Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

A Search Engine Architecture Based on Collection Selection

Diego Puppin

Supervisor
Domenico Laforenza

Supervisor
Marco Vanneschi

Referee
Ophir Frieder

Referee
Gerhard Weikum

September 10, 2007

Viagiar descanta,
ma chi parte mona, torna mona.
Antico detto veneziano

Abstract

In this thesis, we present a distributed architecture for a Web search engine, based on the concept of collection selection. We introduce a novel approach to partition the collection of documents, able to greatly improve the effectiveness of standard collection selection techniques (CORI), and a new selection function outperforming the state of the art. Our technique is based on the novel query-vector (QV) document model, built from the analysis of query logs, and on our strategy of co-clustering queries and documents at the same time.

Incidentally, our partitioning strategy is able to identify documents that can be safely moved out of the main index (into a supplemental index), with a minimal loss in result accuracy. In our test, we could move 50% of the collection to the supplemental index with a minimal loss in recall.

By suitably partitioning the documents in the collection, our system is able to select the subset of servers containing the most relevant documents for each query. Instead of broadcasting the query to every server in the computing platform, only the most relevant will be polled, this way reducing the average computing cost to solve a query.

We introduce a novel strategy to use the instant load at each server to drive the query routing. Also, we describe a new approach to caching, able to incrementally improve the quality of the stored results. Our caching strategy is effectively both in reducing computing load and in improving result quality.

By combining these innovations, we can achieve extremely high figures of precision, with a reduced load w.r.t. full query broadcasting. Our system can cover 65% results offered by a centralized reference index (competitive recall at 5), with a computing load of only 15.6%, *i.e.* a peak of 156 queries out of a shifting window of 1000 queries. This means about 1/4 of the peak load reached when broadcasting queries. The system, with a slightly higher load (24.6%), can cover 78% of the reference results.

The proposed architecture, overall, presents a trade-off between computing cost and result quality, and we show how to guarantee very precise results in face of a dramatic reduction to computing load. This means that, with the same computing infrastructure, our system can serve more users, more queries and more documents.

Acknowledgments

Many people have been important to this work, with their guidance, suggestions and corrections. First of all, I would like to thank my advisors. Domenico Laforenza has encouraged, motivated and supported my research during my years at CNR. He has been a great travel mate, and an important guide in my growth as a researcher. Marco Vanneschi, at University of Pisa, has been an important reference since we first collaborated ten years ago.

The HPC lab has been a very nice environment to work at, and I would like to thank everybody there: those guys have been able to bear with me for 4 years! I had a splendid time. They all have been a source of fun, scientific advice, and personal growth.

In particular, I wouldn't have studied Information Retrieval, and probably I wouldn't have been hired by Google, if it wasn't for Fabrizio Silvestri. His support, help, suggestions and ideas have been invaluable to complete this work. He is one of the brightest minds at CNR, and I wish him a most successful career. Raffaele Perego has also been very important for the second part of this research, with his precious comments. My biggest thanks to both of them.

I would like also to greatly thank Ricardo Baeza-Yates, for inviting me at Yahoo! Research in Barcelona and for supporting (and co-authoring) my recent results. His collaboration has been very fruitful and I hope to keep in touch with such a great researcher and person.

Moreover, thanks to the external reviewers, Ophir Frieder and Gerhard Weikum, for taking the effort to contribute to this work by reviewing it: I am really honored of having their endorsement. I am also grateful to Abdur Chowdury and all the colleagues worldwide that contributed to this thesis with their comments on my papers and my presentations.

History is repeating. Like seven years ago, when completing my thesis for my *Laurea*, I am ready to pack to move to Boston. Again! So, as I did seven years ago, I would like to thank all the long-time friends in Pisa. I have known some of you for about 12 years now, and your friendship has been very valuable to me: this period in Pisa has been really enjoyable. I will never forget you.

This thesis is dedicated to my family. So many thanks to my parents, my brother Cristiano, and Elisa, for their unconditional support: I would have never reached this goal without their help. It has always been important to me to know that there is a safe harbor where I can always rest. Their trust in me has been really motivating. Also, thanks a lot to the Bottinelli family, for welcoming me in their house.

But, most important, I would like to thank Silvia and Giulia for being such a great company to explore the world with! I love you so much.

Reviews and Comments to This Thesis

Per your invitation to assess the quality of Diego Puppins thesis draft, I provide the following comments. Let me begin by applauding Diego Puppin for a solid, innovative, and interesting thesis, one that advance significantly a domain of distributed architectures for search engines. By exploiting a novel collection selection approach, his system efficiently and effectively answer queries including under heavy load. His approach sustains sufficient computational power economically (requiring fewer servers), while preserving search accuracy.

Diego outlines the benefits of a trade-off between result quality and computing costs. Having worked closely with leading industrial people in the Web search community, I am well aware of the need to balance search accuracy and cost particularly to meet scalability concerns. The solution proposed in this thesis can be used to absorb unexpected query peaks, or to reduce the computing load in case of failure. At the same time, it can be used to offer even better results via his novel caching approach.

These results are reached through some novel contributions: a new document model, justified by information theory, which leads to a more effective document clustering; a new collection selection function, able to outperform the established CORI algorithm; a new vision of collection selection, called collection prioritization, that permits the system to stop contacting IR cores, if they are overloaded and not relevant for a query; a novel incremental caching technique, able to effectively exploit prioritization.

In his work, Diego outlines the main elements of a distributed search architecture based on collection selection, addressing some of the most critical problems that IR literature is facing today (load balancing, caching, updating) and identifies original solutions. For this reasons, I believe that Diego's work presents a significant and critical result in the field of IR, which is worth a doctoral degree.

I enthusiastically endorse this dissertation.

Ophir Frieder (July 27, 2007)

The thesis addresses a very important and timely topic: distributed indexing of Web pages with the goal of scalable and efficient search and index maintenance. The thesis presents a novel way of data partitioning, by clustering the pages in a specific way and assigning each cluster to one of the servers in the distributed system. At query time, it routes the query to the most promising servers, based on a specific similarity measure between queries and page clusters. This novel architecture is further enhanced by heuristics for load balancing, heuristics for maintaining a cache of query results, and a folding-in technique for updating the distributed index. The thesis emphasizes experimental studies as the means for evaluating the relative performance of different models and algorithms. Overall, this is a nice and insightful piece of original research.

Gerhard Weikum (July 11, 2007)

Diego's work is in the line of current research on high performance architecture for web search engine. I think that his work has improved current schemes for partitioning indices. In fact, the work that he did during his visit to Yahoo! Research was published at Infoscale 2007 and I was one of the co-authors.

His latest results addressed one of the issues that I mentioned in my earlier mail, that is, a detailed analysis on load balancing: it is fundamental to prove that the suggested technique can actually reduce the overall load of a search engine, in a balanced way, while maintaining competitiveness in other issues such as performance, space usage, etc. This issue is one of the main results of the thesis.

Considering the comments above, Diego's thesis is a good contribution to the field of distributed information retrieval.

Ricardo Baeza-Yates (July 10, 2007)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Managing a Growing Web

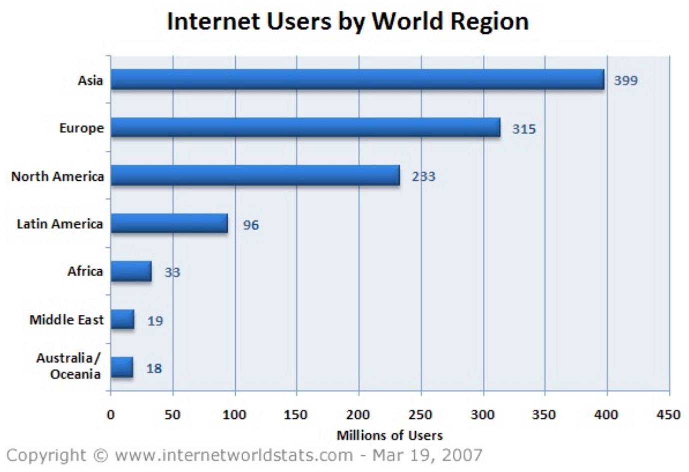


Figure 1.1: Estimated worldwide Internet Users, as of March 10, 2007 (from [48])

Today, about 1.1 billion people have access to the Internet and use it for work or leisure. The World Wide Web is one of the most popular applications, and millions of new Web pages are created every month, to spread information, to advertise services or for pure entertainment. The Web is getting richer and richer and is storing a vast part of the information available worldwide: it is becoming, to many users in the world, a tool for augmenting their knowledge, supporting their theses, comparing their ideas with reputable sources.

To navigate this abundance of data, users rely more and more on search engines for any information task, often looking for their needs directly in the result page (the so-called *information snacking*[74]). This is why successful search engines have to crawl, index and search quickly billions of pages, for millions of users every day. Very sophisticated techniques are needed to implement efficient search strategies for very large document collections.

Similar problems are posed by other IR applications. One is, for instance, the Google Book Search project[1], which aims at scanning, indexing and making searchable millions of printed books from libraries worldwide, including, among several others, the University of California Library System, the University Libraries of Princeton, Harvard, Stanford and Lausanne, the Bavarian State Library and the National Library of Catalonia. Another example is given by the large, and

[1]Available at http://books.google.com/.

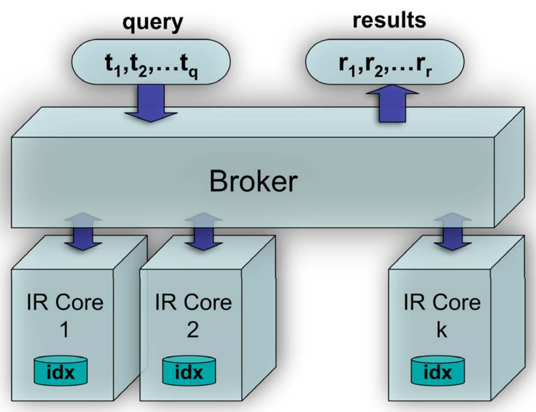


Figure 1.2: Organization of a parallel information retrieval system.

very popular, archives of media, such as Flickr or Youtube[2], which are expected to find and deliver a wealth of information to users worldwide. Very complex problems of scalability are to be tackled in order to offer an effective service to billions of daily queries.

Parallel and distributed information retrieval systems are a way to implement a scalable search service. A parallel information retrieval system is usually deployed on large clusters of servers running multiple *IR core* modules, each of which is responsible for searching a partition of the whole index. When each sub-index is relative to a disjoint sub-collection of documents, we have a *document-partitioned* index organization, while when the whole index is split, so that different partitions refer to a subset of the distinct terms contained in all the documents, we have a *term-partitioned* index organization. Each organization of the index requires a specific process to evaluate queries, and involves different costs for computing and I/O, and network traffic patterns.

In both cases, in front of the cluster, we have an additional machine (or set of machines) hosting a *broker*, which has the task of scheduling the queries to the various servers, and collecting the results returned back. The broker then merges and sorts the results received on the basis of their relevance, produces the ranked list of matching documents, and builds the results page containing URLs, titles, snippets, related links and so on.

Figure 1.2 shows the logical organization of a parallel IR system, where k is the number of servers hosting IR core modules, q the number of terms in a given query, and r the number of ranked results returned for the query.

Document partitioning is the strategy usually chosen by the most popular web search engines [15, 11]. In the document-partitioned organization, the broker may choose among two possible strategies for scheduling a query. A simple, yet very common, way of scheduling queries is to broadcast each of them to all the underlying IR cores. This method has the advantage of enabling a good load balancing among all of the servers [4]. On the other hand, it has the major drawback of utilizing every server for each query submitted, this way increasing the overall system load.

The other possible way of scheduling is to choose, for each query, the most authoritative server(s). By doing so, we reduce the number of IR cores queried. The relevance of each server to a given query is computed by means of a selection function that is built, usually, upon statistics computed over each sub-collection. This process, called *collection selection* is considered to be an effective technique to enhance the capacity of distributed IR systems [8].

In this thesis, we present an architecture for a distributed search engine, based on collection selection. We discuss a novel strategy to perform document partitioning and collection selection, which uses the query log to drive the assignment of documents. Our architecture has effective strategies for load balancing and result caching, and is able to return very high quality results

[2]Available at http://www.flickr.com/ and http://www.youtube.com/.

using a fraction of the computing load of a traditional document-partitioned search engine.

To achieve this result, this thesis analyzes several aspects of a distributed information retrieval system:

- several document partitioning strategies are analyzed, and compared with our solution based on the *query-vector document model*; incidentally, we develop an effective way to select a set of documents that can be safely moved out of the main index (into the supplemental index);
- the quality of collection selection is improved by a new model based on the results of the clustering phase;
- we consider the problem of load balancing, and we design a strategy to reduce the load differences among the cores; our load-driven routing strategy can accommodate load peaks by reducing the pressure on the servers that are less relevant to a query;
- we study the problem of adding documents to the collections, and we verify the effectiveness of a simple strategy based on our collection selection function;
- we study how to utilize a caching system when performing collection selection.

By addressing all these problems, we will show that our system is able to serve a larger volume of queries. Its strength is the ability to use only a subset of the available IR cores for each query: this will reduce the average computing load to answer each query, and open the possibility to serve more users without adding computational power to the system.

Our solution does not explicitly discuss the performance of a single query, but rather focuses on increasing the capacity of an IR system. While our techniques can contribute to a small overhead to each query, we believe this overhead is small and largely compensated by the gain in query throughput.

This study is centered on a trade-off among result quality and computing load. We will show that, by sacrificing a small fraction of the relevant results, we can dramatically decrease the computing load of the system. If enough computing power is available at a given instant, the system can return non-degraded results by polling all servers as in traditional document-partitioned systems, but it can adjust to a load peak by selecting only the most relevant servers and reducing the computing pressure on the other ones.

1.2 A Search Engine Architecture Based on Collection Selection

The overall system architecture is shown in Figure 1.3. All queries from users are received by a query broker, an access point to the system that can be possibly replicated for performance reasons.

First, the caching system verifies if the results are already available and, if so, it can return them to the user. Alternatively, in case of a hit, the cache can update the stored results in an incremental way, as will be shown in Chapter 5.

The query broker stores the query in a log, along with the results (for training purposes), and performs collection selection. At this point, it forwards the query to the most relevant search cores. This selection can be driven by the instant load of the system (see Chapter 4).

The documents are crawled by another server (or set of servers). A simple but effective strategy to add documents to existing partition is discussed in Chapter 6. We use information coming from the query log to build the query-vector representation and perform partitioning (see Chapter 3). The results of this process are sent to the broker, which uses them for collection selection.

Each document block is assigned to a server which performs local indexing on the documents it was given. The local index is then used to answer queries. Each server collects statistics about the local collection so that the system can guarantee uniform document ranking across the different servers (see Section 2.5).

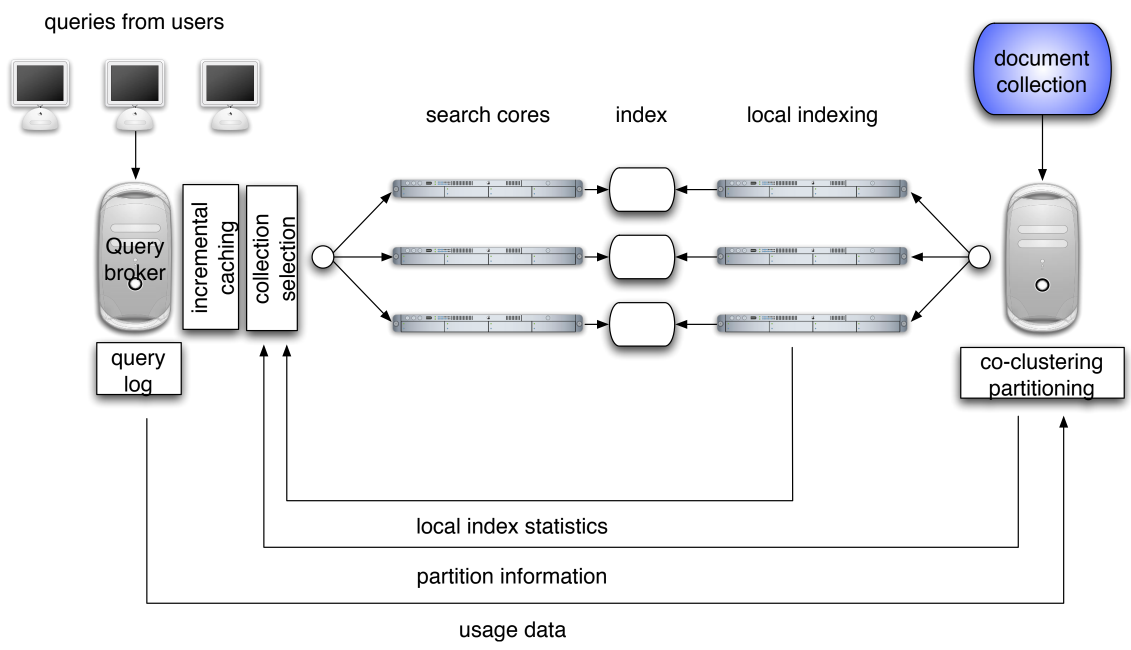


Figure 1.3: Overall architectural scheme.

Our system was tested through an extensive simulations of its main aspects, using a set of real query logs (about 500,000 queries for training and 800,000 for test), on a crawl of real Web pages (about 6,000,000), as described in Chapter 8. Our simulation includes the real cost (timing) of the different queries on each document partition (see Chapter 7).

1.3 Main Contributions of This Thesis

The main contributions of this thesis are:

1. a new representation of documents as *query-vectors*, *i.e.* as the list of queries that recall them, out of a query log; this representation is very compact and allows us to perform a very effective clustering;

2. a novel representation of collections based on the co-clustering results; this representation is about 5 times smaller than the representation needed by CORI [22], and very competitive with it;

3. a new strategy for document partitioning and collection selection, based on the results of the clustering phase, that proved to be more effective than several other approaches;

4. a way to identify rarely asked-for documents: we showed that about half of the documents of our collection are not returned as top-scoring results of any query in the log and contribute only to a very small fraction of the results of the test queries;

5. a load-driven collection selection strategy, which can effectively contribute to balance the system load;

6. an incremental caching technique, which can collaborate with collection selection to retrieve high-quality results;

7. a simple but effective approach to add documents to the collection, using the collection selection strategy.

1.4 Structure of This Thesis

This thesis is organized as follows. After an overview of the current state of the art in distributed search architecture, collection selection and indexing techniques (Chapter 2), we introduce the query-vector document model, and we analyze its performance and precision (Chapter 3).

After that, we discuss how to address load balance (Chapter 4) and caching (Chapter 5) in our architecture and, in Chapter 6, we analyze how to add documents to the collection. Then, in Chapter 7, we introduce a more realistic model, and we confirm our experimental results with this new framework.

Our experimental infrastructure is described in detail in Chapter 8. Finally, we conclude with the summary of our results, an overview of potential applications and future work.

Chapter 2

State of the Art

In this thesis, we discuss how to build an effective distributed architecture for a search engine based on collection selection. In this chapter, we present and discuss some of the main results in literature from which this work starts.

First, we discuss the limitations of standard document- and term-partitioned architectures that motivated this work. We also comment on some advanced IR architectures that are of interest to this thesis. Then, we analyze some techniques for document clustering and collection selection, which are used as a starting point and comparison for our study. We also present an overview of previous literature in the field of result ranking in distributed architectures, and of usage mining and caching in search engines. Finally, we discuss some solutions to update the index that are relevant to our work.

2.1 Document and Term-Partitioning

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 Web full-text IR systems [10]. A simple implementation consists of two data structures, namely the *lexicon* and the *posting lists.* The former stores all the distinct terms contained in the documents. The latter are represented as 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 *sorting* 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 [100], or *single-pass* algorithms [59], 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.

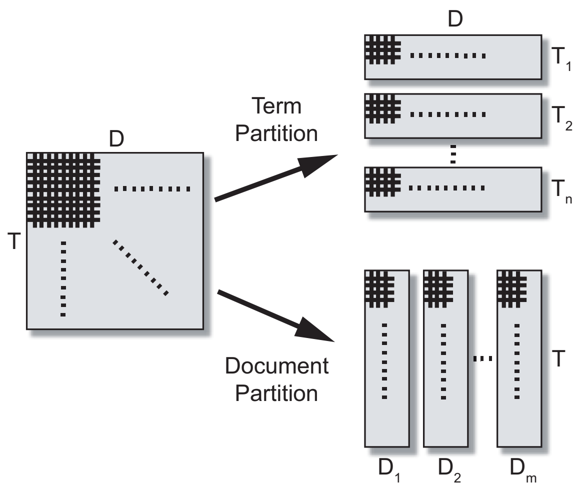


Figure 2.1: The two different types of partitioning of the term-document matrix (from [8]).

According to the way servers partition the $T \times D$ matrix, we can have two different types of distributed indexes (Figure 2.1). 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 the need of building 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. The most important is that only a limited number of servers, *i.e.* those holding the terms comprising a query, are used to compute the results. In a *pipelined* architectures, the query is answered by sending the potential list of results through a pipeline of servers, which intersect it with the posting lists of the terms they hold. Webber *et al.* show that term partitioning results in lower utilization of resources [99]. 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 2.2 (originally from [99]) illustrates the average 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.

Another negative side of term-partitioning is the higher communication cost to solve each query. While in a document-partitioned system, each server has to return the list of local results, in a term-partitioned system the servers holding the relevant terms must share the full posting lists to be intersected. While several optimizations are available, still this is considered a limiting factor of performance.

These are two of the main reasons why document partitioning is considered to be superior to term partitioning [18, 68, 17, 6, 5]. To overcome the typical load skewness of term-partitioned systems, 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. Approaches like the ones discussed in [67, 105] are ways to limit the drawbacks of the term-partitioned approach.

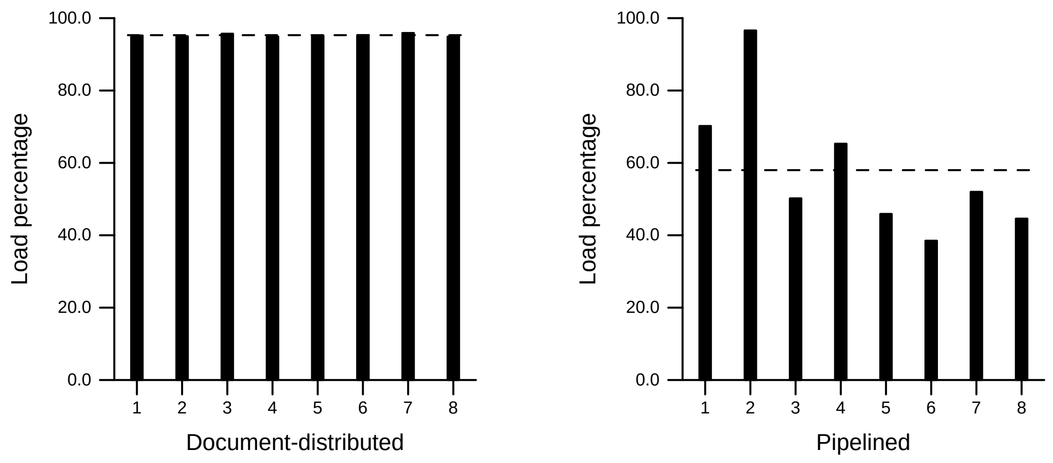


Figure 2.2: Distribution of the average load per processor in a document-partitioned, and a pipelined term-partitioned IR systems [99].

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.

In this thesis, we try to exploit the main benefit of a term-partitioned system, *i.e.* the fact that only a subset of servers is polled for every query, with a document-partitioned architecture. This is done by a careful assignment of documents to servers.

2.2 Advanced IR Hierarchies

On the top of partitioning, search engines often uses hierarchical approaches, with several stages of caching and/or indexing. One of the first thorough analysis of caching performance in Web search engines is given in [9]. The paper presents a detailed analysis of how to organize a single-server search engine in a three-level architecture: cached results, a part of the index in memory and the remaining on the disk.

After this article, several works have discussed more complex organizations, adapted to work within distributed systems. In [64], the authors propose an effective three-layered approach to caching in distributed search architectures. The first cache level stores query results. This is done by the broker, before broadcasting the queries to each servers. Then, at the server level, some memory space is reserved for caching intersections of common term pairs. Last, some posting lists are cached.

The biggest commercial engines are also believed to use complex multi-layered approaches, with several layers of caching, of processing, of data partitioning. Unfortunately, details about them are not disclosed. They are rumored to be structured with multiple *document frontiers* (see Figure 2.3). In this architecture, the most popular pages and sites are available on a first set of servers. If no results are available from the first frontier, the query is forwarded to a second layer, where many more documents are available, with reduced replication. Further frontiers can

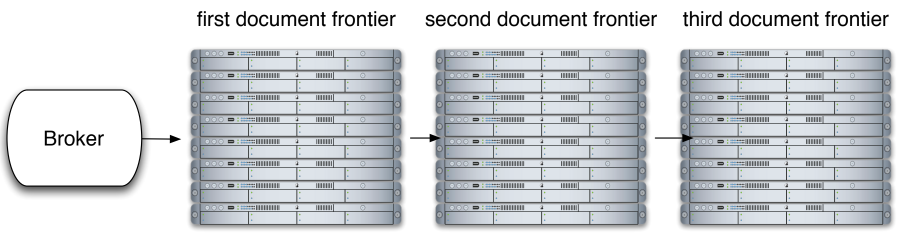


Figure 2.3: Multiple document frontiers in high-performance search engines.

be available. The last frontier can have a simplified index, storing only keywords from titles or URLs. Unfortunately, the details of this strategy are not publicly available, even if in some case users are aware that results are coming from the deeper levels: some commercial engines label the second-layer results as *supplemental*.

Our work improves on the state of the art by introducing an integrated approach to distributed search engines, which introduces novel technique to address load balancing and caching with a collection selection strategy. It also introduces an effective way, based on the analysis of the query logs, to identify documents that are rarely asked for: these documents can be moved to the supplemental index, with a very limited loss in result quality.

2.3 Document Clustering and Partitioning

Partitioning potentially enhances the performance of a distributed search engine in terms of capacity. 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 servers that hold the information on the particular terms of the query. Upon receiving a query, it is forwarded only to the servers responsible for maintaining the subset of terms in the query. In the case of document partitioning, the problem is fairly more complex since we are not able to know in advance the servers that contain the most relevant results.

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. The great number of queries submitted every day to Web search engines enables the detection of user behavior patterns from the resulting query logs. Compared to traditional distributed IR systems [19], Web search engines offer the opportunity to mine user behavior information from the millions of queries submitted every day.

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. There are several ways to exploit this information to enable partitioning the document collection and routing queries more efficiently.

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;
- precision is very high, compared to a centralized system.

For a term partitioned system, Moffat *et al.* [72] 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.

Similarly, Lucchese *et al.* [67] 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. The original bin-packing problem simply aims at balancing the weights assigned to the bins. In this case, however, the *objective function* depends both on the single weights assigned to terms (the objects), and on the co-occurrence of terms in queries. The main goal of this function is to assign 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, the problem of creating good document clusters is also very complex. In [37], the authors present a model for the problem of assigning documents to a multiprocessor information retrieval system. The problem is shown to be NP complete and a solution based on genetic algorithms is discussed. Clearly, faster solutions are needed for large scale systems. 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. Nonetheless, distributing documents randomly across servers, however, does not guarantee an even load balance [4].

Another drawback of random partitions is that servers may execute several operations unnecessarily when querying sub-collections, which may contain only few or no relevant documents. A different, more structured approach is to use k-means clustering to partition a collection according to topics [57, 63].

A radically different approach to partitioning is followed by pSearch [95] and SSW [61]. pSearch performs an initial LSI (*Latent Semantic Indexing*) transformation that is able to identify *concepts* out of the document base. LSI works by performing a *Singular Value Decomposition* (SVD) of the document-term matrix, which projects the term vector onto a lower-dimensional semantic space. Then, the projected vectors are mapped onto a *Content Addressable Network* (CAN) [83]. Search and insertion is done using the CAN space. In other words, when a query is submitted, it is projected, using the SVD, onto the CAN, and then neighbors are responsible for answering with the documents they hold.

SSW [61] also uses LSI to reduce the dimensionality of data, but then uses a small-world network rather than a CAN. Again, a query is answered by the neighbors of the node responsible for the vector representing the query, once mapped on the LSI-transformed space.

Another type of document clustering is given by P2P search systems [13, 76, 94, 93]. In these systems, users independently collect data about their interests, this way creating document collections focused on certain topics. These collections can clearly be incomplete and/or overlapping. All these data are then made available to all peers in a networks, who can then perform queries to retrieve documents coming from any of these sets. Suitable routing and searching algorithms are needed to bring the query to the most promising servers, and to match the query with the documents that are locally available to each peer.

