing technique is proposed by Moffat et al. (2007). In this technique, a query

is sequentially processed over the nodes with intermediate results being pipelined between them. Unfortunately, the performance remained below that of document-based partitioning.

There are a high number of works that analyze the performance of distributed query processing systems (Cacheda et al. 2007; Chowdhury and Pass 2003) as well as works that compare query processing techniques on document-based- and term-based-partitioned indexes via simulations (Ribeiro-Neto and Barbosa 1998; Tomasic and Garcia-Molina 1993) and experimentation on real systems (Badue et al. 2001; MacFarlane et al. 2000). Although these studies are not very conclusive, in practice, document-based partitioning is superior to term-based partitioning as it achieves better load balancing and lower query processing times, despite the fact that term-based partitioning provides higher query throughput. Indeed, search engines are known to employ a document-based partitioning strategy as it also provides better fault tolerance in case of node failures (Barroso et al. 2003).

If documents are partitioned over the search nodes via topical document clustering, collection selection techniques are used to decide on the nodes where the query will be processed. In collection selection, the broker node maintains summaries of collections in each search node and issues a query to search nodes selectively, according to the relevance of collections to the query. The objective is to reduce the workload of search nodes by querying as few nodes (collections) as possible, without significantly degrading the search quality. There is a vast body of research on collection selection (Callan et al. 1995b; D'Souza et al. 2004; Gravano and Garcia-Molina 1995; de Kretser et al. 1998; Puppin et al. 2010; Si et al. 2002a; Tomasic et al. 1997; Xu and Callan 1998; Xu and Croft 1999; Yuwono and Lee 1997). A popular collection selection algorithm, based on inference networks, is presented by Callan et al. (1995b). This work is later adapted to handle partial replicas of the index (Lu and McKinley 1999) and compared with result caching (Lu and McKinley 2000). In a recent work (Puppin et al. 2010), load balancing techniques are proposed for collection selection on a search cluster.

2.5.3 Multi-Cluster Site

A financially more costly strategy is to build multiple clusters within a search site (Brin and Page 1998) to run many instances of the three components (Fig. 2.6). In fact, the issues discussed under this architecture, such as focused crawling, tiering, and index pruning, could have been discussed under the heading of the previous two architectures. However, we prefer to discuss these issues here as this better reflects what is done in practice. From the performance point of view, for the crawling and indexing components, there is no significant difference between constructing C clusters each with K computers or a single cluster with $C \times K$ computers because both crawling and indexing nodes work in an embarrassingly parallel fashion. However, for practical purposes, different crawling clusters are constructed. Each crawling cluster is specialized in fetching a certain type of content (e.g., news pages). For

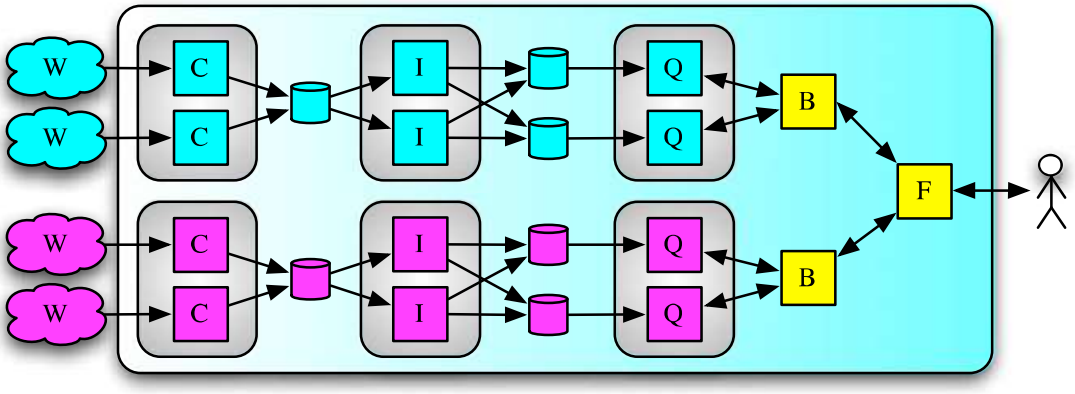


Fig. 2.6 A search architecture with a single site containing multiple clusters, specialized on different crawling, indexing, and query processing tasks

every crawled collection, an indexing cluster builds a corresponding index. These indexes are served by separate search clusters, each with its own broker. Queries are issued to a federator, which blends the results retrieved from different brokers and returns the final results to users.

