Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval

T. Suel C. Mathur Jo-Wen Wu J. Zhang
A. Delis M. Kharrazi X. Long K. Shanmugasundaram

Department of Computer and Information Science

Technical Report
TR-CIS-2003-01
06/20/2003

ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval*

Torsten Suel† **Chandan Mathur** **Jo-Wen Wu** **Jiangong Zhang**
Alex Delis **Mehdi Kharrazi** **Xiaohui Long** **Kulesh Shanmugasundaram**

Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201

Abstract

We consider the problem of building a P2P-based search engine for massive document collections. We describe a prototype system called ODISSEA (Open DIStributed Search Engine Architecture) that is currently under development in our group. ODISSEA provides a highly distributed global indexing and query execution service that can be used for content residing inside or outside of a P2P network. ODISSEA is different from many other approaches to P2P search in that it assumes a two-tier search engine architecture and a global index structure distributed over the nodes of the system.

We give an overview of the proposed system and discuss the basic design choices. Our main focus is on efficient query execution, and we discuss how recent work on top-k queries in the database community can be applied in a highly distributed environment. We also give some preliminary simulation results on a real search engine log and a terabyte-size web page collection that indicate good scalability for our approach.

*Project homepage: `http://cis.poly.edu/westlab/odissea/`. A preliminary version of this paper appeared at the International Workshop on the Web and Databases, June 2003.

†Contact author. Email: `suel@poly.edu`

1 Introduction

Due to the large size of the Web, users increasingly rely on specialized tools to navigate through the vast volumes of data, and a number of search engines, directories, and other IR tools have been built to fill this need. While there is a plethora of smaller specialized engines and directories, the main part of the search infrastructure of the web is supplied by a handful of large crawl-based search engines, such as Google, Inktomi, AltaVista, and a few others. Such search engines are typically based on scalable clusters, consisting of a large number of low-cost servers located at one or a few locations and connected by high-speed local area or system area networks [7]. A lot of work has focused on optimizing performance on such architectures, which support up to tens of thousands of user queries per second on thousands of machines.

The last few years have also seen an explosion of activity in the area of peer-to-peer (P2P) systems, i.e., highly distributed computing or service substrates built from thousands or even millions of typically non-dedicated nodes across the internet that may join or leave the system at any time. Examples range from widely used unstructured ad-hoc communities such as Napster, Gnutella, and FreeNet to recent academic work on scalable and highly structured peer-to-peer substrates such as Chord [45], Tapestry [53], Pastry [42], or CAN [38] that can support a variety of applications.

From the perspective of search engines and large-scale IR this development raises two interesting issues. First, since an increasing amount of content now resides in P2P networks, it becomes necessary to provide search facilities within P2P networks. Second, the significant computing resources provided by a P2P system could also be used to implement search and data mining functions for content located outside the system, e.g., for search and mining tasks across large intranets or global enterprises, or even to build a P2P-based alternative to the current major search engines. This second issue can be seen in the context of the following more general question: Which of the *Giant Scale Services* [7] currently provided by cluster-based architectures can and should be provided by more highly distributed or P2P systems? It has been established that applications such as the sharing of large static files can be very efficiently implemented in a P2P environment. However, other applications that, e.g., involve frequent updates to massive data, are more challenging, and may turn out to be more appropriately implemented on clusters or on highly-robust distributed systems of dedicated nodes with limited changes in topology (due to faults, or nodes joining or leaving).

In this paper, we describe a prototype system called ODISSEA (Open DIStributed Search Engine Architecture) that is currently under development in our group. ODISSEA attempts to address both of the above issues, by providing a "distributed global indexing and query execution service" that can be used for content residing inside or outside of a P2P network. ODISSEA is different in several ways from many other approaches to P2P search, as explained below. It encounters some basic challenges typical of those that arise when implementing more dynamic applications involving frequent updates on P2P systems, leading to interesting algorithmic problems and solutions. We describe and discuss the basic design choices and motivation and give some initial results, with focus on the issue of efficient distributed query processing.

1.1 ODISSEA Design Overview

ODISSEA is a distributed global indexing and query execution service, i.e., a system that maintains a global index structure under document insertions and updates and node joins and failures, and that executes simple but general classes of search queries in an efficient manner. This system provides the lower tier of a proposed two-tier search infrastructure. In the upper tier, there are two classes of clients that interact with this P2P-based lower tier:

1. *Update clients* insert new or updated documents into the system, which stores and indexes them. An update client could be a crawler inserting crawled pages,[1] or a web server pushing documents into the index, or a node in a file sharing system.

[1] Thus, crawling is not included as part of the lower tier, as justified by our subsequent discussion.

2. *Query clients* design optimized query execution plans, based on statistics about term frequencies and correlations, and issue them to the lower tier. Ideally, the lower tier should enable query clients to use or implement a variety of different ranking methods.

There are two main differences that distinguish ODISSEA from most other P2P search systems. First, the assumption of a two-tier architecture that aims to give as much freedom as possible to clients to implement their own user interfaces and search and ranking policies. This is motivated by the goal of providing an "open" search infrastructure that allows the creation of a rich variety of client-based search and navigation tools running on user desktops. There are trade-offs between efficiency and flexibility that may limit the full realization of this goal, and one of our main research goals is to investigate these trade-offs.

The second difference is our assumption of a *global inverted index* structure. Many current approaches[2] to full-text search in P2P systems assume a *local inverted index*, where each node maintains an inverted index for all local documents (or the documents of a few surrounding nodes), and queries have to be broadcast to all, or on average at least a significant fraction, of the nodes, in order to get the best results. In a global index, the inverted index for a particular term (word) is located at a single node, or partitioned over a small number of nodes in some hybrid organizations. Thus, queries with multiple keywords require "combining" the data for the different keywords over the network, at a cost that can be quite considerate. We discuss this decision in detail later, and it has some consequences for the overall design.

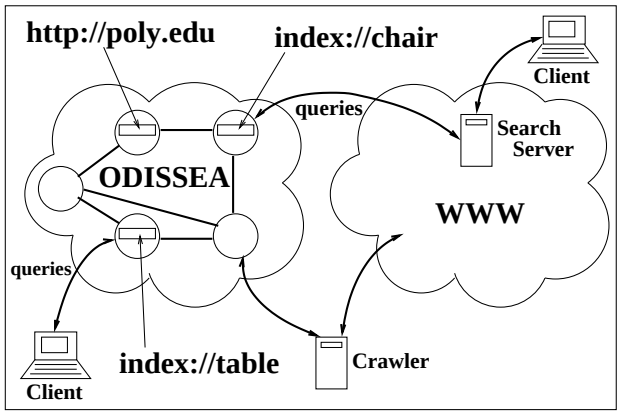

Figure 1: ODISSEA as a web search infrastructure, with a web crawler as update client, and two query clients (one client-based and one as a web-based search service). Also shown are indexes for the words "chair" and "table", and a node holding the document `http://poly.edu`.

Figure 1 shows the basic design. We decided to implement the system on top of an underlying global address space provided by a DHT structure, in particular Pastry [42]. Each object is identified by a hash of its *name*, where the name is the URL of a document or a string such as `index://chair` for the index structure for the term "chair", and is assigned a location determined by the DHT mapping scheme. Thus, the only way to move an object is to rename it, resulting in a mapping to a random other node. (We note that the real mapping scheme used by us is actually more complicated, to enable replication and load balancing.)

1.2 Target Applications

We have four main application scenarios that motivate our research, in particular:

(1) **Search in P2P networks:** To provide full-text search facilities for large document collections located within P2P communities.

(2) **Search in large intranet environments:** Large organizations may use distributed search applications to share machine resources, within a more controlled and possibly less bandwidth-constrained environment.

[2] See [20, 29, 39] for exceptions.

(3) **Web search:** Our most ambitious application is a P2P-based search infrastructure for the web that provides an alternative to the major search engines, with a powerful API (more low-level than, e.g., the Google API) that supports the anticipated shift towards client-based search tools that exploit the resources of todays desktop machines. We admit that this scenario may not be feasible in the near future but we believe it still deserves study.

(4) **Search middleware:** Instead of inserting documents, clients could directly insert "postings", i.e., index entries. The system would then act as "global middleware" on top of a system of local index structures, where nodes might periodically insert some of their postings into the system. The middleware could then use a combination of local and global indexes, resulting in increased efficiency for certain types of queries.

Paper outline: In the next section we justify our main design decisions and assumptions. Section 3 gives more details about the system design. Technical details and preliminary experimental results on query processing are provided in Section 4. Section 5 discusses related work, and finally Section 6 mentions some open problems. Updated information on this project is available at `http://cis.poly.edu/westlab/odissea/`.

