Optimized Inverted List Assignment in Distributed Search Engine Architectures*

Jiangong Zhang Torsten Suel

CIS Department
Polytechnic University
Brooklyn, NY 11201
zjg@cis.poly.edu suel@poly.edu

Abstract

We study efficient query processing in distributed web search engines with global index organization. The main performance bottleneck in this case is due to the large amount of index data that is exchanged between nodes during the processing of a query, and previous work has proposed several techniques for significantly reducing this cost. We describe an approach that provides substantial additional improvement over previous techniques. In particular, we analyze search engine query traces in order to optimize the assignment of index data to the nodes in the system, such that terms frequently occurring together in queries are also often collocated on the same node. Our experiments show that in return for a modest factor increase in storage space, we can achieve a reduction in communication cost of an order of magnitude over the previous best techniques.

1 Introduction

A number of researchers have recently studied the problem of implementing scalable search engines based on highly distributed or peer-to-peer (P2P) architectures. However, current large-scale web search engines are based on a fairly centralized architecture; while these engines consist of machines at a number of locations around the world, each location is typically a fairly large data center that forms its own self-contained index and query processing engine. It is still an open question to what degree this architecture could be replaced by a highly distributed or peer-to-peer design, where each participating machine is in a separate location and each query involves cooperation between a number of such machines over the internet. Such a design would be useful not only for the purpose of trying to compete with the current more centralized engines on web search tasks, but also for probably more realistic challenges such as the efficient indexing and search of textual data residing in P2P systems.

One of the main bottlenecks in such a highly distributed design is the efficiency of search engine query processing, i.e., the problem of determining the highest-scoring, say, 10 or 100 documents for a given set of query terms under some appropriate scoring function. This problem has been extensively studied in the Information Retrieval and Web Search communities. Query processing consumes a significant amount of resources even in the current centralized engines, but additional challenges arise in a highly distributed environment with bandwidth and latency constraints.

In this paper, we focus on the problem of efficiently processing such queries on textual collections up to multiple terabytes in size. On such collections, each of the terms in a typical user query has hundreds of thousands or even hundreds of millions of occurrences in the collection; this results in very long inverted list index structures that slow down query processing. As a result, this scenario is quite different from search in smaller textual collections, or in multimedia collections where each large multimedia object (video, audio) is accompanied by a much smaller amount of textual meta data (such as titles, descriptions, or tags). In the latter scenarios, the main focus is usually on how to locate the fairly small index structures that can be used to retrieve possible matches for the query (often in very dynamic environments), while in our case the main challenge is to compute the top-k results on many millions of possible matches (a hard problem even in relatively stable wide-area environments) once the index data has been located.

We now state a few assumptions. We consider the case of a highly distributed system, where each machine is likely to be located in a different local network, and where communi-

*Work supported by NSF ITR Award CNS-0325777.

cation occurs over fairly slow wide-area connections (e.g., the public internet). We do not directly address challenges that are particular to P2P systems, where nodes can join and leave the network at any time; our approach is largely orthogonal to these issues, and can be implemented on top of standard P2P substrates such as [33, 41, 30] and many others. Several different ways to organize a text index (inverted index) structure in a distributed environment have been proposed, in particular *local index organization*, *global index organization*, and several hybrids. We focus on the case of a global index organization, and consider a very general class of ranking functions that includes common variants of the widely used Cosine and Okapi measures (among many others). The main bottleneck in this case is the amount of communication (bandwidth) required during query evaluation, and we will show gains by almost an order of magnitude over previous results.

In the next section, we provide some technical background. Section 3 describes related work, and Section 4 summarizes our contribution. In Section 5, we formally define the problem of assigning inverted lists to machines and describe possible algorithms, while Section 6 provides an experimental evaluation.

2 Technical Background

Text Index Structures: Search engines use a text index structure called *inverted index*, which allows efficient retrieval of documents containing a particular set of words (or *terms*). We assume that each document (e.g., web page in the case of a web search engine) is identified by a unique *document ID* (docID) assigned, e.g., through hashing or enumeration. An inverted index consists of many *inverted lists*, where each inverted list I_w contains the IDs of all documents in the collection that contain the word w. More precisely, each list I_w is a sequence of *postings*, where each posting consists of the docID of a document D containing w, its frequency (how often w occurs in D), and sometimes also the position and context of each occurrence. The postings in each inverted list are often sorted by docID. Inverted indexes are usually stored in highly compressed form on disk, and many compression techniques have been studied [39, 31].

Term-Based Ranking: We assume that each query consists of a set of words (query terms). The most basic form of ranking involves simply comparing the words contained in the document and in the query. More precisely, documents are modeled as unordered bags of words, and a ranking function assigns a score to each document with respect to the current query, based on the frequency of each query word in the document and in the overall collection, the length of the document, and maybe the context of the occurrence (e.g., higher score if in title or bold face) or the proximity between terms (e.g., higher score if two search terms appear close to each other in the query or document). Formally, given a query $q = \{t_0, t_1, \ldots t_{d-1}\}$ with d terms, a *ranking function* F assigns to each document D a score $F(D, q)$. The system then determines the docIDs of the k documents with the highest scores; this is usually done by fetching and traversing the inverted lists for all query terms.