2.5.3.1 Multi-Cluster Crawling

In practice, a search site contains multiple crawling clusters of varying sizes. Typically, there is a very large cluster that crawls the entire Web and there are other small-scale clusters that run focused web crawlers to harvest the Web selectively according to a selected theme (e.g., news pages, blogs, images, academic papers). Focused web crawling allows accessing high-quality content faster and is especially important for vertical search engines (Chakrabarti et al. 1999; Diligenti et al. 2000). As mentioned before, if the target is to crawl the entire Web, however, there is no significant difference between building a large crawling cluster or many small-sized clusters in terms of crawling performance.

2.5.3.2 Multi-Cluster Indexing

In addition to creating replicas of the same index on multiple search clusters, index pruning and tiering techniques can be employed to reduce the workload incurred by queries. In index pruning, the objective is to create a small web index containing the postings of documents that are more likely to appear in future search results and process the queries only over this index (Carmel et al. 2001). It is also possible to create a two-level index architecture, where the queries are first processed over the pruned index and optionally on the full index (Ntoulas and Cho 2007).

In principle, tiering is similar to pruning. In case of tiering (Risvik et al. 2003), however, the index is disjointly partitioned into multiple tiers based on the quality of documents, i.e., the full web index is not maintained in a single cluster. Each

subindex may be stored on a separate search cluster, which forms a tier. Tiers are ordered in increasing order of average document quality and hit by queries in this order.

2.5.3.3 Multi-Cluster Query Processing

In contrast to crawling and indexing, constructing multiple copies of the same query processing cluster and replicating the same index on these clusters brings performance benefits. More specifically, the peak query processing throughput that can be sustained by the query processor is increased (Barroso et al. 2003). We note that the optimum query processing throughput is achieved with a certain C and K value pair, which depends on the size of the document collection and hardware parameters.

A two-tier index pruning architecture that guarantees correctness of results (relative to the results obtained over the full index) is proposed by Ntoulas and Cho (2007). In this architecture, a pruned index is placed in a separate search cluster in front of the main cluster that contains the full web index.

Queries are first processed over the pruned index in the small cluster. If obtained results are found to be unsatisfactory based on a certain criterion, the query is processed over the full web index in the main search cluster. This approach is shown to result in significant savings in query processing workload. The impact of result caching on the above-mentioned architecture is investigated by Skobeltsyn et al. (2008), showing that pruning is not really effective.

In case of tiering, a query is sequentially processed over the tiers, in increasing quality order of the tiers (Risvik et al. 2003). The challenge in tiering is to identify the optimum number of tiers to be hit in query processing. Naïve approaches may decide on the stopping condition based on the number of retrieved results from tiers (Risvik et al. 2003). For example, if a sufficiently large result set is obtained, further tiers may not be hit. More complicated approaches employ machine learning strategies to predict to which tiers a query should be submitted and then increase the parallelism of the system and decrease the response time (Baeza-Yates et al. 2009b). Similar to static index pruning, the main benefit of tiering is the reduced workload.

2.5.4 Multi-Site Engine

The final and most sophisticated architecture we now discuss distributes all three components over multiple, geographically distant sites (Baeza-Yates et al. 2009a; Cambazoglu et al. 2009) (Fig. 2.7). In this architecture, sites crawl the web pages in their geographical neighborhood. Indexes are built over local document collections and certain documents are replicated on multiple sites. Each user query is routed to an initial local search site, based on geographical proximity. Queries are selectively processed on a subset of search sites via effective query forwarding mechanisms.