2 Discussion and Justification

Two-tier approach: This choice was originally motivated by the web search application scenario. Given the expected increases in speed and bandwidth of desktop systems, we see the potential for a rich variety of novel search and navigational tools and interfaces that more fully exploit client computing resources, and that rely on access to a powerful lower-level web search infrastructure. These tools may perform a large number of web server or search engine accesses during a single user interaction, in order to prefetch, analyze, aggregate, and render content from various sources into a highly optimized form. Existing early examples of these types of client-based tools are browsing assistants such as the Alexa and Google Toolbars, Zapper, Leticia and PowerScout [30], the Stanford Power Browser [11], or tools built with the Google API. In addition, specialized search engines (Google News, citeSeer) or meta search engines could also be supported by such an infrastructure.

Thus, the proposed system could be used to provide such a lower-level search infrastructure, with an powerful open and *agnostic* API that is accessed by client- and proxy-based tools. By *agnostic*, we mean an API that is not limited to a single method for ranking pages (e.g., the Google API, which returns pages according to Google's ranking strategy), but that ideally allows clients to implement their own ranking strategies. There clearly are limits and trade-offs to this goal. The most general solution of performing most of the ranking at the client requires large amounts of data to be transferred. On the other hand, we conjecture that limited but powerful classes of ranking functions could be efficiently supported by providing appropriate "hooks" and algorithmic techniques inside the system.

Given such an API a variety of client-based tools could share the same lower-level search infrastructure. This issue is also related to the perceived "barrier of entry" in the search engine market. A lot has been written in the trade press about the consolidation towards a few major players, and in particular about the dominating role of the Google engine. We note that for a simple large-scale search engine with a limited query load, the obstacles are actually not that high, and a small group of determined programmers could build such a system in a few months on maybe a dozen low-cost machines.[3] Nonetheless, we believe this still presents a significant obstacle to the creation of a wider variety of client-based tools, since developers would rather prefer to focus on the tool itself than on building lower-level infrastructure. The wide response to the release of the Google API shows the need for such an infrastructure, and we are envisioning are more low-level and general API based on a P2P infrastructure.

Global vs. local index: The other important decision is the use of a global index instead of the more

[3] In fact, this is how many of the current major players started out, and there are several attempts underway to build scalable open source search engines.

commonly used local index organization.[4] We now define some terms. First, an *inverted index* for a document collection is a data structure that contains for each word in the collection a list of all its occurrences, or a list of *postings*. Each *posting* contains the document ID of the occurrence of the word, its position inside the document, and possibly other information such as whether the word is in the title or in bold face. Each postings list is best visualized as a simple array, maybe sorted by document ID.

In a local index organization, each node creates its own index for all documents that are locally stored. Thus, every node will have its own small postings list for common words such as "chair" or "table", and a query "chair, table" is first broadcast to all nodes and then the results are combined. In a global index organization, each node will hold a complete global postings list for a subset of the words, as determined, e.g., by a hash function. Thus, every node will have a smaller number of longer lists, and under the standard query evaluation strategy a query "chair, table" is first routed to the node holding the list for "chair" (the shorter list), which then sends its complete list to the node holding the list for "table". We emphasize here that our approach does in fact not send the entire list, as explained later. The two index organizations are illustrated in Figure 2. There have been a number of performance comparisons between local and global index organizations and several hybrid organizations on parallel architectures; see, e.g., [4, 12, 48], but these studies do not directly apply to widely distributed environments.

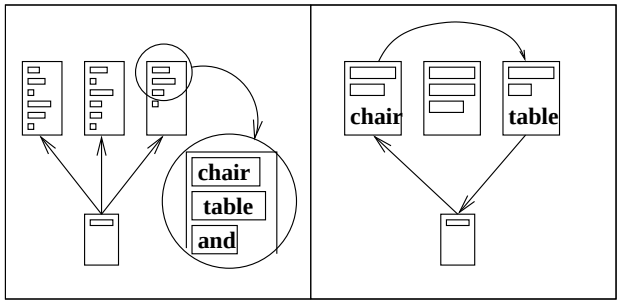

Figure 2: Query processing in a local (left) and global index organization.

The main issue with local index organizations is that all or most nodes need to be contacted for most queries, and thus such schemes are unlikely to scale beyond a few hundred nodes. There have been attempts to overcome this issue by routing queries only to those nodes that are likely to have good results[5] or are in the vicinity [27, 43, 16]. However, we do not believe that this approach will scale at all if result quality is a major concern, since document collections are simply not naturally clustered in a way that allows queries to be routed to only a small fraction of the nodes. This is certainly the case for the current web, where a search infrastructure based on local indexes at each site would be extremely inefficient. This could be somewhat improved by clustering the entire document collection appropriately, though this seems quite challenging to do [29]. Moreover, the statistics needed to intelligently route queries will be quite large for large collections and many nodes since the number of distinct words grows with collection size; the existing literature has only evaluated fairly small collections of maybe a few gigabytes.

In a global index organization, however, large amounts of data need to be transmitted between nodes, since large document collections result in lists of megabytes or more for all except fairly rare words. This problem has led some people to reject global indexes as unrealistic for environments with more limited bandwidth, and for moderate numbers of nodes a local index is probably a better choice. However, we believe that this problem can be overcome by the use of smart algorithmic techniques. One such technique was recently applied in this context in [29, 39], where Bloom filters are used to decrease the cost of intersecting lists of postings over the network, though this only improves results by a constant factor. We investigate in Subsection 4.1 how

[4] These two organizations are also sometimes referred to as vertical and horizontal index partitioning [32]. We avoid these terms here as they tend to lead to confusion with standard database terminology.

[5] This problem is also known as the database selection problem in the meta search community [33].

recent results on top-k queries in the database literature [18] can be applied to our scenario to asymptotically reduce communication requirements. We believe that these techniques, combined with other query optimization techniques, allow interactive response times even on very large data sets.

Updates are another challenge for a global index structure. A new or updated document will result in a large number of index updates, one per word in the document, that now have to be routed to the different nodes responsible for these words. Thus, there will be a persistent very fine-grained communication pattern among the nodes of the system that needs to be implemented carefully. Finally, we note that the choice between local and global index structures also depends on the types of queries and the frequency of document updates; this motivated the "middleware" application in the previous section where the best combination of index structures is used.

Crawling punt: As mentioned, we assume that in the web search application crawling would be performed by *crawling clients* that fetch and insert documents. The main reason is that from our own experiences with large-scale crawling [44] we are not sure a P2P solution is appropriate. Large crawls generate many management issues due to queries or complaints from web site operators and network administrators. It is important to be able to reconfigure a crawler quickly to avoid certain web sites or subnetworks or to modify its behavior, and failure to do so can result in problems with local administrators or upstream network providers.[6] Moreover, smart crawling strategies beyond BFS are hard to implement in a P2P environment unless there is a centralized scheduler. We refer to [6, 14, 15] for work on highly distributed crawling.

Thus, we would expect that a handful of powerful crawling clients would provide most documents, and we plan to use our Polybot crawler [44] to initially populate the system with data. It might be more appropriate to incorporate recrawling into the system, though. Thus, an inserted page could be labeled with an expiration date, after which it is automatically refreshed by the node holding the page. Alternatively, web sites could also push their pages into the system.

P2P systems and fault tolerance: Utilizing idle computing, network, or disk resources is one of the main motivations for building P2P systems. However, there is a fundamental challenge facing applications that use large amounts of disk space on remote nodes, such as a search engine. Given current network speeds, it would take days or weeks to transfer enough data to a newly joined node to utilize any significant fraction of a 200 GB hard disk, and during this time the node would probably consume more resources than it adds to the system. Thus, it would appear that such applications are maybe best restricted to the more stable end of the P2P spectrum, where most nodes remain in the system for an extended period of time, and would be wasteful to implement on highly dynamic P2P systems.

Our system design also relies on this assumption of a more stable system. However, we distinguish between nodes that are temporarily unavailable and nodes that have permanently left the system. When a node rejoins the system after an extended period of unavailability, an interesting problem arises: how do we efficiently *synchronize* its data structures, in this case the index structures, with an up-to-date copy held by another node, to incorporate any changes that were missed during unavailability? We discuss this problem in more detail later. Other questions involve distinguishing between failed and unavailable nodes, when to rebuild data on failed or unavailable nodes, and how quickly data should be pushed to newly joined nodes.

3 System Design Details

In this section we describe our current system design in more detail. Note that query processing is described in the next section. Also, many additional details have to be omitted due to space constraints. We are currently implementing a first version in Java, using Pastry as a P2P substrate. Each node runs a modified version of a high-performance indexer that is being developed within our group, which stores inverted lists in compressed form in Berkeley DB. All documents are also stored in BDB.

[6] Of course, for certain types of crawling activities, e.g., to surreptitiously monitor certain web sites, a P2P solution may be preferable for the very same reasons.

Names and Hashing: As mentioned, a document has a name such as `http://www.cnn.com/` while an inverted list for a term (e.g., "chair") has a name such as `index://chair`. Using MD5, we hash these names to 80-bit IDs. All lookups into the system are performed using these IDs.