There are many different classes of ranking functions in the literature, including the well-known Cosine Measure and Okapi families of functions. Our approach here can be applied to many different ranking functions, including functions that include global scores such as Pagerank or measures based on user feedback (clicks), or the use of contexts or term proximities in the queries and documents, with two limiting assumptions. First, we assume that a query can be efficiently evaluated on the collection by traversing the inverted lists of all the query terms, and maybe also accessing additional smaller data structures containing term or document statistics. Second, we assume *intersection* semantics, i.e., that only documents containing all query terms are returned. However, our techniques would also be useful in cases where almost all terms have to occur in a document [11]. Both of these assumptions appear to be largely true for current large-scale web search engines as well as many other IR search tools.

Index Partitioning: In a parallel or distributed search engine, the inverted index structure is partitioned over a number of machines. There are two basic distributed inverted index organizations, called *local* and *global* index organization, shown in Figure 2.1.

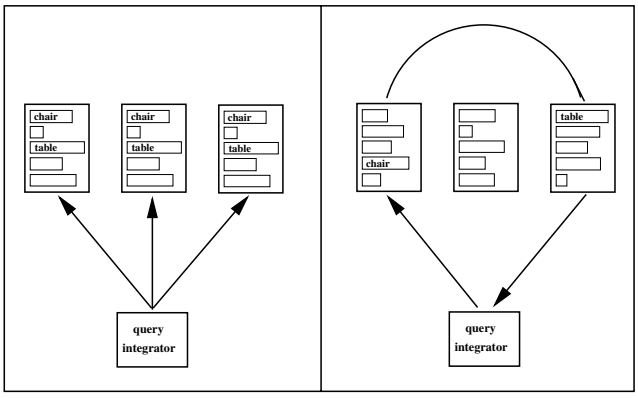

Figure 2.1. Query processing with a local (left) and global (right) index organization.

In a local index organization, each node is assigned a subset of the document collection and creates its own inverted index on these documents only. Thus, every node has its own (shorter) inverted list for words such as "chair" or "table", and a query "chair, table" is first broadcast by a frontend called *query integrator* to all nodes; then the top-

k results from each node are merged into a global top-k list at the frontend. In a global index organization, each node holds the complete inverted lists for a subset of the words as determined, e.g., by hashing. Thus, every node has a smaller number of (longer) lists, and under the most basic query execution strategy a query "chair, table" is first routed to the node holding the shorter list, say "chair", which then sends its complete list to the node holding the list for "table". Hybrids between the two organizations are also known, and performance comparisons of local, global, and hybrid organizations on centralized parallel systems appear, e.g., in [2, 13, 38].

In a nutshell, the main challenge in a local index is that many nodes need to be contacted for each query, resulting in a large number of small messages. On other hand, a global index may require an amount of data proportional to the length of the shortest inverted list in the query to be transmitted, resulting in a few very large messages. Thus, at least under the naive execution strategies outlined above, a local index is unlikely to scale beyond a few hundred nodes, and a global index is unlikely to scale beyond a few million documents.

To show the magnitude of this challenge, we provide some rough cost estimates for a collection of 4 billion web pages.[1] The average page contains about 750 words, including about 250 distinct words. Using state-of-the-art index compression techniques we might get an inverted index size of about 0.4 KB per page if only docIDs and frequencies are stored in each posting, or 1.3 KB if positions are stored as well. Thus, the total index size is on the order of several TB. In a centralized engine, such an index might be distributed over, say, a hundred machines. In a P2P environment, where each machine contributes only part of its CPU and disk resources and the machine configuration is not optimized towards search, a thousand or more machines may be needed. Thus, in a local index organization it would be infeasible to require results from all nodes. Moreover, the inverted lists accessed by a single search query would have an average size of about 1.2 GB (docID and frequency) to 3.9 GB (including positions). In fact, just the shortest list in each query would still have an average size of about 40 MB (docID and frequency) to 100 MB (including positions), making transmission of the list during query processing in a global index organization infeasible.

Query Processing Optimizations: A number of researchers have proposed techniques for reducing the high costs of distributed query processing, and thus the picture is not quite as bleak as suggested by the above numbers. Several approaches for local index organizations return results by contacting only a carefully chosen subset of the nodes, see, e.g., [14, 21, 32, 36, 24, 5]. For global index organizations, a series of techniques in [28, 18, 22, 34, 40] have resulted in very substantial reductions in the total communication cost through the use of Bloom Filters and top-k pruning techniques, though the remaining costs are still high. Techniques for a hybrid index organization were recently proposed in [35], while [42] contains a comparative study of some of the known algorithms for different index organizations.

In this paper we assume a global index organization, and we propose a new technique that achieves additional substantial improvements in the communication cost of query processing. Our approach is orthogonal to the techniques in [28, 18, 22, 34, 40], and can be easily combined with these for best results. Note that we are not trying to advocate any particular index organization, and there are many scenarios where a local organization may be preferable. We believe that both organizations merit further study, and that in some cases hybrid organizations that combine ideas from both directions may turn out to be the best choice.