In these systems, the algorithm designer can not expect a very high degree of collaboration: full statistics describing the collection may be unavailable. Also, the documents may be partitioned inefficiently, imposing the need for querying a high number of peers in order to recall a reasonable set of results.

In this thesis, we present a coordinated strategy to partition documents according to the information coming from a query log. We will show that our document clusters are very focused, and only a limited set of clusters is sufficient to retrieve very high-quality results. Also, our training procedure allows us to choose very efficiently the set of most promising servers for each query, with results outperforming the state of the art.

2.4 Collection Selection

Today, it is commonly held that realistic parallel IR systems will have to manage distinct indexes. In our opinion, a possible way to ensure timely and economic retrieval is designing a query broker module so that it will forward a given query only to workers managing documents related to the query topic. In other words, we are speaking about adopting *collection selection* techniques aimed at reducing a query search space by querying only a subset of the entire group of workers available. Nevertheless, particular attention should be paid in using this technique. In fact, it could result in a loss of relevant documents thus obtaining degradation in the effectiveness figure.

In the last ten years a large number of research works dealt with the collection selection problem [66, 21, 20, 30, 35, 33, 34, 38, 39, 45, 81, 49, 53, 55, 56, 65, 77, 79, 86].

The most common approaches to distributed retrieval exploit a number of heterogeneous collections, grouped by source and time period. A *collection selection index (CSI)* summarizing each collection as a whole, is used to decide which collections are most likely to contain relevant documents for the query. Document retrieval will actually only take place at these collections. In [45] and [27] several selection methods are compared. The authors show that the naïve method of using only a collection selection index lacks in effectiveness. Many proposals tried to improve both the effectiveness and the efficiency of the previous schema.

Moffat *et al.* [73] use a centralized index on blocks of B documents. For example, each block might be obtained by concatenating documents. A query first retrieves block identifiers from the centralized index, then searches the highly ranked blocks to retrieve single documents. This approach works well for small collections, but causes a significant decrease in precision and recall when large collections have to be searched.

In [42, 43, 44, 97], Garcia–Molina *et. al.* propose GlOSS, a broker for a distributed IR system based on the boolean IR model. It uses statistics over the collections to choose the ones which better fits the user's requests. The authors of GlOSS make the assumption of independence among terms in documents so, for example, if term A occurs f_A times and the term B occurs f_B times in a collection with D documents, than they estimated that $\frac{f_A}{D} \cdot \frac{f_B}{D} \cdot D$ documents contain both A and B. In [41] the authors of GlOSS generalize their ideas to vector space IR systems (gGlOSS), and propose a new kind of server hGlOSS that collects information for several GlOSS servers and select the best GlOSS server for a given query.

In [22], the authors compare the retrieval effectiveness of searching a set of distributed collections with that of searching a centralized one. The system they use to rank collections is an inference network in which leaves represent document collections, and nodes represent the terms that occur in the collection. The probabilities that flow along the arcs can be based upon statistics that are analogous to *tf* and *idf* in classical document retrieval: document frequency *df* (the number of documents containing the term) and inverse collection frequency *icf* (the number of collections containing the term). They call this type of inference network a *collection retrieval inference network*, or *CORI* for short. They found no significant differences in retrieval performance between distributed and centralized searching when about half of the collections on average were searched for a query.

CVV (*Cue-Validity Variance*) is proposed in [104]. It is a a new collection relevance measure, based on the concept of *cue-validity* of a term in a collection: it evaluates the degree to which terms discriminate documents in the collection. The paper shows that effectiveness ratio decreases as very similar documents are stored within the same collection.

In [102] the authors evaluate the retrieval effectiveness of distributed information retrieval systems in realistic environments. They propose two techniques to address the problem. One is to use phrase information in the collection selection index and the other is query expansion. In [29], Dolin *et al.* present Pharos, a distributed architecture for locating diverse sources of information on the Internet. Such architectures must scale well in terms of information gathering with the increasing diversity of data, the dispersal of information among a growing data volume. Pharos is designed to scale well w.r.t. all these aspects, to beyond 10^5 sources. The use of a hierarchical metadata structure greatly enhances scalability because it features a hierarchical network organization.

[103] analyzes collection selection strategies using cluster-based language models. Xu *et al.*

propose three new methods of organizing a distributed retrieval system based on the basic ideas presented before. These three methods are *global clustering*, *local clustering*, and *multiple-topic representation*. In the first method, assuming that all documents are made available in one central repository, a clustering of the collection is created; each cluster is a separate collection that contains only one topic. Selecting the right collections for a query is the same as selecting the right topics for the query. This method is appropriate for searching very large corpora, where the collection size can be in the order of terabytes. The next method is *local clustering* and it is very close to the previous one except for the assumption of a central repository of documents. This method can provide competitive distributed retrieval without assuming full cooperation among the subsystems. The disadvantage is that its performance is slightly worse than that of global clustering. The last method is *multiple-topic representation*. In addition to the constraints in local clustering, the authors assume that subsystems do not want to physically partition their documents into several collections. A possible reason is that a subsystem has already created a single index and wants to avoid the cost of re-indexing. However, each subsystem is willing to cluster its documents and summarize its collection as a number of topic models for effective collection selection. With this method, a collection may correspond to several topics. Collection selection is based on how well the best topic in a collection matches a query. The advantage of this approach is that it assumes minimum cooperation from the subsystem. The disadvantage is that it is less effective than both global and local clustering.

In this thesis, we will show how to use the information coming from a query log to build a very effective collection selection function.

2.5 Result Ranking

A problem closely related to collection selection is how to re-rank the matching documents, coming from independent servers, in a coherent way, so that a *global ranking* is built out of the ranking performed by each server.

CORI [22] bases its selection and ranking algorithms on the availability of statistics from the collections: for each collection, CORI needs to collect the frequency of terms and of documents holding each term. This is a very strong hypothesis, as the participating collections may choose not to cooperate, by giving wrong or no statistics about their data. Also, the collections could change (due to crawling) and precise information could be not available at certain times.

Subsequent works have removed this assumption, by using statistical sampling or using information from previous queries in the query log. [82] presents a technique to determine a global ranking, using the results coming from each collection. By using probabilistic methods, the technique can infer the *skewness* of each collection's ranking, and can re-adjust documents' score. ReDDE, presented in [89], shows a strategy that uses information from previous queries to build statistics over the collections. It is shown to out-perform CORI over non-balanced collections.

The work discussed in [88], instead, uses information from a query log to actively samples the collection. The data resulting from sampling are then used for collection selection. ROSCO, presented in [24], improves on this by removing the assumption of non-overlapping collections. Active sampling and probing are also used in [52] to improve collection selection, in the context of P2P networks.

All these results, in any case, do not apply to the system we presented in this thesis, as collections are not only cooperating, but they are built in a coordinated way. If the servers use the standard vector space model to rank the documents, each pair term-document is weighted by the frequency of term t in document d (the term frequency *tf*) and the inverse document frequency (*idf*) of the term t among the documents in the whole collection. Clearly, we want *idf* to be consistent across all the servers. This can be done, in a simple way, by having the servers exchanging their local *idf* after creating the local index for the documents they are assigned. The servers can compute the global *idf* from the contribution of each server [84]. This can clearly be extended to ranking strategies including more complex metrics (*e.g.* PageRank).

If we want to let the local collections change freely over time (*e.g.* due to continuous crawling), we can build consistent ranking with a *double round* approach to queries. In the first round, the query is sent to each server, which returns the local statistics for the terms in the query. The broker combines them and send them back to the servers, who can now compute a consistent ranking. This solution is clearly more costly but can be highly engineered so to introduce a minimal overhead, *e.g.* by caching value or by a periodical polling.

In this thesis, we assume that the statistics are exchanged after the local indexing phase, and that the servers simply return consistent scores for their results.

2.6 Web Usage Mining and Caching

Caching is a useful technique on the Web: it enables a shorter average response time, it reduces the workload on back-end servers and the overall amount of utilized bandwidth. When a user is browsing the Web, both his/her client and the contacted servers can cache items. Browsers can cache Web objects instead of retrieving them repeatedly, while servers can cache pre-computed answers or partial data used in the computation of new answers. A third possibility, although of less interest to the scope of this work, is storing frequently requested objects in the proxies to mediate the communication between clients and servers [75].

Query logs constitute a valuable source of information for evaluating the effectiveness of caching systems. While there are several papers analyzing query logs for different purposes, just a few consider caching for search engines. As noted by Xie and O'Hallaron [101], many popular queries are in fact shared by different users. This level of sharing justifies the choice of a server-side caching system for Web search engines.

Previous studies on query logs demonstrate that the majority of users visit only the first page of results, and that many sessions end after just the first query [90, 51, 50, 12]. In [90], Silverstein *et al.* analyzed a large query log of the AltaVista search engine, containing about one billion queries submitted in more than a month. Tests conducted included the analysis of the query sessions for each user, and of the correlations among the terms of the queries. Similarly to other works, their results show that the majority of the users (in this case about 85%) visit the first result page only. They also show that 77% of the users' sessions end up just after the first query. A similar analysis is carried out in [51]. With results similar to the previous study, they concluded that while IR systems and Web search engines are similar in their features, users of the latter are very different from users of IR systems. A very thorough analysis of users' behavior with search engines is presented in [50]. Besides the classical analysis of the distribution of page-views, number of terms, number of queries, etc., they show a topical classification of the submitted queries that point out how users interact with their preferred search engine. Beitzel *et al.* [12] analyzed a very large Web query log containing queries submitted by a population of tens of millions users searching the Web through AOL. They partitioned the query log into groups of queries submitted in different hours of the day. The analysis, then, tried to highlight the changes in popularity and uniqueness of topically categorized queries within the different groups.

One of the first papers that exploits user query history is by Raghavan and Sever [78]. Although their technique is not properly caching, they suggest using a *query base*, built upon a set of persistent optimal queries submitted in the past, in order to improve the retrieval effectiveness for similar queries. Markatos [69] shows the existence of temporal locality in queries, and compares the performance of different caching policies.

Lempel and Moran propose *PDC* (Probabilistic Driven Caching), a new caching policy for query results based on the idea of associating a probability distribution with all the possible queries that can be submitted to a search engine [58]. PDC uses a combination of a *segmented LRU* (SLRU) cache [54, 69] (for requests of the first page) and a heap for storing answers of queries requesting pages after the first. Priorities are computed on previously submitted queries. For all the queries that have not previously seen, the distribution function evaluates to zero. This probability distribution is used to compute a priority value that is exploited to order the entries of the cache: highly probable queries are highly ranked, and have a low probability to be evicted from the cache. Indeed, a replacement policy based on this probability distribution is only used for queries regarding pages subsequent to the first one. For queries regarding the first page of results, a *SLRU* policy is used [69]. Furthermore, *PDC* is the first policy to adopt prefetching to anticipate user requests. To this purpose, *PDC* exploits a model of user behavior. A user session starts with a query for the first page of results, and can proceed with one or more *follow-up* queries (*i.e.* queries requesting successive page of results). When no follow-up queries are received within τ seconds, the session is considered finished. This model is exploited in *PDC* by demoting the priorities of the entries of the cache referring to queries submitted more than τ seconds ago. To keep track of query priorities, a priority queue is used. *PDC* results measured on a query log of AltaVista are very promising (up to 53.5% of hit-ratio with a cache of $256,000$ elements and 10 pages prefetched).

Fagni *et al.* show that combining static and dynamic caching policies together with an adaptive prefetching policy achieves a high hit ratio [31]. In their experiments, they observe that devoting a large fraction of entries to static caching along with prefetching obtains the best hit ratio. They also show the impact of having a static portion of the cache on a multithreaded caching system. Through a simulation of the caching operations they show that, due to the lower contention, the throughput of the caching system can be doubled by statically fixing one half of the cache entries.

Another very recent work [7] confirms that static caching is robust over time, because the distribution of frequent queries changes very slowly.

The Adaptive Replacement Cache (ARC), introduced in [70], is a very flexible caching system able to adapt automatically both to streams preferring accesses to recent entries (*e.g.* instructions in program loops) and streams preferring frequent entries (*e.g.* disk accesses to the allocation table). This results is reached by simulating two LRU caches of N entries, one for one-time hits, and one for repeated hits. When an entry in the first LRU cache is requested, it is promoted to the second cache. The system will actually store only $N1$ entries for the first cache, and $N2$ for the second, with $N = N1 + N2$, but will record the keys for all $2N$. Over time, the values $N1$ and $N2$ are changed according to the relative hit-ratio of the two caches. ARC was proved to outperform several online algorithms, and to compete with several offline algorithms (*i.e.* caching systems tuned to the workload by an offline analysis).

In this thesis, we build over these results by adding the cache the capability of updating the cached results. In our system, based on collection selection, the cache stores the results coming only from the selected servers. In case of a hit, the broker will poll further servers and update the cache entry with the new results. Our incremental caching is compared with ARC and SDC.

2.7 Updating the Index

The algorithms to keep the index of an IR system up to date are very complex, and have been the interest of several works in the scientific literature.

In [3], the opening article on the first issue of ACM TOIT (2001), Arasu *et al.* comment on the open problems of Web search engines, and clearly identify the update of the index as a critical point. They suggest to rebuild the index after every crawl, because most techniques for incremental index update do not perform very well when they have to deal with the huge changes (in term of the number of documents added, removed and changed) that are commonly observed between successive crawls of the Web. The work by Melnik*et al.* [71] (cited in [3]) gives a measure of the phenomenon, and emphasizes the need to perform research in this area. The opportunity of a batch update of the index (rather than incremental) is also supported by a previous work by Frieder *et al.* [36], who show that the result precision is not affected by a delayed update of the index.

The problem is made more complex in distributed system. In a term-partitioned system, when a document is added/changed/removed, we need to update the index in all servers holding at least one term from the document. While this update can be fast because the update is executed in parallel in several locations of the index (several different posting lists), the communication costs are very high, and they make this approach unfeasible. On a document-partitioned system, instead, all the changes are local to the IR server to which the document is assigned: when a document is assigned to one server, only the local index of that server needs to be changed. This is another reason why document-partitioned systems are preferred to term-partitioned systems in commercial installations [15, 8].

One of the first papers to address the incremental update of an index is [96], which, in 1993, presented a a dual-structure index to keep the inverted lists up to date. The proposed structure dynamically separated long and short inverted lists and optimized retrieval, update, and storage of each type of list. The authors show an interesting trade-off between optimizing update time and optimizing query performance. The result is a starting point for later papers, which explicitly address a very high rate of change.

In [62], the authors acknowledge that, while results in incremental crawling have enabled the indexed document collection of a search engine to be more synchronized with the changing World Wide Web, the new data are not immediately searchable, because rebuilding the index is very time-consuming. The authors explicitly study how to update the index for changed documents. Their method uses the idea of *landmarks*, *i.e.* marking some important spots in the inverted lists, together with the *diff* algorithm, to significantly reduce the number of postings in the inverted index that needs to be updated.

Another important result was presented in [60]. The authors dicuss the three main alternative strategies for index update: in-place update, index merging, and complete re-build. The experiments showed that re-merge is, for large numbers of updates, the fastest approach, but in-place update is suitable when the rate of update is low or buffer size is limited. The availability of algorithms targeted to different speed of change is important because, in a document-partitioned system, different collections could change at a different speed.

A very different approach to update is analyzed in [87]. Following previous literature, the authors suggest to book some sparing free space in an inverted file, so to allow for incremental updates. They propose a run-time statistics-based approach to allocate the spare space. This approach estimates the space requirements in an inverted file using only some recent statistical data on space usage and document update request rate. Clearly, there is a trade-off between the cost of an update and the extra storage needed by the free space. They show that the technique can greatly reduce the update cost, without affecting the access speed to the index.

All these results are surely crucial in order to have a real-world implementation of a system as the one proposed in this thesis. When more documents become available, it is important to have efficient ways to update the index in the IR cores. Nonetheless, the issues of managing the local index are not the central point of this research. Our collection selection system can use any underlying IR strategy for the IR cores as a black box: our work is focused on choosing the right server, and not on the performance and management of each server. In this thesis, we will show a simple technique that allows our system to choose a partition for each new document so that the document clustering is not degraded.

Chapter 3

The Query-Vector Document Model

Commercial search engines, as discussed in the previous chapter, usually adopt a document-partitioned approach, where documents are partitioned using a random/round-robin strategy aimed at offering a good load-balance. More complex strategies, *e.g.* a partitioning based on k-means [103], are rarely used because of the great computing costs on large collections. In both cases, anyway, documents are partitioned without any knowledge of what queries will be like.

We believe that information and statistics about queries may help in driving the partitions to a good choice. The goal of our partitioning strategy is to cluster the most relevant documents for each query in the same partition. The cluster hypothesis states that *closely associated documents tend to be relevant to the same requests* [98]. Clustering algorithms, like the k-means method cited above, exploit this claim by grouping documents on the basis of their content.

We instead base our method on the novel *query-vector (QV) document model.* In the QV model, documents are represented by the weighted list of queries (out of a training set) that recall them: the QV representation of document d stores the scores that d gets for each query in the query set. The set of the QVs of all the documents in a collection can be used to build a query-document matrix, which can be normalized and considered as an empirical joint distribution of queries and documents in the collection. Our goal is to co-cluster queries and documents, so to identify queries recalling similar documents, and groups of documents related to similar queries. The algorithm we adopt is described by Dhillon *et al.* in [28], and is based on a model exploiting the empirical joint probability of picking up a given couple (q, d), where q is a given query and d is a given document (see Section 3.2).

The results of the co-clustering algorithm are used to perform collection selection. We will prove that our method has a very small representation footprint, and is competitive with CORI, outperforming it in critical conditions, *i.e.* when only a small numbers of collections are selected. In this chapter we show that our strategy, based on the QV model, is very effective and is very robust to topic shift, noise, changes in the training set.

3.1 Model of the Problem

Our goal is to partition the documents of our collection into several clusters, using user requests from a query log as our driver. This way, all documents matching a given query are expected to be located in the same cluster. To do this, we perform co-clustering on the queries from our query log and the documents from our base.

The way we consider documents for co-clustering can be seen as a new way of modeling documents. So far, two popular ways of modeling documents have been proposed: *bag-of-words*, and *vector space.* Since we record which documents are returned as answers to each query, we can represent a document as a *query-vector.*

Query/Doc	d1	d2	d3	d4	d5	d6	...	dn
q1	-	0.5	0.8	0.4	-	0.1	...	-
q2	0.3	-	0.2	-	-	-	...	0.1
q3	-	-	-	-	-	-	...	-
q4	-	0.4	-	0.2	-	0.5	...	0.3
...	...	...	...	...	...	...	...	...
qm	0.1	0.5	0.8	-	-	-	...	-

Figure 3.1: In the query-vector model, every document is represented by the query it matches (weighted with the rank).

The query-vector representation of a document is built out of a query log. A reference search engine is used in the building phase: for every query in the training set, the system stores the first N results along with their rank.

Figure 3.1 gives an example. The first query $q1$ recalls, in order, $d3$ with rank 0.8, $d2$ with rank 0.5 and so on. Query $q2$ recalls $d1$ with rank 0.3, $d3$ with rank 0.2 and so on. We may have empty columns, when a document is never recalled by any query (in this example $d5$). Also, we can have empty rows when a query returns no results ($q3$).

We can state this more formally.

Definition 1. ***Query-vector model.*** *Let Q be a query log containing queries $q_1, q_2, \ldots, q_m$. Let $d_{i1}, d_{i2}, \ldots, d_{in_i}$ be the list of documents returned, by a reference search engine, as results to query q_i. Furthermore, let r_{ij} be the score that document d_j gets as result of query q_i (0 if the document is not a match).*

A document d_j is represented as an m-dimensional query-vector $\overline{d_j} = [\overline{r_{ij}}]^T$, *where $\overline{r_{ij}} \in [0,1]$ is the normalized value of r_{ij}:*

$$\overline{r_{ij}} = \frac{r_{ij}}{\sum\limits_{i \in Q} \sum\limits_{j \in D} r_{ij}}$$

In our model, the underlying reference search engine is treated as a black box, with no particular assumptions on its behavior. Internally, the engine could use any metric, algorithm and document representation. The QV model will be simply built out of the results recalled by the engine using a given query log.

Definition 2. ***Silent documents.*** *A silent document is a document never recalled by any query from the query log Q. Silent documents are represented by* null query-vectors.

The ability of identifying silent documents is a very important feature of our model because it allows us to determine a set of documents that can safely be moved to a supplemental index (see Section 2.2). In this work, we will use the supplemental index as the last resource to answer a query. It will be successfully accessed by our incremental caching algorithm, when the other servers are not able to return valid results.

The $\overline{r_{ij}}$ values form a contingency matrix R, which can be seen as an empirical joint probability distribution and used by the cited co-clustering algorithm. This approach creates, simultaneously, clusters of rows (queries) and columns (documents) out of an initial matrix, with the goal of minimizing the loss of information. To improve performance, empty columns and rows are removed from the matrix before clustering. Documents corresponding to empty columns are put together in an *overflow* document cluster.

Co-clustering considers both documents and queries. We thus have two different kind of results: (i) groups made of documents answering to similar queries, and (ii) groups of queries with similar results. The first kind of results is used to build the document partitioning strategy, while the second is the key to our collection selection strategy (see below).

The result of co-clustering is a matrix $\widehat{P}$ defined as:

$$\widehat{P}(qc_a, dc_b) = \sum_{i \in qc_b} \sum_{j \in dc_a} \overline{r_{ij}}$$

In other words, each entry $\widehat{P}(qc_a, dc_b)$ sums the contributions of $\overline{r_{ij}}$ for the queries in the query cluster a and the documents in document cluster b. We call this matrix simply PCAP, from the LaTeX command `\cap{P}` needed to typeset it.[1] The values of PCAP are important because they measure the relevance of a document cluster to a given query cluster. This induces naturally a simple but effective collection selection algorithm.

3.2 Information-Theoretic Co-clustering

The co-clustering algorithm described in [28] is a very important block of our infrastructure. In this section, we quickly summarize its main features. The paper describes a very interesting approach to cluster contingency matrices describing joint probabilities.

Several IR problems are modeled with contingency matrices. For instance, the common vector-space model for describing documents as vector of terms can be described as a contingency matrix, where each entry is the probability of choosing some term and some document. Obviously, common terms and long documents will have higher probability.

Given a contingency matrix, co-clustering is the general problem of performing simultaneously a clustering of columns and rows, in order to maximize some clustering metrics (e.g. inter-cluster distance). In the cited work, the authors give a very interesting theoretical model: co-clustering is meant to minimize the loss of information between the original contingency matrix and its approximation given by the co-clustered matrix.

They extend the problem by considering the entries as empirical joint probabilities of two random variables, and they present an elegant algorithm that, step by step, minimizes the loss of information. The algorithm is guaranteed to find an optimal co-clustering, meaning that every other co-clustering with the same number of clusters shows a bigger loss of information.

More formally, given a contingency matrix $p(x, y)$, describing a joint probability, and a target matrix size $N \times M$, an optimal co-clustering is an assignments of rows to one of the N row-clusters, and columns to one of the M column-clusters, that minimizes the loss of information between the original matrix $p(x, y)$ and the clustered matrix $p(\hat{x}, \hat{y}) = \sum_{x \in \hat{x}, y \in \hat{y}} p(x, y)$, as follows:

$$\min_{\hat{X}, \hat{Y}} I(X; Y) - I(\hat{X}; \hat{Y})$$

where I is the mutual information of two variables:

$$\begin{aligned} I(X; Y) &= \textstyle\sum_x \sum_y p(x, y) \log \frac{p(x,y)}{p(x)p(y)} \\ I(\widehat{X}; \widehat{Y}) &= \textstyle\sum_{\widehat{x}} \sum_{\widehat{y}} p(\widehat{x}, \widehat{y}) \log \frac{p(\widehat{x},\widehat{y})}{p(\widehat{x})p(\widehat{y})} \end{aligned}$$

By defining:

$$q(x, y) = p(\hat{x}, \hat{y}) p(x|\hat{x}) p(y|\hat{y}) = p(\hat{x}, \hat{y}) \frac{p(x)}{p(\hat{x})} \frac{p(y)}{p(\hat{y})}$$

it can be proved (see [28]) that:

$$I(X; Y) - I(\hat{X}; \hat{Y}) = D(p(X, Y) || q(X, Y))$$

where D is the Kullback-Leibler distance of the two distributions

$$D(p(X, Y) || q(X, Y)) = \sum_x \sum_y p(x, y) \log \frac{p(x, y)}{q(x, y)}$$

[1]The correct LaTeX command for $\widehat{P}$ is actually `\widehat{P}`, but we initially thought it was `\cap{P}`, which incorrectly gives $\cap P$. Hats, caps... whatever.

q has the same the marginal distributions of p, and by working on q we can reach our goal. Given an assignment C^t_X and C^t_Y, the optimal co-clustering can be found by assigning a row x to the new cluster $C_X^{(t+1)}(x)$ as follows:

$$C_X^{(t+1)}(x) = \quad \text{argmin}_{\hat{x}}\, D(p(Y|x)||q^{(t)}(Y|\hat{x}))$$

The co-clustering algorithm in [28] works by assigning each row and each column to the cluster that minimizes the KL distance $D(p(Y|x)||q^{(t)}(Y|\hat{x}))$ for rows, and $D(p(X|y)||q^{(t)}(X|\hat{y}))$ for columns. The algorithm iterates this assignment, monotonically decreasing the value of the objective function. It stops when the change in the objective is smaller than a given threshold.

Unfortunately, the algorithm only find a local minimum, which is influenced by the starting condition. It is possible to build artificial examples where the algorithm reaches only sub-optimal results. Also, it may happen that one of the row (or column) clusters become empty, for a fluctuation in the algorithm. In this case, the algorithm is not able to fill it up anymore, and the cluster is *lost*: the result will have $N - 1$ clusters.

We addressed these issues by introducing a small random noise, which randomly re-assigns a small number of rows and columns if one cluster gets empty or if there is no improvement in the objective function. In our tests, we found that ten iterations are usually enough to reach convergence.

3.2.1 Our Fast Implementation

Our implementation[2] is written in C++, and uses a representation of the sparse matrix based on linked lists: every row (and column) is represented by a list holding the non-empty entries. In order to speed up the scanning of the matrix, we keep in memory a representation of both the matrix and its transposed. While this causes an overhead of 2X in memory, the access is much faster and more direct. Our algorithm takes a matrix as an input, and returns an assignment to a chosen number of row and column clusters. The user can also choose the termination condition (convergence, or number of iterations) and the presence/absence of noise.

We were able to speed up the algorithm by simplifying the main computations. When we iterate on rows, we choose the assignment for the next iteration as follows:

$$\begin{aligned}
C_X^{(t+1)}(x) &= \text{argmin}_{\hat{x}}\, D(p(Y|x)||q^{(t)}(Y|\hat{x})) && \text{definition of D(.)}\\
&= \text{argmin}_{\hat{x}} \textstyle\sum_y p(y|x) \log \frac{p(y|x)}{q^{(t)}(y|\hat{x})} && \text{conditional prob.}\\
&= \text{argmin}_{\hat{x}} \textstyle\sum_y \frac{p(x,y)}{p(x)} \log\left(\frac{p(x,y)}{p(x)} \frac{1}{q^{(t)}(y|\hat{y})q^{(t)}(\hat{y}|\hat{x})}\right) && \text{properties of q}\\
&= \text{argmin}_{\hat{x}} \textstyle\sum_y \frac{p(x,y)}{p(x)} \log\left(\frac{p(x,y)}{p(x)} \frac{p(\hat{y})}{p(y,\hat{y})} \frac{p(\hat{x})}{p(\hat{x},\hat{y})}\right) && \text{properties of q and p}\\
&= \text{argmin}_{\hat{x}} \textstyle\sum_y \frac{p(x,y)}{p(x)} (\log p(x,y) + \log p(\hat{y}) + \log p(\hat{x}) + \\
&\qquad - \log p(x) - \log p(y,\hat{y}) - \log p(\hat{x},\hat{y})) && \text{properties of log}
\end{aligned}$$

where $\hat{y} = \hat{y}(y)$ is the cluster to which the running y belongs. Now we remove terms that do not affect the value of argmin, and we change to argmax:

$$\begin{aligned}
C_X^{(t+1)}(x) &= \text{argmin}_{\hat{x}} \textstyle\sum_y p(x,y)\,(\log p(\hat{x}) - \log p(\hat{x},\hat{y}))\\
&= \text{argmax}_{\hat{x}} \textstyle\sum_y p(x,y)\,(\log p(\hat{x},\hat{y}) - \log p(\hat{x}))
\end{aligned}$$

The values of $p(\hat{x})$, $p(\hat{x},\hat{y})$, and their logarithm, do not change during one iteration. We can precompute them once per iteration. The formulas to compute assignments over y can be simplified the same way.

