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 network.

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 preliminary simulation results on a real search engine log and a terabyte web collection that indicate good scalability for our approach.

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, AllTheWeb, 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 LANs or SANs [4]. 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 [31], Tapestry [39], Pastry [28], or CAN [25] 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* [4] 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, a web server pushing documents into the index, or a node in a file sharing system.
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 various ranking methods.

There are two main differences that distinguish ODISSEA from 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

*Contact author. Email: suel@poly.edu. Research partly supported by NSF CAREER Award NSF CCR-0093400 and by the Othmer Institute for Interdisciplinary Studies at Polytechnic University.

International Workshop on the Web and Databases (WebDB).
June 12–13, 2003, San Diego, California.

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 (see [15, 19, 26] for exceptions) to full-text search in P2P systems assume a *local inverted index*, where each node maintains an index for all local documents (or the documents of a few surrounding nodes), and queries are 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 possibly significant cost. We discuss this decision later as it has consequences for the overall design.

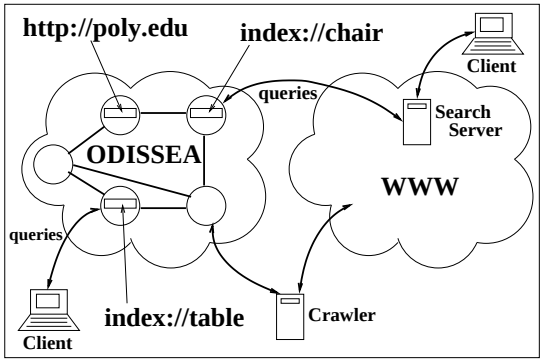

Figure 1: ODISSEA as a web search infrastructure, with a web crawler as update client, and two query clients (one client-based and one as a web-based search service). Also shown are indexes for the words "chair**" and "**table**", and a node holding the document** `http://poly.edu`**.**

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 [28]. Each object is identified by a hash of its *name* (i.e., a URL 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. 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 is actually more complicated, to enable replication and load balancing.)

1.2 Target Applications

We have four main application scenarios that motivate our research:

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

(2) **Search in large intranet environments:** Large organizations may use distributed search to share machine resources within more controlled and maybe less bandwidth-limited environments.

(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. This scenario may not be feasible in the near term but we believe 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 for query processing, resulting in increased efficiency for certain queries.

Paper outline: In the next section we justify our main design decisions and assumptions. Technical details and preliminary experimental results on query processing are provided in Section 3. Section 4 discusses related work, and Section 5 mentions some open problems. A more detailed version of this paper appears in [32], and up to date information on the ODISSEA 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. Early examples of these types of client-based tools are browsing assistants such as the Alexa and Google Toolbars, Zapper, Leticia and PowerScout [20], the Stanford Power Browser [8], or tools built with the Google API. Specialized search engines (Google News, citeSeer) or meta 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.

Global vs. local index: The other important decision is the use of a global index instead of the more commonly used local index organization. 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 other information such as whether the word is in the title or in bold face. Each postings list is best visualized as an array 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 holds a complete global postings list for a subset of the words, as determined, e.g., by a hash function. Thus, every node has 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. There have been a number of performance comparisons between local and global index organizations and several hybrid organizations on parallel architectures [3, 9, 34], but these do not directly apply to widely distributed environments.

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[1] or are in the vicinity [11, 18, 29]. However, we do not believe that this approach will scale 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, though this seems quite challenging to do [19]. Moreover, the statistics needed to intelligently route queries are quite large for large collections and many nodes as the number of distinct words grows with collection size; the existing literature has only evaluated collections up to a few gigabytes.

In a global index organization, however, large amounts of data need to be transmitted between nodes, since large collections result in lists of megabytes or more for all except fairly rare words. This has led some people to reject global indexes as unrealistic for environments with 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 through smart algorithmic techniques. One technique was recently applied in this context in [19, 26], 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 study in Subsection 3.1 how recent results on top-k queries in the database literature [13] can be applied to asymptotically reduce communication requirements. We believe that these techniques, combined with other query optimization techniques, allow interactive response times even on massive data sets.