3 Discussion of Related Work

We now give a more detailed discussion of related work. First, for background on indexing and query execution in search engines, we refer to [1, 3, 9, 39], and for parallel search engine architecture we refer to [8, 4, 29]. Discussions of local and global index partitioning schemes and the resulting query performance on parallel architectures are given, e.g., in [2, 13, 19, 25, 38].

As mentioned, there has been a significant amount of recent work on query processing in P2P search engines with local index organizations, such as [14, 21, 32, 37, 36, 24, 5] and others. Much of this work is concerned with the problem of finding good results without broadcasting each query to all peers in the network, by selecting a subset of nodes that are particularly promising for the given query (or in some cases, nodes in the local neighborhood of the overlay structure). This is often done using approaches similar to the *database selection* problem studied in the context of distributed databases and meta search engines; see [26] for a survey of some techniques.

One potential problem with this approach is that its performance relies on the collection being nicely clustered and the queries being well-behaved with respect to this clustering. Given the very diverse types of information needs evident in real search engine query traces, it could be argued that there may be no way to cluster collections of billions of pages over a few thousand nodes such that good results can be obtained by contacting only a small subset of the nodes. We are not aware of any experimental results for such data sizes. However, the proposed approaches appear to perform well on more limited data sizes, and there are sometimes significant advantages to local index organiza-

[1] This is not as large as the leading engines, but probably large enough to give decent results for most queries.

tions that create and maintain index structures at the node where the content resides.

Several authors have studied P2P query processing under a global index organization [28, 22, 18, 34, 40]. In particular, [28, 22] investigate the use of *Bloom Filters* [7, 12, 27] for efficient intersection of inverted lists located on different machines, influenced by earlier work on distributed joins in databases. A Bloom Filter is a data structure that supports membership test queries on a set of elements (i.e., "Is this element in the represented set?"). A Bloom Filter uses significantly less space than a dictionary or hash table, but at the cost of a small false positive rate that can be traded off against space. Thus, we can compute the intersection between two lists by first sending a small Bloom Filter representing the docIDs in the shorter list to the other machine, which can then eliminate from its own list most of the postings that do not match. As shown in [22], Bloom Filters can reduce communication costs by an order of magnitude over the naive approach. Of course, this assumes that only documents containing all query terms are returned as results.

A second approach, proposed in [34], uses a top-k pruning technique from [15, 16] during query processing. This approach can also give significant reductions in communication costs, but is limited to certain classes of ranking functions. In particular, it is necessary that the score $F(D, q)$ of a document D with respect to query $q = \{t_0, \ldots, t_{d-1}\}$ is a monotone combination (e.g., the sum) of scores $f(D, t_i)$ for some function $f()$. This is true for the Cosine Measure and many popular cases of Okapi, but not for certain other functions. Under this approach, only a subset of the shortest inverted list has to be communicated over the network. Hybrid algorithms that combine Bloom Filters and top-k pruning are studied in [40], which shows further cost reductions in this case. One observation from [40] is that top-k pruning performs best for short queries but degrades for queries with 4 or more terms, while Bloom Filters are better for longer queries. Thus, the two techniques complement each other well.

Our approach in this paper is complementary to both techniques. The basic idea is very simple. Consider the naive algorithm for query processing in a global index organization, i.e., ignoring the optimizations based on Bloom Filters and top-k pruning. The communication cost of this algorithm is roughly proportional to the length of the shortest inverted list involved in the query, since the data size typically decreases significantly after the first intersection. However, if it so happens that two of the inverted lists are located on the same machine, then we could locally intersect these two lists and the result might be much smaller than the shortest list. Thus, the idea is to make this happen more often, through a careful assignment of inverted lists to machines. This idea was inspired by an observation mentioned to us by Andrei Broder [10] that, given enough machines and storage space, one could replicate inverted lists such that for any pair of keywords there is a machine containing both corresponding lists.

A naive implementation of this idea could result in a very large amount of extra machines and storage space. Our main insight here is that by analyzing search engine query traces, we are able to allocate inverted lists with only a small amount of space overhead such that for many (though not all) queries, there is at least one pair of lists that is on the same machine. This idea is also related to two other previous works. In [6], Bhattacharjee et al. propose to temporarily cache intermediate results of intersections at the receiving node in order to reduce communication costs. Also, [23] studies the caching of previously computed intersections of pairs of lists in a centralized search engine architecture.

4 Contributions of this Paper

We study the problem of search engine query processing in highly distributed or P2P-based architectures with global index organization, and describe a new technique that results in significant bandwidth savings over previous approaches. In particular, our contributions are:

(1) We motivate and study the problem of assigning/replicating inverted lists over a set of nodes in a way that minimizes communication costs during query processing, for a given distribution of queries.

(2) We describe and implement heuristic algorithms for this problem that result in a good assignment of lists.

(3) We evaluate the performance of the algorithms on large data sets consisting of real web pages and associated query traces. Our results show substantial decreases in communication cost with moderate space overhead.

