Efficient Query Evaluation on Large Textual Collections in a Peer-to-Peer Environment

Jiangong Zhang ***Torsten Suel***

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

Abstract

We study the problem of evaluating ranked (top-k) queries on textual collections ranging from multiple gigabytes to terabytes in size. We focus on the case of a global index organization in a highly distributed environment, and consider a class of ranking functions that includes common variants of the Cosine and Okapi measures. The main bottleneck in such a scenario is the amount of communication required during query evaluation. We propose several efficient query evaluation schemes and evaluate their performance. Our results on real search engine query traces and over 120 million web pages show that after careful optimization such queries can be evaluated at a reasonable cost, while challenges remain for even larger collections and more general classes of ranking functions.

1. Introduction

Peer-to-peer (P2P) architectures have received a tremendous amount of attention over the last five years, and several classes of applications such as file sharing, storage, and media streaming have been implemented and in some cases adopted by large user communities. However, many other classes of internet applications are still predominantly provided by more centralized systems. In particular, one type of application that appears to be very challenging to implement in a P2P architecture is the efficient searching of very large textual collections, as required by web search engines and large-scale digital library systems. This issue has recently begun to receive some attention in the research community, but there are still many obstacles that need to be overcome.

In particular, one of the main obstacles is the efficiency of top-k query processing in P2P systems, i.e., the problem of retrieving 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 search engines, but additional challenges arise in a wide-area distributed environment with bandwidth and latency constraints.

In this paper, we study the problem of efficiently executing such top-k queries on textual collections ranging from multiple gigabytes to terabytes in size. On such large 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 only a fairly small amount of textual meta data. In a nutshell, in the latter scenario the main problem is to locate the 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 out of many millions of possible matches once the index data has been located (a hard problem even in relatively stable wide-area environments).

There are several different ways to organize a text index (inverted index) structure in a distributed environment, in particular *local index organization* and *global index organization* and several proposed hybrids. We focus on the case of a global index organization in a highly distributed environment, and consider a class of ranking functions that includes common variants of the widely used Cosine and Okapi measures. The main bottleneck in such a scenario is the amount of communication (bandwidth) required during query evaluation. Our results on real search engine query traces and over 120 million crawled web pages show that after careful optimization such queries can be evaluated at a reasonable cost. On the other hand, challenges remain for even larger collections and for more general classes of ranking functions.

2. Technical Background

Text Index Structures: Most Text Information Retrieval systems 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 search engine) is identified by a unique *document ID* (docID), e.g., assigned 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, sorted by document ID or some other measure, plus possibly some additional information about each occurrence. Inverted indexes are usually stored in highly compressed form on disk, and many compression techniques have been studied [28, 19].

Term-Based Ranking: We assume that each query consists of a set of words (query terms). The most common way to perform ranking is based on 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 page 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). 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 returns the docIDs of the k documents with the highest score. One popular class of functions is the *Cosine Measure* [28], for example

$$F(D,q) = \sum_{i=0}^{d-1} \frac{w(q,t_i) \cdot w(D,t_i)}{\sqrt{|D|}},$$

where $w(q,t) = \ln(1 + n/f_t)$, $w(D,t) = 1 + \ln f_{D,t}$, and $f_{D,t}$ and f_t are the frequency of term t in document D and in the entire collection, respectively. We note that a query under such a ranking function can be efficiently evaluated on the collection by traversing the inverted lists of all the query terms and computing the scores of any encountered documents in passing, using the information embedded in the index plus some global tables with information on document sizes and global term frequencies.

A crucial observation for our work here is that $F(D, q)$ is the sum (or other simple combination) of scores f_{D,t_i}, in the above case $f_{D,t_i} = w(q,t_i) \cdot w(D,t_i)/\sqrt{|D|}$. Many other common ranking functions, such as Okapi, also share this property. For our purposes, it is often convenient to think of each inverted list I_w as a sequence of items $(D, f_{D,w})$ where D is the docID of a document containing w, sorted by either D or $f_{D,w}$ (i.e., we may assume that the $f_{D,w}$ are precomputed though in a real implementation this is usually done on the fly).