[2]The initial implementation of the algorithm was written by Fabrizio Silvestri, after a deep mathematical analysis performed in collaboration. The author of this thesis later extended and optimized the core computation.

The resulting algorithm is very fast and can compute a whole iteration of about 170,000 rows and 2.5M columns in 40 seconds. Moreover, the algorithm has the potential to be run in parallel on several machines, and could scale to bigger data sets.

3.3 Applications to Document Clustering

IR systems face the problem of clustering documents, with different goals, including: finding similar documents to be removed, partition the documents into collections, suggest related documents to users. The query-vector model offers important features that result to be beneficial to this task:

- the query-vector model consider only terms that appears in queries, and are likely to be more discriminant than others, and surely more interesting to the users;
- it also naturally considers terms that are co-occurring in queries, this way boosting the weight of short, hot *phrases* in documents, more than the composing terms (e.g. “college application”, “movies by Wood”, “Drawn Together”);
- in the clustering process, high-ranking documents are *heavier* than low-ranking documents: they will have a priority in the clustering choices;
- the query-vector representation is shorter, allowing a faster algorithm;
- it helps coping with the curse of dimensionality, by reducing the space size, in our experiments, from 2.7 million to 190,000.

3.4 PCAP Collection Selection Algorithm

The queries belonging to each query cluster are chained together into *query dictionary* files. Each dictionary file stores the text of each query belonging to a cluster, as a single text file. When a new query q is submitted to the IR system, the BM25 [85] metric is used to find which clusters are the best matches: each dictionary file is considered as a document, which is indexed in the vector space, and then queried with the usual BM25 technique. This way, each query cluster qc_i receives a score relative to the query q, say $r_q(qc_i)$.

This is used to weight the contribution of PCAP $\widehat{P}(i,j)$ for the document cluster dc_j, as follows:

$$r_q(dc_j) = \sum_i r_q(qc_i) \cdot \widehat{P}(i,j)$$

Table 3.1 gives an example. The top table shows the PCAP matrix for three query clusters and five document clusters. Suppose BM25 ranks the query-clusters respectively 0.2, 0.8 and 0, for a given query q. After computing $r_q(dc_i)$, we will choose the collection dc3, dc1, dc2, dc5, dc4 in this order.

An improved strategy could be to use CORI to perform collection selection when PCAP is not able to make a choice. This can happen if the terms in a query are not present among the queries in the training set. In that case, CORI can perform a choice based on all the terms of the collection dictionary, and could choose to access the overflow cluster. We tested this solution initially, but did not bring to any improvement and we did not pursue it. Some results of this will be shown in Section 4.4.

3.5 Collection Selection at the Center of an IR Architecture

We used the ideas presented in the previous sections to design a distributed IR system for Web pages. Our strategy is as follows.

PCAP	dc1	dc2	dc3	dc4	dc5	$r_q(qc_i)$
qc1		0.5	0.8	0.1		0.2
qc2	0.3		0.2		0.1	0.8
qc3	0.1	0.5	0.8			0

$$\begin{array}{lcrcrcccc}
r_q(dc_1) & = & 0 & + & 0.3 \times 0.8 & + & 0 & = & 0.24 \\
r_q(dc_2) & = & 0.5 \times 0.2 & + & 0 & + & 0 & = & 0.10 \\
r_q(dc_3) & = & 0.8 \times 0.2 & + & 0.2 \times 0.8 & + & 0 & = & 0.32 \\
r_q(dc_4) & = & 0.1 \times 0.2 & + & 0 & + & 0 & = & 0.02 \\
r_q(dc_5) & = & 0 & + & 0.1 \times 0.8 & + & 0 & = & 0.08
\end{array}$$

Table 3.1: Example of PCAP to perform collection selection.

First, we train the system with the query log of the *training period*, by using a reference centralized index to answer all queries submitted to the system. We record the top-ranking results for each query. Then, we perform co-clustering on the resulting query-document matrix. The documents are then partitioned onto several IR cores according to the results of clustering.

We partition the documents into 17 clusters: the first 16 clusters are the clusters returned by co-clustering, and the last one holds the silent documents, *i.e.* the documents that are not returned by any query, represented by null query-vectors. A smaller number of document clusters would have brought to a situation with a very simple selection process, while a bigger number would have created artificially small collections.

After the training, we perform collection selection as shown above. The cores holding the selected collections are queried, and results are merged. In order to have comparable document ranking within each core, we distribute the global collection statistics to each IR server. So, the ranking functions are consistent, and results can be very easily merged, simply by sorting documents by their score, as described in Section 2.5.

3.6 Metrics of Result Quality

One of the problems that appeared when we tried to test our approach was to properly measure the quality of the results returned by our system, after collection selection. Due to the nature of data, we do not have a list of human-chosen relevant documents for each query. Our test set (see Section 8.1) includes only 50 evaluated queries. Following the example of previous works [102], we compare the results coming from collection selection with the results coming from a centralized index. In particular, we will use the metrics described in [23].

Let's call G_q^N the top N results returned for q by a centralized index (*ground truth*, using the authors' terms), and H_q^N the top N results returned for q by the set of servers chosen by our collection selection strategy. The *competitive recall at N* for a query q is the fraction of results retrieved by the collection selection algorithm that appear among the top N documents in the centralized index:

$$CR_N(q) = \frac{|H_q^N \cap G_q^N|}{|G_q^N|}$$

The competitive recall, or simply *intersection*, measures the relative size of the intersection of H_q^N and G_q^N. Given N, the intersection at N will be abbreviated as INTER N. If $|G_q^N| = 0$, $CR_N(q)$ is not defined, and we do not include it in the computation for average values. Over a query stream Q:

$$CR_N(Q) = \frac{\sum\limits_{q:CR_N(q)\neq 0} CR_N(q)}{|q : CR_N(q) \neq 0|}$$

Now, when performing queries over Web documents, the user can be interested in retrieving high-relevance documents, even if they are not exactly the top N. When asking for 10 results, s/he could be happy if result #8 is missing, and result #12 is present instead, but less happy if result #100 is substituting it. We adapted the *competitive similarity*, introduced in [23], as follows. Given a set D of documents, we call *total score* the value:

$$S_q(D) = \sum_{d\in D} r_q(d)$$

with $r_q(d)$ the score of d for query q. The competitive similarity at N (COMP N) is measured as:

$$CS_N(q) = \frac{S_q(H_q^N)}{S_q(G_q^N)}$$

This value measures the relative quality of results coming from collection selection with respect to the best results from the central index.[3]. Again, if $S_q(G_q^N) = 0$, the similarity is not defined and do not contribute to the average. Over a query stream Q:

$$CS_N(Q) = \frac{\sum\limits_{q:CS_N(q)\neq 0} CS_N(q)}{|q : CS_N(q) \neq 0|}$$

In our experiments, we measure competitive recall (intersection) and competitive similarity for different values of N.

3.7 Experimental Results

We performed our test using the WBR99 collection. WBR99 is a crawl of the Brazilian web (domain .br), as spidered by the search engine www.todo.com.br in 1999. It also includes a complete query log for the period January through October 2003. As a search engine, we used Zettair[4], a compact and fast text search engine designed and written by the Search Engine Group at RMIT University. We modified it so to implement our collection selection strategies (CORI and PCAP). The reader can find more details of the experimental data in Section 8.1.

To test the quality of our approach, we performed a clustering task aimed at document partitioning and collection selection, for a parallel information retrieval system. We will compare different approaches to partitioning, some based on document contents:

random : a random allocation;

shingles : documents' signature were computed using shingling[16], on which we used a standard k-means algorithm;

URL-sorting : it is a very simple heuristics, which assign documents block-wise, after sorting them by their URL; *this is the first time URL-sorting is used to perform document clustering*; we will show that this simple technique, already used for other IR tasks in [80, 14, 92] can offer an improvement over a random assignment;

[3]In the original paper:

$$CS_N(q) = \frac{S_q(H_q^N) - worst(q)}{S_q(G_q^N) - worst(q)}$$

where $worst(q)$ is the total score of the worst N documents for the query q. In our case, this is negligible, as there are clearly documents completely irrelevant to any query, and their total score would be 0

[4]Available under a BSD-style license at http://www.seg.rmit.edu.au/zettair/.

INTER5

	1	2	4	8	16	OVR
CORI on random	6	11	25	52	91	100
CORI on shingles	11	21	38	66	100	100
CORI on URL sorting	18	25	37	59	95	100
CORI on kmeans qv	29	41	57	73	98	100
CORI on coclustering	31	45	59	76	97	100
PCAP on coclustering	34	45	59	76	96	100

INTER10

	1	2	4	8	16	OVR
CORI on random	5	11	25	50	93	100
CORI on shingles	11	21	39	67	100	100
CORI on URL sorting	18	25	37	59	95	100
CORI on kmeans qv	29	41	56	74	98	100
CORI on coclustering	30	44	58	75	97	100
PCAP on coclustering	34	45	58	76	96	100

INTER20

	1	2	4	8	16	OVR
CORI on random	6	12	25	48	93	100
CORI on shingles	11	21	40	67	100	100
CORI on URL sorting	18	24	36	57	95	100
CORI on kmeans qv	29	41	56	74	98	100
CORI on coclustering	30	43	58	75	97	100
PCAP on coclustering	34	45	58	75	96	100

Table 3.2: Comparison of different clustering and selection strategies, using competitive recall at 5, 10, 20.

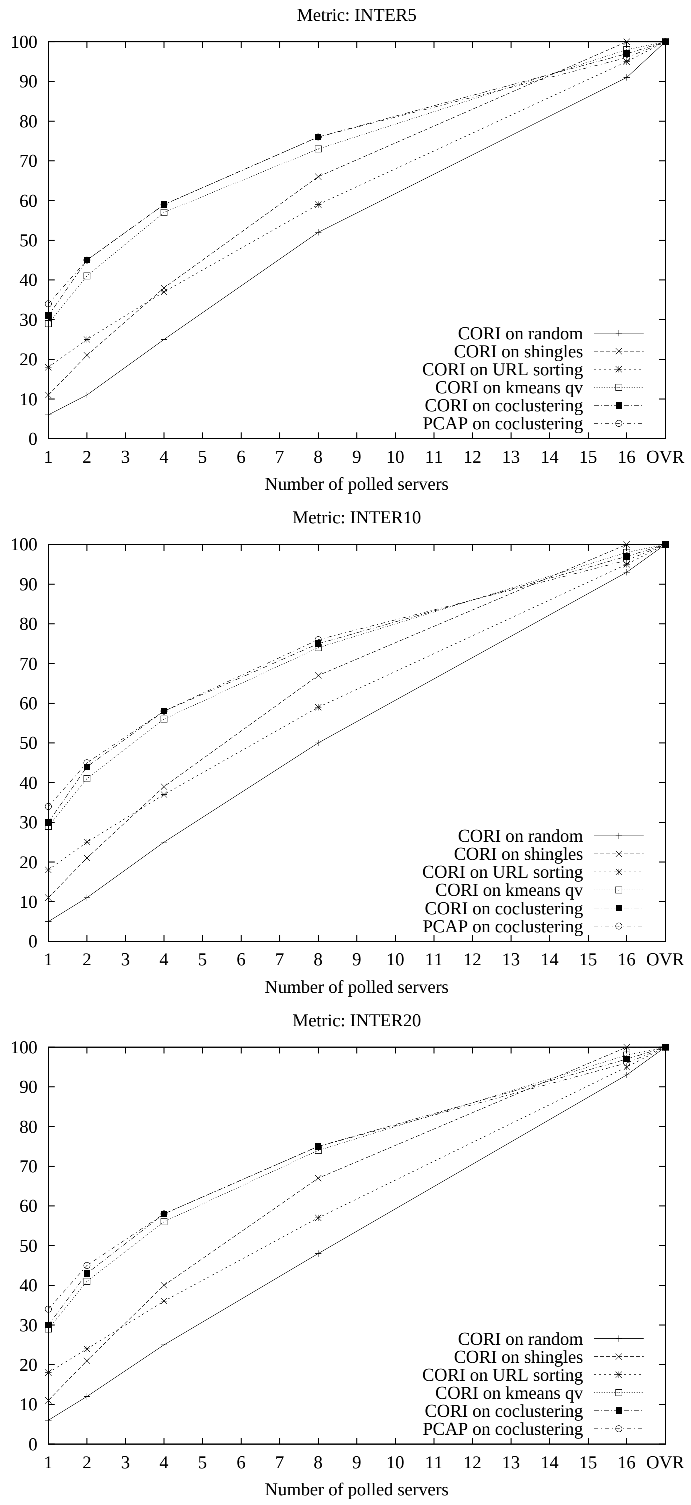


Figure 3.2: Comparison of different clustering techniques.

and other based on the query-vector representation:

k-means : we performed k-means over the document collection, represented by query-vectors;

co-clustering : we used the co-clustering algorithm to compute documents and query clusters.

CORI was used as the collection selection function on these assignments. For the last one, PCAP was also used.

3.7.1 CORI

We implemented CORI as described in a later work [19]. $df(term, C_i)$ is the document frequency, *i.e.* the number of document in collection i holding *term*; $cf(term)$ is the number of collections with documents containing *term*; $cw(C_i)$ is the total word count of collection i; $\overline{cw}$ is the average value of $cw(C_i)$ across the collections; C is the number of collection.

$$\begin{aligned} T = & \quad \frac{df(term, C_i)}{df(term, R_i) + 50 + 150 \cdot \frac{cw(C_i)}{\overline{cw}}} \\ I = & \quad \frac{\log \frac{c+0.5}{cf(term)}}{\log(c+1.0)} \\ b = & \quad 0.4 \\ p(term|C_i) = & \quad b + (1-b) \cdot T \cdot I \end{aligned}$$

In CORI, $p(term|C_i)$ is an estimation of the *belief* that the term is observed in the chosen collection i. For single-term queries, CORI ranked the collection according to the value $p(term|C_i)$. To implement more complex queries, we used the operator described in [19]. For a query of the form $term_1$ *operator* $term_2$ *operator ...* $term_n$, with $p_i = p(term_i|C_i)$ the belief for C_i is:

$$\begin{aligned} bel_{or}(Q) = & \quad 1 - (1-p_1) \cdot (1-p_2) \cdot \ldots \cdot (1-p_n) \\ bel_{and}(Q) = & \quad p_1 \cdot p_2 \cdot \ldots \cdot p_n \\ bel_{not}(Q) = & \quad (1-p) \end{aligned}$$

3.7.2 Query-Driven Allocation vs. Random Allocation

Our first experiments were aimed at showing that our document partitioning strategy boosted the results of any standard collection selection algorithms. In order to do this, we compared the performance of CORI on our query-driven partition with that on a random document allocation.

In the test, our system was trained with the first three weeks of queries. In other words, we recorded the results of the queries submitted to the system for this period, and we used this to build the query-vector representation of documents and to perform the co-clustering algorithms. The first three weeks of the log comprise about 190,000 unique queries.

The record of the training queries takes about 130 MB in a binary representation. The co-clustering algorithm took less than 10 minutes to run on a single machine.

To speed up our experimental phase, we simulated the document allocation by running Zettair on the full index of documents, and by filtering the results according to our allocation. This let us quickly change the allocation and estimate the results without actually moving the data around. As said before, this strategy does not change the ranking of documents on the IR cores because it is possible to send the collection statistics to each core so to normalize the ranking.

Along with this allocation, we considered a random allocation, where documents are assigned randomly to the servers, an equal share to each one.

We used CORI as our collection selection strategy for the two allocations. We tested it with the queries for the fourth week, *i.e.* we used the queries for one week following the training period.

The test comprise about 194,200 queries. In this case, we considered the statistics of repeated queries, because they represent the effective performance as perceived by user.

On random partition, CORI performs rather poorly (see Figure 3.2 and Table 3.2). The results are actually very close to what we would get with a random collection selection . This means that, with a poor document partition strategy, CORI does not perform an effective collection selection. It performs dramatically better with a carefully prepared partition (see below).

3.7.3 CORI vs. PCAP

In this subsection, we measure the performance of our collection selection strategy w.r.t. CORI. In this case, we test two different selection strategies on the same document allocation, as generated by the co-clustering algorithm.

The experimental results (Figure 3.2 and Table 3.2) show that, in the fourth week (the first after training), PCAP is performing better than CORI: the precision reached using only the first cluster is improved of a factor between 11% and 15% (competitive recall improving from 31% to 35% on the first cluster). The improvement is smaller when using two or more partitions, and clearly it gets to zero when all partitions are used. So, while CORI has a very good performance on the partitions we created using the co-clustering algorithm, PCAP has an edge when only one cluster is used.

3.7.4 Other Partitioning Strategies: Shingles

Shingles have already been used in the past for clustering text collections, [16] and for detecting duplicate pages [25, 47]. They have been shown to be a very effective document representation for identifying document similarity: they offer the advantage of being very compact, fast to compute, while being very reliable in identifying similar documents. They can be safely used in our experiments to compute a fast approximation of a clustering based on document contents. Instead of using the full body of the documents, we perform k-means on this new representation.

The k-shingling of a document is a set of sequences of k consecutive words from that document. For example, let us consider a document composed by the following terms: *(a,rose,is,a,rose,is,a,rose)*. The 4-shingling of this document is the set { *(a,rose,is,a), (rose,is,a,rose), (is,a,rose,is)* }. If $S(A)$ is the set of shingles contained by A, we can compute the resemblance of A and B like this:

$$\frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|}$$

The problem is that the intersection is hard to compute, so it has to be estimated. The estimation is done by considering a smaller set obtained by a hashing operation. This can be considered a fingerprint, and it is usually called *sketch*. With sketches the resemblance of two documents become clearly approximate, but very efficient, and it is estimated as:

$$\frac{|F(A) \cap F(B)|}{|F(A) \cup F(B)|}$$

We used the irudiko library[5] to create document sketches, of length 1024, which were then clustered using k-means. Using this technique, the documents were split into 16 + 1 clusters: the 17th cluster is composed of empty documents. We assigned each document cluster to a server of our distributed IR system, and then we used the standard CORI technique to perform collection selection. CORI collects some statistics about the distribution of terms in the collections, and then weights the collections accordingly.

Results are shown in Figure 3.2 and Table 3.2. Shingles offer only a moderate improvement over a random allocation, more evident when a large number of collections, about half, are chosen. Shingles are not able to cope effectively with the curse of dimensionality.

[5] Kindly provided by Angelo Romano.

3.7.5 Other Partitioning Strategies: URL Sorting

In this thesis, for the first time, URL-sorting is used to perform document clustering.

The solution of considering the sorted list of URLs has already been used in [80] to enhance the compressibility of Web Graphs. Web graphs may be represented by using adjacency lists. They are basically lists that contain, for each vertex v of the graph, the list of vertices directly reachable from v. It has been observed that almost 80% of all links are local, *i.e.* point to pages of the same site. Starting from this observation, it is obvious that by assigning closer identifiers to URLs referring to the same site, we will have that adjacency lists will contain the around 80% of IDs very close among them. Representing these lists using a d-gapped representation will thus lead to d-gapped adjacency lists having long runs of 1's. Starting from this assumption, in [80] and [14], authors show that exploiting the lexicographical ordering of URLs leads to an enhancement in performance of Web graphs encoding algorithms.

URL sorting was also used by Silvestri, in [92], where he showed that this method can be used to assign document identifiers to the pages in a collection, resulting in a better compression ratio for the posting lists in the main index.

After sorting the documents by their URLs, we simply allocated documents blockwise. Our experimental results (Figure 3.2 and Table 3.2) show that URL-sorting is actually a good clustering heuristic, better than k-means on shingles when a little number of servers is polled. URL-sorting is even better if we consider that sorting a list of a billion URLs is not as complex as computing clustering over a billion of document. This method, thus, could become the only one feasible in a reasonable time within large scale web search engines.

3.7.6 Other Partitioning Strategies: More on QV

Our precision results improve dramatically when we shift to clustering strategies based on the query-vector representation. Figure 3.2 and Table 3.2 show the results obtained on partitions created with k-means on query-vectors, and clearly proves that our representation greatly helps the clustering process.

Even better does co-clustering behave. Both CORI and PCAP over co-clustered documents are superior to previous techniques, with PCAP also outperforming CORI by about 10%.

3.8 Footprint of the Representation

Every collection selection strategy needs a representation of the collections, which is used to perform the selection. We call dc the number of different collections (document clusters), qc the number of query clusters, t the number of terms, t' the number of distinct terms in the query log, d the number of documents, q the number of queries.

CORI representation includes:

- $df_{i,k}$, the number of documents in collection i containing term k, which is $O(dc \times t)$ before compression,
- cw_i, the number of different terms in collection i, $O(dc)$,
- cf_k, the number of resources containing the term k, $O(t)$.

This is in the order of $O(dc \times t) + O(dc) + O(t)$, before compression, *i.e.* about 48.6 million entries. On the other side, the PCAP representation is composed of:

- the PCAP matrix, with the computed $\widehat{p}$, which is $O(dc \times qc)$,
- the index for the query clusters, which can be seen as $n_{i,k}$, the number of occurences of term k in the query cluster i, for each term occurring in the queries — $O(qc \times t')$.

This is in the order of $O(dc \times qc) + O(qc \times t')$, before compression, *i.e.* about 9.4 million entries. This is significantly smaller (more than 5 times) than the CORI representation as $dc << d$, $t' << t$, and $qc << t$ (see Table 3.3).

dc:	number of document clusters	17
qc:	number of query clusters	128
d:	number of documents	5,939,061
t:	number of distinct terms	2,700,000
t':	number of distinct terms in the query dictionary	74,767

Table 3.3: Statistics about collection representation.

3.9 Supplemental Results in the Overflow Cluster

At the end of the training period, we observed that a large number of documents, around 52% of them (3,128,366), are not returned among the first 100 top-ranked results of any query. This means that, during the training period, no user was invited to access these documents as a result of his/her query.[6]

Separating these documents from the rest of the collection allows the indexer to produce a more compact index, containing only relevant documents that are likely to be requested in the future. This pruning process can speed up the overall performance significantly.

The pruned documents can be stored in another server, used, for instance, when the user browses for low-relevance documents. This is actually done by commercial engines, as seen in Section 2.2. In our test, we showed that the contribution to recall and similarity is very low for the last cluster. In Figure 3.2 and Table 3.2, the change in intersection (competitive recall) when adding also the last cluster is in the range of 2%–3%. More dramatically said, less than 50% of the documents contribute more than 97% of the relevant documents.

The overflow cluster, in our simulation, is always ranked last by the PCAP selection function. This means that it will be accessed with low priority by the load-driven selection (Chapter 4) and the incremental caching function (Chapter 5). The queries in the log do not directly access it, even if we can imagine that users could be interested in obscure results. It is easy to imagine an extended query syntax that triggers the access to the supplemental index.

3.10 Robustness of the Co-clustering Algorithm

The co-clustering algorithm is very fast and robust. We tested the final performance of our collection selection strategy on clustering assignments created with different configurations of the co-clustering algorithm.

3.10.1 Reducing the Input Data

In the full configuration, the co-clustering algorithm has to assign all the non-empty queries and non-empty documents, *i.e.* the queries recalling at least one document, and the documents recalled by at least one query. In our tests, out of 187,080 queries, 174,171 were non-empty; and out of 5,939,061 documents, 2,810,705 were non-empty. So, the full configuration had to manage a sparse matrix of about 174,000 x 2,810,0000, holding the score for the matching documents. The matrix had 16,603,258 non-zero entries.

We reduced the input size by considering only non-unique queries, *i.e.* queries appearing at least twice in the training set. This reduces the number of queries to 68,612, and the number of recalled documents to 1,965,347. The matrix has 6,178,104 non-zero entries.

We reduced the input size in another way, *i.e.* by recording only the top 20 results, rather than the top 100 results for each training query. The number of recalled documents drops to 1,192,976 (the number of queries clearly does not change). The resulting matrix is much smaller, with only 3,399,724 entries.

[6]We record the top 100 results for each query. Clearly, more results could be available after them. Nonetheless, it is verified that only a very small fraction of users go beyond the first result page.

Last, we performed the co-clustering algorithms with the full data set, but using boolean values instead of the scores. In other words, rather than using a matrix in $\mathbb{R}$, we store only the ID of the recalled documents for each query. At the logical level, in this case, the co-clustering algorithm works on a matrix over $\{0, 1\}$. The matrix has the same 16M original entries. Clearly, the co-clustering algorithm is heavily simplified. To determine an assignment, the formula simplifies from:

$$C_X^{(t+1)}(x) = \quad \mathrm{argmax}_{\hat{x}} \sum_y p(x,y) \left(\log p(\hat{x},\hat{y}) - \log p(\hat{x})\right)$$

to:

$$C_X^{(t+1)}(x) = \quad \mathrm{argmax}_{\hat{x}} \sum_{y:p(x,y)\neq 0} \left(\log \mathrm{count}(\hat{x},\hat{y}) - \log \mathrm{count}(\hat{x})\right)$$

where count() is simply the count of the entries assigned to clusters $\hat{x}$ and $\hat{y}$ (or $\hat{x}$ alone). This is due to the fact that any normalization factor does not affect the final value of argmax. Clearly, the formulas for y can be simplified the same way.

Figure 3.3 show the comparison of these four configurations:

- *10iter*: is the initial configurations, with 10 iterations of co-clustering, with 128 query-clusters, 16 document-clusters + overflow, on the full scores;
- *nonunique*: is the result obtained training only on repeated queries (queries appearing at least twice in the training set);
- *top20*: uses only the top 20 results of each query for training;
- *boolean*: builds co-clustering on boolean values.

We show both the competitive recall (intersection) and the competitive similarity at 5 and 50. The reader can clearly see that using only boolean values, or limiting to the top queries, is not affecting the final result. This means that:

- the input matrix is extremely redundant, and the algorithm can use the count of non-zero entries instead of scores to perform a very good assignment;
- queries that appears just once in the query log are not contributing to the training; this can be a very important factor to reduce the training effort.

On the other side, the quality of results degrades by a margin if we store only the top 20 results, instead of the top 100. In this case, the number of non-zero entries in the input matrix dropped severely, as well as the number of recalled documents. The size of the overflow collection is now much bigger, and a big fraction of good documents is lost. For instance, when we measure the intersection at 50, *i.e.* the number of the top-ranking 50 documents that are available when using collection selection, we can see that the overflow cluster holds about 1/4 of the relevant documents when we perform training with only the top 20 results, while this is about 12% when we train with only the repeated queries, and about 5% with full or boolean data.

Also, the recalled documents are not assigned properly. This is because there is much less information if we cut the result list to 20: all documents with low relevance are lost.

3.10.2 Changing the Number of Query-Clusters

Another experiment measures the effect of the number of query clusters in the training process. Clearly, the smaller the number of query clusters we create, the faster the co-clustering algorithm is. In detail, the co-clustering algorithm needs to test the assignment of every query to each query cluster. The running time is linear with the number of query clusters.