5 The Inverted List Assignment Problem

In this section, we define and discuss the problem of assigning inverted lists to machines, and then describe some simple algorithms for the problem.

5.1 Problem Definition

Recall from our informal discussion that our goal is to distribute and replicate inverted lists over nodes so that for many queries, there exists some node that contains two or more of the inverted lists for the query terms. For simplicity, assume that we are using the naive query processing algorithm for global index organization, where the smallest list is sent over the network in the first step, and that this first step dominates the total cost of a query. We also assume that the size of a list is given by the number of postings. We now add a preprocessing phase to this algorithm where

any lists that are located on the same node are first intersected locally (with zero communication cost), and then the naive algorithm is applied to the resulting reduced query with fewer and shorter inverted lists.

For example, given a query "armadillo, alligator, dog" such that there exists a node containing both lists $I_{alligator}$ and I_{dog}, we would first intersect these two lists into a temporary list $I_{alligator \cap dog}$. Then the cost of the naive algorithm on the reduced query would be given by the size of the smallest remaining list, which might be either $I_{alligator \cap dog}$ or $I_{armadillo}$. As a special case, the communication cost is zero if all lists in the query are on the same node; this includes all single-keyword queries. Based on this, we can now define a simplified version of the problem:

Definition 5.1 Inverted List Assignment Problem: *Assume that we are given a set of n terms $W = \{w_0, \ldots, w_{n-1}\}$ and associated inverted lists (modeled as sets) $I_{w_0}, \ldots, I_{w_{n-1}}$ of total size $S = \sum_{i=0}^{n} |I_{w_i}|$. We are also given m nodes $N_0, \ldots, N_{m-1}$ with storage capacity C each such that $m \cdot C \geq S$, and a set of queries Q where each query $q \in Q$ is a subset of W. Then our goal is to assign to each node N_i a set of terms $T_i \subset W$ such that*

(1) each term is assigned to one or more nodes,

(2) for each node N_i, we have $\sum_{w \in T_i} |I_w| \leq C$, and

(3) the assignment minimizes the total cost of the set of queries Q, $\sum_{q \in Q} c(q)$, where the cost $c(q)$ of a query is:

(a) $c(q) = 0$ if $q \subseteq T_i$ for some i, or

(b) $c(q) = \min_{\{q' \subseteq q \mid q' \neq \emptyset \wedge \exists i: q' \subseteq T_i\}} \left(\left| \bigcap_{w \in q'} I_w \right| \right)$ otherwise.

We now discuss some of the assumptions in this definition. First, we note that to achieve good performance in practice, it is important to combine our approach with the optimizations based on Bloom Filters and top-k pruning in [28, 22, 34, 40]. As a result, the cost of the reduced query becomes significantly more difficult to estimate and is not really linear in the size of the shortest list anymore. For example, given a query "armadillo, alligator, dog" such that $I_{alligator}$ and I_{dog} are together on one node and $I_{armadillo}$ and I_{dog} on another, we could further decrease communication costs by first performing both local intersections even if we already know which one will result in the smallest intersection. The reason is that when we use Bloom Filters and top-k pruning, the cost depends (in a non-trivial way) on the lengths of both the sending and the receiving list. This also leads to possible trade-offs between local computation and communication costs, and raises the question of which copies of the inverted lists to select in case there are several copies. As we discovered in our experiments, however, our simple assumption that the cost is proportional to the number of postings in the shortest list appears to perform quite well for the purpose of assigning lists to nodes.

Second, in reality we do not know the set of queries ahead of time. Instead, we would have as input a probability distribution over all possible queries, which itself would be derived from a sufficiently large query trace. This is also the approach we will take in our experiments: We divide a query trace into two parts, a *training set* that is used to derive a probability distribution (possibly using appropriate smoothing and density estimation techniques) that is used for the list assignment, and a smaller *testing set* that is used to evaluate the quality of the resulting assignment.

5.2 Algorithmic Approach

Having defined the basic problem, we now consider possible solutions. First, we observe that, not surprisingly, it is NP-Complete to find an optimal solution to the list assignment problem. In fact, this is true even if all lists before intersection have the same length, all machines can hold only two lists, and no query has more than 3 terms. The proof is by a fairly straightforward reduction from the Vertex Cover problem [17]. Given a graph $G = (V, E)$, we create one list I_v for each $v \in V$, plus one extra list I_α, and create for each $(u, v) \in E$ a query $\{u, v, \alpha\}$. If we assume that all lists are of the same (sufficienctly large) size s, and that $|I_u \cap I_v| \geq (1 - \epsilon) \cdot s$ and $|I_u \cap I_\alpha| \leq \epsilon \cdot s$ for all $u, v \in V$ and some small ϵ, then there exists an assigment with cost less than $(1 - \epsilon) \cdot s$ onto x machines that can each hold two lists if and only if there exists a Vertex Cover of size x.

Thus, a precise solution is probably out of our reach. We decided to attack the problem through two different approaches: through greedy algorithms that assign inverted lists to nodes until the available space is exhausted, and as a partitioning problem on a suitably constructed graph. We will use the latter approach to assign the first copy of each list to the nodes, while additional replicas are then assigned using the greedy approach.