Crawling punt: We assume in the web search application that crawling is performed by *crawling clients* that fetch and insert documents. The main reason is that from our own experiences with large-scale crawling [30] 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 web sites or subnetworks or to modify its behavior, and failure to do so can result in problems with local administrators or upstream providers.[2] Moreover, smart crawling strategies beyond BFS are hard to implement in a P2P environment without a centralized scheduler.

Thus, we would expect that a handful of powerful crawling clients would provide most documents, and we plan to use our Polybot crawler [30] to initially populate the system with data. It might be more feasible 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 remote 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 disk, and during this time the node would probably consume more resources than it adds to the system. Thus, such applications are maybe best restricted to the more stable end of the P2P spectrum, where most nodes remain in the system for longer times.

Our system design 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 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 updates missed while unavailable? Other problems 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.

[1]This is also known as the database selection problem [21].

[2]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.

3. 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 in WAN environments are on the order of a few hundred Kb/s, this would result in response times of many seconds or even minutes. We now describe how to adapt recent techniques by Fagin and others [12, 13, 14] to our scenario, and give measurements of the expected savings 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.

3.1 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 [5] 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,

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

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 includes the well-known class of *cosine measures*; see, e.g., [37]. 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 terms in a document would lead to an additional function $h(d, q_0, \ldots, q_{m-1})$ that depends on all terms; this would impact the efficiency of our methods.

Queries to search engines have on average less than three terms, and 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.[3] An 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 list. However, recent work in the database community [13] shows how to evaluate top-k queries without scanning the entire intersection.

Fagin's Algorithm (FA): We now describe the first algorithm, which was originally proposed in [12] 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

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

an item that is ranked in the top is likely to be ranked very high in at least one contributing subcategory.

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 $(d, f(d, q_i))$, $i \in \{0, 1\}$, where d is an integer identifying the document and $f(d, q_i)$ is real-valued. Assume each inverted list is sorted by the second attribute, so that documents with largest $f(d, q_i)$ 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.

It is not difficult to see that this indeed returns the top-k results overall. It is shown in [12] 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 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 could potentially exploit special distributions such as Zipfians.

Threshold Algorithm (TA): The following refinement was proposed by several authors; see [13] for a discussion. We again simultaneously scan both lists, so that in each step we read an item $(d, f(d, q_0))$ from the first 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 list, we immediately perform a lookup into the other list to compute its complete score. As soon as we have found k items with score larger than the current t, we return these as results. It can be shown that *TA* is correct and always terminates at least as early as *FA*, though the asymptotic bounds are 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 this should increase efficiency, as it introduces significant correlation between the orderings 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 [27] 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 [27] 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 a problem since we are interested in minimizing bandwidth consumption rather than CPU cost.

3.2 Experimental Results on Real Data

We run some initial experiments to determine the potential savings of these schemes. Note that these experiments are in a centralized setting; we consider distributed implementations in the next subsection. There have been previous evaluations of the *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 the experiments, we use queries selected from a log of over 1 million queries posted to the Excite search engine on December 20, 1999. Our document collection consists of about 120 million web pages crawled by the Polybot crawler [30] in October 2002, for a total of about 1.8 TB of data. The numbers reported here are limited to a few hundred queries with two terms. Also, stop words were removed, as done by many engines, and we removed queries with less than 100 results in the intersection. 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.

	Top 1	Top 10	Top 100
Lists	1,056,746	1,056,746	1,056,746
Intersect	654	1,536	8,954
FA(cosine only)	7,860	17,087	47,024
TA(cosine only)	2,445	6,353	17,962
CA(cosine only)	932	2,978	13,585
FA(cosine + pagerank)	6,137	11,046	37,991
TA(cosine + pagerank)	1,652	4,651	16,533
CA(cosine + pagerank)	529	1,785	11,025

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

Table 1 shows the average number of postings that have to be scanned from each list under the various algorithms. 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 is a reasonable lower bound[4] 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 $\Theta(\sqrt{n})$ where we might have to scan most of the lists. 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. 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. Including the Pagerank score usually results in improved performance. 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 massive data sets.

3.3 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, 24] 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 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 $g(d)$).