Parsing and Routing Postings: A newly inserted or updated document is parsed at the node where it resides, as determined by the DHT mapping. The parser generates a set of postings for the document, where each posting is a tuple containing the term (word), the 80-bit document ID, the position within the document, the length of the document, and the context of the term occurrence (in bold face, in the title, etc.). It is best to think about these postings as update commands that have to be applied to the index structure. In particular, there are commands to add and delete a term occurrence from the index, and for shifting an occurrence within a document by a given number of positions (for the case of updated pages). During transfer, postings are transformed into a slightly more succinct format.

Each posting must be propagated to the node holding the corresponding index in the ODISSEA network. Given the very fine-grained communication pattern resulting from this, it would not be a good idea to open direct TCP connections to the destinations. Instead, postings are routed via several intermediate destinations, as determined by the topology of the Pastry network. Once they arrive at the destination, postings are queued and then applied to the appropriate index structure. Due to their semantics, it does not matter in which order postings arrive and are applied, as long as we do not parse two different versions of the same document within a short period of time (in which case the resulting index for this document could become inconsistent.) Locally, index structures are organized to allow index updates with low amortized complexity. Thus, updates are inserted into a small index structure that is eventually merged with larger and larger structures on disk; similar techniques are, e.g., used by the Lucene indexer of the Apache Jakarta project and the text indexer of SQL Server.

Groups and Splits: Recall that objects are identified by an 80-bit ID. However, we do not directly present this ID to Pastry to determine the location of an object. Instead, we group a large number of objects into a *group*. Initially, all objects (documents, indexes) whose first w bits coincide (say, for $w = 16$) are placed into a common group identified by this w-bit string. Pastry then uses this string to determine where the group is located. Locally, each group maintains a Berkeley DB database with all objects it contains.

When a group becomes too large (say, larger than 1 GB), it is split into two groups identified by a $(w+1)$-bit string and objects are divided between the two groups in the obvious way. As a result, the new groups will be assigned to new locations by Pastry. However, a *stub* structure will remain accessible under the old w-bit label, and this stub structure maintains a table of all its descendants. In fact, each descendant strictly speaking also consists of a stub and the actual data. When a new node takes over some part of the Pastry address space, the appropriate stubs are immediately transferred to it, while the actual data can be moved later in a lazy fashion; see Figure 3.

We also have splitting rules for index structures. When the inverted list for a common term grows beyond a certain size (e.g., 100 MB), the list is split into two lists, containing postings with document IDs starting with "0" and "1", respectively. These new lists are renamed appropriately and thus assigned to new groups, with a stub remaining under the old name. Note that performing the split based on the document ID in this way results in simple and efficient query execution plans during query processing. However, only the largest lists are split.

Replication: Replication for fault tolerance is performed at the group level. For example, we may decide to have four replicas of each group, which are named by attaching "/0", "/1", "/2", and "/3" to its group label; this new label is then what is *really* presented to Pastry during lookups. We assume that all replicas of a group form a clique and periodically communicate to update their status; if a group replica fails, the others are in charge of detecting this and if necessary performing repair. Each node can contain a number of group replicas, and thus participates in a number of cliques; see Figure 3. Postings from a new or updated web page are first routed to only one replica of the appropriate index structure. This replica is then in charge of forwarding the postings to the other currently available replicas over a period of a few minutes.

Faults, Unavailability, and Synchronization: When a node leaves the system, its group replicas eventually have to be replaced to maintain the desired degree of replication. A problem is that we may not always be able

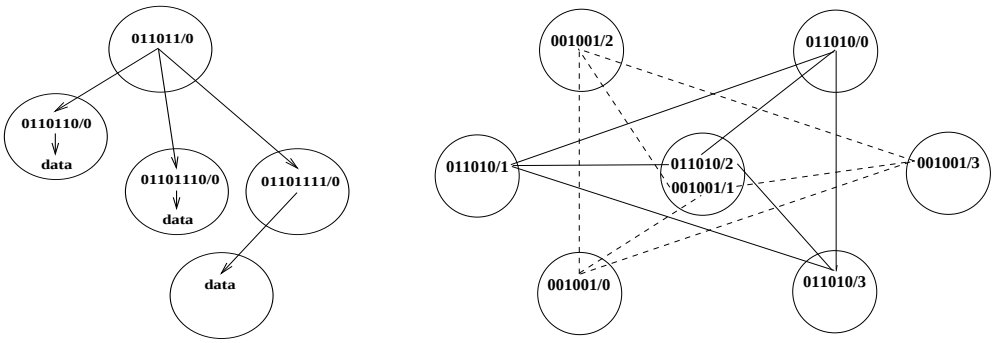

Figure 3: **Left:** A group has been split, resulting in three descendants. The node holding one of the descendants has only recently joined the Pastry address space, and thus the actual data associated with the descendant is still at its old location. **Right:** A node holding two group replicas, each shown with its clique.

to distinguish between failed and temporarily unavailable nodes. One solution is to declare a node failed if it has been unavailable for an extended period, and to create a new replica whenever a node has failed or more than a certain number of nodes are unavailable.

Once a node has been unavailable for a longer period of time, it needs to synchronize its index structures when it rejoins the system. Ideally, this should be done in a bandwidth-efficient manner, so that the amount of communication is proportional to the number of missed updates to the index. We note that there are a number of traditional approaches to such problems that rely on logs of previous updates or appropriate time stamps. We are currently studying the use of recently studied set reconciliation techniques to solve this problem without logs or time stamps; see [26, 35] for recent work and [46] for a survey.

4 Efficient Query Processing in ODISSEA

In this section we describe query processing in the proposed system. A naive implementation of ranked queries with a global index structure would result in transfers of many megabytes of data for many queries from a typical query load. Since realistic bandwidths for P2P applications in WAN environments are on the order of maybe a few hundred kilobytes per second, this would result in response times of many seconds or even minutes. We now describe how to adapt a recent set of techniques by Fagin and others [17, 18, 19] to our scenario, and give measurements of the expected savings from this approach based on a real search engine query log and a set of 120 million web pages from a recent crawl that we have carried out.

4.1 Some Background and Algorithmic Techniques

Ranking in search engines: We first give some background on ranking in search engines. Search engines rank pages based on many criteria, including classical term-based techniques from IR, global page ranks as provided by Pagerank [8] and similar methods, whether text is in bold face or within a hyperlink, and distances between the search terms in the documents, among others. Formally, a *ranking function* is a function F that, given a query consisting of a set of search terms $q_0, q_1, \ldots q_{m-1}$, assigns to each document d a score $F(d, q_0, \ldots, q_{m-1})$. The top-$k$ ranking problem is then the problem of identifying the k documents in the collection with the highest scores. We focus on two families of ranking functions, of the forms

$$F(x) = \sum_{i=0}^{m-1} f(d, q_i) \quad \text{and} \quad F(x) = g(d) + \sum_{i=0}^{m-1} f(d, q_i).$$

The first family includes the common families of term-based ranking functions used in IR, where we add up the scores of each document with respect to all words in the queries. In particular, this also includes the well-known class of *cosine methods*; see, e.g., [51]. The second formula adds a query-independent value $g(d)$ to the score of each page; this could for example be a suitably normalized Pagerank value. Thus, these two families include

many important ranking functions, and we could in fact use any other monotone function instead of addition to combine the various functions in the above formula. Note however that techniques using the distances between the query terms in a document would lead to an additional function $h(d, q_0, \ldots, q_{m-1})$ that depends on all query terms, and that this would impact the efficiency of the methods described in the following.

Queries to search engines have on average less than three terms, and search engines typically evaluate a query by considering all documents in the intersection of the inverted lists, i.e., all documents that contain all search terms.[7] An easy information-theoretic argument shows that determining the intersection of two lists located at different nodes requires transmitting an amount of data linear in the size of the shorter of the two lists. However, recent work in the database community [18] shows how to evaluate top-k queries without scanning over the entire intersection.

Fagin's Algorithm (FA): We now describe the first algorithm, which was originally proposed in [17] for the case of *multimedia queries*, e.g., to retrieve images from an image database. We will state them directly for our scenario, first for the case of the first family of ranking functions without $g(d)$. Intuitively, the algorithm exploits the fact that an item (document, university) that is ranked in the top is likely to be ranked very high in at least one of the contributing subcategories (term scores, departmental rankings).

Consider the inverted lists for a search query with two terms q_0 and q_1. For the moment, assume they are located on the same machine, and that the postings in the list are pairs of the form $(d, f(d, q_i))$, $i \in \{0, 1\}$, where d is an integer identifying the document and $f(d, q_i)$ is real-valued. Assume that each inverted list is sorted by the second attribute, so that documents with largest value of $f(d, q_i)$ are at the start of the list. Then the following algorithm, called *FA*, computes the top-k results:

(1) Scan both lists from the beginning, by reading one element from each list in every step, until there are k documents that have each been encountered in both of the lists.

(2) Compute the scores of these k documents. Also, for each document that was encountered in only one of the lists, perform a lookup into the other list to determine the score of the document. Return the k documents with the highest score.