Greedy Approach: List-Driven and Node-Driven Algorithms We first describe the greedy algorithms. A first very naive idea was to replicate each inverted list a fixed number k of times, as determined by the available space in the system. Typical values for k are 2 or 3, i.e, the total amount of space needed is 2 or 3 times the amount needed for a single copy of the index without replication. In this first algorithm, called *List-Driven*, we round-robin over the lists, and for each list we pick the node that should receive the list. More precisely, we determine which node would provide the largest overall reduction in communication cost on the given query trace if we add a copy of the list to that node. As we will see later, this approach has several shortcomings, though.

A second approach, called *node-driven*, selects in each round the node with the most available space. This node now adds to its index a copy of the inverted list that gives the most benefit, i.e., that gives the greatest overall reduction in communication cost. A variant called *node-driven ratio* considers the ratio between the overall reduction in cost and the size of the list. In a naive implementation, this means that to choose a list, we have to iterate over all terms in the query trace, and compute for each query containing this term the sizes of various list intersections and the resulting savings. However, there are ways to make this more efficient by using a suitable priority queue data structure.

Graph Approach: An alternative approach is to try to reduce the list assignment problem to that of determining a good (overlapping) partitioning of a suitably constructed weighted graph. In particular, we create a vertex for every term w, with weight $|I_w|$. We also create an undirected weighted edge (w, w') between all $w, w' \in W$, where the weight $c(w, w')$ is proportional to the benefit of having w and w' on the same node. Then the problem is how to partition the vertices into possibly overlapping neighborhoods, where each neighborhood corresponds to a node, such that (i) the total vertex weight on each node is less than its capacity, and (ii) we minimize the total weight of those edges that are "completely cut", i.e., where there is no node containing both endpoints.

There are two issues with this approach. First, it is not obvious what the right edge weights are. Not all benefits are pairwise in nature, e.g., when a single node contains three inverted lists used in the same query. In a first naive approach, we initialize all weights to 0, and then iterate over all edges and assign to each edge as weight the difference between the size of the shorter of the two corresponding lists and their intersection. In another approach, we iterate over the queries instead. Whenever a pair (u, v) has the smallest intersection among all pairs of terms in the current query, we increase the weight $c(u, v)$ by the benefit, defined as the difference between the size of the shortest list in the entire query and this intersection (if positive). Or alternatively, we can assign to *each* edge that is contained in the query an appropriate weight. Note that none of the resulting three graph partitioning problems is strictly equivalent to the original instance of the Inverted List Assignment Problem, but merely a heuristic approximation.

Second, for the case where we only have a single copy of each list, the problem is equivalent to a standard graph partitioning problem studied, e.g., in parallel computation, where we partition a workload into equal size pieces in a way that minimizes interprocessor communication; thus, we can use available graph partitioning software for this task. However, we were unable to find tools for the case of an overlapping partitioning, i.e., where each node may be replicated several times. We note that there has been some interest in such overlapping graph neighborhoods in the algorithms and parallel computing communities, though in somewhat different contexts.

Combining Graph and Greedy: One problem with the greedy algorithm is how to get a good initial assignment of lists from which to start the greedy process. Ideally, we would start with each node having one or several good cluster heads or seeds to which other lists can then be added. This suggests the following combination of the graph-based and greedy approaches: First, partition a single copy of the entire index over the nodes based on the graph-partitioning approach; this can be done using available and highly tuned graph partitioning software. Then, use the greedy approach to add additional copies of some of the lists to the nodes until all available space is used.

6 Experimental Evaluation

In this section, we provide a preliminary experimental evaluation of the various algorithms on large data sets.

6.1 Experimental Setup

We first describe the data set and experimental setup. The results presented in this section are based on the *GOV2* data set used in the TREC Terabyte Track. It consists of 25.2 million pages from 17186 hosts in the `gov` top-level domain, and a set of $100,000$ queries made available as part of the efficiency competition. In the following table, we show the distribution of the number of keywords in the queries; as we see, most queries have between two and five keywords.

# terms k	1	2	3	4	5	6	> 6
% queries	2.10	19.63	32.25	25.26	12.67	4.87	3.22

Table 6.1. Percentage of queries with k terms for the GOV2 query set.

We precomputed the intersection sizes of all pairs of terms that occur in a common query, by issuing appropriate queries to a full-text index of the collection. These sizes are then used to determine the benefit of a particular list placement during the computation of the list assignment. This means that only pairwise benefits are considered by our assignment algorithms; i.e., we do not consider the additional benefits that occur when 3 or more lists are on the same node (though these benefits are considered in the evaluation). We note that the extra benefits are small relative to the pairwise case, while computing all intersections of sets of three terms would have been quite expensive.