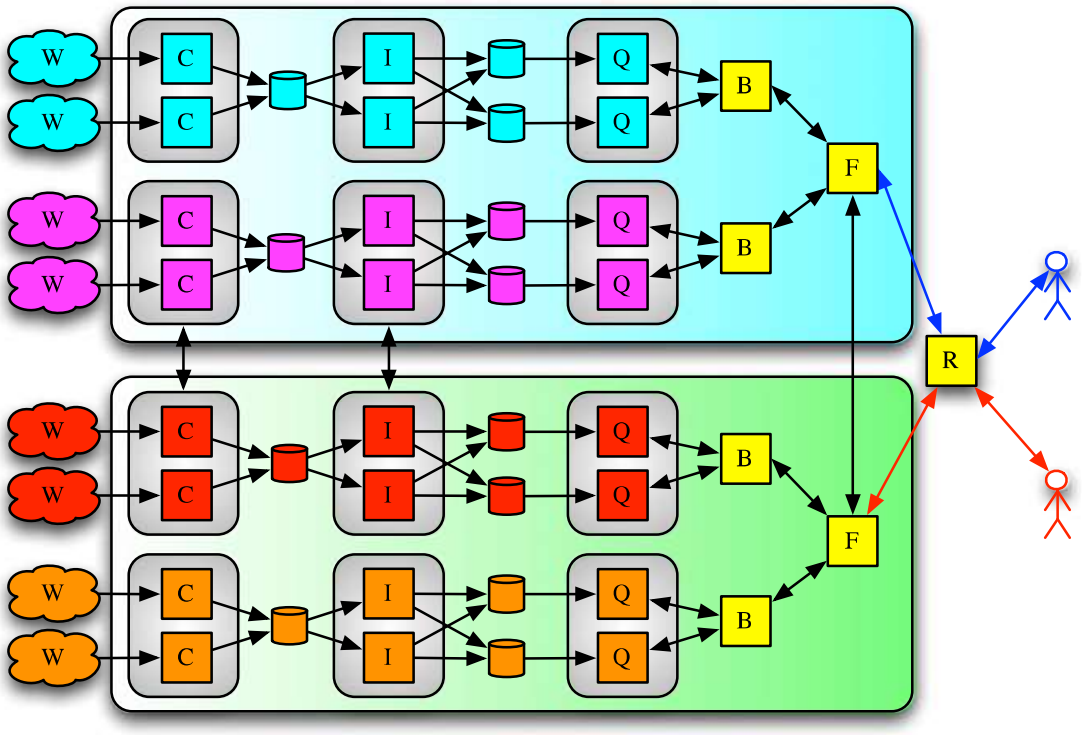


Fig. 2.7 A multi-site search engine with geographically distributed crawling, indexing, and query processing

We note that the techniques discussed herein are taken from the research literature. They do not necessarily exist in current multi-site web search engines. However, commercial search engines are known to span multiple sites, where the full web index and search clusters are replicated.

2.5.4.1 Multi-Site Crawling

In most search engines, crawling is performed by a single site. However, a single-site crawler suffers from the standard scalability issues in centralized systems. Moreover, since the network topology cannot be efficiently utilized, achieved download rates are limited. A feasible alternative is to crawl the Web from multiple, geographically distributed sites. Multi-site crawlers can increase the page download rate relative to a centralized system as the network latency between crawlers and web sites is lower. Feasibility of multi-site crawlers and associated performance issues are discussed by Cambazoglu et al. (2008).

In case of multi-site crawlers, web partitioning and repartitioning problems take a different shape as hash-based assignment is no longer meaningful. A successful partitioning function tries to maximize the download rate while keeping the load imbalance across sites under a satisfactory level. In (Exposto et al. 2005) and (Exposto et al. 2008), a multi-objective graph partitioning approach is employed to partition the Web, taking into account the geodesic distances between web sites. In (Gao et al. 2006), the problem is formulated as a geographically focused web

crawling problem. A complementary problem to the web partitioning problem is the site placement problem, where a fixed number of crawling sites are tried to be placed, taking into account the distribution of web sites, in order to better utilize the network.