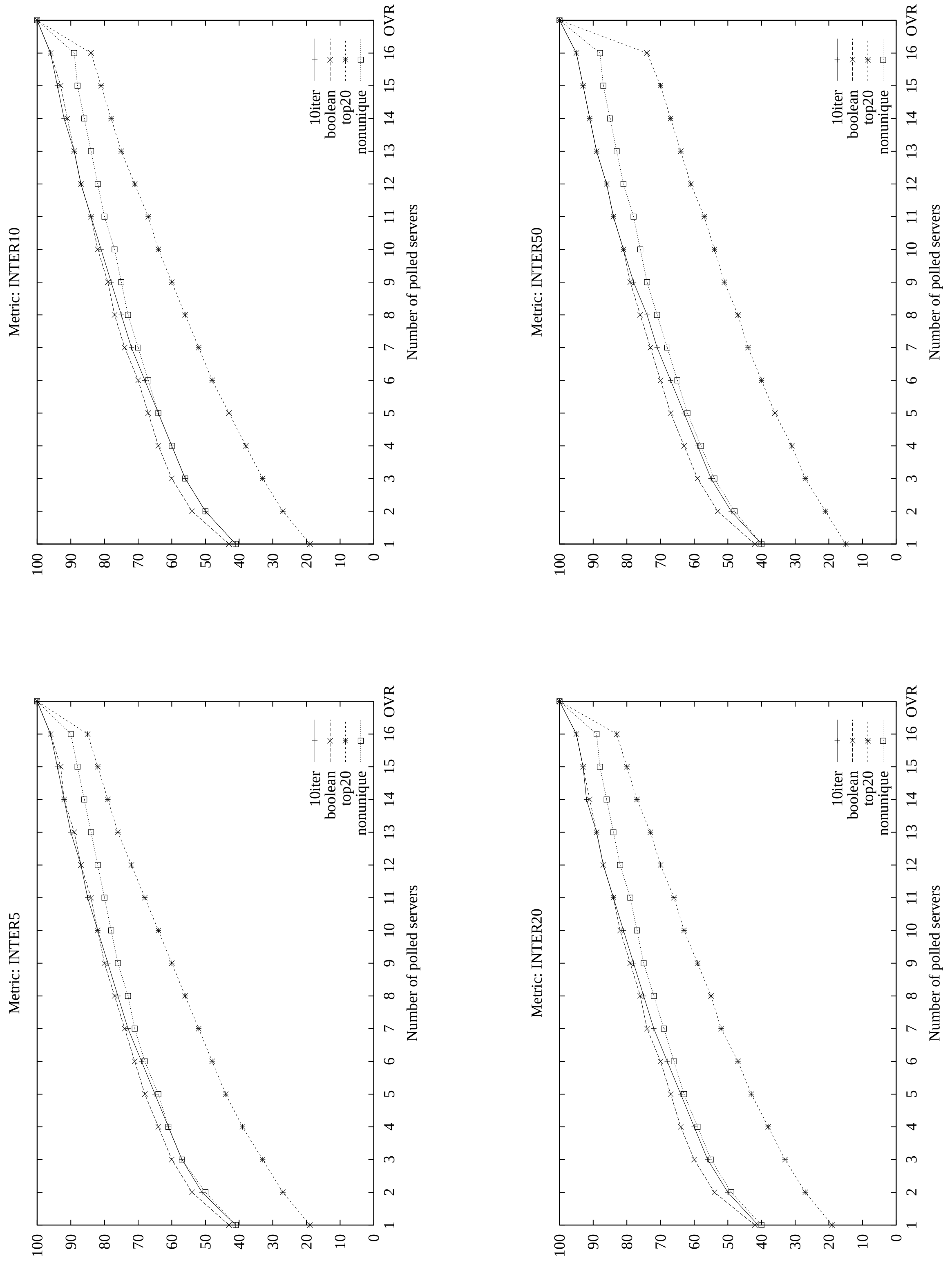


Figure 3.3: Comparison with different training sets. (continues on the next page)

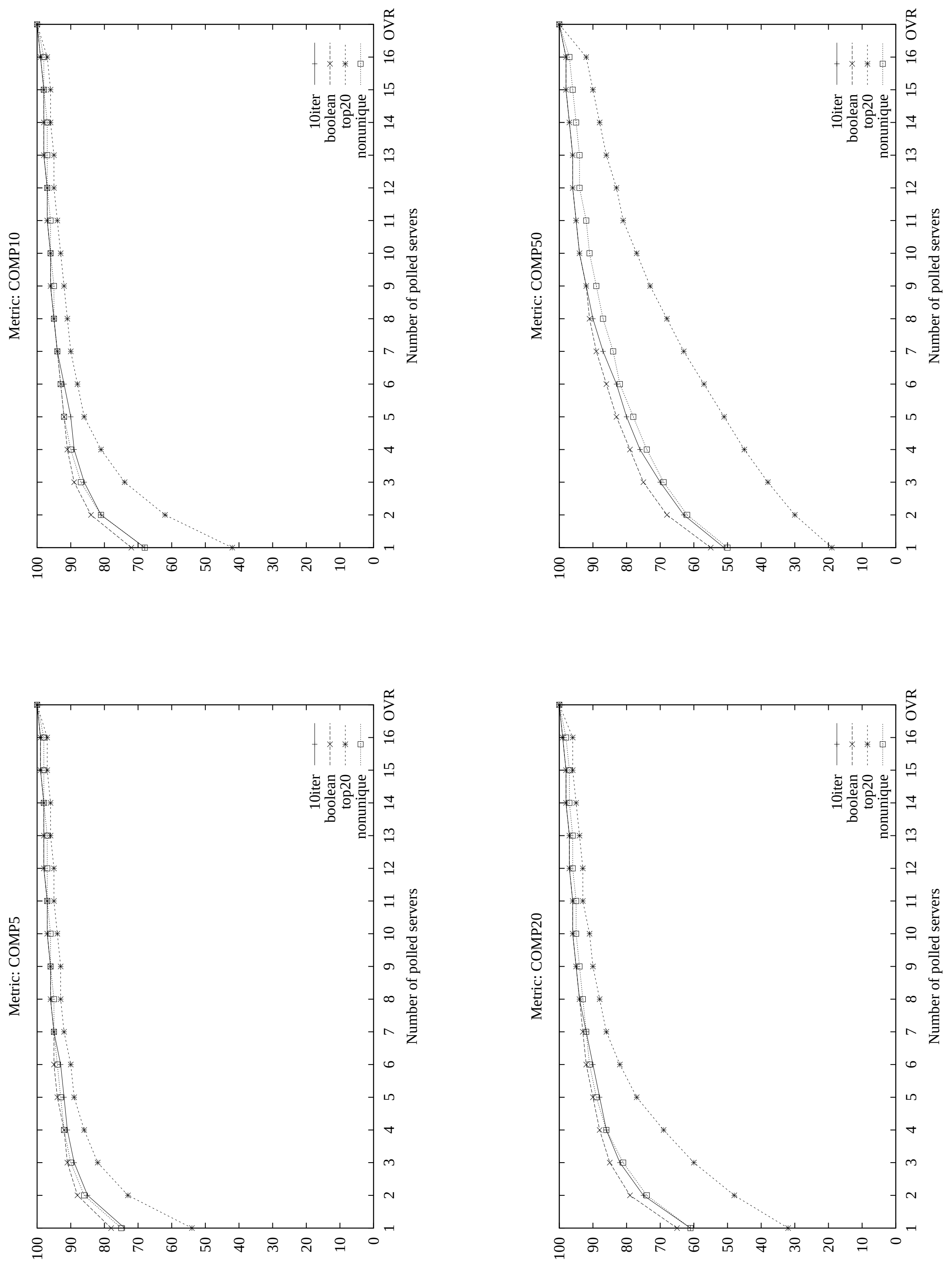


Figure 3.4: Comparison with different training sets.

We measured that 32, 64 or 128 query clusters (the last one, represented by *10iter*) shows a very similar behavior, with 128 appearing to be the optimal choice. When this value is increased, the query clusters lose their identity and the collection selection function performs poorly. Figure 3.5 shows the comparison.

3.10.3 Robustness to Topic Shift

The quality of the results achieved in the test phase clearly depends on the quality of the partitions induced by the training set. If, over time, the type and frequency of queries change heavily, the partitions could turn out to be sub-optimal. This is an important problem in Web IR systems, as the user could be suddenly interested in new topics (*e.g.* due to news events). This phenomenon is usually referred to as *topic shift*.

We keep logging the results of each query, also after the end of the training period, in order to further train the system and also to accommodate any *topic shift*. While CORI, among others, perform selection based on term and document statistics, our model is trained to follow the taste of the users: it learns out of the query log. If the range and the topic of queries change substantially over time, it might happen that documents relevant to the new queries are not clustered together, with the net result of a loss of precision if only the first few IR cores are queried.

To adjust to this possibility, the co-clustering algorithm can be periodically performed by an off-line server and documents can be moved from one cluster to another in order to improve precision and reduce server load. One interesting thing is that there is no need for a central server running a centralized version of the search engine because the rank returned by the individual IR cores is consistent with the one a central server would return. We will discuss this more in detail in Chapter 6.

In this section, we want to measure the robustness of training over time. To do this, we tested our partitioning against queries coming from subsequent weeks. In the previous sections, the tests showed the results when using the queries for the week following the training. Now, we measure the results when using queries for the second, third and fourth week after the training. Results are totally comparable (see Figure 3.7). The users do not perceive a degradation of results over time. This confirms the results of [31] and [7], which showed that training on query log is a robust process over time, because the distribution of frequent queries changes very slowly.

3.11 Summary

In this chapter, we presented a novel approach to document partitioning and collection selection, based on co-clustering queries and documents. Our new representation of documents as query-vectors (QV) allows us to perform, very efficiently, an effective partitioning. We show that partitioning strategies based on the QV model (in our test, k-means on QV and co-clustering) greatly outperforms other techniques based on content and meta-data (URL sorting and shingles), with a jump in the recall quality: with methods based on QV, we can retrieve more than 29% of the top documents that we would retrieve from the full index (only 18% with other methods).

The QV model also induces a very compact representation of the resulting collections. We showed that our selection strategy out-performed CORI by a factor of 10%, with a footprint about 1/5 of that needed by CORI.

Also, we showed a way to select rarely asked-for documents. The process of pruning these documents could improve the performance of the system, because less than 50% contributes more than 97% of the relevant documents: around 52% of the documents (3,128,366) are not returned among the first 100 top-ranked results of any query.

Our experimental results proved that the co-clustering is very robust, and that can be used to create well defined partitions. In our tests, 10 iterations and 128 query clusters are the best configuration for our purpose. It is important to say that, by tuning our co-clustering algorithm, *we are not artificially improving the performance of PCAP over CORI.* In fact, CORI's performance

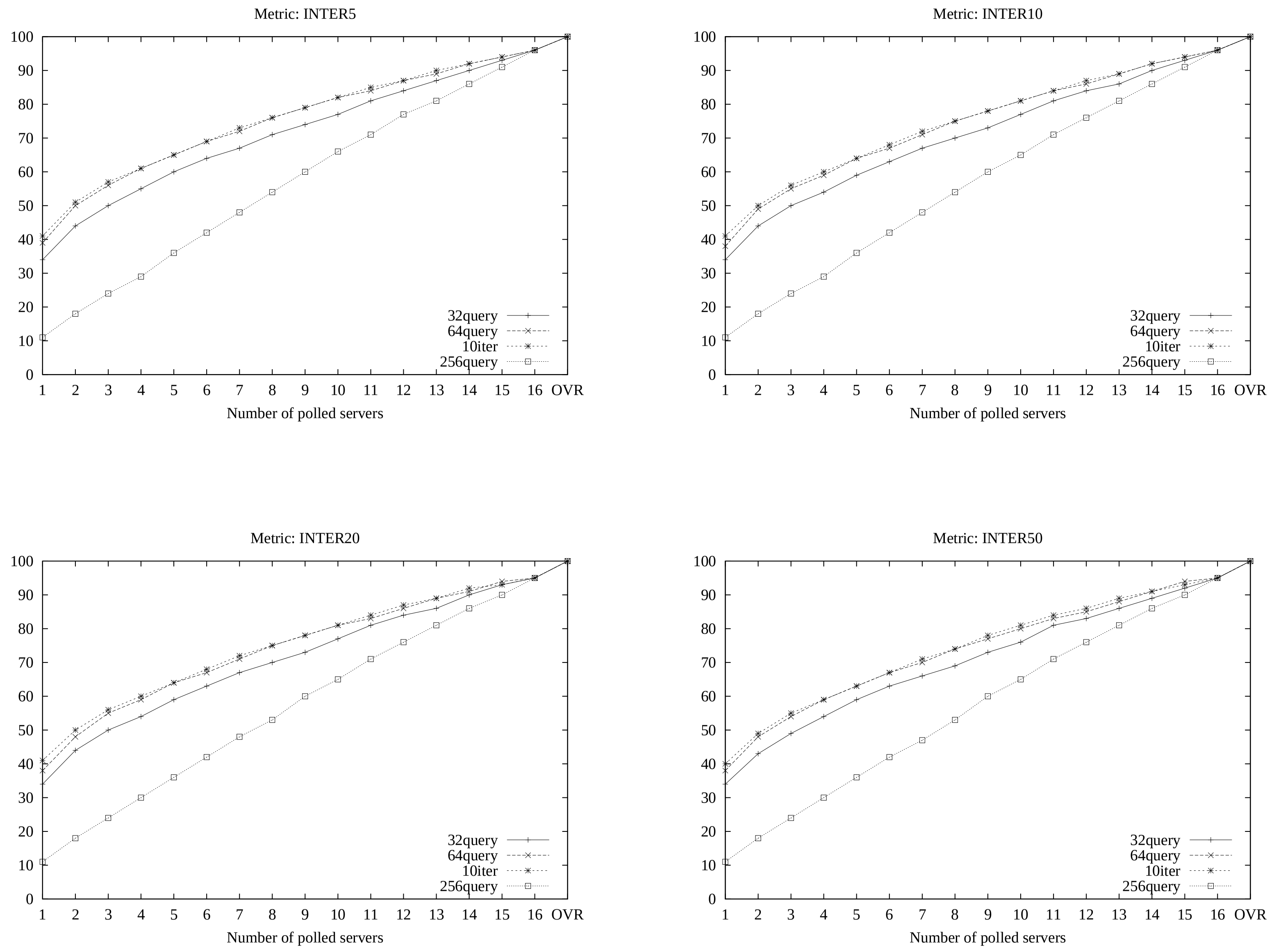


Figure 3.5: Comparison using different number of query clusters. (continues on the next page)

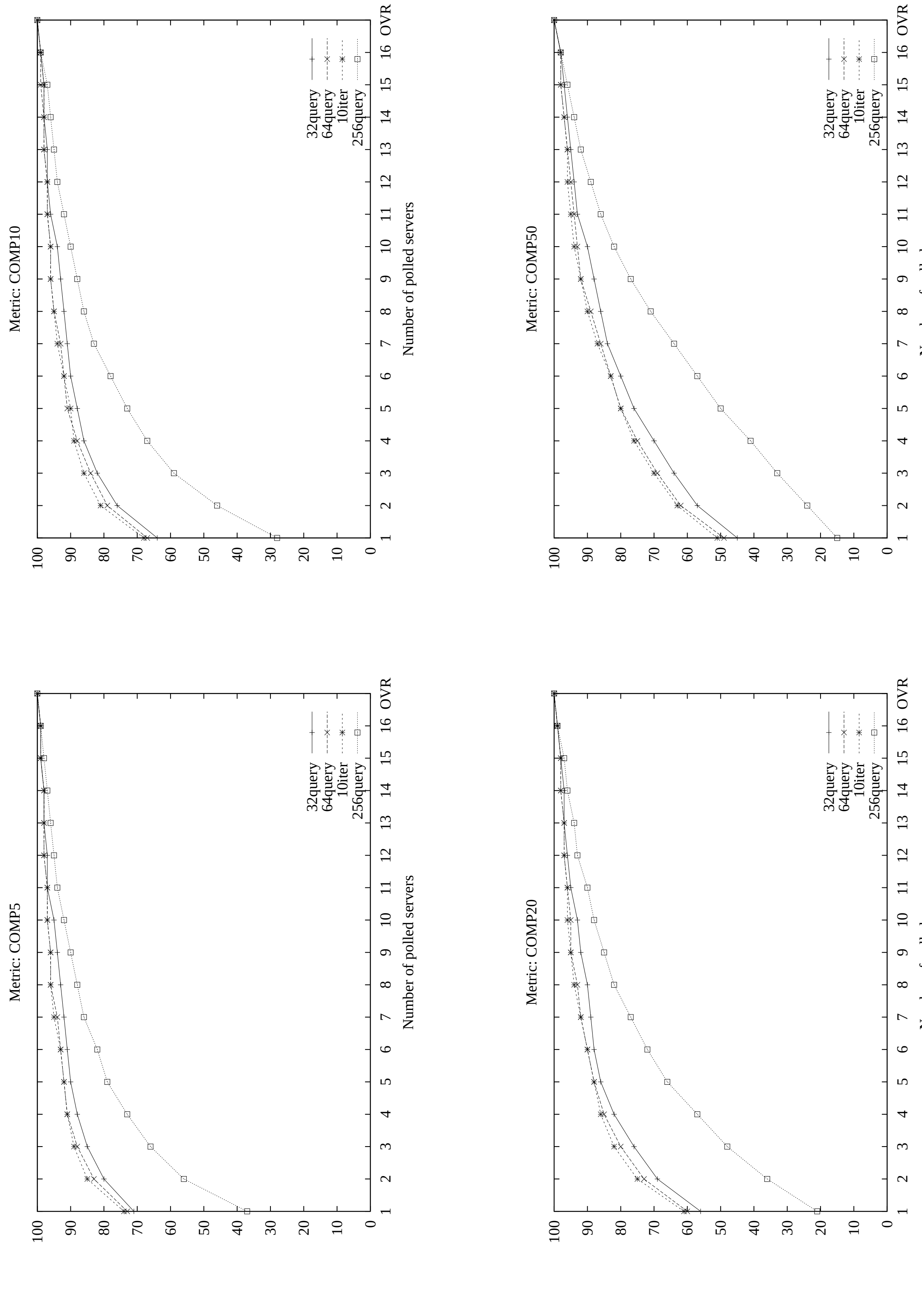


Figure 3.6: Comparison using different number of query clusters.

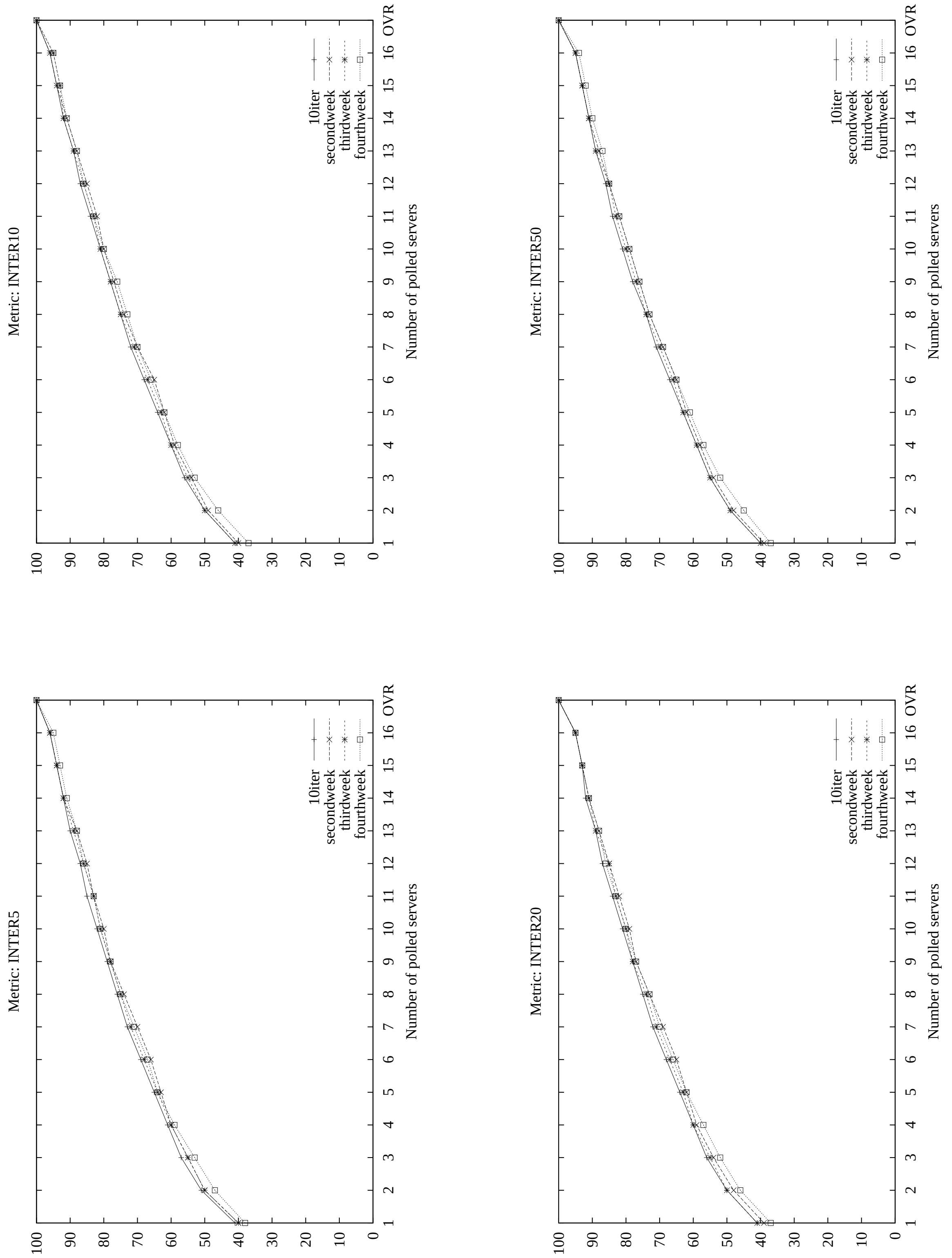


Figure 3.7: Comparison of results over different weeks. (continues on the next page)

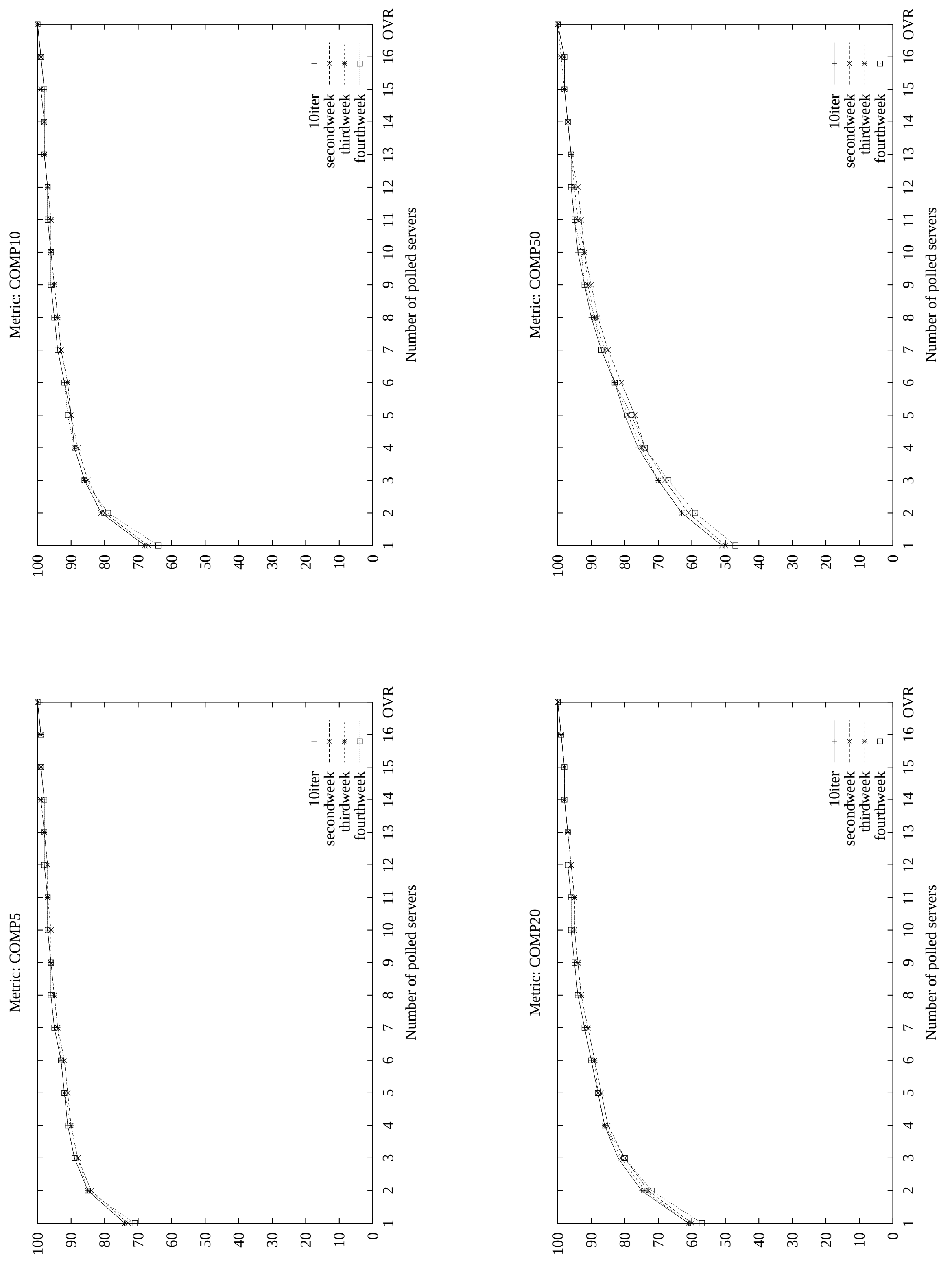


Figure 3.8: Comparison of results over different weeks.

greatly benefits from our partitioning strategy, over competing solutions (shingles, URL sorting, k-means over QV).

This chapter did not address the issues of load balancing and performance of this architecture. In the next chapters, we will see how to use load-driven selection and caching to have a better utilization of the computing servers.

Chapter 4

Load Reduction and Load Balancing

In the previous chapter, we showed how collection selection can be used to reduce the number of IR cores polled to answer a query, while retaining a large fraction of the relevant results. The quality of results, as perceived by the user, is not heavily degraded if the system chooses to poll only a few servers, rather than broadcasting the query to all of them.

So far, we only measured the relative quality of the selection function, by comparing the results coming from the chosen subset of servers with the full set of available results. In this chapter, we will discuss the impact on the overall system load, and how we can address the need of a balanced computing load among the cores. So, we present here a set of strategies to reduce the computing load in a search engine, by exploiting the strength of the QV model in collection selection.

If the search engine broadcasts the query to each server, we have that each of them has to elaborate 100% of the queries. If we choose to query only one of N servers (the most promising one), each server, on average, will elaborate a fraction $1/N$ of the queries. There can be peaks when one server is hit more often than the others, which can raise the fraction it has to elaborate in a time window.

To be more precise:

Definition 3. *For each IR core server in the system, given a query stream Q, and a window size W, we call* instant load at time t $l_R^{i,t}(Q)$ *the fraction of queries answered by server i from the set of W queries ending at t. We define the* peak load for i $l_R^i(Q)$ *as the maximum $l_R^{i,t}(Q)$ over t, and the* maximum load $\bar{l}_R(Q)$ *as the maximum $l_R^i(Q)$ all over the k servers.*

In our experiments, we set W equal to 1000, *i.e.* we keep track of the maximum fraction of queries that hit a server out of a rotating window of 1000 queries. With no cache and no collection selection, we expect to have a maximum load of 100%.

There is a clear trade-off in quality and computing load. If we want to increase the result quality, we can query more servers each time, with a growth in the average load to each server. In this chapter, we are interested in measuring the computing load needed to guarantee a chosen level of result quality, and strategies to reduce it without degrading the retrieval performance.

4.1 Strategies for Load Balancing

In this chapter, we introduce the novel concept of *load-driven collection selection*. To the best of our knowledge, this is the first time a strategy for collection selection is designed to address load-balancing in a distributed IR system.

We have shown in the previous chapter that, by choosing a fixed number of servers to be polled for each query, the system can reach a certain level of competitive recall or similarity. If we want, for instance, to reach 90% competitive similarity at 10, we can poll the best 3 or 4 servers for each query.

This strategy can bring to a strong difference in the relative computing load of the underlying IR cores, if one server happens to be more hit than another one, but still we need to size our computing platform so to be fast enough to handle this load. The less loaded servers will just *wait*, without contributing to answering the query.

In a load-driven collection selection system, the broker can poll more servers, if they are momentarily under-loaded, *i.e.* if their load is below a chosen threshold. This way, the broker tries to exploit the load differences among servers to gather more results.

We compare three query routing strategies:

- *Fixed* $< T >$: it chooses the T most relevant servers, according to a collection selection function, with T given once for all. This allows us to measure the computing power required to sustain a guaranteed number of queried servers (and a guaranteed result quality).

- *Load-driven basic (LOAD)* $< L >$: the system contacts all servers, with different *priority.* The most relevant server (the one with top priority) will be polled and will have to serve the query if, chosen a window size W, its current load $l_R^{i,t}(Q)$ is below L. Each server i has a linearly decreasing priority p_i from 1 (for the most relevant) to 1/16 (for the least relevant). The broker will poll a server i if: $l_R^{i,t}(Q) \times p_i < L$. This is done to *prioritize* queries to the most promising servers: if two queries involve a server s, and the load in s is high, only the query for which s is very promising will be served by s, and the other one will be dropped. This way, the overloaded server will be not hit by queries for which it is only a second choice.

  We set the load threshold L to match the values measured with the previous strategy. For instance, if we measure that with with FIXED $< 4 >$, the peak load is 40%, we compare it with LOAD $< 40\% >$.

- *Load-driven boost (BOOST)* $< L, T >$: same as load-driven, but here we contact the first T servers with maximum priority, and then the other ones with linearly decreasing priority. By boosting, we are able to keep the lower loaded servers closer to the load threshold. Boosting is valuable when the available load is higher, as it enables us to use the lower loaded servers more intensively. If the threshold L is equal to the load reach by FIXED $< T >$, we know that we can poll T servers every time without problems. The lower-priority will be dropped when we get closer to the threshold.