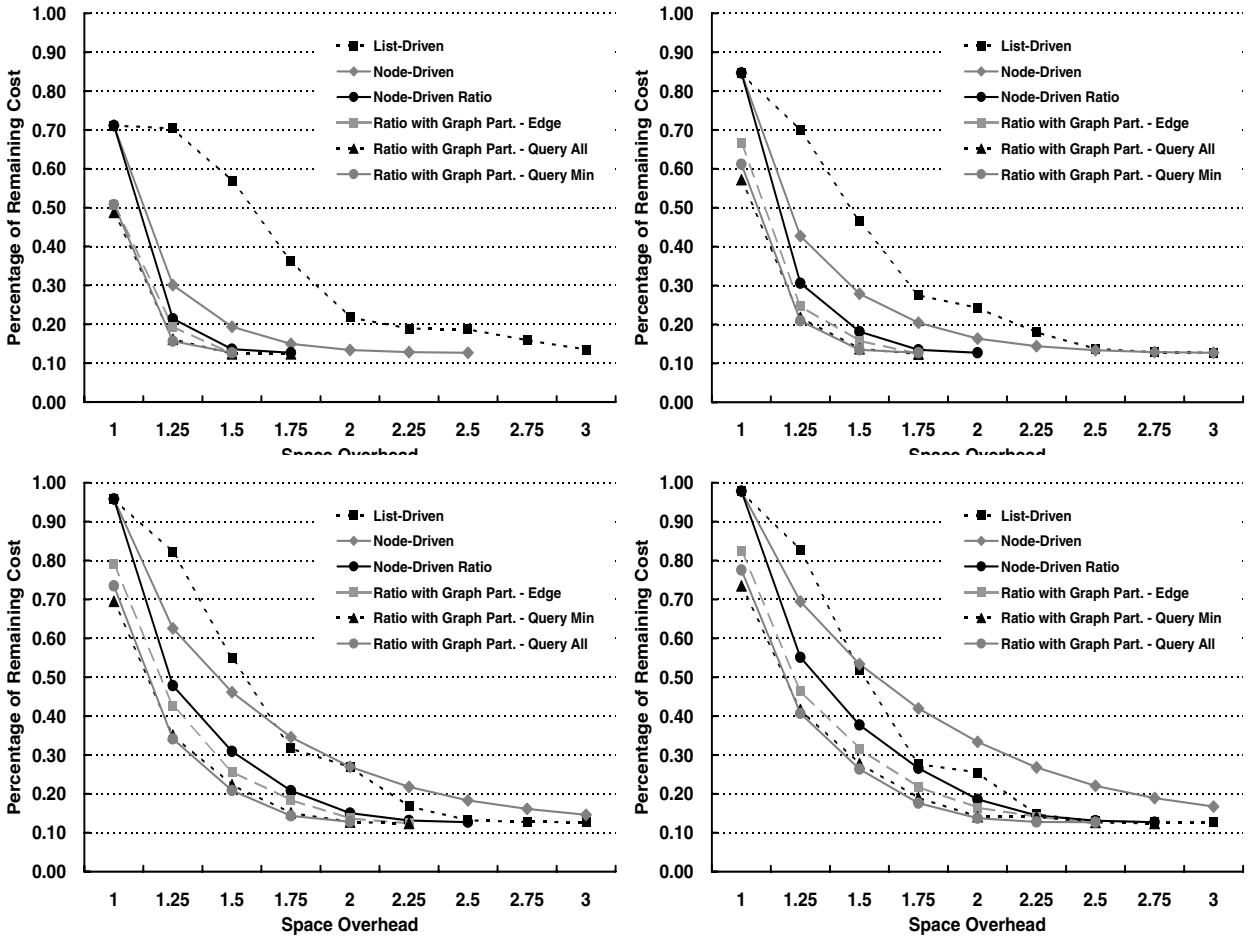

Figure 6.1. Relative communication costs of our algorithms on 8 (top left), 16 (top right), 64 (bottom left), and 128 (bottom right) nodes.

For the initial graph partitioning computation, we used Version 4.0.1 of the METIS package[2] based on a multi-level partitioning approach described in [20]. We also experimented with random initial assignments for comparison.

We first evaluate the algorithms in terms of the objective function used during the list assignment, i.e., the number of elements in the shortest list or shortest intersection of collocated lists. In the second part of our experiments, we then show the effect of combining our technique with the optimized algorithms based on Bloom Filters and top-k pruning described in [40]. Throughout this section we focus on the communication costs of the various techniques and omit local computation; note however that all local steps can be very efficiently processed using a fairly standard search engine query processor.

[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/download

6.2 Performance Results

In Figure 6.1 we show the relative communication costs of six methods, compared to the baseline method that does not exploit collocation of lists and simply sends the shortest list.

The methods consist of three methods without initial graph partitioning, *list-driven*, *node-driven*, *node-driven with ratio*, and three versions of *node-driven with ratio and initial graph partitioning*, one where we iterate over edges and two where we iterate over the queries to assign edge weights, as described before. For the first three methods, we assume that the first copy of each list is assigned at random. We show four different graphs, for 8 (top left), 16 (top right), 64 (bottom left), and 128 (bottom right) nodes. We note that in these graphs, we are showing the value of the objective function, i.e., the cost of the resulting assignment in terms of the number of elements in the shortest list or shortest intersection.