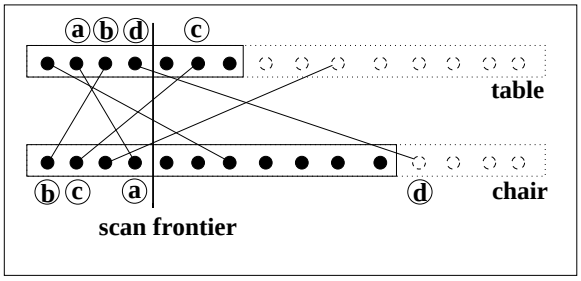

Figure 4: Fagin's Algorithm on the terms "chair" and "table". At this point, the first 4 postings in each list have been scanned, and two documents (a and b) have been encountered in both of the lists. If $k = 2$, then the scan of Step (1) is complete at this point. Other documents such as c and d have been encountered in only one list, and a random lookup is now needed to find them in the other list. In the case of d, the random lookup fails because d does not contain the word "chair". Documents not containing a term are shown in outlines at the end of each list, though they would not actually be stored in an inverted list implementation.

The algorithm is illustrated in Figure 4. It is not difficult to see that this indeed returns the top-k results overall. It is shown in [17] that if the orderings of documents in the two lists are independent, then the algorithm terminates after looking at only $O(\sqrt{kn})$ entries in each list, where n is the number of documents in the

[7] This is in contrast to "traditional" IR systems that tend to consider the union of the inverted lists, and where typical queries consist of a dozen terms or more. The results in this subsection do not really depend on this choice.

collection (not the length of the list). In the case of queries with m terms, the bound becomes $O(n^{\frac{m-1}{m}} k^{\frac{1}{m}})$. Thus, for long lists this significantly improves over scanning the entire list. If terms are positively correlated, then the result improves, while it gets worse for negatively correlated terms. Note that the result is independent of the actual "shapes" of the distributions of the $f(d, q_i)$, though refinements to the basic technique could potentially exploit special distributions such as Zipfians.

Threshold Algorithm (TA): The following refinement to the algorithm was proposed by several authors; see [18] for a discussion. In the refinement, we again simultaneously scan both lists, so that in each step we read an item $(d, f(d, q_0))$ from the first list and an item $(d', f(d', q_1))$ from the second list. In each step we compute $t = f(d, q_0) + f(d', q_1)$; note that d and d' will usually be different documents. Also, whenever we encounter a document in one of the lists, we immediately perform a lookup into the other list to compute its complete score. As soon as we have found k documents with score larger than the current value of t, we return these as the results. It can be shown that *TA* is correct and always terminates at least as early as *FA*, though the asymptotic bounds remain the same.

Integrating query-independent scores: We can naively adapt both algorithms to the second family of ranking functions as follows. Instead of sorting each list by $f(d, q_i)$, we sort by $f(d, q_i) + \frac{1}{2} \cdot g(d)$, so that the total score is the sum of the sort attributes from both lists. Note that from a performance point of view, this should in fact increase efficiency, as it introduces significant correlation between the ordering of the two lists.

However, in reality we cannot combine term-based and link-based scores simply by adding them up. Instead, it is preferable to normalize the scores in a query-dependent way that minimizes the effect of outliers. Following [40] we do this by normalizing using the mean of the top-100 term-based and link-based scores that appear in the two (or more) lists; see [40] for details. This means that the inverted lists cannot be completely organized in sorted order before the arrival of the query, though they can usually be kept approximately sorted. In our distributed setting this is not really a problem since we are interested in minimizing bandwidth consumption and not processing cost at the node. One further issue concerns the Pagerank value that we use in this context. After some experimentation, we concluded that rather than using the raw value as output by the iterative Pagerank computation, it is more appropriate to use the logarithm of this value; the resulting value distribution is more similar to that of the cosine values, while the raw scores are extremely skewed. However, we ran experiments for both cases and in general there are many possible approaches here. We note that Pagerank is only one of many possible global orderings; others might be based on a different link analysis technique, text analysis, or user access data.

A natural question that we do not address is how the global ordering is computed and made available to all nodes in the system. Again, Pagerank is only one of many possible orderings and we do not propose here to run Pagerank inside a P2P system. One alternative that fits into our vision of client-determined ranking functions is to allow clients to compute their own global orderings which can depend on their personal preferences [23, 24]. This ordering can then be treated as an additional term in each query posed by a client, and the query can be evaluated using the protocol proposed in this paper (though with slightly diminished efficiency due to the additional term, and with minor internal changes to incorporate normalization). An experimental evaluation of this approach remains to be done.

4.2 Experimental Results on Real Data

We run some initial experiments to determine the potential savings due to the above schemes. We note that these experiments are still in a centralized setting; we consider distributed implementations in the next subsection. There have been previous evaluations of the basic *FA* and *TA* algorithms on data sets from other application domains, but not on large-scale web data or in conjunction with global measures such as Pagerank. For such large collections, indexes typically reside on disk, and the random lookups make the proposed schemes unsuitable in a centralized scenario. The situation is different, however, in the distributed case or for scenarios where the indexes can fit into the large main memories of state-of-the-art machines.

For the experiments, we use queries selected from a log of over 1 million queries posted to the Excite search engine on December 20, 1999. For the reported experiments, we used 900 two-term queries selected from this

	top1	top4	top10	top100
Length of shorter list	879,010	879,010	879,010	879,010
Intersect	786	1,171	1,705	8,033
FA (cosine only)	4,638	8,012	12,445	39,306
TA (cosine only)	1,114	2,199	3,441	13,014
CA (cosine only)	371	931	1,729	9,429
FA (cosine + pagerank)	3,038	5,083	8,453	34,029
TA (cosine + pagerank)	677	1,346	2,255	11,333
CA (cosine + pagerank)	245	578	1,075	8,075
FA (cosine + log(PR))	944	1,410	2,137	9,709
TA (cosine + log(PR))	228	373	625	3,721
CA (cosine + log(PR))	97	166	288	2,102

Table 1: Average costs of the various methods on a data set of 120 million pages.

set. Our document collection consists of about 120 million web pages crawled by the Polybot crawler [44] in October 2002, for a total of about 1.8 TB of data. We removed queries containing words with more than 50 million postings (stop words such as "a", "the", "with", "as"), as is done by many engines, and we removed queries with less than 100 results in the intersection of the lists. For the first family of ranking functions, we used a standard cosine measure. For the second family, we defined $g(d)$ as an appropriately normalized Pagerank score computed from a web graph extracted from our crawl.

Table 1 shows the average number of postings that have to be accessed from each list under various algorithms on the 120 queries. In the first row we have the number of postings in the shorter of the two inverted lists; this represents the cost incurred by the unoptimized algorithm where we transmit the entire list. In the next line, we have the number of postings that are scanned if we are only interested in getting an arbitrary k elements that contain both query terms. This might be considered a reasonable lower bound[8] on what we could hope to achieve with the optimized methods, and was measured by ordering indexes by document ID and scanning from the beginning until k elements in the intersection are found. We note that this cost can in some cases be quite significant, say for two inverted lists of length $\Theta(\sqrt{n})$ with no correlation where we might have to scan most of the lists to find a single document containing both words. In the experiments, we also include an idealized algorithm called *CA* (Clairvoyant Algorithm) that stops as soon as it has encountered the top-k elements; this shows the cost between finding the top-k results and being certain that we have found them.

We show results for *FA*, *TA*, and *CA*, with and without Pagerank. As we see from the data, all three algorithms perform significantly better than the basic algorithm. The results for *TA* and *CA* show that we can usually terminate the scan much earlier without impact on the result. We also see that including the Pagerank score results in improved performance, in particular when using the logarithm of the raw score, while the extremely skewed raw Pagerank score gives less benefit. We also see that in a number of cases the algorithms perform better than the algorithm for finding k elements in the intersection; this is due to the fact that many of the terms in the queries are correlated so that documents scoring high in one term are also more likely to score high in the other term.

Table 2 gives a more detailed look at the results, grouped by the length of the shorter of the two inverted lists, i.e., for the one-fifth of queries with the shortest shorter inverted lists, the next one-fifth, and so on. We see that as predicted by the asymptotic results, we get the most benefit when it really counts, i.e., for queries that would be very expensive otherwise. Interestingly, including Pagerank helps even more for longer than for shorter lists. Figures 5 to 9 in the Appendix provide even more details, and show the costs of the methods for various combinations of list lengths. E.g., Figure 7 shows results for the middle quintile of the shorter lists, for different lengths of the other, longer list. As we see, performance varies quite a lot among the different

[8] If we discount correlations between query terms.

	shortest 20%		shorter 20%		middle 20%		longer 20%		longest 20%	
Shorter list	10,401		63,853		222,948		666,717		3,371,176	
	top10	top100	top10	top100	top10	top100	top10	top100	top10	top100
FA (cosine only)	5,298	7,452	16,115	34,786	17,361	59,105	13,541	52,214	10,017	43,058
TA (cosine only)	2,057	4,978	4,083	15,304	2,904	12,677	4,417	16,942	3,754	15,244
CA (cosine only)	705	3,031	2,251	9,609	1,285	9,332	2,495	13,126	1,916	12,131
FA (cos + log(PR))	3,691	7,582	3,990	21,730	1,672	11,109	865	5,623	498	2,615
TA (cos + log(PR))	889	3,922	922	6,762	610	3,587	407	2,834	304	1,534
CA (cos + log(PR))	227	1,696	403	3,283	352	2,309	261	2,032	197	1,200

Table 2: Average number of accesses for different quintiles of the length of the shorter list.

cases, and we do not yet have a complete explanation for the behavior. We note also that it is not clear that one should scan both lists at the same speed as done in Fagin's algorithm; maybe the shorter list should be scanned at higher (or lower ?) speed. We are working on a model to explain this behavior.