The interaction with a caching system is very interesting. If we have a fixed selection, the cache will store the results from the chosen T servers. In case of a hit, the stored results will be used to answer the query. In the case of load-driven selection, the cache stores the results of the servers that accepted the query. As just said, many servers can be dropped if they are overloaded or the priority is low. In the next chapter, we will discuss a strategy to improve the results stored by the caching system over time.

In our architecture, the broker needs to model the computing load of every IR core in order to determine the best routing for a query. There a several ways to do so. The first and simplest strategy is to assume that the time needed by any server to answer any query is fixed. The broker can model the load simply by counting the number of queries that are forwarded to each server within a time window. This is the model used in this chapter.

More complex models (*e.g.* the one shown in [67]) may use the number of terms in the query, the posting list length, the disk latency. In this case, the broker needs to keep track of the estimated computing load of every server. Still, the variety of IR models are often inadequate to properly model the complex interaction of the query stream with all the cache levels or with the disk. In Chapter 7, we will show a new model, which pragmatically uses the measured timing of queries on a set of real local indices holding the document collections. The extended model properly manages the high variance in the response time, and confirms our results with data closer to reality.

Num. of servers	1	2	3	4	5	6	7	8
Measured overall peak	15.6	18.0	20.0	24.7	29.2	33.7	38.4	44.0

Num. of servers	9	10	11	12	13	14	15	16
Measured overall peak	47.7	51.5	53.5	55.6	56.5	57.6	58.2	58.2

Table 4.1: Peak load with fixed collection selection

4.2 Experimental Setup

In our tests, we simulated the broker's role of modeling the load of the IR cores, doing some assumptions.

- The computing load of an IR core to answer one query on its local index is counted as 1, independently from the query and the queried server. This is justified by the fact that the sub-collections are of similar size. The overflow collection, which holds about 50% documents, is considered to be 16 times slower than the other clusters.
- We measure the computing load of a server as the number of queries that are forwarded to it within a window of the query load. Unfortunately, we could not use information on query timing because they are not available for our query log. So, a server contacted for all query in a given window, has 100% load. A server contacted by one query out of two, has 50% load. We imagine to have very big machines that can handle the full load, and then we cap the maximum load so to use smaller, low-fat machines.
- The search engine front-end can merge the results coming from the servers and rank them. This can be done just by computing the global term statistics at the end of the indexing (see Section 2.5).
- The broker has a LRU cache of 32,000 results.

Our simulation works as follows. Each query is submitted to a centralized search engine that indexes all the documents. We used Zettair on the WBR99 collection.

Using the clustering results of the co-clustering algorithm, we know how many results from the centralized index are assigned to each sub-collection. We use our collection selection strategy to rank the document clusters, and then we choose to which servers we should broadcast the query, according to our different strategies. At this point, we can record an increase in the load for the servers hit by the query, and we compute the quality metrics.

4.3 Experimental Results

In this test, we measure what is the load peak when a fixed number T of servers is queried every time, using a LRU cache of 32,000 elements. For each server, the peak is measured as the maximum fraction of queries, from a window of 1000, that are sent to it. For the overflow cluster, each query was supposed to be 16 times slower.

The overall peak is the maximum among the server peaks. This measure is the reference for sizing our system. If we choose a certain T, we need beefed-up, fat machines able to absorb the overall computing peak.

Table 4.1 shows the peak load of our system for different values of T. $T = 16$ is the case when all regular collections are polled (but not the overflow cluster).

We use these data to test our two load-driven strategies. In Figure 4.1, we compared the different strategies according to our different metrics. While the basic load-driven strategy (LOAD) outperforms FIXED for smaller computing loads, it is not able to fully exploit the remaining computing power from the less loaded servers, because it is, in a sense, too conservative. BOOST is able to reach extremely good results. With a computing load equivalent to that of FIXED $< 1 >$, BOOST improves the intersection at 50 from 40% to 60%, and the competitive similarity from 51% to a whopping 76%. In other words, by sizing the system so that it can guarantee the access to the most relevant server for each query, the user perceives a 75% quality for the top 50 results. This goes to 90% with the computing load to guarantee 5 servers.

4.4 Comparison with CORI

In the previous chapter, we have shown that the PCAP collection selection strategy outperformed CORI for the collection selection task. Here, we want to compare the effect on load reduction of PCAP with the standard CORI collection selection algorithm. Our comparison will use the same partitions, created with co-clustering. Subcollections will be ranked using CORI. Still, we can use different routing strategies, after the subcollections have been ranked.

Also, we are interested in using CORI to complement PCAP. When a query is composed of terms not present in the training log, the ranking returned by PCAP will be not significant (with the net result of a random selection). CORI can be used in this case, because it is designed to have complete knowledge of the dictionary of the collection.

The comparisons are shown in Figure 4.3.[1]

When the load cap is smaller, *i.e.* in more critical conditions, the results than can be reached using CORI instead of PCAP are slightly inferior, because PCAP is more able to identify the most promising clusters. When the load cap increases, CORI gets a small advantage, because apparently is more able to identify (and exclude) the worst clusters. Differences are in, any case, smaller than 3% when measuring the competitive recall at 100 (INTER100).

The reader must remember however that the PCAP representation of collections has a footprint equal to about 1/5 of CORI, so one could use the extra space to save extra statistics, or to use more than 100 results in the building phase of PCAP. This tuning is clearly part of the engineering trade-off of a real world search engine.

On the other side, CORI might be used to complement PCAP. When a query is composed of terms not present in the training log, PCAP cannot distinguish among the various collection. We implemented a mixed PCAP+CORI strategy, that resorts to standard CORI for the unknown query terms. This offers a tiny improvement to the simple PCAP strategy (less than 1% competitive recall), which does not seem to justify the extra complexity.

4.5 Other Partitioning Strategies

We wanted to test the quality of different clustering strategies. As a benchmark, we took an approach based on content (shingles), and one loosely based on the information of the web structure (URLs). The QV document partitioning strategy is clearly competing with other strategies that implements different approaches to the grouping choice.

In our tests, we computed the shingle representation of our document base, over which we performed a standard k-means algorithm. We created 16 document clusters. The documents with empty body were stored in the overflow cluster.[2] Approximately, each collection is twice as large as in the previous case, because we do not prune documents. We perform query routing by ranking collections with CORI (PCAP is clearly not available in this case), and by using load-driven boost.

The system has the same 32000-entry LRU cache and we still measure the load in the same way (fraction of queries in a window), even if this favors shingles (each server has a much larger collection). Even with this advantage, the partitions created by shingles are not very good for the task at hand, and show a smaller recall for critical load levels (under 9) (see Figure 4.3).

[1]For this comparison, we had to use a slightly different infrastructure, and only the INTER100 metric is available.

[2]Some of these documents are empty due to the presence of graphics or other multimedia contents, stripped before indexing. They could be recalled because of their URL or because of incoming links.

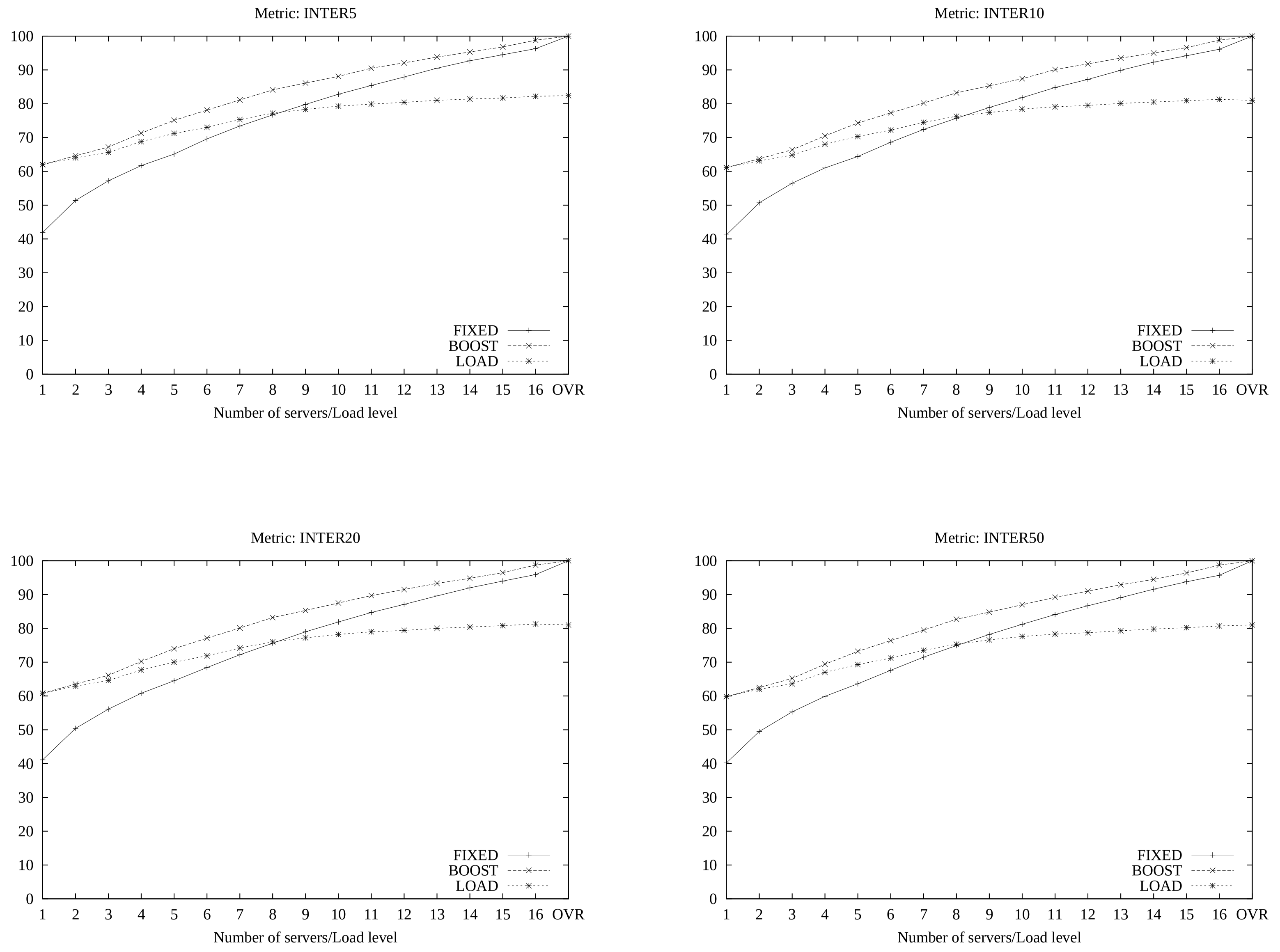


Figure 4.1: Results quality, varying the query routing strategy. For the fixed strategy, we vary the number of polled servers. For load-driven, the load cap. For boost, the combination load-cap/number of servers. (continues on the next page)

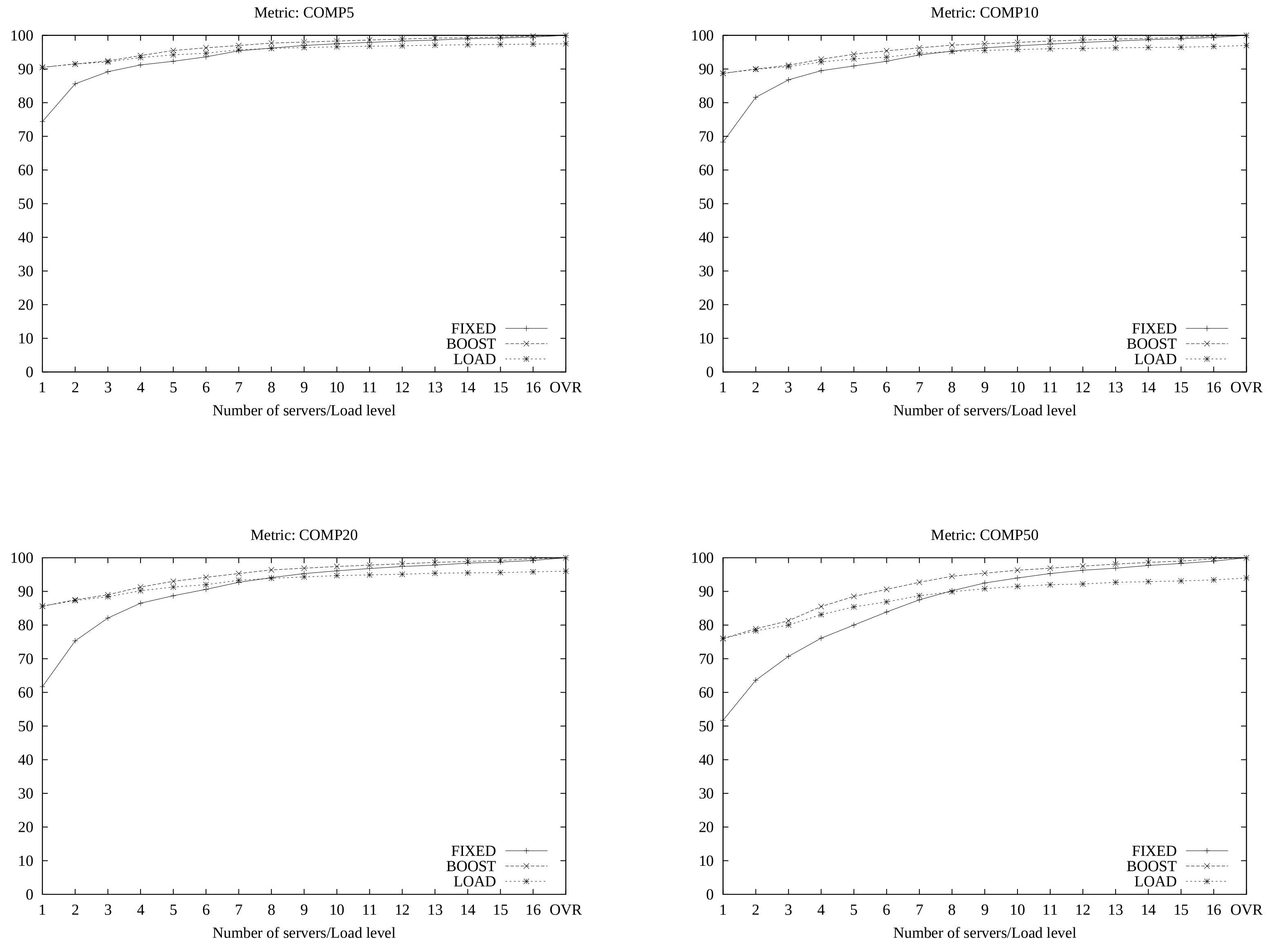


Figure 4.2: Results quality, varying the query routing strategy. For the fixed strategy, we vary the number of polled servers. For load-driven, the load cap. For boost, the combination load-cap/number of servers.

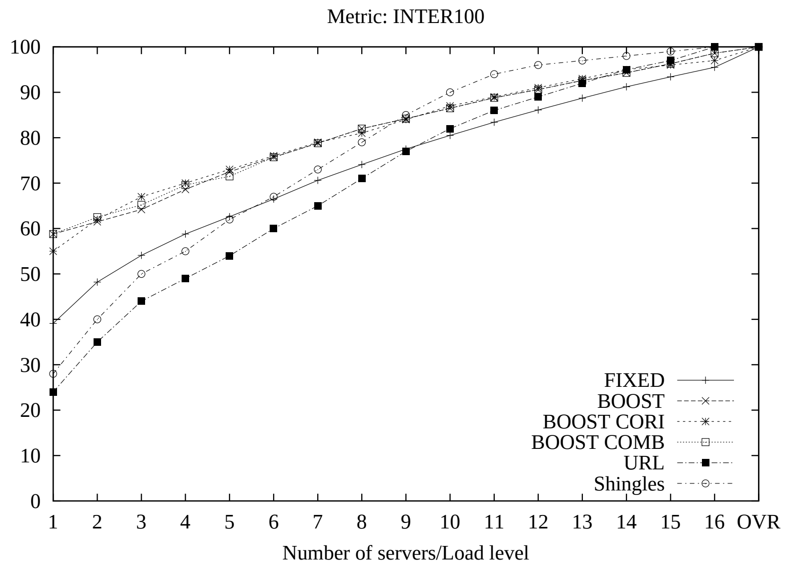


Figure 4.3: Result quality with different partitioning and selection strategies.

We tested our system also over an allocation generated with URL sorting, as discussed in Section 3.7.5. URL-sorting was also used to perform document clustering: after sorting, we simply allocated documents blockwise. The experiment was run by creating 17 clusters of documents, with the same LRU cache, ranking procedure (CORI), selection strategy (incremental boost) and load levels. Results are shown in the same figure. URL-sorting falls well behind PCAP in terms of recall.

4.6 Summary

In this chapter, we show how collection selection can heavily reduce the computing load of a search engine. By using a smart strategy for document partition, collection selection can find very good results to a given query using only a limited number of servers: the best three servers cover more than 50% of the results you would get from a centralized index (FIXED $< 3 >$ with INTER50 metric). Load-driven collection selection can exploit the idle servers: we can reach about 60% recall (INTER50) with only 15% load or more than 70% with only 24% load. Using the fixed strategy, the system would need to handle 24% and 38% load instead.

Let's try to restate the results in a more dramatic way. After pruning about half of the collections (moved to a slow overflow server), we partitioned the collection into 16 pieces: every server is holding about 3% of the collection. If these servers are loaded up to 15%, *i.e.* a peak of 150 queries in a window of 1000, less than 1 out of 6, we can cover 60% of the results coming from the full index. Each server is holding a very small part of the collection and is contacted with low frequency. One can easily imagine to map several IR cores onto one machine, or to use a smaller machines, or to index many more pages.

We believe this strategy could be used successfully to reduce the computing load in a distributed search engine, allowing to use lighter, low-fat machines, reducing costs, power consumption and global warming!

In the next chapter, we will see how this technique can be integrated with caching to extract even higher quality results with the same very limited load.

Chapter 5

Incremental Caching

In the previous chapter, we anticipated the fact that, in a system based on collection selection, the caching system has a complex interaction with the selection function and the load-driven routing. In fact, the cache can only store the results coming from the selected servers. If some servers are not polled because they are not highly relevant and/or they are heavily loaded, all the potential results stored there will not be available to the broker for caching. This means that an entry in the cache could be degraded, and it will be returned every time in case of a hit.

In this chapter, we introduce the novel concept of *incremental caching*. In case of a hit, the broker will try to poll more servers among the ones that were not polled at the time of the first request of a given query. Over time, the broker will try to poll more and more servers, with the result of storing *non degraded* results, *i.e.* the full set of results from all the servers, for the repeated queries.

While traditional caching system are aimed exclusively at reducing the computing load of the system and at improving the overall system efficiency, the incremental caching is also able to *improve the result quality*. In fact, every time there is a hit, more results are added to the stored entry, and the user will get an answer of higher quality. The incremental caching, in other words, is addressing *both the computing load and the result quality*.

5.1 Model of Incremental Caching

To formalize in detail our incremental caching system, we need to redefine the type of entries stored in a cache line.

Definition 4. *A* query-result record *is a quadruple of the form $< q, p, \overline{r}, \overline{s} >$, where: q is the query string, p is the number of the page of results requested, $\overline{r}$ is the ordered list of results associated to q and p, $\overline{s}$ is the set of servers from which results in $\overline{r}$ are returned. Each result is represented by a pair $< doc\_id, score >$.*

As anticipated, if we cache an entry $< q, p, \overline{r}, \overline{s} >$, this means that only the servers in $\overline{s}$ were selected and polled, and that they returned $\overline{r}$. Also, since in an incremental cache the stored results might be updated, we need to store the score (along with the document identifier) to compare the new results with the old ones.

We can see the first difference between a traditional and an incremental cache: results in an incremental cache are continuously modified by adding results from the servers that have not been queried yet. The set $\overline{s}$ serves to this purpose and keeps track of the servers that have been contacted so far.

For a caching system, a key measure is the hit-ratio:

Definition 5. *For a query stream Q, and an incremental caching policy R, the hit-ratio $h_R(Q)$ is the number of queries answered by the incremental cache divided by the number of queries in Q.*

The parameters that will help us in measuring the performance of an incremental caching system, are clearly;

1. the *hit-ratio* $h_R(Q)$;
2. the *competitive recall and competitive similarity* $CR_N(Q)$ and $CS_N(Q)$ (see Section 3.6);
3. the *maximum load* of the system $\bar{l}_R(Q)$ (see Chapter 4).

5.2 Different Types of Incremental Caches

We will present a list of possible cache approaches, starting from a very simple system to more complex incremental caches. With this, we would like to explore the design-space of this important component of an IR system.

5.2.1 No Caching, No Collection Selection

This policy consists simply of forwarding each query to each IR core, with no caching at all.

```
1. Forward the query to all the $k$ servers;
2. collect results $\overline{r}$;
3. return $\overline{r}$.
```

5.2.2 No Caching, Selection Policy $\rho(q)$

This policy applies collection selection, in order to reduce the computing load, but does not perform caching. The system selects a subset $\overline{s}$ of the k IR core servers using the collection selection strategy $\rho(q)$, and then forwards them the query. The selection policy can be as simple as a random choice, or more complex as CORI or our PCAP strategy. Both could be implemented using load-driven routing.

```
1. Select $\overline{s}$ using $\rho(q)$;
2. forward the query to $\overline{s}$;
3. collect results $\overline{r}$ from $\overline{s}$;
4. return $\overline{r}$.
```

5.2.3 Caching Policy P, No Collection Selection

This caching policy corresponds to apply a standard cache (using replacement policy P) to a pure document-partitioned distributed IR system. If the results for a query are not found in cache, then the query is forwarded to all the servers.

```
1. Look up $(q,p)$.
2. If found $< q, p, \overline{r} >$:
    (a) update the cache according to $P$ (e.g. updating the hit-time);
    (b) return $\overline{r}$.
3. If not found:
    (a) forward the query to all servers;
    (b) let $\overline{r}$ be the result set;
    (c) select and remove the best candidate for replacement;
    (d) store $< q, p, \overline{r} >$;
    (e) return $\overline{r}$.
```

The cache is hosted by the broker, which uses it to filter queries. Clearly, a very big result cache could have a non-negligible access cost, and this needs to be carefully considered when the IR system is engineered. Usually, a cache is considered beneficial, because the little loss in latency time is greatly compensated in increased throughput.

Please note that, the caching policy could choose *not* to remove any stored entry and drop the new results instead, *e.g.* if the entry is believed to be a one-timer. This is partially performed by ARC [70] and SLRU [54, 69], but can be done in a more systematic way by heuristics based on query log analysis. This is planned to be part of our future work.

5.2.4 Caching Policy P, Selection Policy $\rho(q)$

In this case, the cache is storing the results coming from the selected servers, for the query q. Subsequent hits are not changing the result set.

```
1. Look up $(q,p)$.
2. If found $< q, p, \overline{r} >$:
    (a) update the cache according to $P$ (e.g. updating the hit-time);
    (b) return $\overline{r}$.
3. If not found:
    (a) forward the query to a subset of servers, $\overline{s}$, selected as $\rho(q)$;
    (b) let $\overline{r}$ be the result set;
    (c) select and remove the best candidate for replacement;
    (d) store $< q, p, \overline{r} >$;
    (e) return $\overline{r}$.
```

5.2.5 Incremental Caching Policy $\widetilde{P}$, Selection Policy $\rho(q)$

This approach differs from the previous because, at each hit, the system adds results to the cached entry, by querying further servers each time.

```
1. Look up ($q$,$p$).
2. If found $< q, p, \overline{r}, \overline{s} >$:
    (a) forward the query to the appropriate subset of servers $\rho(q)$ (excluding $\overline{s}$);
    (b) add them to $\overline{s}$;
    (c) add the results to $\overline{r}$, and update $\overline{s}$;
    (d) update the cache according to $\widetilde{P}$;
    (e) return $\overline{r}$.
3. If not found:
    (a) forward the query to a subset of the $k$ servers, $\overline{s}$, selected as $\rho(q)$;
    (b) let $\overline{r}$ be the result set;
    (c) select and remove the best candidate for replacement;
    (d) store $< q, p, \overline{r}, \overline{s} >$;
    (e) return $\overline{r}$.
```

Once chosen the main caching structure, several options are still available for the collection selection approach and the cache structure.

5.2.6 Load-driven Collection Selection

There are two main ways of performing collection selection, as shown in the previous section: *fixed* or *load-driven* (possibly with *boost*). In the second case, clearly, the broker has to model the dynamic load of each server, *e.g.* by counting the number of queries forwarded to each in a time window. The priority is used to prevent queries from overloading low-relevance servers, leaving them free to answer when the priority is higher.

When a load-driven selection is used, only the results coming from the polled servers are available for caching. In case of a subsequent hit, the selection function will give top priority to the first server that was not polled before.

Let's say, for example, that for a query q, the cores are ranked in this order: $s4$, $s5$, $s1$, $s3$ and $s2$. $s4$ will have priority 1. Now, let's say that only $s4$ and $s1$ are actually polled, due to their lower load. When q hits the system a second time, $s5$ will have the top priority, followed by $s3$ and $s2$. Their results will be added to the incremental cache.

5.2.7 Replacement Policy and Static Cache

Several choices are available for the replacement policy. The algorithms for cache management have been widely studied in literature, but due to limited resources we chose to test only two very representative strategies. The first is the simple LRU cache, and the second is the state-of-the-art Adaptive Replacement Cache (ARC [70]). ARC manages two LRU caches, one for entries with exactly one hit in the recent history, and one for entries with multiple hits. The size of these two caches, combined, is equal to the chosen cache size, but the former can grow at the expenses of the latter if the request streams shows a dominance of recency vs. frequency, *i.e.* most requests are for recently requested entries instead of common entries. If common entries are dominant, the second cache can grow at the expenses of the first one.

The system could also use the Static-Dynamic Cache (SDC) strategy introduced by Fagni *et al.* [31], and reserve a part of the resources to a static cache. The static entries are chosen as the most frequent in the training set, and can be pre-loaded with the full set of results (*i.e.* results coming for all servers). SDC is orthogonal to the replacement policy, and can be used with both LRU and ARC.

All these choices are orthogonal and can be combined in any variety of ways, generating a very large design-space (see Table 5.1).

5.3 Testing ARC and SDC

Several caching strategies have been proposed for web search engines. In particular, we tested two main algorithms, the standard LRU replacement policy, and ARC. Both can be used in combination with SDC.

We configured our experimental infrastructure to measure the hit ratio. We tested both LRU and ARC with different cache size. Part of the cache (ranging from 0% to 90%) was devoted to a static cache. The entries of the static cache were chosen as the most frequent queries in the training set. We tested the system with a query log of four weeks, starting right after the training. In our configuration, the system performs an initial training, where documents and queries are co-clustered, and query frequencies are measured, and then accept queries from the users without re-tuning the caches. This measures the robustness of our training.

The data reported in Table 5.2 show a little advantage of ARC over LRU (less than 1%), and show that the use of SDC can bring a small improvement with big cache sizes: with bigger caches, the best results are achieved if a larger part is devoted to a static cache. Anyway, in all configurations, except with very small caches, about 1% is gained by using a small static cache (about 10-20%). On big caches, a larger fraction can be devoted to static lines.

Differences between ARC and LRU are smaller when a big fraction of the cache is devoted to static entries. This is because the static part of a LRU cache plays the role of the second part of the cache (frequent entries) in ARC.

Some important advantages of SDC, not measured by the hit ratio, are:

- static entries are never changed: their management is simplified and they can be accessed faster by read-only threads;
- its implementation is very simple, using an efficient hash table;
- we can pre-compute the full set of results from all servers in the system (skipping collection selection), this way improving the final result quality.

All these considerations push for the adoption of a static cache in any configuration. On the other side, the little performance gains of ARC over LRU do not seem to justify its extra complexity.

5.4 Experimental Results

In this section, we give a full measurement of the performance of the combined approach a load-driven collection selection and incremental caching. The tests were performed on the WBR99 data sets.

We tested five system configurations, with a LRU cache of 32000 entries.

Caching	Yes / No / Incremental result update
Collection Selection	CORI / PCAP / Random / None
Routing Strategy	Fixed number (choosing N)
	/ Load driven (choosing load)
Replacement policy	LRU / ARC (+ SDC)

Table 5.1: The design-space for a collection selection system with caching.