	16				128			
	Min Lists	Max Lists	Min Blocks	Max Blocks	Min Lists	Max Lists	Min Blocks	Max Blocks
List-Driven	3,664	10,066	21,100	58,672	421	10,904	2,832	62,453
Node-Driven	2,371	3,336	23,224	23,547	279	1,202	2,898	3,362
Node-Driven Ratio	3,593	5,040	20,224	20,292	192	1,534	3,521	3,630
Ratio with Graph Part. - Edge	942	7,435	21,432	21,485	131	1,562	3,765	3,973
Ratio with Graph Part. - Query All	1,730	4,826	21,432	21,485	73	1,140	2,889	3,094
Ratio with Graph Part. - Query Min	2,529	4,359	19,453	19,465	189	948	3,352	3,420

Table 6.2. Resulting imbalance in the number of inverted lists per and in the index size per node, for all six algorithms and for 16 and 128 nodes. Shown are for each setting the maximum and minimum number of lists and 64 KB blocks of compressed index data held by any of the nodes.

# of terms k	1	2	3	4	5	6	> 6
% of queries (before)	2.10	19.63	32.25	25.26	12.67	4.87	3.22
% of queries (2 Nodes)	69.94	30.06	0	0	0	0	0
% of queries (8 Nodes)	38.88	43.96	14.42	2.34	0.41	0	0
% of queries (32 Nodes)	20.10	46.50	23.55	7.41	1.32	0.71	0.41
% of queries (128 Nodes)	11.37	41.62	29.85	12.18	3.45	0.81	0.71

Table 6.3. Percentage of queries with k terms before and after computing local intersections using *node-driven with ratio and initial graph partitioning - edge***, for different numbers of nodes. The space overhead is limited to 2.0 in this table.**

We can make several observations from the graphs. Looking at the point for space overhead 1.0 (i.e., only one copy of each list, as in the baseline method) we can see that graph partitioning of course gives a better assignment of lists than the random assignment. (The first three methods are identical at this data point as the greedy phase has not yet been reached.) Thus, some benefit can be obtained with no space overhead at all. Overall, the method that assigns benefit only to the edge with the smallest intersection performs best. However, the other methods eventually achieve the same improvements as the best method, though with a higher space overhead. We note that there is an upper limit to the benefit for all the methods, partly due to limitations of the query trace: as the space overhead approaches 1.75 to 2.0 for the node-driven methods, and about 2.75 to 3.0 for the other methods, there is no additional benefit as all interesting pairs in the query trace already exist somewhere on the same machine. We would expect moderate additional benefits for larger query traces (as well as after some additional fine-tuning of our policies).

The list-driven method performs worst. In fact, in the results in Figure 6.1, we give the list-driven approach an advantage by not enforcing a bound on storage size at each individual machine. This results in significant imbalances in index size at the nodes, as shown in Table 6.1. Basically, slight imbalances are magnified by this method, as more and more lists are attracted to the node with the most data. If we enforce a bound on storage size at each node, the effect will be that the nodes are filled one after the other, and the benefits are even less. The other methods, by design, result in a fairly balanced allocation of list data in terms of their actual compressed size on disk (which we used to select the node with the most remaining space in each step).

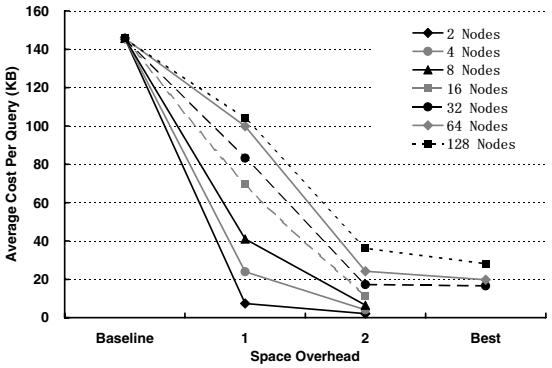

Figure 6.2. Total communication cost per query in KB, for 2 to 128 nodes, and including the Bloom Filter and top-k pruning optimizations from [40]. The baseline in this figure is the set of techniques in [40] without the list assignment approach in this paper.

In the following, we focus on the method *node-driven with ratio and initial graph partitioning - edge*, where we iterate over the edges to compute edge weights for the parti-

tioning. Note that by first performing local intersections on each node, we a really transforming each query into another query with fewer terms and shorter inverted lists. Thus, the average number of terms per query decreases, and in particular queries with many terms are very likely to have at least one pair that can be locally intersected at some node. In Table 6.3, we see the the resulting changes in the keyword distribution of queries, with most queries now having 3 or less keywords.

So far, we have only evaluated the methods with respect to the size of the shortest list or intersection (number of elements). We now look at the actual total communication cost in KB per query when we combine our technique with the Bloom Filter and top-k pruning techniques from previous work. The results are shown in Figure 6.2, which plots cost versus space overhead for configurations with 2 to 128 nodes. The most interesting cases are probably between 16 to 128 nodes: we note that we get a benefit of more than a factor of 10 (12 KB versus 145 KB) for 16 nodes, and a factor of about 6 for 128 nodes. Again, we point out that we expect additional benefits with larger training query traces, better fine-tuning of techniques, and also with the use of data sets other than TREC GOV2 (which we believe to be more challenging than typical search engine traces, e.g., in terms of the number of terms per query).