2.5.4.2 Multi-Site Indexing

One approach in multi-site indexing is to replicate documents on all sites and build copies of the same web index. However, this approach suffers from several scalability issues, such as space problems in data centers, maintenance overheads, hardware costs, and lengthy index deployment cycles. A feasible alternative to full replication is to disjointly partition the index across data centers, instead of fully replicating. Among the two partitioning strategies mentioned before, the term-based partitioning strategy is not suitable for multi-site search engines because transferring long result sets between search sites is too costly for wide area networks. Two possible variants of document-based partitioning may be feasible: language-based or region-based partitioning. In the former case, each search site is assigned a set of languages that match the languages of queries submitted to the search site. Documents are partitioned and indexed across the search sites according to their languages. In the latter case, each site is given a geographical region, preferably close to its location. All documents obtained from a region are indexed by the site corresponding to the region. So far, all works investigated the region-based partitioning strategy (Baeza-Yates et al. 2009a; Cambazoglu et al. 2009, 2010b).

The partitioning strategies mentioned above can be coupled with partial replication of documents across the search sites. A naïve replication approach is to sort documents according to their popularity, which can be estimated using their past access frequencies, and replicate a small fraction of most popular documents on all sites (Baeza-Yates et al. 2009a). A similar approach is taken by Cambazoglu et al. (2010b), but also taking into account the storage costs of documents. In (Kayaaslan et al. 2010), more advanced partial document replication techniques are proposed for geographically distributed web search engines.

2.5.4.3 Multi-Site Query Processing

One motivation behind building multiple data centers and processing queries over replicated indexes is to provide fault tolerance. Moreover, this approach brings gains in query response times as network latencies between users and search sites become shorter. In case the web index is partitioned across the sites, reduction in workload and some other scalability benefits can also be achieved (Cambazoglu et al. 2009).

In multi-site query processing on partitioned indexes, assuming the user space is partitioned among the sites (e.g., according to the geographical proximity of sites to users), each search site acts as a local site for queries originating from a specific region. Queries are first processed in the local site to generate a set of local results.

Due to the partitioned nature of the web index, achieving good search quality requires evaluating queries on non-local sites as well. Therefore, some queries are selectively forwarded to non-local sites for further evaluation. Contacted non-local sites return the results they computed over their local indexes to the initial local site, which merges all retrieved results and return them to the user.

The main problem in multi-site query processing is to accurately identify non-local search sites that can contribute good results into the final result set. As a solution, in (Baeza-Yates et al. 2009a), a threshold-based query forwarding strategy is proposed. In an offline phase, this strategy computes the maximum possible score that each term in the vocabulary can receive from a search site. In the online phase, these score thresholds are used to set upper-bounds on potential query scores that can be produced by non-local sites. Queries are forwarded to only the sites with a threshold larger than the locally computed top kth score. One of the main results of this work is that a distributed search engine can achieve the same quality of a centralized search engine at a similar cost. In (Cambazoglu et al. 2010b), a linear-programming-based thresholding solution is proposed to further improve this algorithm, as well as several other optimizations. The impact of index updates on freshness of computed thresholds is studied in (Sarigiannis et al. 2009).

In this architecture, a side issue is the placement of search sites. In practice, locations of search sites are selected based on various characteristics of the host country, such as tax rates, climate, and even political issues. It is possible to enhance these decisions by taking into account distribution of web sites and users. This problem can be seen as an instance of the facility location problem. In the research literature, replica placement algorithms are proposed for distributed web servers and CDNs (Radoslavov et al. 2002).

We note that, herein, we assume that the assignment of users to search sites is static. That is, queries are simply routed to their local data centers by means of a router. If the assignment of users is dynamic (e.g., depends on the workloads of search sites at the query time), the router may act like a load balancer or more generally, like a scheduler.

2.6 Open Problems

We provide a number of open problems that are related to the scalability issues in search engines. We should note that the list we provide is not exhaustive. Moreover, this list might be biased toward our current research interests.

2.6.1 Crawling

An open problem is the so-called push-based crawling. In this technique, web pages are discovered and pushed to the crawler by external agents, instead of being discovered by the crawler itself through the link structure of pages (see for example Castillo 2003). The external agents can be ISPs, toolbars installed on users'

browsers, mail servers, and other software or hardware agents where URLs may be collected. Push-based crawling enables discovery of content, which is not easy to discover by a regular crawler (e.g., hidden web sites Raghavan and Garcia-Molina 2001), and rare or newly created content.