(1) The node holding the shorter list, called node A, sends the first x postings of its inverted list to node B. (Assume for the mo-

[4] If we discount correlations between query terms.

	shortest 30%	middle 30%	longest 30%
Lists	23,620	305,557	3,092,772
Postings A to B	5,105	7,405	4,264
Postings B to A	5,572	7,360	4,183
Tot. bytes sent	85,336	118,120	67,576
Time 400kbps (ms)	2,456	3,151	2,077
Time 2mbps (ms)	977	1,151	882

Table 2: Communication costs and times for top-10 queries.

ment 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 lookups into its own list to compute the total scores of the corresponding documents. Retain the k documents with the highest score. Let r_k be the smallest score among these.

(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 performs lookups into its own list for the postings received from B, and determines the overall top k.

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 [6] 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.

3.4 Evaluation of *DPP*

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 [36] with typical parameters for a broadband connection between the East and West coast of the US.[5] 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 each posting is transmitted in 8 bytes, 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 [37]. This results in at most 40 to 48 bits per hash; the remaining bits are used for an approximation of the term value $f(d, q_0)$. The protocol could be adapted to recognize when a collision occurs, in which case an additional roundtrip is 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 2 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.

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 $\sqrt{kN}$ 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.

[5]The model in [36] is similar to others that have been proposed.

3.5 Optimizing Query Execution Plans

The above protocol is a first step towards efficient query execution. There are two ways to get further improvements: (1) use of Bloom filters as studied in [26], and (2) use of a hybrid partitioning where large inverted lists are split among several nodes [32]. We note that the second approach does not actually decrease the total cost of a query, but it can improve 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.

The design of a good query plan is up to the query client in our system, and is done in two phases. The client first inquires basic statistics such as term frequencies, mean values for the normalization, and possibly samples [6] to estimate term correlations from the system. The system returns the statistics and 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 term has the shortest inverted list, and which of the lists are partitioned between several nodes. Next, 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 operation to be performed, e.g., send complete list if small, send a Bloom filter of the list, send a prefix as done in the baseline *DPP* protocol, or send a Bloom filter of a prefix.

4. RELATED WORK

There has been a lot of recent interest in the pruning techniques of Fagin et. al [12, 14]; see also [13] for a survey and [10] 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 [7, 17, 24, 35, 38], 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. Thus, recent pruning techniques from IR typically restrict access to scans, resulting in more limited savings. We are mainly 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 [11, 18, 29, 33]. 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 [22], STARTS [16], and the Z39.50 standard [23], 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 [15, 19, 26]. The work by Reynolds and Vahdat [26] considers the benefits of using Bloom filters instead of sending an entire inverted list during query execution. Subsequent work in [19] 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 [13] for additional improvements, but no details are provided. Combining Fagin's technique with those in [19] is possible, as indicated in Subsection 3.5, but the details are tricky and the returns diminish as more techniques are applied.

5. OPEN QUESTIONS AND FUTURE WORK

In this paper, we have given an overview of the ODISSEA system, and presented some early results on query processing in the system. There are numerous open questions for future work. 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. Once this is complete, we plan to perform a more thorough experimental evaluation for queries with multiple keywords and phrase searches, and for ranking functions that use term distance within documents.

We are also studying techniques for synchronizing outdated indexes and for load balancing and rebuilding of lost replicas in an environment where nodes hold 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 simple applications such as file sharing.

Acknowledgements: We thank Hojun Lee and Malathi Veeraraghavan for their help with the TCP performance model.

6. REFERENCES

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

[2] V. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. 21st Annual SIGIR Conf.*, 1998.

[3] 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.

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

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th WWW Conference*, 1998.

[6] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997.

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] O. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, MIT, 2002.

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

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

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

[19] 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.

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

[21] W. Meng, C. Yu, and K. Liu. Building efficient and effective metasearch engines. *ACM Computer Surveys*, March 2002.

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

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

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

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

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

[27] 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.

[28] 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.

[29] 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, 2002.

[30] 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.

[31] 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.

[32] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasunderam. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. TR-CIS-2003-01, Polytechnic University, 2003.

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

[34] 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.

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

[36] 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.

[37] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann, second edition, 1999.

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

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