Finally, we attempt to extrapolate our results to a collection of 2.5 billion pages, using the same method as in [40]. We observe again significant benefits by about a factor of 10 over the baseline method. We note that the raw baseline number cannot be compared directly to the numbers in [40], due to use of a different data set.

6.3 Discussion and Open Questions

As our preliminary results have shown, significant savings in communication cost can be obtained through careful assignment of inverted lists to nodes. While the results are promising, there are still a number of shortcomings in our evaluation that we are working to resolve. We plan to run experiments on other data sets, including the commonly used Excite query traces on a set of crawled data. Extrapolating from the results in [40], which used this data set, we would expect slightly improved numbers (say by about a factor of 2), and also separate improvements due to the larger size of the query trace (giving us a larger training data set).

In addition, there are various details in our current implementation that could be further tuned, such as appropriate smoothing of the query probability distribution which will become more important for longer traces. Finally, while our methods balance the space consumption between nodes while minimizing the overall communication cost, we have not yet considered how to also balance local computation and communication costs between the different nodes in the system.

As we saw, performance deteriorates slightly as the number of nodes increases, and we expect that for very large collections, a hybrid organization might perform best. For example, a 4 billion page collection might be partitioned into 20 sets of 200 million pages, each handled by a separate cluster of say 64 nodes. Within each cluster, we might use our approach to distribute lists among nodes. With a space overhead of 2.0 and postings containing position information, each node would receive about 8 GB of inverted index data in this scenario. Note that this hybrid does not actually save communication bandwidth over our approach, but it allows parallel processing and transmission in different clusters, thus making sure query latency is within a few seconds or less.

As an open question for future research, it would be interesting to study the basic list assignment problem, and the graph-based variation of it, in more detail and to derive more formal algorithms with performance guarantees (or show that none exist). Improvements in our results might also be possible by a more precise modeling of the actual query processing cost during the list assignment phase. Finally, there are many other research challenges that need to be overcome to enable large P2P-based search engines, including robustness of such large-data-set applications against changes in the network structure, and efficient index updates in such systems.

Acknowledgements

We would like to thank Andrei Broder for sharing with us his idea of using replication of inverted lists to reduce communication costs, and for allowing us to use this idea as the starting point for our work here. We also thank Xiaohui Long for his work on the indexing and query execution software used in our experiments.

References

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

[2] 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.

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

[4] L. Barroso, J. Dean, and U. Hoelzle. Web search for a planet: The google cluster architecture. *Micro IEEE*, 23(2):22–28, 2003.

[5] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Minerva: Collaborative p2p search. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (demo paper)*, pages 1263–1266, 2005.

[6] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems*, 2003.

# of nodes	baseline	2	4	8	16	32	64	128
Cost in bytes	7,074,615	22,754	63,483	120,504	249,551	400,217	509,887	829,153

Table 6.4. Estimated average cost in bytes per query when scaling the data set to 2.5 billion pages, for varying numbers of nodes.

[7] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

[9] 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.

[10] A. Broder. Informal communication, 2002.

[11] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th Int. Conf. on Information and Knowledge Management*, pages 426–434, 2003.

[12] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Proc. of the 40th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, 2002.

[13] 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.

[14] 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.

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

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

[17] M. Garey and D. Johnson. *Computer and Intractability: A Guide to the Theory of NPCompleteness*. WH Freeman and Company, 1979.

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

[19] 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.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, August 1999.

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

[22] 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.

[23] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of the 14th Int. World Wide Web Conference*, May 2005.

[24] B. Mayank, R. Bayardo, S. Rajagopalan, and E. Shekita. Make it fresh, make it quick — searching a network of personal webservers. In *Proc. of the 12th Int. World-Wide Web Conference*, 2003.

[25] 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.

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

[27] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. on Networking*, 10(5):604–612, 2002.

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

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

[30] 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.

[31] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 222–229, August 2002.

[32] 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.

[33] 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.

[34] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. ODISSEA: A peer-to-peer architecture for scalable web search and information retrieval. In *International Workshop on the Web and Databases (WebDB)*, June 2003.

[35] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proc. of the 1st Symp. on Networked Systems Design and Implementation*, pages 211–224, March 2004.

[36] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. of ACM SIGCOMM*, pages 175–186, August 2003.

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

[38] 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.

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

[40] J. Zhang and T. Suel. Efficient query evaluation on large textual collections in a peer-to-peer environment. In *Fifth IEEE Int. Conf. on Peer-to-Peer Computing*, pages 225–233, 2005.

[41] 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.

[42] M. Zhong, J. Moore, K. Shen, and A. Murphy. An evaluation and comparison of current peer-to-peer full-text keyword search techniques. In *Proc. of the Eight Int. Workshop on the Web and Databases*, pages 61–66, 2005.