Another problem that requires further research attention is effective partitioning of the Web to improve the scalability of multi-site web crawlers (Cambazoglu et al. 2008). The main challenge in this problem is to come up with techniques to accurately identify locations of web sites and map them to the closest crawling sites (Exposto et al. 2005; Gao et al. 2006, 2008). Although the primary objectives in a successful partitioning are minimization of the crawling time and load balancing across the sites, the partitioning should also take into account the potential overheads in distributed indexing.

2.6.2 Indexing

A challenge for the search engines is to present in their result pages the content created in real-time. With wide-spread use of blogging sites and social networking sites (e.g., Twitter and Facebook), indexing such content becomes important. An open research problem is to develop scalable techniques for processing and indexing vast amounts of real-time streaming text data (e.g., tweets).

Static pruning architectures have already taken some research attention, but more research is needed on tiering. It is still unclear how the number of tiers and sizes of individual tiers can be selected in an optimum manner. Furthermore, better measures are needed to decide on the placement of documents in the tiers.

Some works have considered region-based index partitioning for query processing on multi-site search engine architectures. However, language-based partitioning has not taken much attention so far. Hence, a pros and cons analysis of these two partitioning techniques may be valuable. Another possibility here is hybrid partitioning, which mixes language and region-based partitioning. Hybrid partitioning can also mix document and term based partitioning.

A final open problem that has not taken any research attention so far is multi-site distributed indexing. Although there are works on parallel index creation, it is not clear how these works can be extended to multi-site architectures, which are connected by wide-area networks. The performance overheads of distributed indexing in such architectures need further study.

2.6.3 Query Processing

A recently introduced problem is the freshness issue in result caches of web search engines (Blanco et al. 2010; Cambazoglu et al. 2010c). The problem here is to identify stale cache entries and refresh them before they are requested by queries. Refreshing stale entries is not a trivial problem as this should be done without incurring

too much computational overhead to other parts of the search engine (e.g., the back-end search system).

Most research so far has focused on efficiency of ranking over inverted indexes. In addition to this type of ranking, however, commercial search engines employ complex ranking mechanisms, based on machine learning, over a small set of filtered documents (Cambazoglu et al. 2010a). Efficiency of these mechanisms is important to satisfy query processing time constraints. Moreover, given more time for ranking, search result qualities can be improved (Cambazoglu et al. 2009). The research problems here is to find effective early termination algorithms (Cambazoglu et al. 2010a) and learning algorithms that take into account the efficiency of the generated models (Wang et al. 2010).

In web search engines, queries are processed with certain fixed time budgets. Queries that cannot be processed within their budget lead to degraded result quality. An interesting problem is to identify these time budgets on a per-query basis, rather using fixed thresholds. The budgets can be shrunk based on the geographical proximity of users and extended based on predicted query difficulties or obtained partial result qualities.

For multi-site search engines, better replication strategies can be developed. Current strategies assume that documents are statically assigned and periodically partitioned. One possible improvement could be to replicate documents as the need arises and queries are processed.

Another important issue is to minimize the energy spendings of the search engine. Typically, electricity prices show variation across countries and also during the day within a country. Query processing and forwarding algorithms that are aware of this variation need to be developed to reduce the electricity bill of the search engine.

2.7 Conclusions

We have presented the current scalability challenges in large-scale web search engines. We also discussed a number of open research problems in the field. We envision that the scalability of web search engines will continue to be a research challenge for some time.

We feel that, unlike the past research, the current scalability research is mainly driven by the needs of commercial search engine companies. We must admit that the lack of large hardware infrastructures and datasets make conducting scalability research quite difficult, especially for the researchers in the academia. However, we still find it appropriate to make the following recommendations to those who work in the field:

- The trends in the Web, user bases, and hardware parameters should be closely followed to identify the real bottlenecks in scalability.
- The newly emerging techniques whose primary target is to improve the search quality should be followed and their implications on efficiency should be well understood.

- In solving scalability problems, rather than reinventing the wheel, solutions should be adapted from previously solved problems in related, more mature research fields, such as databases, computer networks, and distributed computing.
- Evaluating new solutions may imply simulating large systems and using synthetic data, as the cost of using real distributed systems is quite large.