In summary the results indicate that an appropriate distributed protocol based on these algorithms might have the potential to achieve interactive response times in WAN environments even for the fairly large data set that we used.

4.3 A Simple Distributed Protocol

We now adapt these techniques to a highly distributed environment with limited bandwidth as well as high latency. Thus, we have to limit ourselves to one or a few roundtrips between the nodes holding different inverted lists. There is also a potential bottleneck in the random lookups performed by the *FA* and *TA* algorithms. In a high-bandwidth environment, this is a serious drawback of the algorithms since large index structures have to reside on disk. As a result, other pruning methods have been proposed for this case [1, 37] that avoid such accesses but instead need to scan a significant part of the inverted lists. In a P2P environment this is less of a concern, and a large set of random lookups could always be resolved by performing a local scan over the inverted list. Following is our proposed distributed implementation, called *DPP* (Distributed Pruning Protocol), for the case of two search terms and a ranking function from the first family (i.e., without the term $g(d)$).

Algorithm DPP:

(1) The node holding the shorter list, called node A, sends the first x postings of its inverted list to node B. (Let's assume for the moment that A somehow knows the best value of x.) Also, let r_{min} be the smallest (last) value $f(d, q_0)$ transmitted.

(2) Node B receives the postings from A, and performs a lookup into its own list in order to compute the total scores of the corresponding documents. Retain the k documents with the highest score among these. Let r_k be the smallest score among these documents.

(3) Node B now transmits to A all postings among its first x postings with $f(d, q_1) > r_k - r_{min}$, together with the total scores of the k documents from Step (2).

(4) Node A now performs lookups into its own list for the postings received from B, and determines the overall top k documents.

One remaining question is how to choose the value of x. This could be done by deriving appropriate formulae based on extensive testing. Alternatively, we could use sampling-based methods [9] to estimate the number of documents appearing in both prefixes. In either case, a wrong estimate could be corrected at the cost of an extra roundtrip.

	shortest 20%	shorter 20%	middle 20%	longer 20%	longest 20%
Shorter lists	10,401	63,853	222,948	666,717	3,371,176
Number of postings sent from A to B	2,057	4,083	2,904	4,417	3,745
Number of postings sent from B to A	1,486	4,084	2,891	4,413	3,745
Total bytes transferred	28,344	65,336	46,360	70,640	59,920
Total communication time (400 Kbps)	1,052	1,477	1,216	1,550	1,405
Total communication time (2 Mbps)	833	1,368	1,107	1,441	1,295
Shorter lists	10,401	63,853	222,948	666,717	3,371,176
Number of postings sent from A to B	889	922	610	407	304
Number of postings sent from B to A	792	923	612	407	307
Total bytes transferred	13,448	14,760	9,776	6,512	4,888
Total communication time (400 Kbps)	720	757	648	463	426
Total communication time (2 Mbps)	612	648	538	353	317

Table 3: Communication costs and estimated times of the protocol for top-10 queries, for cosine measure (top 6 lines) and cosine + log(PR) (bottom 6 lines).

4.4 Evaluation of *DPP*

To estimate the performance of *DPP* on our data and query set, we need an appropriate model of communication cost. In our system, we open a new TCP connection between the participating nodes for each query. To model the effect of the TCP congestion window on performance, which is significant in our scenario, we use a model for file transfer cost under TCP recently proposed in [50] with typical parameters for a broadband connection between the East and West coast of the US.[9] In particular, we assume a roundtrip signaling delay of $50ms$, and a bandwidth limit between 400 kbits and 2 mbits per second on the first and last leg. For both directions, we incur the cost due to the congestion window, and for the first message we have the additional cost of establishing the connection.

We assume that each posting is transmitted in 8 bytes on average, as follows: We hash the 80-bit document IDs down to z bits, where z is chosen such that the likelyhood of a collision between the transmitted prefix and the other list is less than, say, 0.1%. We then encode the hashes using standard gap compression techniques [51]; this results in at most 40 to 48 bits per hash; the rest of the 8 bytes is used for an approximation of the term value $f(d, q_0)$. We note that our protocol could be adapted to recognize when a collision occurs, in which case an additional roundtrip can be used to fix the problem. (Observe that the scheme is a bit like using a very precise compressed Bloom filter with one hash function.)

Table 3 shows the estimated cost of the algorithm, using the same data set as before. There are two assumptions in the measurements. First, we choose the length x of the prefix that should be sent from A to B by using the results of the experiments on the *TA* algorithm. This is optimistic since the parties do not have these results available; on the other hand, the results from the *CA* algorithm indicate that even a low estimate would often return the correct result (or we could choose an additional roundtrip to be sure). Second, we do not measure internal computation within nodes. Of course, this internal computation is also incurred by standard (non-P2P) search engines, and most of it is overlapped with communication anyway. We believe that neither of these assumptions changes the measurements fundamentally.

The results indicate that interactive response times are possible on terabyte size collections. Note that large engines such as Google in fact use data sets that are 20 to 30 times larger than ours. According to the theoretical bound of $\sqrt{N}$ this would result in an additional factor of about 5 on the amount of data transmitted. Use of more than two keywords would also increase communication. On the other hand, the above algorithm is really only a baseline as discussed in the following.

[9] The model in [50] is similar to other TCP models that have been proposed.

4.5 Optimizing Query Execution Plans

The above protocol is just a first step towards efficient query execution. There are two other ways to get further improvements: (1) use of Bloom filters as studied in [39], and (2) use of the hybrid partitioning described in Section 3 where large inverted lists are split among several nodes. We note that the second approach does not actually decrease the total cost of a query, but it can improve total latency by splitting communication and computation among several nodes. As it turns out, Bloom filters can be combined in several interesting ways with our protocol. The end result is that there are a large number of possible ways to execute a query on three or more search terms. We are currently studying in detail how to derive the best possible plans.

Recall that the design of a good query plan is up to the query client in our system. This would be done in two phases. The client first inquires basic statistics such as term frequencies, mean values for the normalization, and possibly samples [9] to estimate correlations between terms from the system. The system would return the statistics and also the IP addresses of the nodes holding the lists. This type of information can be very efficiently cached in the system as it is small compared to the rest of the data. In fact we really only need to keep statistics for inverted lists of significant length (e.g., more than a few thousand postings).

Given the statistics, the client knows which search term has the shortest inverted list, and which of the longer inverted lists are partitioned between several nodes. In the second phase, a query plan is designed as a directed labeled graph, where the nodes are nodes in the network identified by address, and the edges are labeled with the type of protocol to be used, e.g., send complete list if small, send a Bloom filter of the list, send a prefix of a list as done in the baseline *DPP* protocol, or send a Bloom filter of a prefix.

5 Related Work

For background on indexing and query execution in IR and search engines, we refer to [3, 5, 51], and for issues in parallel search engine architecture we refer to [7, 8, 28, 41]. Discussions and comparisons of local and global index partitioning schemes and the resulting query performance on parallel architectures are given, e.g., in [4, 12, 25, 31, 32, 48].

There has been a lot of recent interest in the pruning techniques of Fagin et. al [17, 19]; see also [18] for a survey and [13] for early related ideas. Most of the interest has been focused on multimedia and meta search scenarios, and we are not aware of previous applications in a peer-to-peer environment. On the other hand, there has also been significant work in the IR community, much of it preceding the above, on pruning techniques for vector space queries. Some early work is described in [10, 22, 37, 49, 52], and more closely related recent work is in [1, 2]. One difference between these two strains of work is that in the IR case, a random lookup of a posting is much more expensive than scanning it. Thus, the recent pruning techniques from IR typically restrict index access to scans, resulting in more limited savings. In our case, we are primarily concerned with bandwidth, making this less of an issue.

There has been significant interest in search in distributed and P2P systems over the last few years. We note, however, that the problem of full-text search on terabyte-size collections is different from that on smaller collections or on systems that only index titles and keywords for multimedia objects (e.g., mp3 files). Some recent work on text search in P2P systems with local index organization appears in [16, 27, 43, 47]. As explained, the global index organization is one of the aspects that distinguish our system from others. Another very different approach to distributed search is taken by systems such as JXTA [34], STARTS [21], and the Z39.50 standard [36], which are mainly concerned with issues of combining outputs from diverse search tools.