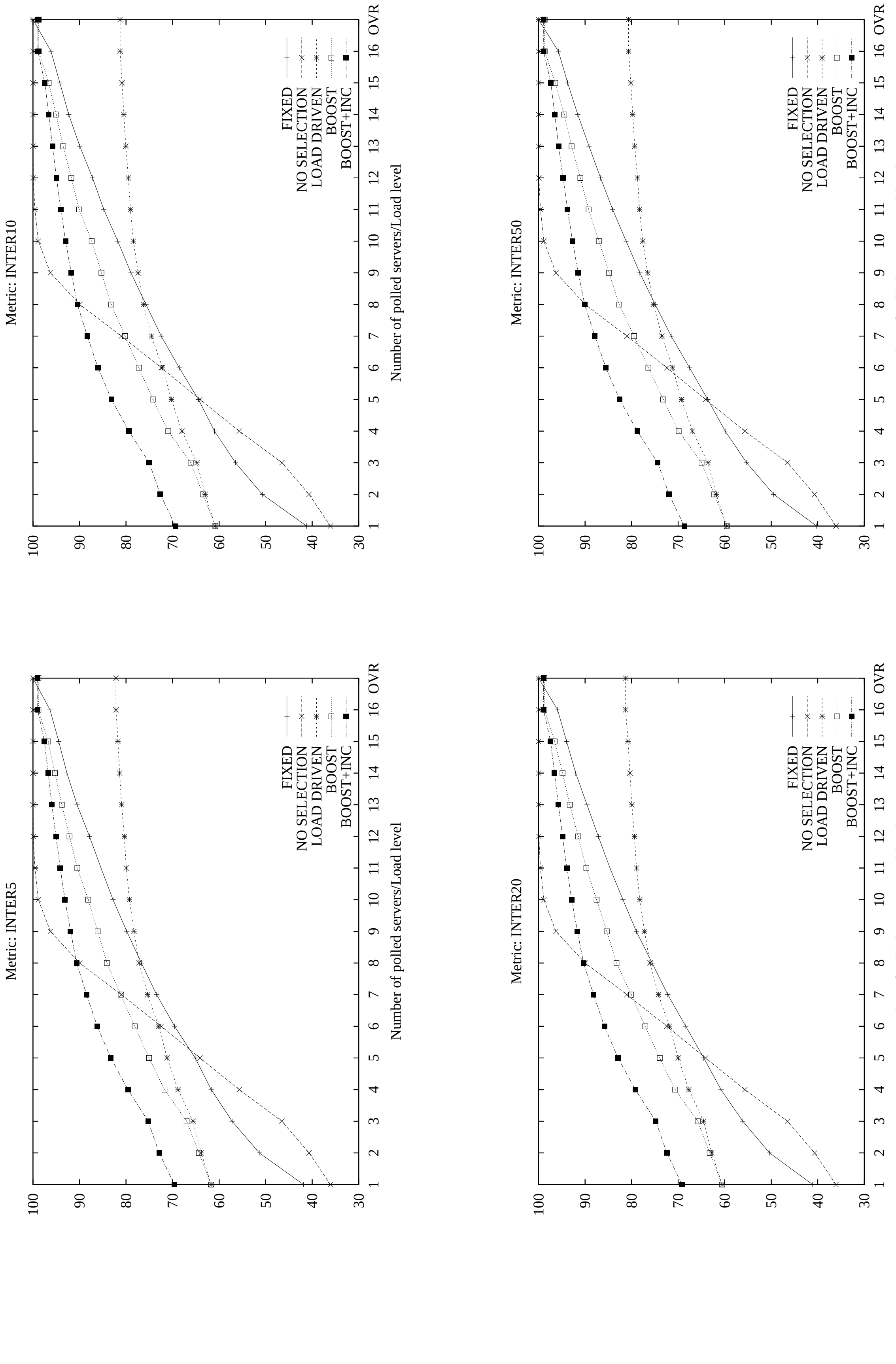


Figure 5.1: Results with a LRU cache of 32000 entries. (continues on the next page)

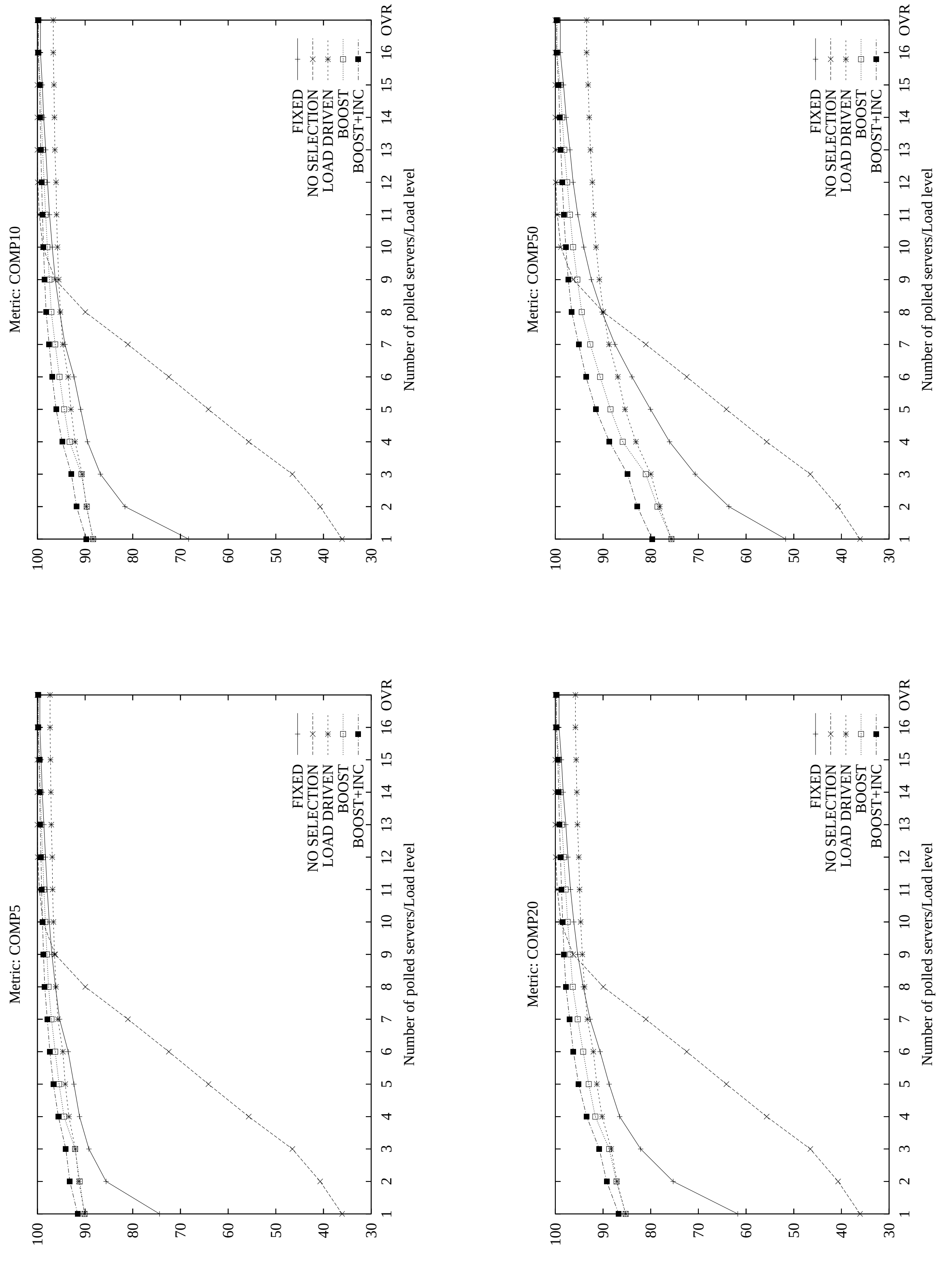


Figure 5.2: Results with a LRU cache of 32000 entries.

Algorithm and	Size of static cache (%)									
total cache size	0	10	20	30	40	50	60	70	80	90
ARC 100	**41.6**	41.5	41.0	40.3	39.4	38.1	36.3	33.7	29.4	22.8
ARC 500	48.3	48.9	49.2	**49.4**	49.3	49.1	48.7	47.9	46.1	41.7
ARC 1000	49.6	50.5	51.0	51.3	**51.5**	51.5	51.4	51.0	50.1	47.5
ARC 2000	50.8	51.8	52.4	52.8	53.0	53.1	**53.2**	53.2	52.9	51.5
ARC 4000	52.1	53.2	53.8	54.1	54.4	54.7	54.8	**54.9**	54.9	54.4
ARC 8000	54.1	55.0	55.6	55.8	56.0	56.2	56.4	56.5	**56.6**	56.4
ARC 16000	56.2	57.0	57.4	57.6	57.9	58.0	**58.3**	58.2	58.2	58.2
ARC 32000	58.4	59.2	59.6	59.7	60.0	60.0	60.1	60.1	**60.2**	59.9
LRU 100	41.5	**41.7**	41.2	40.4	39.5	38.2	36.4	33.8	29.2	22.0
LRU 500	48.2	48.8	49.2	**49.4**	49.4	49.2	48.8	48.1	46.4	41.8
LRU 1000	49.6	50.5	51.0	51.3	**51.6**	51.6	51.5	51.1	50.3	47.7
LRU 2000	50.7	51.8	52.5	52.8	53.0	53.2	**53.3**	53.2	52.9	51.7
LRU 4000	52.1	53.3	53.8	54.2	54.5	54.7	54.9	**55.0**	55.0	54.5
LRU 8000	53.7	54.8	55.4	55.8	56.1	56.3	56.5	**56.6**	56.6	56.5
LRU 16000	55.7	56.7	57.2	57.6	57.8	58.0	58.2	**58.3**	58.3	58.3
LRU 32000	57.7	58.7	59.2	59.6	59.8	60.0	**60.1**	60.1	60.1	60.0

Table 5.2: Cache hit ratio using different algorithms. Test over four weeks. The best choice for the size of the static cache is shown in bold.

1. LRU caching, fixed collection selection;
2. LRU caching, no collection selection; in this case, still we drop queries for overloaded servers, *i.e.* all servers are polled with priority 1;
3. LRU caching, selection policy PCAP, load-driven routing;
4. LRU caching, selection policy PCAP, load-driven (boost) routing;
5. Incremental caching, selection policy PCAP, load-driven (boost) routing.

Initially, we use the fixed collection selection, for different values of T, to determine the peak load needed to guarantee that T servers are used to answer each query. These load levels (15.6, 18.0, 20.0, 24.7 etc.) represent the peak percentage of queries that hit the most loaded server within a sliding window of 1000 queries.

For instance, when using FIXED $< 1 >$, with a LRU 32000, we have that, in a specific moment, about 150 queries out of the latest 1000 hit a server. During the rest of the execution of the test, no query window generated a higher peak.

Given these load limits, we test different routing procedure, i.e. broadcast (no collection selection), load-driven and boost. All queries that would break the load threshold are not served by the overloaded queries.

The test cache has a hit ratio of about 55%. This is actually a good result, as an infinite cache would reach a 57% hit ratio on our query log, with more than 84,000 unique entries. The caching system alone query reduces the load of about 40% (from 100% down to 58.1%). An interesting datum is that, in an infinite cache, only about 54,000 of cache entries would be read after their creation, because about 30,000 queries are one-timer. The caching system of a Web IR system would greatly benefit from being able to identify (and filter out) one-timer queries. Currently, some groups are trying to address this issue [91].

The benefits given by collection selection and load-driven routing are clear, using any of our metrics. At INTER5, for instance, using the computing power needed to sustain FIXED $< 1 >$ (i.e. about 15%), using load-driven routing we can surpass 60% coverage, reaching 70% when using the boost strategy with incremental caching. Incremental caching contributes to about 10% results.

Clearly, if the load threshold is low, it is very important to use collection selection. If we simply broadcast the queries, the IR cores will drop them, with no respect for the relative priority. In fact, with very low threshold, broadcast is worse than FIXED. When more computing power is available, the difference between broadcast (no collection selection) and load-driven routing gets smaller, and actually broadcast can be used when we plan to poll half the servers. In other words, the difference, in peak computing load and result quality, between polling the most relevant 8 servers, or all of them at the same time, is limited.

These results can be presented in several different ways. If we want to guarantee a competitive recall at 5 (INTER5) of 70%, without using collection selection, we need to use a machine able to process about 33% of the query traffic. With this load threshold, query broadcast is roughly equivalent, in term of load and precision, to answering using always the 6 most relevant servers. Alternatively, we can either use a smaller machine, able to sustain only 15% of the query traffic, with boost and incremental caching. With 50% the computing power, our strategy can reach the same performance.

The difference, as perceived by the users, is even more dramatic. If we measure competitive similarity at 5 (COMP5), in order to reach 90% coverage, we need to broadcast the queries and sustain about 45% the load (corresponding to the load reached by FIXED 8), while boost and incremental caching can reach the same quality with only 15% peak load.

Clearly, the biggest benefits are measured with small load threshold levels. When the threshold level is higher, we can retrieve more results with any approach, reducing the differences. With very small load threshold, 15%, we can still retrieve results with 90% similarity to those we would get from the full index. In this configuration, each server is holding a very small part of the collection, about 1/32, and is polled every six queries.

The incremental cache has also a very beneficial effect for queries with hard collection selection. If a query is composed of terms not present in the training set, PCAP will not be able to make a strong choice, and the selection will be ineffective. If the query is popular, subsequent hits will retrieve more results, reducing the effects of a wrong selection: for popular hits, *non degraded* results will be returned to users.

5.5 Experiments on Different System Configurations

The result quality offered by the combined usage of load-driven routing and incremental caching is very beneficial when we want to limit the computing load to a search engine system. If we limit the number of queries that every core can answer, our approach can greatly improve the result quality w.r.t. to query broadcasting. This is because, if we broadcast the query, when the load is limited, there is no control of which queries must be dropped by every core. Using collection selection, the queries for which the server is not relevant will be dropped first.

In Table 5.3, we compare our algorithms under different configuration and computing load. We tested three different cache sizes (4,000, 15,000 and 32,000). We also used SDC: half of the cache was reserved for a static set of queries. Then, we limited the computing load to four levels (10%, 15%, 20% and 25%), which roughly correspond to the computing load reached by strategy FIXED $< 1 >$ through FIXED $< 4 >$. The results confirm our expectation: under every configuration, load-driven routing and incremental caching greatly improves the result quality. This is confirmed by all metrics, and we show only COMP5 for simplicity.

This is also confirmed when testing different query test sets, which corresponds to the queries submitted by users on different times (we tested the queries submitted during the four weeks after training).

5.6 Caching Posting Lists

Some recent works by Long *et al.* [64] and Baeza-Yates *et al.* [7] discussed the effectiveness of caching posting lists as a part of the cache hierarchy in complex IR systems. For this reason, we

Load level:			10%				15%				20%				25%			
Cache Size	SDC Ratio	Week #	Fixed	Boost	Boost + Inc.	No Coll. + Inc.	Fixed	Boost	Boost + Inc.	No Coll. + Inc.	Fixed	Boost	Boost + Inc.	No Coll. + Inc.	Fixed	Boost	Boost + Inc.	No Coll. + Inc.
4000	0	0	40	54	**59**	33	51	60	**67**	43	57	65	**72**	52	61	70	**77**	61
		1	40	53	**61**	36	50	60	**68**	46	55	64	**74**	55	60	70	**78**	64
		2	40	55	**61**	38	50	61	**68**	48	55	66	**74**	57	60	71	**78**	65
		3	38	51	**56**	31	47	59	**64**	41	53	64	**70**	51	59	69	**76**	60
4000	50	0	46	59	**63**	42	55	65	**70**	52	61	70	**75**	60	65	74	**80**	68
		1	47	59	**65**	45	55	65	**71**	54	60	69	**76**	62	64	73	**81**	70
		2	48	60	**64**	46	56	66	**71**	55	62	71	**76**	63	66	75	**80**	71
		3	42	56	**60**	39	50	62	**67**	49	56	67	**73**	58	61	71	**78**	66
15000	0	0	40	54	**61**	37	51	61	**69**	48	57	66	**74**	57	61	71	**79**	66
		1	40	53	**56**	39	50	60	**70**	50	55	65	**75**	59	60	69	**80**	68
		2	40	56	**63**	41	50	63	**70**	51	55	68	**76**	61	60	73	**80**	69
		3	38	51	**58**	34	47	60	**66**	45	53	65	**72**	55	59	69	**77**	64
15000	50	0	49	62	**66**	48	57	67	**73**	57	62	72	**77**	65	66	75	**82**	72
		1	49	61	**67**	49	57	67	**73**	58	62	71	**78**	67	65	74	**82**	75
		2	50	63	**67**	50	58	68	**73**	59	63	73	**78**	67	67	76	**82**	75
		3	44	58	**63**	44	52	64	**70**	54	58	69	**75**	63	63	73	**80**	71
32000	0	0	40	54	**62**	37	51	61	**69**	49	57	66	**75**	58	61	71	**79**	66
		1	40	53	**63**	40	50	60	**70**	51	55	65	**76**	60	61	70	**80**	69
		2	40	57	**63**	42	50	63	**71**	52	55	68	**76**	62	60	73	**81**	71
		3	38	52	**59**	36	47	60	**67**	47	53	65	**73**	56	59	70	**78**	66
32000	50	0	51	64	**68**	51	59	69	**74**	60	63	73	**79**	68	67	77	**83**	76
		1	51	62	**69**	53	58	68	**75**	61	63	72	**79**	70	66	75	**83**	77
		2	52	64	**69**	53	59	69	**75**	62	64	74	**80**	70	68	77	**83**	78
		3	46	60	**66**	48	53	66	**71**	57	59	70	**77**	66	64	74	**81**	74

Table 5.3: Competitive similarity at 5 (COMP5 %) of our different strategies on several cache configurations, over four weeks. We tested caches of size 4,000, 15,000, and 32,000, of which 0% and 50% was devoted to static entries. The combined strategy of load-driven (boost) routing and incremental caching is greatly beneficial under any configuration.

Cache size	Min	Max	OVR
64	31.2 %	38.5 %	21.5 %
128	36.5 %	44.7 %	25.0 %
256	41.8 %	51.2 %	30.2 %

Table 5.4: Hit-ratio of local caches for posting lists. Reported are the minimum and maximum value across the main 16 clusters, and the hit-ratio for the overflow cluster.

measured the performance of small caches of posting lists in the IR cores. Due to the fact that the cores *specialize*, in a sense, to certain topics, we expect that the hot terms can be effectively cached.

As shown in Table 5.4, a very small cache of only 64 entries is already able to exhibit more than 30% hit-ratio, raising to more than 40% with a larger cache (256 entries).

While these results are very promising, we chose not to study this cache level further, because, to the best of our knowledge, there are no strong models to compare, among other things, the cost of scanning a cached posting lists vs. a non-cached list. Still, we believe that a local cache could greatly contribute to reducing further the computing load of the system.

5.7 Possible Developments

The concept of incremental cache is very general, and we see some possible extensions.

5.7.1 Impact-Ordered Posting Lists

Anh and Moffat [1, 2] have introduced a very efficient strategy to score documents in an IR system by using precomputed *impacts*. The impact $\omega_{d,t}$ of a document d for a query term t is an approximation of the contribution of the occurrences of term t in d to the final score of the document d for a query comprising term t. The authors show a way to compute the value $\omega_{d,t}$ as an approximation of the exact results obtained with a vector-space model such as Okapi BM25 [85]. They also show how to reduce the cost of a query by sorting the posting lists by document impact, and by scanning only a part of the list (the most promising) for every query.

Their approach has much in common with ours, as a set of documents (those with the highest impact for each query term) are analyzed before the other, less relevant documents. The authors also show a halting condition that lets the system determine when to stop scanning. Clearly, there is a trade-off between computing cost (*i.e.* the depth of the scanning) and result accuracy.

Our incremental cache could be used also with this type of systems, as follows. When a query is submitted for the first time, only the initial part of the posting list for each term (*i.e.* documents with high impact for that term) is scanned and cached in a cache of posting lists. When the query is repeated (or when the term is requested again), more entries from the posting lists are scanned and added to the cache. Over time, the posting lists for the most requested terms will be entirely stored in the cache: the system will be able to return non-degraded results for queries comprising those common terms.

5.7.2 Different Value of Cache Lines

The replacement policies used in our tests (LRU and ARC) for the result cache are based on query timing: the entries that are not recently used are evicted from the cache. We can devise a strategy that combines the timing information with the *value* of the stored information. In our case, cache lines that store the results coming from a large number of servers are probably more valuable than cache lines where we store results coming from only a small number of servers. The replacement policy could combine the recency with the value, and try to preserve more valuable entries even if they are relatively old.

5.7.3 Application to P2P Systems

P2P Web search systems [13, 76, 94, 93] use sophisticated techniques to perform collection selection and to route queries in a network of (possibly) uncooperative peers. In order to reduce the pressure on the communication medium and the computing load on peers, only a limited set of peers can be used to answer each query. Our incremental caching strategy can clearly be adapted to this type of systems, with very beneficial effects.

5.8 Summary

In this chapter, we analyzed the design space of a distributed IR system featuring caching and collection selection. The cache can be designed to interact with the selection system, by updating results in an incremental way. Every time there is a cache hit, further servers are polled, and their results are added to the cached entries. An incremental cache, over time, stores non degraded results for frequent queries, as more and more servers will be polled on each subsequent hits. In other words, our incremental cache system is improving *both the computing load*, by filtering repeated queries, *and the result quality*, by refining the stored results for common queries.

The results are dramatic. We used PCAP to move 50% of our base in an overflow server, and then we partitioned the remaining into 16 clusters: each server is holding about 1/32 of the base. By limiting the peak load to about 15%, we are able to reach an INTER5 of more than 2/3, or a COMP5 of 80%. This result is reached by using a small LRU cache (32000 elements), load-driven collection selection and incremental result updating. Alternatively, we can cover 3/4 (INTER5) of the results with a peak load of 18%.

We believe that collection selection can be a key element in improving the performance and scalability of modern search engines, by offering a very nice trade-off between result quality and query cost (load). It is also able to adapt dynamically to load peaks, by temporarily reducing the result quality.

This strategy can also be used on bigger systems, on each sub-collection. Commercial systems can index billions of pages, with sub-clusters holding millions at a time. Our strategy can be used on each sub-cluster, in a hierarchical architecture: each set of millions of documents can be partitioned and then collection selection can be used to reduce the computing load.

Chapter 6

Updating the Index

Several researchers have investigated the problem of keeping the document base and the index up to date, in a changing environment like the Web. Two main issues have been investigated. First, the IR system needs to plan the crawling strategy so to find new pages, to identify new versions of pages and to prevent frequently changing pages (*e.g.* news sites, blogs) from becoming stale in the index. Second, the new documents, or their new versions, must be parsed and indexed, and the main index must be updated.

In a distributed, document-partitioned configuration, new documents are added to one of the IR cores. The core must extract the posting lists for the new pages and make them searchable. One simple way to do it is to create a second index with the new postings, and query it in parallel with the main index. The second, smaller index can be updated when new documents are available and, after a chosen time interval, the main and the second index can be merged into one. This server, during this merging procedure, becomes unavailable to answer queries, but this is usually not problematic if several replicas are deployed to provide enough computing bandwidth.

If the query is broadcasted to all cores, as in traditional document-partitioned systems, the choice of the core to which a new document is mapped does not have any effect on the precision of the system. It can have an impact on performance if the partitions are heavily unbalanced, but this can be simply prevented by keeping track of the collection size in each server.

In our architecture, on the other side, the choice is more important: it is crucial to assign new documents to the *correct* collection, *i.e.* to the server where most similar or related documents are already assigned. In other word, we want to store the new document where the collection selection function will find it. This is complex because the document could be relevant for several queries, which will rank the IR servers differently.

6.1 A Modest Proposal

An ideal approach would be as follows. First, all queries from the training set are performed against the new documents. This way, we can determine which documents are matching which queries, and create the QV representation for the new documents. The vectors are added to the QV vectors of all the other existing documents, and co-clustering is performed on the new matrix (using the previous clustering as a starting point). This is clearly unfeasible, as the training set can be composed of several thousand queries.

A very quick approximation can be reached using the PCAP collection selection function itself. *The body of the document can be used in place of a query: the system will rank the collections according to their relevance.* The rationale is simple: the terms in the document will find the servers holding all the other documents relative to the same broad topics. Our collection selection function is able to order the document collection according to the relevance to a given query. If the body of the document is used as query, the collection selection function will find the closer collections.

In the first phase, we perform a query, using the document body as a topic, against the query dictionaries, and we determine the matching query clusters, *i.e.* the clusters comprising query with terms appearing also in the document. Then, the PCAP matrix will be used to choose the best document clusters, starting from the query cluster ranking, as seen in Section 3.4.

This is somewhat similar to what is done by pSearch [95] and SSW [61], as shown in Section 2.3. These two systems project the documents onto a lower dimensional space using LSI, and then route them to the correct servers using the same infrastructure used by the queries.

Even if we will prove that this approach is robust, there are several issues that could make it fail. For example, some query cluster could be composed of queries for which the document is actually a good matching, but the terms of which do not appear in the document itself. This can happen for instance in the presence of synonyms, or if the document itself uses graphics to represent words and the query terms are used, instead, in links pointing to the page.

This problem is limited by the redundancy of the QV matrix and the robustness of the algorithm, as we discussed in Section 3.10. Even if one good query, in some query clusters, may use terms not present in the document, the query clusters will be comprised of queries retrieving close documents, and those terms might be present in the document. In any case, after the document is mapped to one cluster, a full search algorithm (using links, PageRank etc.) is used on the local index, and the document will be correctly retrieved and ranked.

From the moment of the assignment on, the new document will appear among the results of the new queries. They will contribute to building a new QV matrix, used when the system chooses to have a major update of the whole index. In fact, every once in a while, the system will be momentarily stopped to update the partitioning and the local indexes. While the latter is needed in every document-partitioned architecture, our collection selection architecture also need to update the assignment of documents to clusters. This means moving a document from its initial position to the position suggested by a complete round of co-clustering. We will verify below that this is not strictly necessary, as the approximate choice is good enough to our purposes.

6.2 The Cost Model

Several works have also analyzed the effect of the collection size in information retrieval in general, and in distributed information retrieval systems in particular.

In his work on large English text corpora, Heaps [46] showed that as more instance text is gathered, there will be diminishing returns in terms of discovery of the full vocabulary from which the distinct terms are drawn. This law was verified also for pages coming from the Web [10]. The law can be restated by saying that the number of distinct terms $T(n)$ from a collection of n documents grows as:

$$T(n) = K \cdot n^{\beta}$$

with K between 10 and 100, and β between 0.4 and 0.6 for collection of English language documents. If documents are of average size S:

$$\text{total number of terms} = n \cdot S$$

$$\text{average size of a posting list} = \frac{n \cdot S}{K \cdot n^{\beta}} = \frac{S}{K} n^{1-\beta}$$

According to the Heaps law, the growth of the average posting list is sub-linear. If the length of the posting lists drives the cost of solving a query, this will grow slower than the growth of the collection.

This cost model has been challenged by later works. [26] models the response time as linearly dependent from the collection size. Later experimental tests have shown that the load distribution is more variable, and heavily dependent from the cache access patterns, the structure of the memory hierarchy, and the distribution of the length of the posting lists in the local server [4].

In the previous chapters, we simply assumed a cost equal to 1 for any query on any server (except the overflow cluster). In the next chapter, we will show an extended model that measures

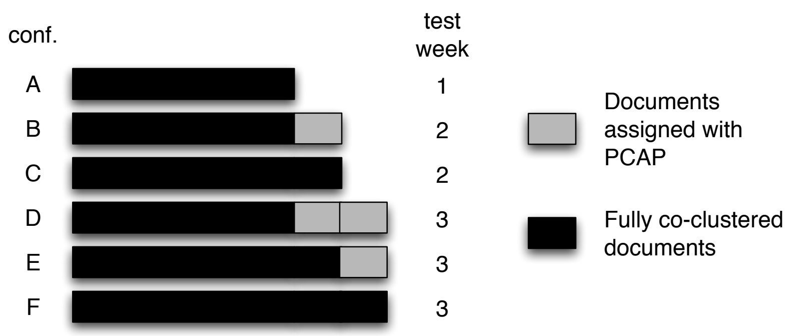


Figure 6.1: Different configurations for the experiment on index update.

the actual cost of queries on the partitions, and we will see that our system can effectively deal with partitions of different sizes.

6.3 Experimental Results

In this section, we try to verify here the feasibility of updating the index. In this experiment, we assume that a part of the documents, about one million, is not initially available, because it represents newer or changed pages collected over the two weeks following the end of the initial training. We performs the usual training on the initially available five million documents, and then we use the strategy suggested above to assign the new documents as they arrive. To speed the process up, only the first 1000 bytes from the document body (stripped of HTML tags, header and so on) were used as a query to determine the best collection assignment.