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. The two index organizations are illustrated in Figure 2.1. 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 [1, 5, 27].

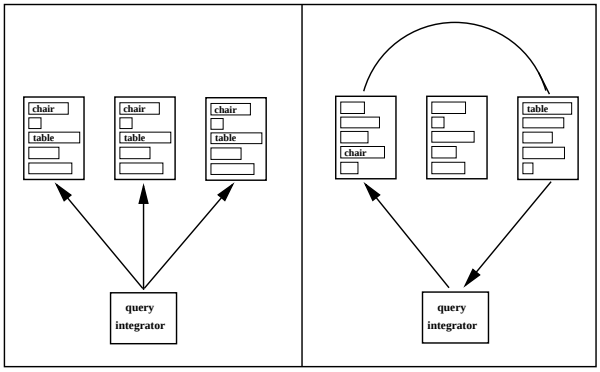

Figure 2.1. Query processing with a local (left) and global (right) index.

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 unlikely to scale beyond a few million documents. In the local index case, there are some optimizations that may allow us to answer many queries by only contacting a carefully chosen subset of "promising" nodes [13, 25, 11, 20, 6].

In this paper we assume a global index organization. As we describe later, there are several techniques that allow query execution in this case without transmitting complete inverted lists, and these are the subject of this paper. 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 are of interest and merit further study, and in some cases hybrid organizations may turn out to be the overall best choice.

Possible Architecture of a P2P Search Engine: An example of a P2P search engine or IR system with a global index organization is shown in Figure 2.2, which is based

on our previous work on the ODISSEA project [23]. Inverted lists are mapped to a set of peers, by employing an underlying DHT substrate such as [22, 29, 18]. Queries are issued by clients outside the P2P system that may act as query integrators or delegate this role to a better connected node within the P2P system. Clients might also act as intermediaries (search servers) that forward queries issued by web clients and relay back the results. In the case of a web search engine, crawling might be done by the peers, or *crawl clients* might fetch documents and then insert them into the P2P system.

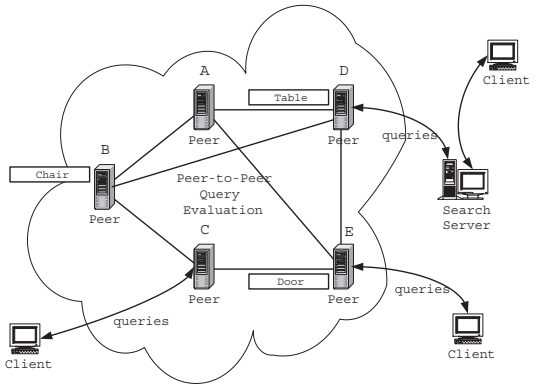

Figure 2.2. A possible P2P search architecture.

Given a query, a query integrator first retrieves the locations of the relevant inverted lists plus maybe certain meta data (e.g., collection statistics) through DHT lookups, and then uses this data to design a good *query execution plan* that specifies how peers communicate with each other during query execution. As we will see, there are many possible execution strategies, and the best choice depends on the particular query.

Bloom Filter: A *Bloom Filter* [3, 4, 15] is a data structure that represents a set of elements and supports membership test queries (i.e., "Is this element in the represented set?"). The advantage of a Bloom Filter is that it uses significantly less space than a dictionary or hash table of the elements in the set. On the other hand, there is a small false positive rate that can be traded off against space (i.e., some elements may be reported as being in the set when they are not), and a Bloom Filter cannot retrieve a list of the elements in the represented set (we can only test if a given element is in the set).