Global index organizations in a peer-to-peer environment have recently been discussed in [20, 29, 39]. The work by Reynolds and Vahdat [39] considers the benefits of using Bloom filters instead of sending an entire inverted list during query execution. Subsequent work in [29] estimates the potential benefit of using a combination of techniques, including Bloom filters, clustering, compression, caching, and adaptive set intersection, compared to the naive algorithm that transfers the entire list. The paper concludes that these techniques together save a significant constant factor and bring the approach close to feasibility for terabyte data sets. The authors also mention the possibility of using Fagin's pruning technique [18] for additional improvements, but

no details are provided. We note that combining Fagin's technique with those in [29] is possible, as indicated in Subsection 4.5, but the details are tricky and the returns will diminish as more techniques are applied.

6 Open Questions and Future Work

In this paper, we have given an overview of the ODISSEA system, and presented some very early results on query processing in the system. There are numerous open questions for future work. In particular, we are currently working on a framework for generating optimized query execution plans for multi-keyword queries based on a combination of pruning techniques, Bloom filters, and compression. We are also studying new algorithmic techniques for the index synchronization problem described in Section 3, and strategies for load balancing and rebuilding of lost replicas in an environment where nodes hold very large amounts of data but may be temporarily unavailable. Beyond these specific items, the general question remains whether the near future will see massive P2P-based systems for challenging applications such as web search and large-scale IR, beyond the current simple applications such as file sharing.

Finally, we are working on some improvements in the experimental evaluation in this paper. This concerns experimental results for queries with more than two keywords and phrase searches, and the effects on efficiency of using term distance in the ranking.

Acknowledgements

We thank Hojun Lee and Malathi Veeraraghavan for their help with the TCP performance model. This work was supported by NSF CAREER Award NSF CCR-0093400 and by the Othmer Institute for Interdisciplinary Studies at Polytechnic University.

References

[1] V. Anh, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in information retrieval*, pages 35–42, September 2001.

[2] V. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. 21st Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 290–297, 1998.

[3] A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, 1(1), June 2001.

[4] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. of the 9th String Processing and Information Retrieval Symposium (SPIRE)*, September 2002.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[6] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Trovatore: Towards a highly scalable distributed web crawler. In *10th Int. World Wide Web Conference*, 2001.

[7] E. Brewer. Lessons from giant scale services. *IEEE Internet Computing*, pages 46–55, August 2001.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World Wide Web Conference*, 1998.

[9] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.

[10] C. Buckley and A. Lewit. Optimization of inverted vector searches. In *Proc. 8th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 97–110, June 1985.

[11] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, and T. Winograd. Power browser: Efficient web browsing for PDAs. In *Proc. of the Human-Computer Interaction Conference*, 2000.

[12] B. Cahoon, K. McKinley, and Z. Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *IEEE Transactions on Information Systems*, 18(1):1–43, January 2000.

[13] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. *Data Engineering Bulletin*, 19(4):45–52, 1996.

[14] J. Cho and H. Garcia-Molina. Parallel crawlers. In *11th Int. World Wide Web Conference*, May 2002.

[15] Grub Distributed Internet Crawler. grub.org. http://www.grub.org/.

[16] F. Cuenca-Acuna and T. Nguyen. Text-based content search and retrieval in ad hoc p2p communities. In *Proc. of The Int. Workshop on Peer-to-Peer Computing*, May 2002.

[17] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of ACM Symp. on Principles of Database Systems*, 1996.

[18] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, June 2002.

[19] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of ACM Symp. on Principles of Database Systems*, 2001.

[20] O. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, 2002.

[21] L. Gravano, C. Chang, H. Garcia-Molina, and A. Paepcke. STARTS: Stanford Proposal for Internet Meta-Searching. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*. ACM Press, 1997.

[22] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, August 1990.

[23] T.H. Haveliwala. Topic-sensitive pagerank. In *Proc. of the 11th Int. World Wide Web Conference*, May 2002.

[24] G. Jeh and J. Widom. Scaling personalized web search. In *12th Int. World Wide Web Conference*, 2003.

[25] B. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[26] M. Karpovsky, L. Levitin, and A. Trachtenberg. Data verification and reconciliation with generalized error-control codes. In *39th Annual Allerton Conf. on Communication, Control, and Computing*, 2001.

[27] A. Kronfol. FASD: a fault-tolerant, adaptive, scalable, distributed search engine. June 2002. Unpublished manuscript.

[28] R. Lempel and S. Moran. Optimizing result prefetching in web search engines with segmented indices. In *Proc. of the 28th Int. Conf. on Very Large Data Bases*, August 2002.

[29] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems*, 2003.

[30] H. Lieberman, C. Fry, and L. Weitzman. Exploring the web with reconnaissance agents. *Communications of the ACM*, 44(8):69–75, August 2001.

[31] Z. Lu and K. McKinley. Partial collection replication versus caching for information retrieval systems. In *Proc. of the 23rd Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 248–255, 2000.

[32] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proc. of the 10th Int. World Wide Web Conference*, May 2000.

[33] W. Meng, C. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Computer Surveys*, 34(1), March 2002.

[34] Sun Microsystems. JXTA. http://www.jxta.org.

[35] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with almost optimal communication complexity. Technical Report TR2000-1813, Cornell University, 2000.

[36] National Information Standards Organization. Information Retrieval (Z39.50): Application Service Definition and Protocol Specification. Technical report, NISO, Bethesda, MD, 1995.

[37] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, May 1996.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conference*, 2001.

[39] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. February 2002. Unpublished manuscript.

[40] M. Richardson and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. In *Advances in Neural Information Processing Systems*, 2002.

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

[42] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Int. Conf. on Distributed Systems Platforms*, pages 329–350, November 2001.

[43] Y. Shen and D. L. Lee. An mdp-based peer-to-peer search server network. In *Proc. of the 3th International Conf. on Web Information Systems Engineering*, pages 269–278, December 2002.

[44] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proc. of the Int. Conf. on Data Engineering*, February 2002.

[45] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM Conference*, August 2001.

[46] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.

[47] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. In *Proc. of ACM HotNets-I*, October 2002.

[48] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *Proc. of the 2nd Int. Conf. on Parallel and Distributed Information Systems (PDIS)*, 1993.

[49] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, November 1995.

[50] M. Veeraraghavan, H. Lee, and R. Grobler. A low-load comparison of TCP/IP and end-to-end circuits for file transfers. In *Proc. of INET 2002*, June 2002.

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

[52] W. Wong and D. Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, September 1993.

[53] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, UC Berkeley, Computer Science Division, April 2000.