We simulate six different configurations, as shown in Figure 6.1:

A This represents the situation right after the training, with 5 million documents available, partitioned using co-clustering. We test against the first set of test queries (queries for the first week after the training).

B This represents the situation one week later, with 5.5 million documents available. Documents are assigned using PCAP as just described. We test against the second set of test queries (queries for the second week after the training).

C This is similar to B, but now old and new documents are clustered using a full run of co-clustering. Same test set.

D This is the situation after two weeks, with all documents available. In this configuration, all new documents are assigned with PCAP. We use the third week of queries.

E Here we imagine that we were able to run co-clustering for the first half million new documents, but not for the latest ones. Same test set as D.

F Finally, all 6 million documents are clustered with co-clustering. Same test set as D.

Figure 6.2 shows the coherence of the resulting partitions, according to our set of metrics. The collection selection function performs with the same effectiveness on every configuration: the quality of the results coming from the selected servers is not degrading if documents are added with our strategy. Still, the system performance could degraded if partitions are not balanced. The next section discusses this.

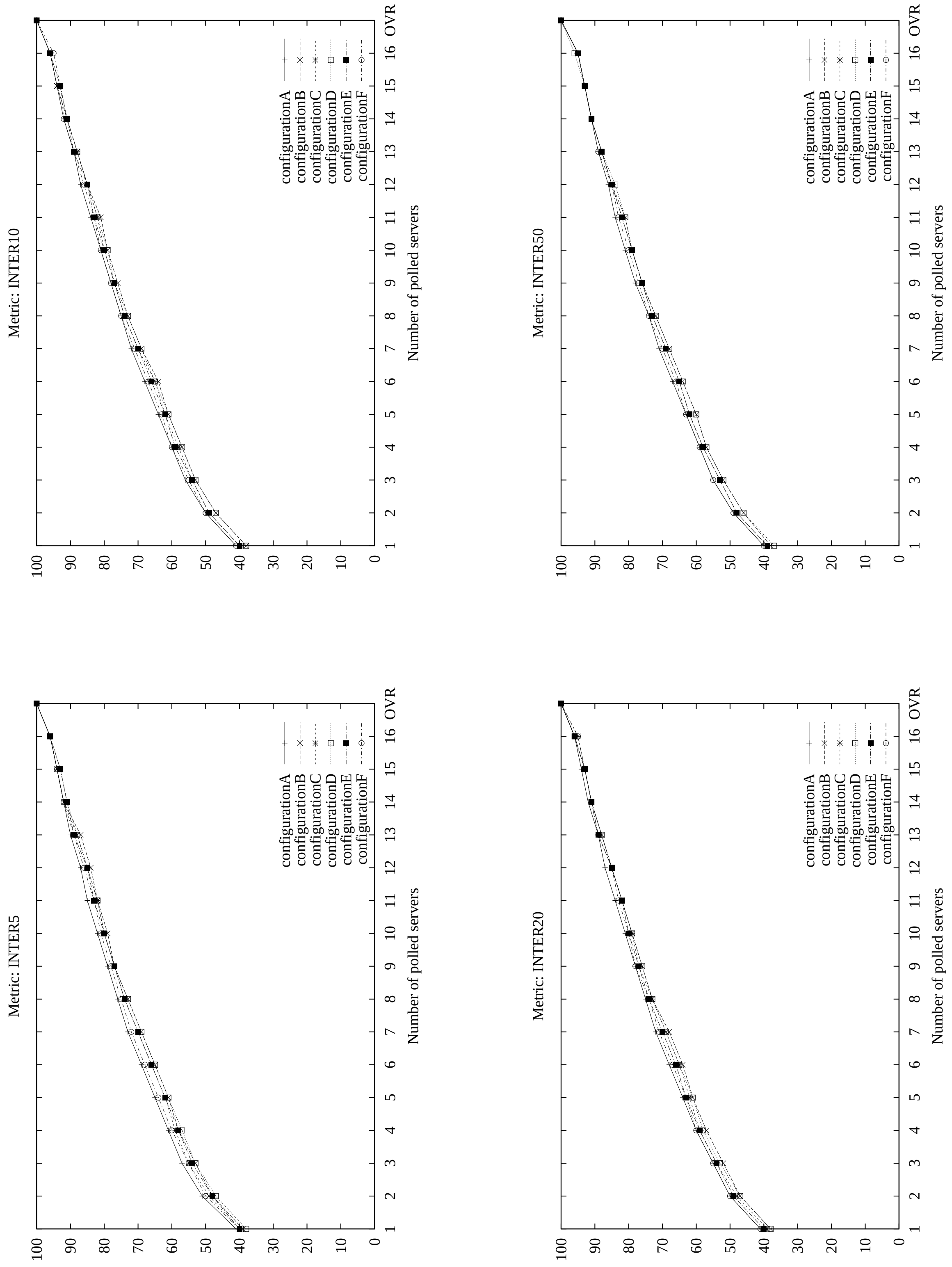


Figure 6.2: Clustering quality during index and collection update. (continues on the next page)

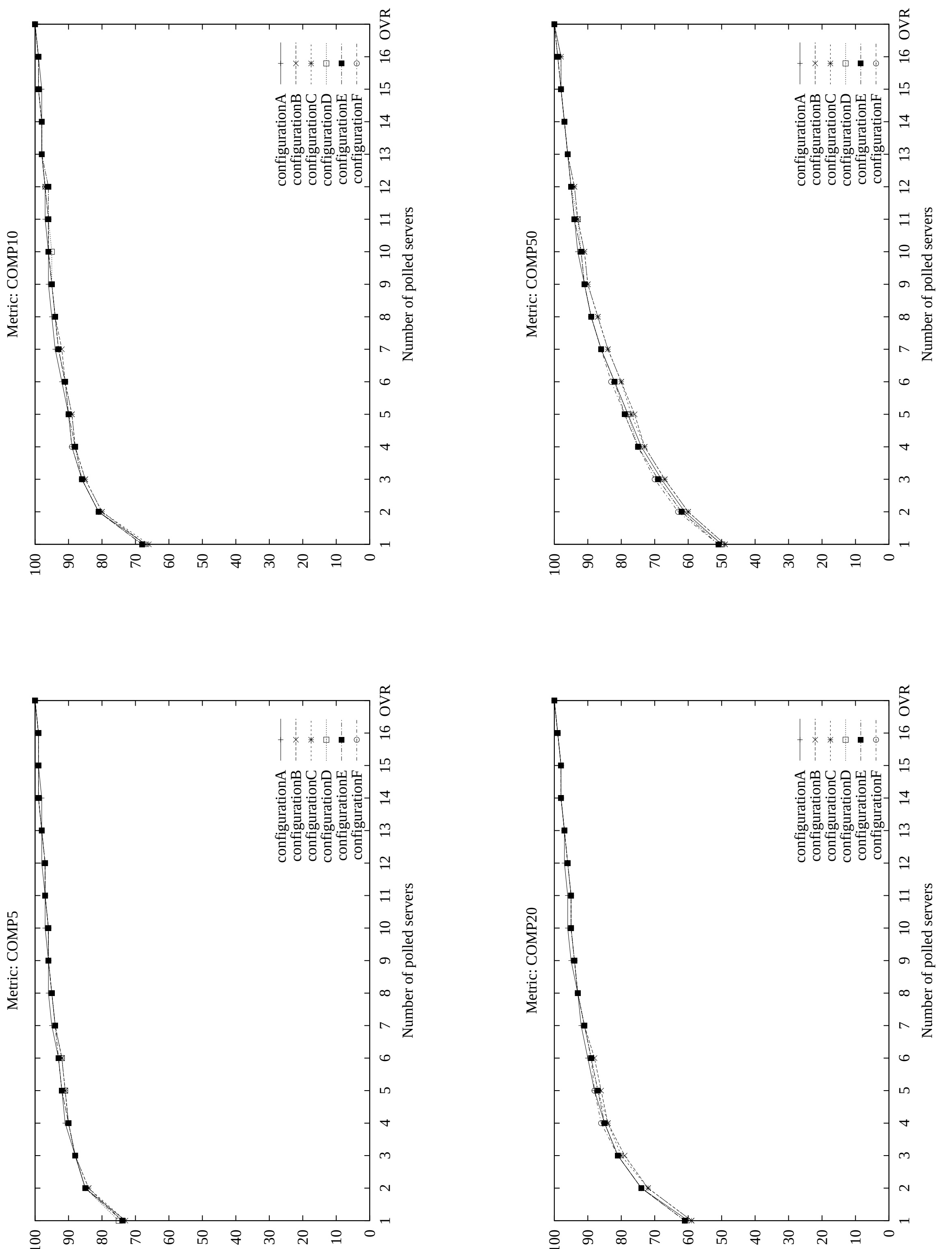
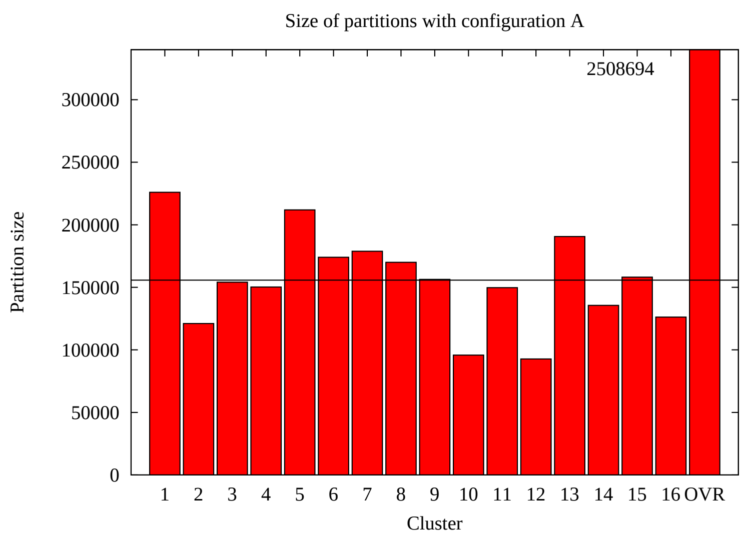
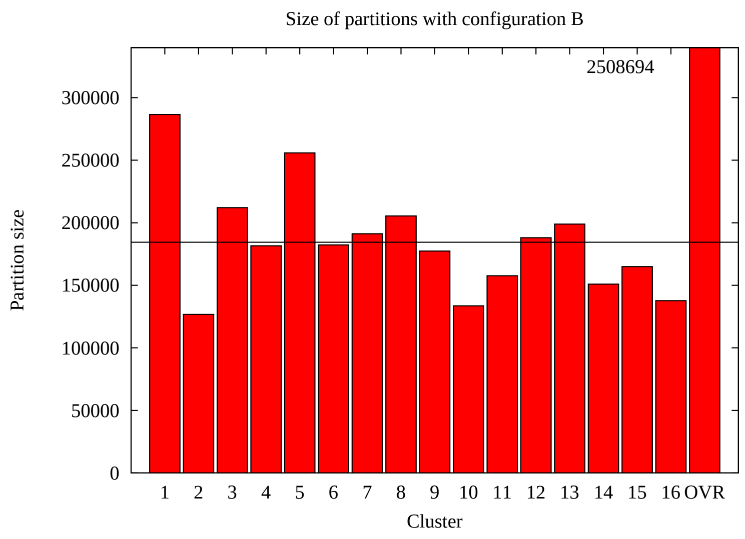


Figure 6.3: Clustering quality during index and collection update.





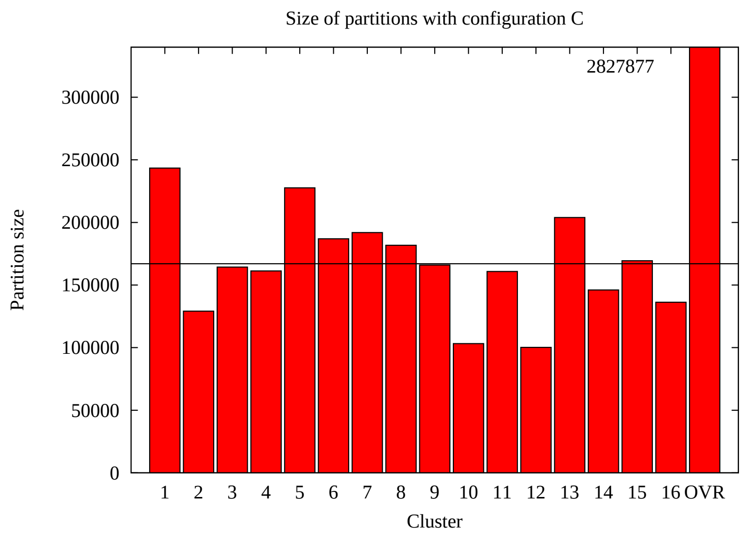


Figure 6.4: Size of the document collections in each cluster.

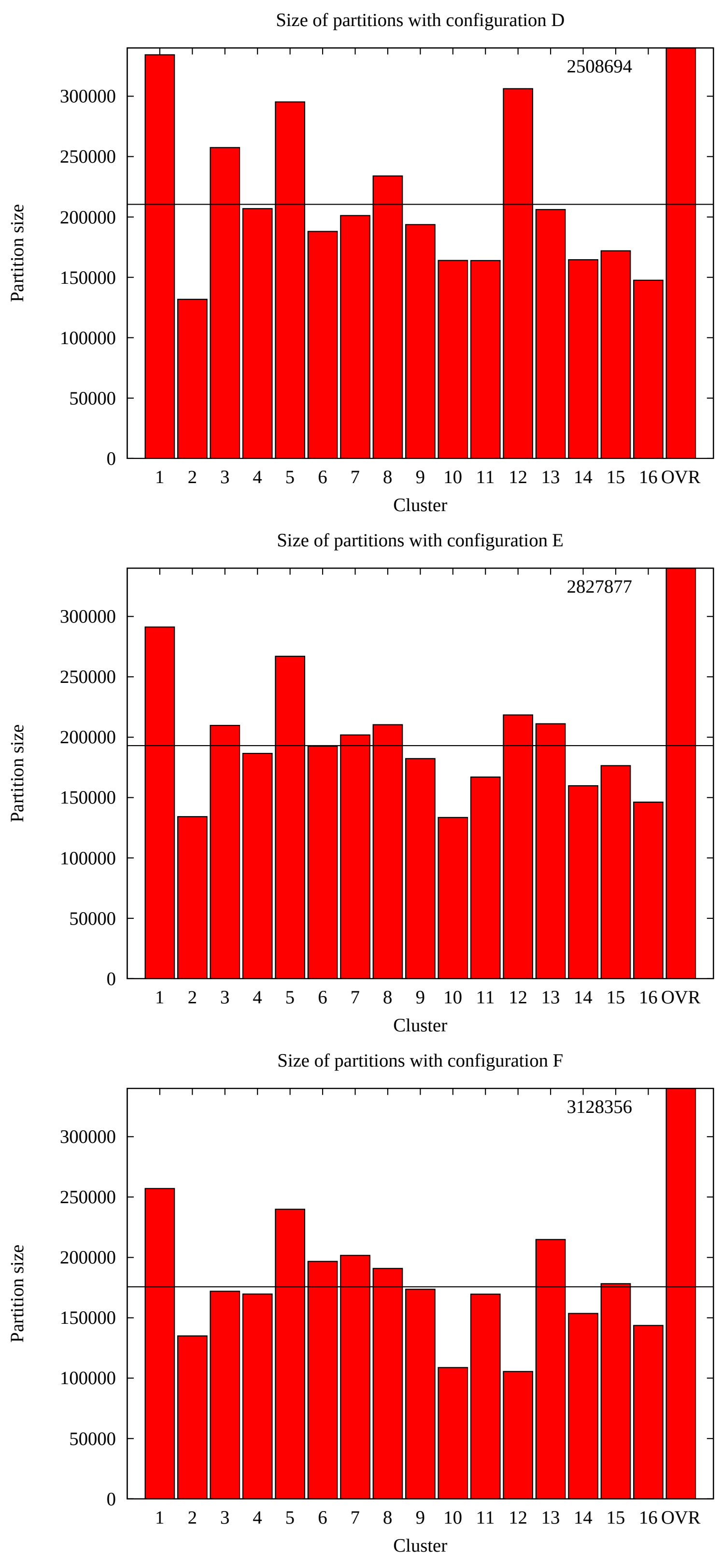


Figure 6.5: Size of the document collections in each cluster.

6.4 Size of Partitions

An important aspect to consider when updating the partitions is their relative sizes: if a partition grows much bigger than another one, the server holding it could become slower than the other ones, contributing to a system-wide degradation of performance. Even if, in our architecture, this problem is limited by the use of a load-driven collection selection, still we want to verify is this can be a problem or not.

We measured the size of the different partitions in each configuration. Results are shown in Figures 6.4 and 6.5. The plots show the size of the collection held by each cluster. The size of the collection held by the overflow cluster is beyond scale and truncated. The horizontal line shows the average size for the clusters, excluding the overflow cluster.

In the worst case, *i.e.* Configuration D, the biggest cluster holds about 2.5 times the documents held in the smallest cluster. In the other cases, this difference is a little bigger than 2.1 times, and the distance from the average is not dramatic. It is not surprising that Configuration D has the worst distribution: it is the configuration with the biggest number of documents not assigned using co-clustering.

According to the Heaps law, the average size of posting lists, in a collection 2.1 times bigger, is about 1.3–1.5 times bigger. We showed that the load-driven collection selection is able to effectively distribute the load across servers, and that a local cache of posting lists is extremely effectively. In the next chapter, we will show that the difference in query timing due to the partition sizes are not detrimental to our performance.

6.5 Summary

In this chapter, we showed a very effective strategy to add new documents to the collection. By using the PCAP collection selection function, we can choose the best cluster for each new document. The proposed strategy is effective is keeping the partitions reasonably balanced (with the smallest partition about half the size of the biggest one) and in maintaining the same result quality we would reach if the documents were fully reassigned using co-clustering. This simple strategy can be very effective in a real implementation because of its very limited cost.

Chapter 7

Evaluation with Real Query Timing

In this chapter, we refine our simulation by using a more detailed cost model. Also, we redefine our architecture, by shifting the responsibility of load-balancing from the broker down to the cores. By doing this, we show an architecture closer to a real implementation, and we prove that our strategy is still applicable and very effective in this extended model.

7.1 Reversing the Control

In our previous discussion, we imagined that the broker modeled the computing load of the IR cores by counting the number of queries that are forwarded to each core within a time window. This is clearly a rough approximation of the load imposed to each core, because it assumes that the cost of each query is independent from the terms comprising it and from the documents held by a core. On the contrary, the cost of a query can widely vary, due to several factors including: number of documents in the collection, popularity of the requested terms, state of the cache and the disk [4].

We can reverse this situation by giving the control of the system load to the cores. We can instruct the cores about the maximum allowed computing load, and let them drop the queries they cannot serve. In this configuration, the broker still performs collection selection, and ranks the collections according to the expected relevance for the query at hand. The query is then *tagged* with the priority, as described in Section 4.1, and forwarded to each query. In other word, the query is again broadcasted, but now every server is informed about the relevance it is expected to have w.r.t. the given query. At this point, each core can choose to serve or drop the query, according to its instant load, the query priority etc.

If the query is accepted, the core will answer with its local results. Otherwise, it can send a negative acknowledgments. Alternatively, the broker can set a time-out, and consider as dropped all the query that are not answered before its expiration.

In this model, the broker, instead of simply performing collection selection, performs a more precise *collection prioritization*[1]. Now, the cores can use real data to determine their load, including the actual cost (time) of queries.

7.2 Extended Simulation

To simulate the proposed architecture, we worked as follows. Using our co-clustering strategy, we created 16 document collections, plus the overflow collection. Then we indexed them independently, and we submitted the stream of queries to each index. On each partition, we recorded the timing

[1]We thank Fabrizio Silvestri for this term.

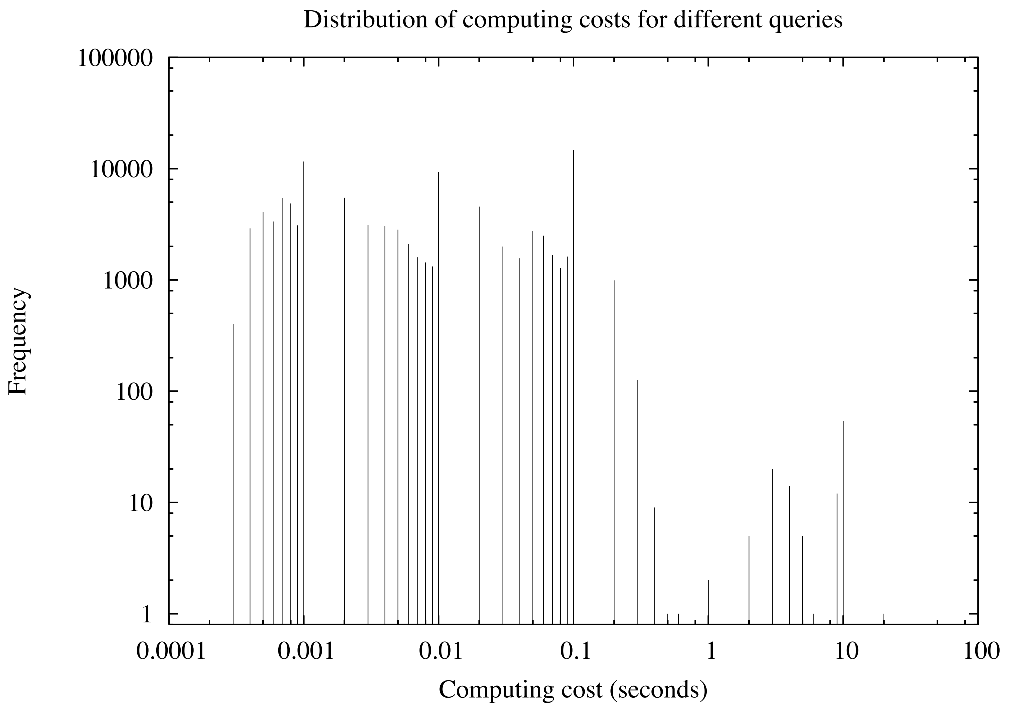


Figure 7.1: Example of distribution of the computing costs of the queries in our test set, on the first document collection.

of each query, as reported by the search engine scanning the local index. This information is used to determine the cost imposed by each query to each index.

The load-driven strategies work a way similar to what we presented before. When the core receives a query tagged with a priority, it verifies if $l_R^{i,t}(Q) \times p_i < L$. The definition of instant load is extended: it is computed as the total cost of the queries served from the window of the last 1000 queries. If a core c is the most relevant for a query q, the broker will give it priority 1, and the core will drop the query only if it is currently overloaded. When priorities are lower, queries will be dropped easier.

A difference with the previous model is that the measured cost of answering a query can widely vary, influenced by several factors and transient conditions at the server. Figure 7.1 shows the distribution of the computing cost (time) needed to answer queries from our test set, on the first document cluster. The time can vary from less than a microsecond to several seconds. While most queries are served in less than 0.1 seconds, the maximum load can be widely affected if a couple of slow queries happen to be seen at short distance. That is why, in the experiment with this model, we capped the load threshold to the average load measured across all servers, across the full test set of queries. In other words, we measured the average computing cost needed to sustain the various configurations with FIXED collection selection. These values are then used as threshold limits in the modified load-driven strategies.

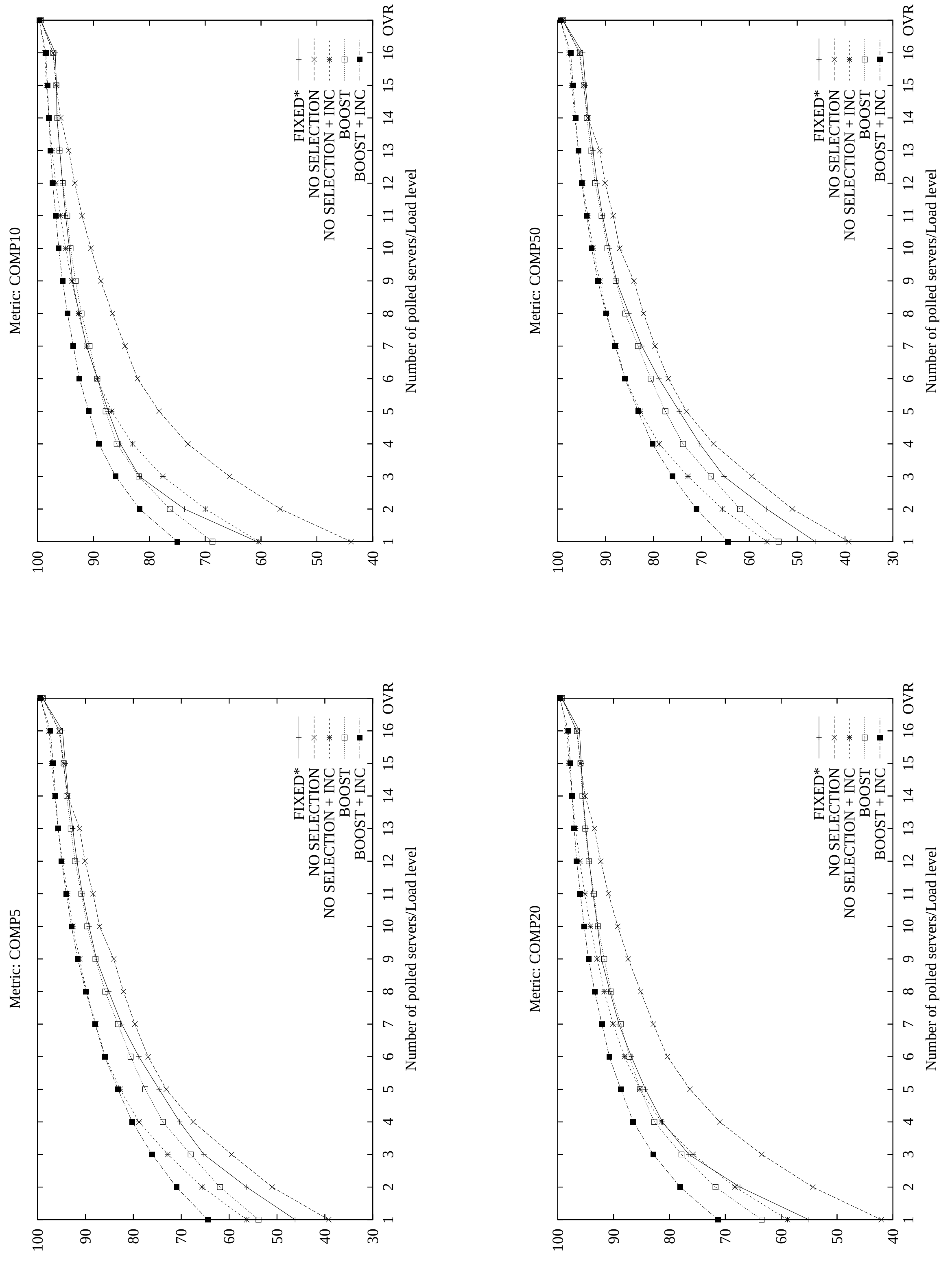


Figure 7.2: Comparison with real query timing information.

	1	2	3	4	5	6	7	8	
FIXED(*)	0.86	1.77	2.75	3.64	4.58	5.56	6.50	7.41	
NO COLL	5.72	7.57	9.17	10.43	11.43	12.08	12.59	13.04	
NO COLL + INC	8.72	10.35	11.57	12.65	13.31	13.90	14.26	14.61	
BOOST	2.04	2.88	3.78	4.68	5.57	6.47	7.30	8.23	
BOOST + INC	5.32	6.52	7.69	8.78	9.70	10.57	11.34	12.05	
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**16+OVR**
FIXED(*)	8.33	9.26	10.23	11.15	12.06	13.04	14.01	15.08	16.34
NO COLL	13.45	13.86	14.16	14.57	14.86	15.19	15.40	15.63	16.34
NO COLL + INC	14.88	15.14	15.37	15.58	15.77	15.92	16.06	16.17	16.62
BOOST	9.14	9.98	10.91	11.93	12.85	13.83	14.71	15.50	16.34
BOOST + INC	12.65	13.17	13.67	14.19	14.68	15.16	15.64	15.98	16.62

Table 7.1: Average number of servers polled with different strategies, fixed the computing load. FIXED(*) polls a fixed number of servers, but queries can be dropped by overloaded servers.

7.3 Experimental Results

In Figure 7.2, we show the competitive similarity of different strategies, under this new simulation model. Due to the fact that the load cap is much lower that in the previous experiments, and does not guarantee the peak load of FIXED (it is the average load, which is significantly smaller than transient peaks), we used a different version of FIXED (FIXED $\star$) where cores can drop the queries if they are overloaded, as in the load-driven strategies.