Bloom Filters have been widely used in network applications [4]. In particular, they can be used to efficiently compute the intersection between two sets stored on different machines. We note that many search engines only return results that contain all of the query terms. As observed in [17, 12], this means that significant savings can be obtained during query execution under a global index organization, by first sending a Bloom Filter of the docIDs in the shortest inverted list, rather than the complete list, to the next node. Many of the docIDs in the shortest list will not find a match in the other lists, and thus we only have to send a small subset of the items in the shortest inverted list in a second round-trip, at which point any false positives can be detected. As shown in [12], an order of magnitude in bandwidth savings can be obtained by using one or more rounds of Bloom Filter exchanges before sending the actual index items. The approach works for a very wide range of ranking functions, but requires that all results contain all query terms.

It is observed in [15] that additional moderate savings can be obtained by applying standard compression techniques to Bloom Filters, resulting in *Compressed Bloom Filters*. In this paper, all Bloom Filters are in compressed form. We note that for small false positive rates, the best Compressed Bloom Filter reduces to simply hashing all elements to a large enough domain, sorting the hash values, and then compressing the gaps between them using either arithmetic or Golomb coding.

Top-k Pruning Algorithm: Over the last decade, a number of algorithms have been proposed in the IR and database communities that allow computation of top-k results without scanning over the entire inverted lists; see e.g., [7, 8, 16]. The basic idea is to order each inverted list in a smart fashion, such that we can find the top results by scanning only a small part of the list. In particular, Fagin [7, 9] described several algorithms that are very useful in our scenario.

We now describe the first algorithm, called *Fagin's Algorithm* (FA). Assume that items $(D, f_{D,w})$ in each inverted list are sorted in descending order by the value $f_{D,w}$. Now simultaneously scan all lists from the beginning, and stop when there are k docIDs that have been encountered in every list. Note that for these k docIDs, we know their precise scores at this point. Now for any docID encountered in only some of the lists so far, perform a lookup into the other lists to determine its complete score. The claim is that at this point we know all the top-k results and their precise scores. The algorithm is illustrated in Figure 2.3.

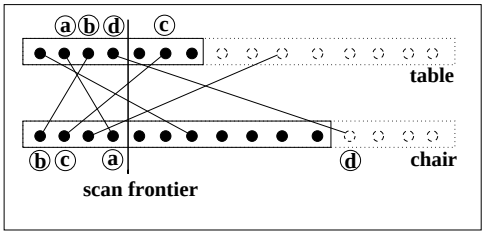

Figure 2.3. Fagin's Algorithm on terms "chair" and "table". The first 4 postings in each list have been scanned, and 2 documents (a and b) have been encountered in both lists. If $k = 2$, then we can stop at this point. Other documents such as c and d were encountered in only one list, and a lookup is used to find them in the other list. For d, the lookup fails because d does not contain "chair". (Documents not containing a term are shown in outlines at the end of each list, but would not be stored in an index.)

A variation of the algorithm, called the *Threshold Algorithm* (TA) [7, 9], performs random lookups into the other lists to compute the precise score as soon as a new docID is discovered during the scan, and the stopping condition is modified as follows: In every step, we combine the scores of the last item scanned in each list; we stop if this value is smaller than the k-th largest score computed thus far. It is easy to show that this also guarantees correct top-k results, and that it never stops later (in fact, usually much earlier) than FA. It is shown in [7] that if the orderings of the documents in the different lists are independent, then the algorithm is expected to terminate after looking at $O(\sqrt{kn})$ entries in each list, where n is the number of documents in the collection (not the length of the list). In the case of m lists, the bound becomes $O(n^{\frac{m-1}{m}}k^{\frac{1}{m}})$. Thus, for long lists and few query terms, this can be significantly better than scanning the entire list. If terms are positively correlated, then the result improves. Note that the result is independent of the actual "shapes" of the distributions of the f_{D,t_i}, though refinements could potentially exploit special cases such as Zipfians.