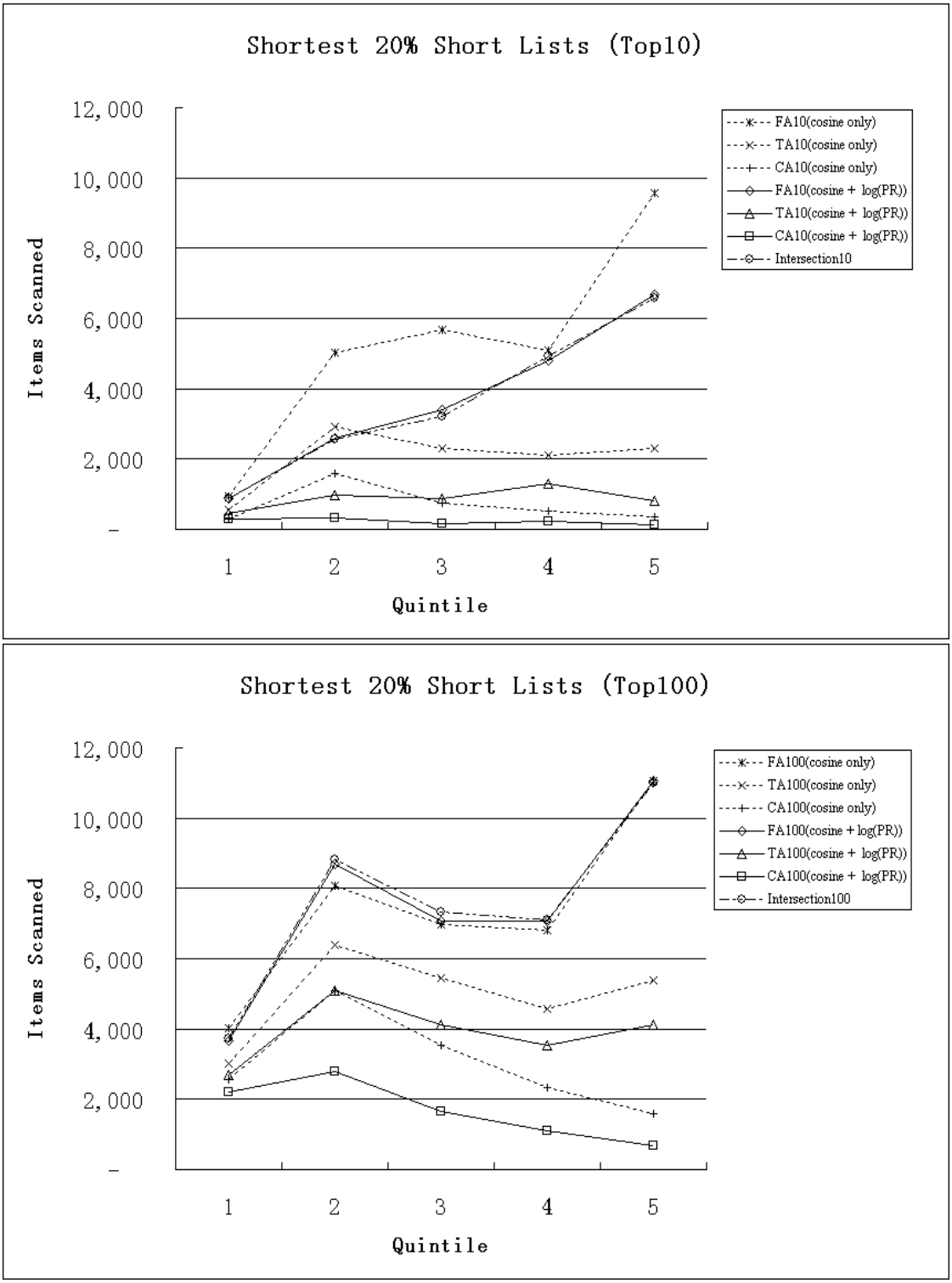

Figure 5: Performance of the various algorithms for top-10 and top-100 queries, for the 20% of the queries with the shortest shorter lists (first quintile). In each chart, the average number of tuples scanned is plotted versus the lengths of the other (longer lists), which are also grouped into quintiles.

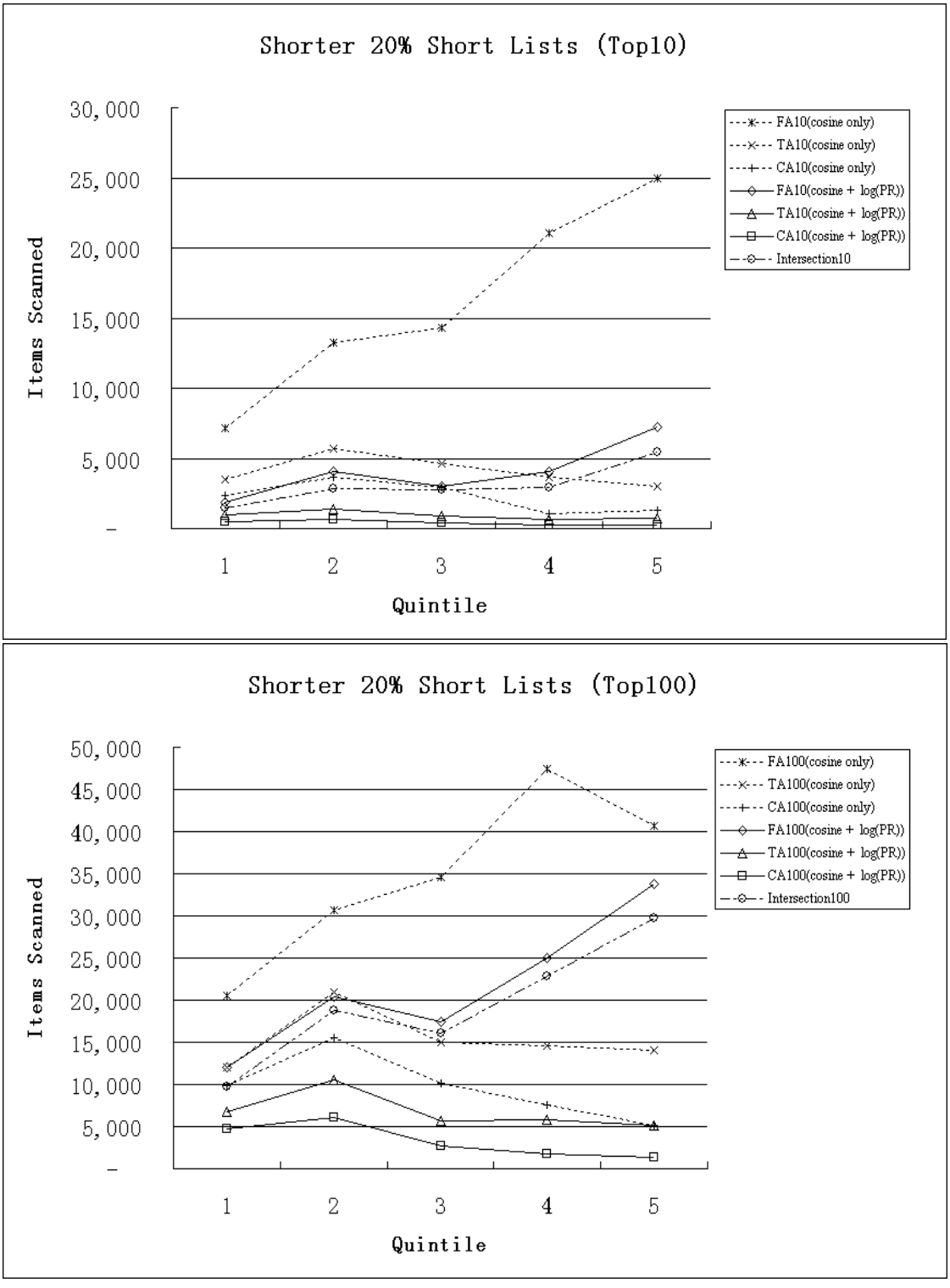

Figure 6: Performance of the various algorithms for top-10 and top-100 queries, for the second quintile of the shorter lists. In each chart, the average number of tuples scanned is plotted versus the lengths of the other (longer lists), which are also grouped into quintiles.

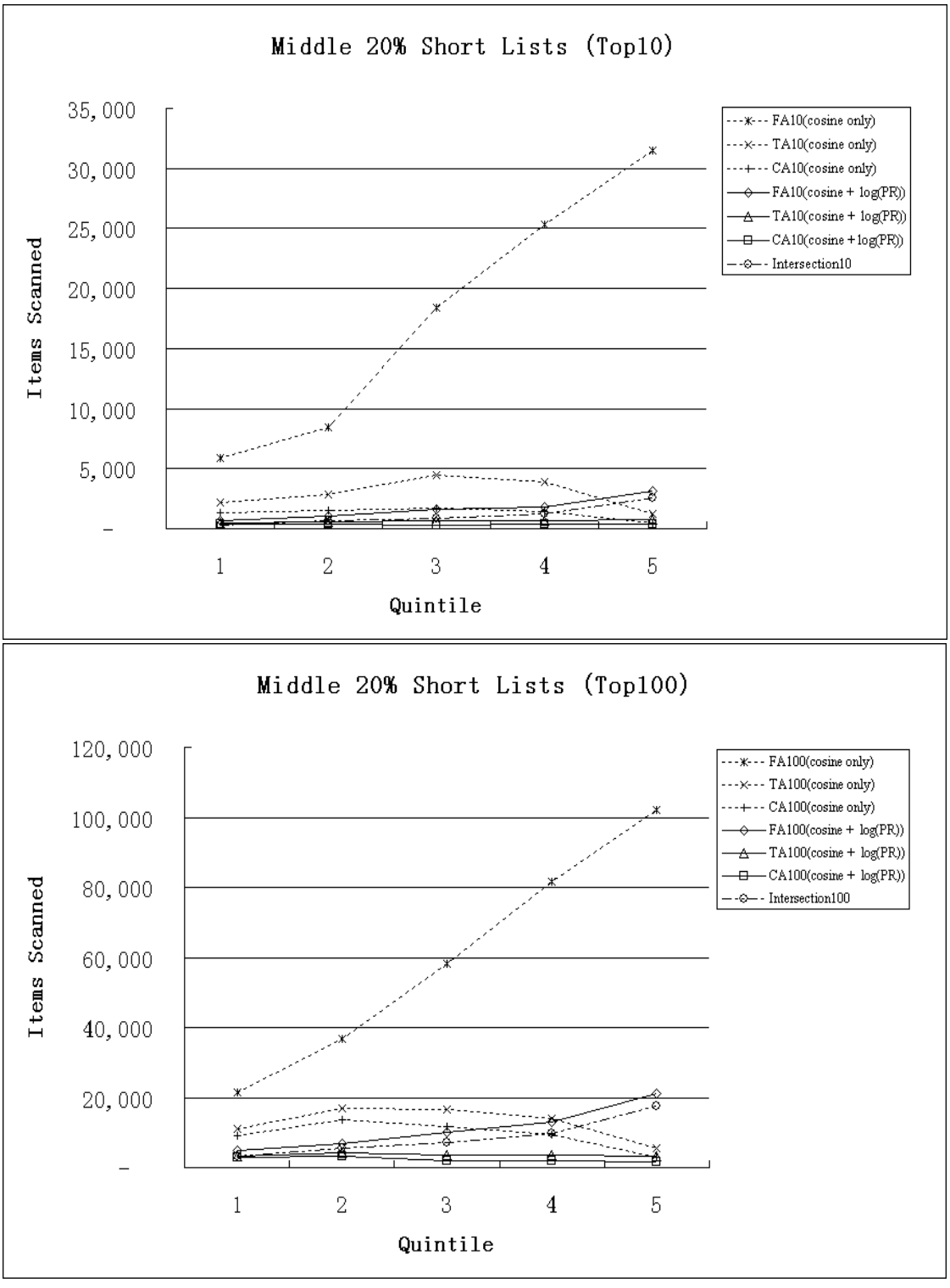

Figure 7: Performance of the various algorithms for top-10 and top-100 queries, for the third quintile of the shorter lists. In each chart, the average number of tuples scanned is plotted versus the lengths of the other (longer lists), which are also grouped into quintiles.