While the configuration without collection selection can poll more servers (see Table 7.1), they are less relevant for the queries, and the final results are better for the approaches based on collection selection, which poll fewer, but more relevant, servers.

With a peak load set to the *average* load of FIXED $< 1 >$, the strategy based on prioritization and incremental caching surpasses a competitive similarity of 65% (with an amazing 80% COMP10).

Chapter 8

Experimental Infrastructure

8.1 The WBR99 Web Collection

The WBR99 Web document collection[1] consists of 5,939,061 documents, *i.e.* Web pages, representing a snapshot of the Brazilian Web (domains .br) as spidered by the crawler of the TodoBR search engine. The collection is about 22 GB of uncompressed data, mostly in Portuguese and English languages, and comprises about 2,700,000 different terms after stemming.

Along with the collection, we utilized a query log of queries submitted to TodoBR, in the period January through October 2003. We selected the first three weeks of the log as our training set. It is composed of about half a million queries, of which 190,000 are distinct. The main test set is composed by the fourth week of the log, comprising 194,200 queries. We also used the following weeks, for the tests on topic shift and training robustness, for a total of about 800,000 test queries.

The collection includes also a limited set of queries evaluated by human experts. Unfortunately, this set is of only 50 queries, and is not sufficient for a large scale test. This is why, following the example of previous works [102, 23], we compared the results coming from collection selection with the results coming from a centralized index.

The main features are summarized in Table 8.1.

8.2 Reference Search Engine

For our experiment, we used Zettair, a compact and fast text search engine designed and written by the Search Engine Group at RMIT University. Zettair is freely available, under a BSD-style license at `http://www.seg.rmit.edu.au/zettair/`. We modified Zettair so to implement the collection strategies discussed in this work, namely CORI and PCAP.

Zettair is used to answer the queries in the training set, so to build the QV representation of our documents. This means that, at the logical level, documents are represented by vectors in $\mathbb{R}^{194,200}$. On the other side, the vector-space representation is a vector in $\mathbb{R}^{2,700,000}$, because we have 2,700,000 distinct terms in our base.

[1]Thanks to Nivio Ziviani and his group at UFMG, Brazil, who kindly provided the collection, along with logs and evaluated queries.

d	5,939,061 documents taking (uncompressed) 22 GB
t	2,700,000 unique terms
t'	74,767 unique terms in queries
tq	494,113 (190,057 unique) queries in the training set
$q1$	194,200 queries in the main test set

Table 8.1: Main feature of our test set.

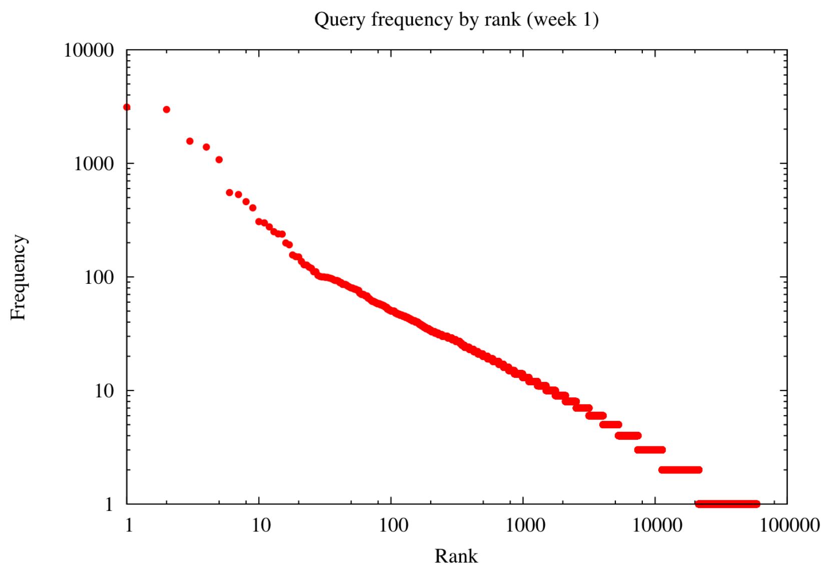


Figure 8.1: Distribution of query frequencies for week 1.

In our simulations, Zettair was used as the *oracle* against which we compare the quality of the results coming from the selected servers. More in detail, every test query was submitted to the centralized index served by Zettair. Then, we performed collection selection and we verified which documents were assigned, by co-clustering, to the selected servers.

Our simulations were run on a Apple Xserver G5, with 9 Dual PowerPC G5 nodes featuring 5 GB RAM each.

8.3 Partitioning the Collection

After removing the silent documents (documents never recalled during training), the collection was clustered into 16 document clusters, *i.e.* we determined to which sub-collection each document was assigned. The silent documents were grouped into an *overflow* cluster.

In our tests, about 50% documents are silent: the overall cluster is holding about half the collection, while the other 16 servers are holding 1/32 each on average.

8.4 Features of the query log

In our test, we used a query log from a later period w.r.t. the crawl. Nonetheless, the fact that it was focused to the same public (the Brazilian web) make it very valuable: most queries have valid results in the collection we were using. The combination, in a sense, is a coherent subset of the Web and of its users' needs.

Figures 8.1 through 8.4 give an example of the pattern of queries. The query distribution is a very typical power-law distribution [32], with some popular queries covering a big part of the traffic. Still, the very top queries are above the curve, and they are rather unexpected: the top query for the first week is "mormons", for the fourth week is "usinagem", while the usual suspect ("sexo", "travestis", "mp3") appears after them.

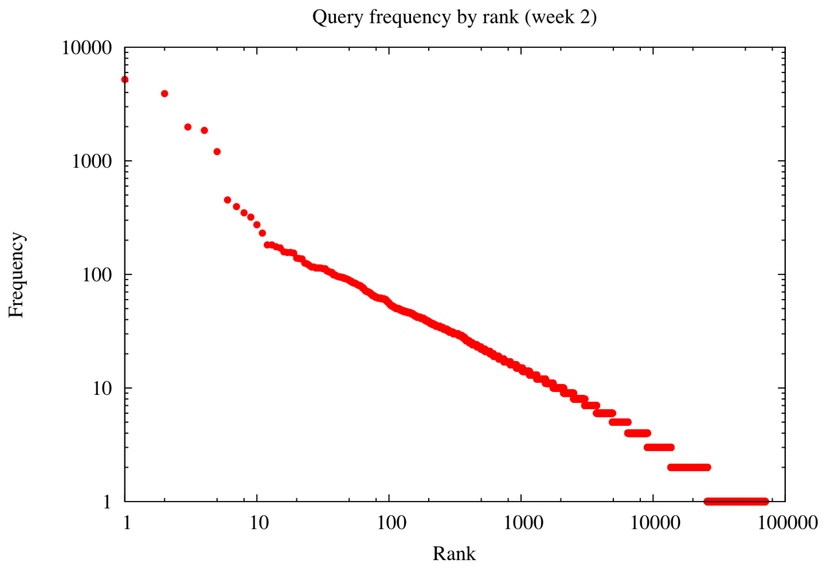


Figure 8.2: Distribution of query frequencies for week 2.

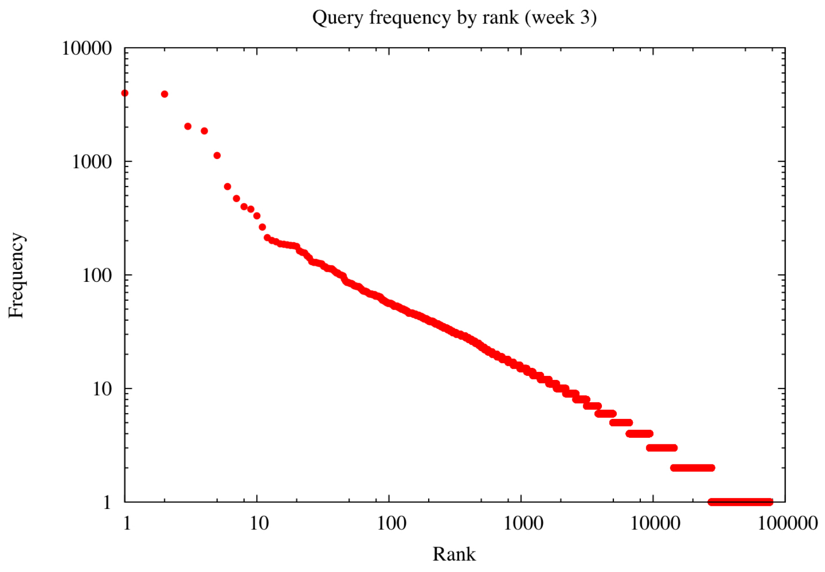


Figure 8.3: Distribution of query frequencies for week 3.

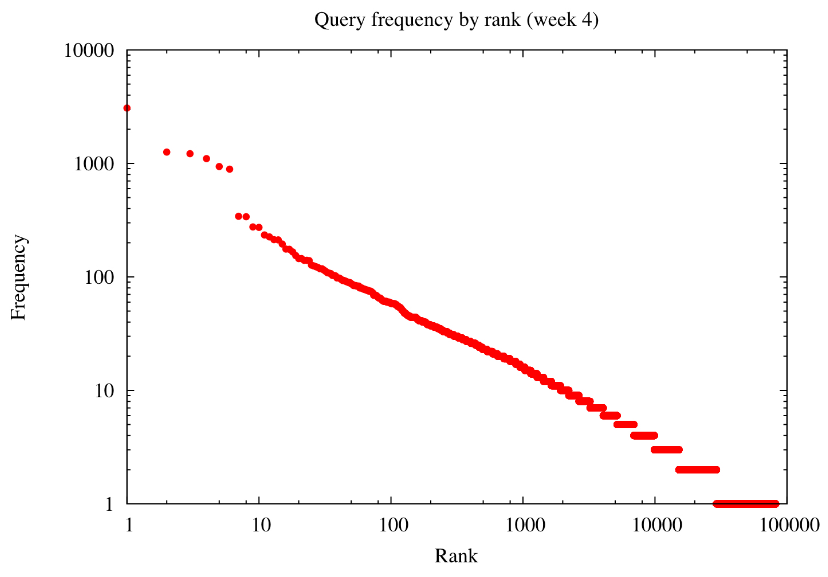


Figure 8.4: Distribution of query frequencies for week 4.

The query "mormons" has a very interesting pattern. Its frequency is overall high, but with big fluctuation. After March 2003, the query completely loses users' interest. Our query logs does not offer any information to explain this phenomenon (see Figure 8.5).

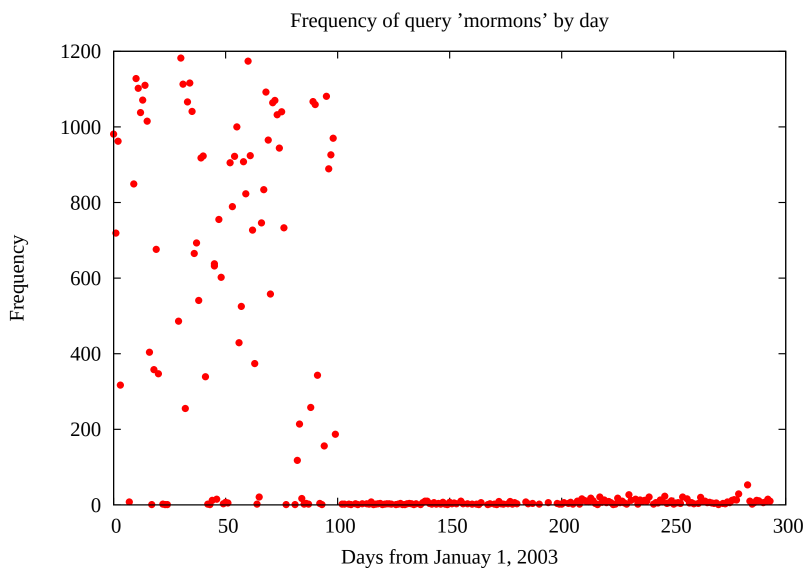


Figure 8.5: Frequency of query "mormons" by day

Chapter 9

Conclusions

In this thesis, we introduced a distributed architecture for a Web search engine, based on the concept of collection selection. We introduced a novel approach to partition documents in a distributed architecture, able to greatly improve the effectiveness of standard collection selection techniques (CORI), and a new selection function outperforming the state of the art. Our technique is based on the analysis of query logs, and on co-clustering queries and documents at the same time.

Incidentally, our partitioning strategy is able to identify documents that can be safely moved out of the main index (into a supplemental index), with a minimal loss in result accuracy.

By suitably partitioning the documents in the collection, our system is able to select the subset of servers containing the most relevant documents for each query. Instead of broadcasting the query to every server in the computing platform, only the most relevant will be polled, this way reducing the average computing cost to solve a query.

Our combined approach to partitioning and selection is very effective. By querying only the first server returned by our selection function, we can retrieve about 40% of the results that a full index would return. The accuracy perceived by the user (competitive similarity, see Section 3.6) is much higher, up to 75% for the similarity of the top 5 documents, because the selected server has highly relevant documents, even if not the very first. Alternatively, we can cover more than 60% documents from the full index by polling only 3 servers (out of 16, plus the overflow cluster).

We introduced also a novel strategy to use the instant load at each server as a driver to query routing: the less loaded servers can be polled, even when they are not the most relevant for a given query. Also, we describe a new approach to caching, able to incrementally improve the quality of the stored results. By combining these two techniques, we can achieve extremely high figures of precision, with a reduced load w.r.t. full query broadcasting.

By combining these two approaches, we can reach a 65% competitive recall or 80% competitive similarity, with a computing load of 15.6%, *i.e.* a peak of 156 queries out of a shifting window of 1000 queries. This means about 1/4 of the peak load reached by a broadcast of queries (reaching, clearly, 100%). The system, with a slightly higher load (24.6%), can reach a whooping 78% recall. We can restate this in a very dramatic way: we can return more than 3/4 of the results, with less than 1/2 the computing load.

All these tests were performed on a large simulator, on a base of 6 million documents, with 190,000 queries for training and 800,000 for tests. We verified that the results are still valid in an extended model which uses the real timing of queries on an actual search engine running on each document partition.

The proposed architecture, overall, presents a trade-off between computing cost and result quality, and we show how to guarantee very precise results in face of a dramatic reduction to computing load. This means that, with the same computing infrastructure, our system can serve more users, more queries and more documents.

We strongly believe that the proposed architecture has the potential of greatly reducing the computing cost of solving queries in modern IR system, contributing to saving energy, money, time and computer administrators!

9.1 Possible Extensions

Our approach to partitioning and selection is very general, as it makes no assumption on the underlying search strategies: it only suggest a way to partition documents so that the subsequent selection process is very effective. We used this approach for implementing a distributed Web search engine, but it could be used with different types of data, or with different architectures.

In fact, our strategy could be used *e.g.* in a image retrieval system, as long as similar queries return similar images, so that the co-clustering algorithm can safely identify the clusters. Or, it could be used in a peer-to-peer search network.

Even more interesting, there is no need for a centralized reference search engine. The training can be done over the P2P network itself, and later documents could be reassigned (or better duplicated).

Another interesting aspect of this thesis is the analysis of the caching system. We tested several algorithms, in combination with our incremental strategy. More aggressive solutions could be tested. The incremental cache could actively try to complete partial results by polling idle servers. This can be particularly valuable for queries that get *tenured* in the cache. In Web search engines, in fact, it is really hard to determine which queries should be cached, which should not, which should be moved to the static section and never be removed (maybe only updated if new results become available).

We did not investigate the interaction of our architecture with some advanced features of modern engines, such as query suggestion, query expansion, universal querying (*i.e.* the ability of returning results of different kind, such as images, videos and books [40]) and so on. We believe that our system has a big potential for query expansion and suggestion, as it is already able to identify similar queries returning similar documents.

Bibliography

[1] Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 226–233, New York, NY, USA, 2005. ACM Press.

[2] Vo Ngoc Anh and Alistair Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 372–379, New York, NY, USA, 2006. ACM Press.

[3] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Trans. Inter. Tech.*, 1(1):2–43, 2001.

[4] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3):592–608, 2007.

[5] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *Proceedings of the Eighth International Symposium on String Processing and Information Retrieval (SPIRE 2001)*, pages 10–20, 2001.

[6] Claudine Santos Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Distributed processing of conjunctive queries. In *HDIR '05: Proceedings of the First International Workshop on Heterogeneous and Distributed Information Retrieval (HDIR'05)*, SIGIR 2005, Salvador, Bahia, Brazil, 2005.

[7] Ricardo Baeza-Yates, Carlos Castillo, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *To Appear in Proceedings of The 30th Annual International ACM SIGIR Conference*, 2007.

[8] Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges in distributed information retrieval (invited paper). In *ICDE*, 2007.

[9] Ricardo Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE 2003 : string processing and information retrieval (Manaus, 8-10 October 2003)*, 2003.

[10] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] L.A. Barroso, J. Dean, and U. Hölze. Web search for a planet: The google cluster architecture. *IEEE Micro*, 22(2), Mar/Apr 2003.

[12] Steven M. Beitzel, Eric C. Jensen, Abdur Chowdhury, David Grossman, and Ophir Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR*, 2004.

[13] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Minerva: Collaborative p2p search. In *VLDB*, 2005.

[14] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 595–602, New York, NY, USA, 2004. ACM Press.

[15] S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.

[16] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[17] Fidel Cacheda, Vassilis Plachouras, and Iadh Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *Inf. Process. Manage.*, 41(5):1141–1161, 2005.

[18] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Trans. Inf. Syst.*, 18(1):1–43, 2000.

[19] J. Callan. Distributed information retrieval, 2000.

[20] J. Callan, A. Powell, J. French, and M. Connell. The effects of query-based sampling on automatic database selection algorithms, 2000.

[21] Jamie Callan and Margaret Connell. Query–Based Sampling of Text Databases.

[22] J.P. Callan, Z. Lu, and W.B. Croft. Searching Distributed Collections with Inference Networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, WA, July 1995. ACM Press.

[23] Flavio Chierichetti, Alessandro Panconesi, Prabhakar Raghavan, Mauro Sozio, Alessandro Tiberi, and Eli Upfal. Finding near neighbors through cluster pruning. In *Proceedings of ACM SIGMOD/PODS 2007 Conference*, 2007.

[24] Bhaumik Chokshi, John Dyer, Thomas Hernandez, and Subbarao Kambhampati. Relevance and overlap aware text collection selection. Technical Report TR 06-019, Arizona State University, Tempe, Arizona, September 2006. Available at http://rakaposhi.eas.asu.edu/papers.html.

[25] Abdur Chowdhury, Ophir Frieder, David Grossman, and Mary Catherine McCabe. Collection statistics for fast duplicate document detection. *ACM Trans. Inf. Syst.*, 20(2):171–191, 2002.

[26] Abdur Chowdhury and Greg Pass. Operational requirements for scalable search systems. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 435–442, New York, NY, USA, 2003. ACM Press.

[27] N. Craswell, P. Bailey, and D. Hawking. Server Selection on the World Wide Web. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, pages 37–46, 2000.

[28] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Proceedings of The Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD-2003)*, pages 89–98, 2003.

[29] R. Dolin, D. Agrawal, L. Dillon, and A. El Abbadi. Pharos: A Scalable Distributed Architecture for Locating Heterogeneous Information Sources. Technical Report TRCS96–05, 5 1996.

[30] Christoph Baumgarten Email. A Probabilistic Solution to the Selection and Fusion Problem in Distributed Information Retrieval.

[31] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore 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.

[32] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262, New York, NY, USA, 1999. ACM Press.

[33] James C. French, Allison L. Powell, James P. Callan, Charles L. Viles, Travis Emmitt, Kevin J. Prey, and Yun Mou. Comparing the Performance of Database Selection Algorithms. In *Research and Development in Information Retrieval*, pages 238–245, 1999.

[34] James C. French, Allison L. Powell, and Jamie Callan. Effective and Efficient Automatic Database Selection. Technical Report CS-99-08, Carnegie Mellon University, 2 1999.

[35] James C. French, Allison L. Powell, Charles L. Viles, Travis Emmitt, and Kevin J. Prey. Evaluating Database Selection Techniques: A Testbed and Experiment. In *Research and Development in Information Retrieval*, pages 121–129, 1998.

[36] O. Frieder, D. Grossman, A. Chowdhury, and G. Frieder. Efficiency considerations in very large information retrieval servers. *Journal of Digital Information*, 1(5), April 2000.

[37] Ophir Frieder and Hava Tova Siegelmann. On the allocation of documents in multiprocessor information retrieval systems. In *SIGIR '91: Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 230–239, New York, NY, USA, 1991. ACM Press.

[38] N. Fuhr. A Decision–Theoretic Approach to Database Selection in Networked IR. In Workshop on Distributed IR, 1996.

[39] N. Fuhr. Optimum Database Selection in Networked IR. In *Proceedings of the SIGIR'96 Workshop Networked Information Retrieval*, Zurich, Switzerland, 1996.

[40] Google Press Center. Google begins move to universal search. Available at http://www.google.com/intl/en/press/pressrel/universalsearch_20070516.html, accessed on May 20, 2007.

[41] L. Gravano and H. Garcia–Molina. Generalizing GlOSS to Vector–Space Databases and Broker Hierarchies. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[42] L. Gravano, H. Garcia–Molina, and A. Tomasic. Precision and Recall of GlOSS Estimators for Database Discovery. Stanford University Technical Note Number STAN-CS-TN-94-10, 1994.

[43] L. Gravano, H. Garcia–Molina, and A. Tomasic. The Effectiveness of GlOSS for the Text Database Discovery Problem. In *Proceedings of the SIGMOD 94*. ACM, September 1994.

[44] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The Efficacy of GlOSS for the Text Database Discovery Problem. Technical Report CS-TN-93-2, Stanford University, 1993.

[45] David Hawking and Paul Thistlewaite. Methods for Information Server Selection. *ACM Transactions on Information Systems*, 17(1):40–76, 1999.

[46] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, Inc., Orlando, FL, USA, 1978.

[47] Timothy C. Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003.

[48] Internet World Stats. Worldwide internet users. Available at http://www.internetworldstats.com/stats.htm, accessed on April 12, 2007.

[49] Paul Ogilvie Jamie. The Effectiveness of Query Expansion for Distributed Information Retrieval.

[50] Bernard Jansen and Amanda Spink. How are we searching the World Wide Web? A comparison of nine search engine transaction logs. *Inf. Proc. & Management*, 42:248–263, 2006.

[51] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Real life information retrieval: a study of user queries on the web. *SIGIR Forum*, 32(1):5–17, 1998.

[52] D. Jia, W. G. Yee, L. Nguyen, and O. Frieder. Distributed, automatic file descriptor tuning in peer-to-peer file-sharing systems. In *IEEE Seventh International Conference on Peer-to-Peer Computing (P2P)*, September 2007.

[53] Trevor Jim and Dan Suciu. Dynamically Distributed Query Evaluation. In *Proceedings of the PODS 2001 conference*, Santa Barbara, CA, May 2001. ACM.

[54] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[55] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computer Survey*, 1999.

[56] Leah S. Larkey, Connel, Margaret, and Jamie Callan. Collection selection and results merging with topically organized u.s. patents and trec data. Proceedings of Ninth International Conference on Information Knowledge and Management, November 6–10 2000.

[57] Leah S. Larkey, Margaret E. Connell, and Jamie 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.

[58] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, 2003.

[59] Nicholas Lester, Alistair Moffat, and Justin 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.

[60] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[61] Mei Li, Wang-Chien Lee, and Anand Sivasubramaniam. Semantic small world: An overlay network for peer-to-peer search. *12th IEEE International Conference on Network Protocols (ICNP'04)*, 2004.

[62] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal. Dynamic maintenance of web indexes using landmarks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 102–111, New York, NY, USA, 2003. ACM Press.

[63] Xiaoyong Liu and W. Bruce 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.

[64] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, 2005.

[65] Z. Lu, J. Callan, and W. Croft. Measures in Collection Ranking Evaluation, 1996.

[66] Zhihong Lu and K.S. McKinley. *Advances in Information Retrieval*, chapter 7. The Effect of Collection Organization and Query Locality on Information Retrieval System Performance and Design. Kluwer, New York. Bruce Croft Editor, 2000.

[67] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Proceedings of the Second Infoscale International Conference*, 2007.

[68] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 209, Washington, DC, USA, 2000. IEEE Computer Society.

[69] Evangelos P. Markatos. On Caching Search Engine Query Results. *Computer Communications*, 24(2):137–143, 2001.

[70] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[71] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.*, 19(3):217–241, 2001.

[72] Alistair Moffat, William Webber, and Justin 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.

[73] Alistair Moffat and Justin Zobel. Information Retrieval Systems for Large Document Collections. In *Proceddings of the Text REtrieval Conference*, pages 85–94, 1994.

[74] J. Nielsen. Information foraging: Why google makes people leave your site faster. Available at: http://www.useit.com/alertbox/20030630.html, 2003.

[75] Stefan Podlipnig and Laszlo Boszormenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[76] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *ICDE*, 2007.

[77] A. Powell, J. French, J. Callan, Connell, and C. M. Viles. The Impact of Database Selection on Distributed Searching. In *Proceedings of the SIGIR 2000 conference*, pages 232–239. ACM, 2000.

[78] Vijay V. Raghavan and Hayri Sever. On the reuse of past optimal queries. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 344–350, 1995.

[79] Anand Rajamaran and Jeffrey D. Ullman. Querying Websites Using Compact Skeletons. In *Proceedings of the PODS 2001 conference*, Santa Barbara, CA, May 2001. ACM.

[80] Keith H. Randall, Raymie Stata, Janet L. Wiener, and Rajiv G. Wickremesinghe. The link database: Fast access to graphs of the web. In *DCC '02: Proceedings of the Data Compression Conference (DCC '02)*, page 122, Washington, DC, USA, 2002. IEEE Computer Society.

[81] Yves Rasolofo. Approaches to Collection Selection and Results Merging for Distributed Information Retrieval.

[82] Yves Rasolofo, Faiza Abbaci, and Jacques Savoy. Approaches to collection selection and results merging for distributed information retrieval, 2001.

[83] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[84] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 182–190, New York, NY, USA, 1998. ACM Press.

[85] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[86] Yipeng Shen, Dik Lun Lee, and Lian Wen Zhang. A Distributed Search System Based on Markov Decision Processes. In *Proceedings of the ICSC 99 conference*, Honk Kong, December 1999.

[87] Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, 2005.

[88] Milad Shokouhi, Justin Zobel, Seyed M.M. Tahaghoghi, and Falk Scholer. Using query logs to establish vocabularies in distributed information retrieval. *Journal of Information Processing and Management*, 43(1), 2007.

[89] L. Si and J. Callan. Relevant document distribution estimation method for resource selection, 2003.

[90] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. In *ACM SIGIR Forum*, pages 6–12, 1999.

[91] Fabrizio Silvestri. Personal communication, 2007.

[92] Fabrizio Silvestri. Sorting out the document identifier assignment problem. In *Proceedings of the 29th European Conference on Information Retrieval*, April 2007.

[93] G. Skobeltsyn and K. Aberer. Distributed cache table: Efficient query-driven processing of multi-term queries in p2p networks. In *P2PIR*, 2006.

[94] T. Suel, C. Mathur, J. wen Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB*, 2003.

[95] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks, 2002.

[96] Anthony Tomasic and Hector Garcia-Molina. Performance of Inverted Indices in Distributed Text Document Retrieval Systems. In *PDIS*, pages 8–17, 1993.

[97] Anthony Tomasic, Luis Gravano, Calvin Lue, Peter Schwarz, and Laura Haas. Data Structures for Efficient Broker Implementation. *ACM Transactions on Information Systems*, 15(3):223–253, 1997.

[98] C.J. Van Rijsbergen. *Information Retrieval.* Butterworths, 1979.

[99] William Webber, Alistair Moffat, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 2006. published online October 5, 2006.

[100] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, May 1999.

[101] Yinglian Xie and David R. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM*, 2002.

[102] Xu, Jinxi, and W.B. Croft. Effective Retrieval with Disributed Collections. In *Proceedings of SIGIR98 conference*, Melbourne, Australia, August 1998.

[103] Jinxi Xu and W. Bruce Croft. Cluster-Based Language Models for Distributed Retrieval. In *Research and Development in Information Retrieval*, pages 254–261, 1999.

[104] Budi Yuwono and Dik Lun Lee. Server Ranking for Distributed Text Retrieval Systems on the Internet. In *Database Systems for Advanced Applications*, pages 41–50, 1997.

[105] Jiangong Zhang and Torsten Suel. Optimized inverted list assignment in distributed search engine architectures. In *IPDPS '07: the 21st IEEE International Parallel Distributed Processing Symposium (IPDPS'07)*, March 2007.