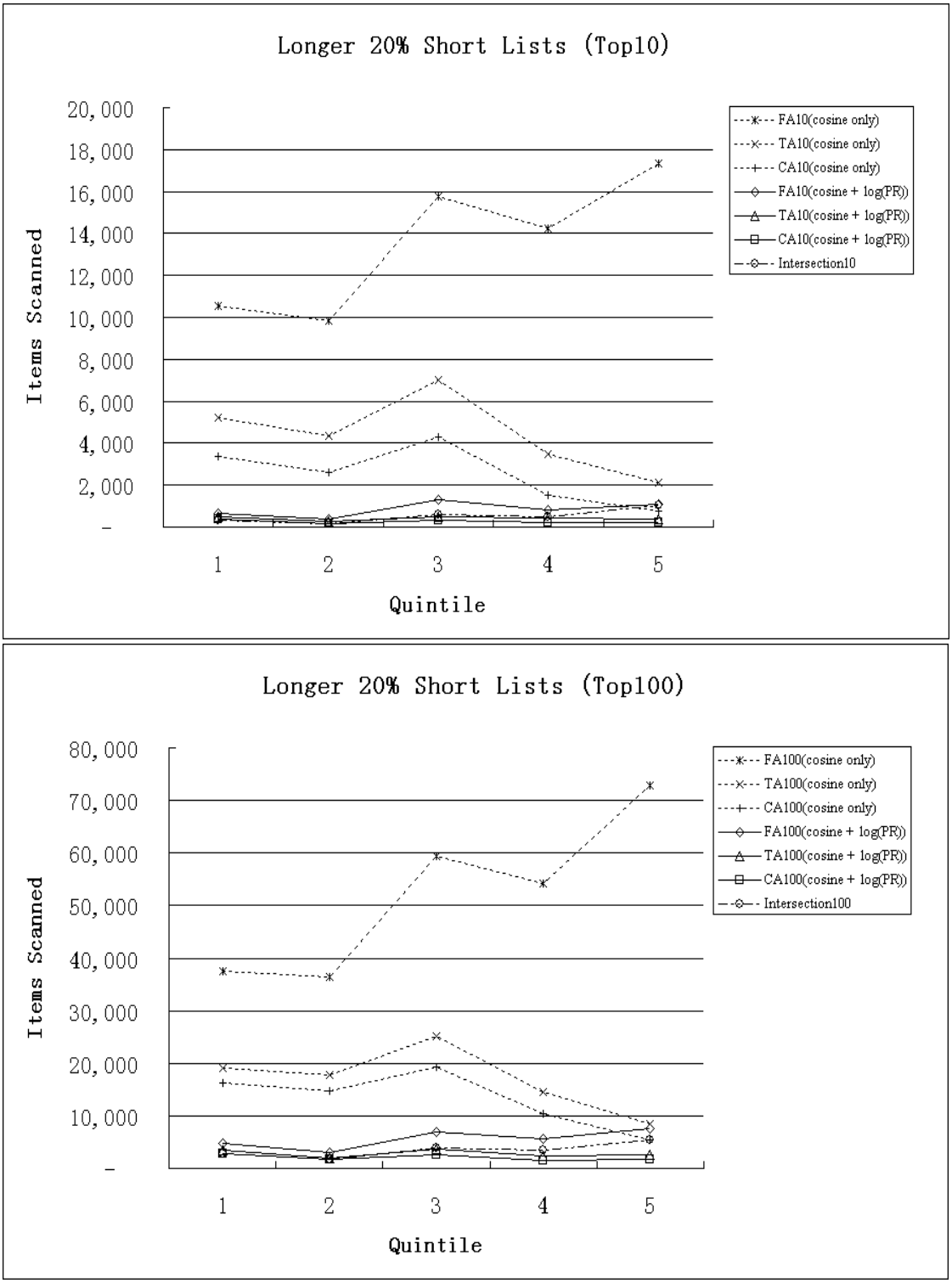

Figure 8: Performance of the various algorithms for top-10 and top-100 queries, for the fourth quintile of the shorter lists. In each chart, the average number of tuples scanned is plotted versus the lengths of the other (longer lists), which are also grouped into quintiles.

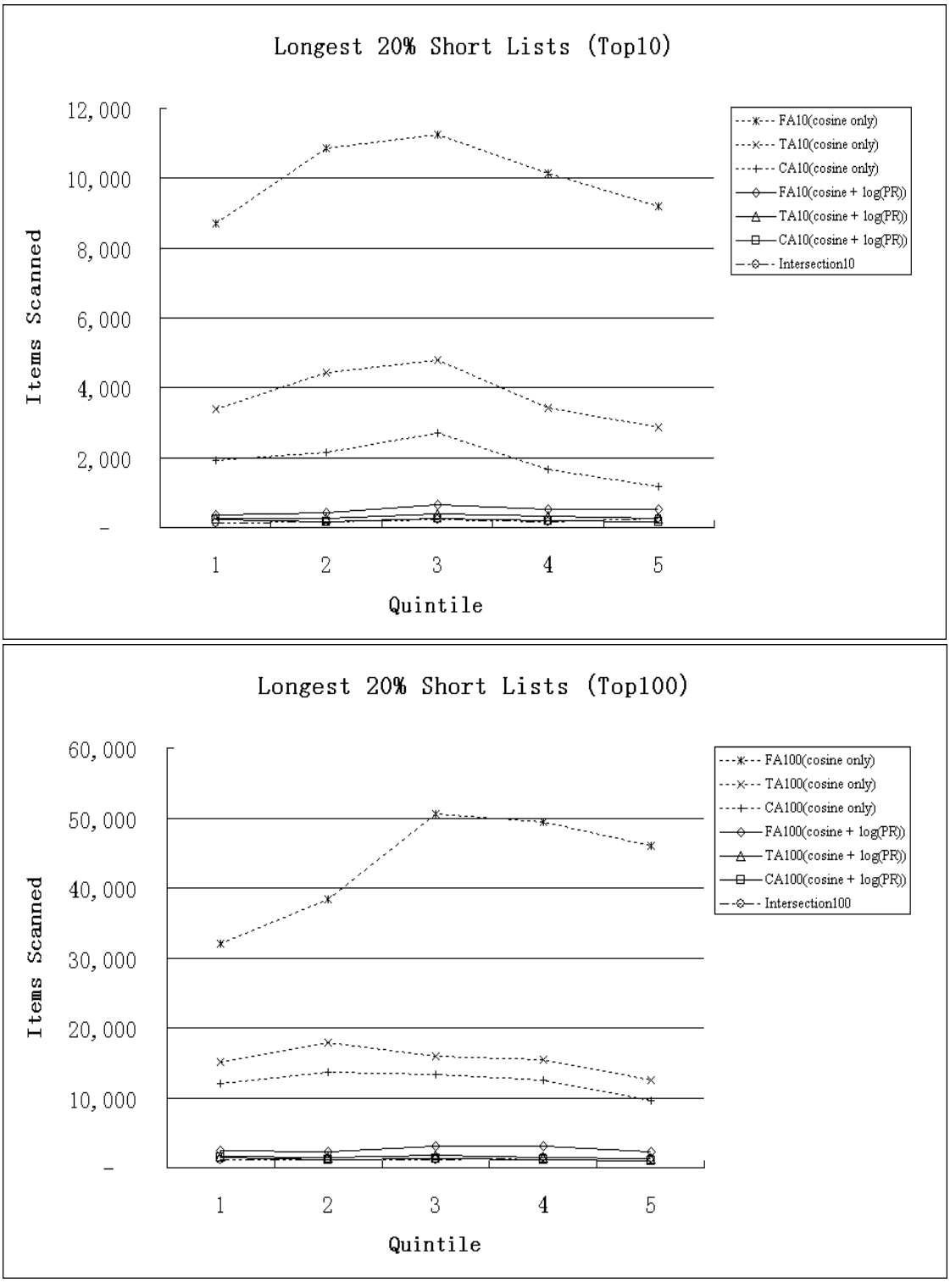

Figure 9: Performance of the various algorithms for top-10 and top-100 queries, for the 20% of the queries with the longest shorter lists (fifth quintile). In each chart, the average number of tuples scanned is plotted versus the lengths of the other (longer lists), which are also grouped into quintiles.