In [23], we applied the Threshold Algorithm to the query processing problem in a P2P environment with global index organization. The preliminary results on 2-term queries in [23] indicated that for this case, very significant benefits can be obtained even with a fairly straightforward implementation. We note that this approach is limited to ranking functions of the form $F(D,q) = \sum_{i=0}^{d-1} f_{D,t_i}$ (or some combining functions other than the sum, such as maximum or minimum). While this does include classes of functions such as Cosine and Okapi that are popular in the IR literature, it does not include the use of term distance (how far apart are the different terms in the document?) as commonly done in web search engines. One the other hand, the approach does allow the integration of global measures such as Pagerank or visit popularity, and it also works for ranking functions with OR semantics, i.e., when we do not require all terms to occur in the results. (In fact, performance tends to improve in these cases.)

3 Discussion of Related Work

We have already discussed much of the relevant previous work on text query processing in P2P systems, and thus we now only provide a very brief summary. Our work is most closely related and also motivated by the work on query processing under a global index organization in [17, 12, 10] and by our own previous work in [23]. Essentially, our goal in this work is to combine the optimizations in [17, 12] based on Bloom filters with the pruning-based algorithms based on [7] that were used in [23], and to evaluate the performance of the resulting new algorithms on different types of queries. In the process, we noticed that there was a large space of possible execution plans, particularly for queries with multiple terms, that deserves to be explored.

There has also been a significant amount of recent work on query processing under local index organizations [6, 11, 20, 26, 25, 13]. Much of this work is concerned with the problem of finding the best 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 that are close by. This is typically done using techniques similar to the *database selection* problem studied in the context of distributed databases and meta search engines; see [14] for a survey of some techniques. The problem with this approach is that it implicitly relies on the collection to be nicely clustered and the queries to be well-behaved with respect to the given clustering, and it could be argued that for terabyte size data sets and the diverse types of information needs evident in real search engine traces there are severe limits to such techniques. On the other hand, the results appear to perform well on more limited data and network sizes, and there are some significant advantages to local index organizations that create and maintain index structures at the place where the content resides. Finally, we note that a very interesting hybrid approach between local and global index organization was recently proposed in [24].

4. Contributions of this Paper

We study top-k query processing on large textual collections in a wide-area distributed environment with global index organization. We refine and combine several previous approaches and show significant performance gains. In particular:

(1) We address the challenges posed by queries with three and more keywords. Such queries account for a disproportionate fraction of the total cost of query processing, and there are many possible and sometimes nonobvious execution plans that can be used.

(2) We combine the Bloom Filter approach in [17, 12] and the top-k pruning approach used in [23] to obtain additional performance gains.

(3) We perform an evaluation on a large data set and real search engine queries. Our results indicate significant benefits from combining several approaches, and show that different algorithms should be used depending on the number of keywords and the lengths of the lists.

There are many open problems and a few loose ends left by our work. We expect moderate additional benefits (say, $20-30\%$) through a few extra optimizations in the Bloom Filter and pruning strategies, and plan to include these in a later version. Our main focus is on total bandwidth consumption. We are confident that our approaches are implementable with low CPU cost and reasonable total latency, but we do not evaluate this aspect here.

5. Policies for Distributed Query Execution

In this section, we describe several different policies for distributed query evaluation. We first give the *Simple Algorithm* (SA) and the *Distributed Threshold Algorithm* (DTA) as our basic algorithms. Then we present three more advanced policies called the *Bloom Circle Threshold Algorithm* (BCTA), the *Bloom Petal Threshold Algorithm* (BPTA), and the *Simple Bloom Petal Algorithm* (SBPA).

5.1. Simple Algorithm (SA)

Recall that the inverted lists are distributed over the different peers in the peer-to-peer environment. To obtain the top-k results, the most straightforward way is to send all lists to one node and then evaluate the query as in a standard centralized search engine. While the length of the inverted lists ranges from less than 1 KB to more than 1 GB, it is typically in the tens to hundreds of MB for our collection of 120 million pages. So the total cost of transferring all the lists would be prohibitive most of the time. One alternative way is to combine lists step by step in order of the lengths of the lists; we call this the *Simple Algorithm*:

(1) All peers are arranged in ascending order by the lengths of the relevant lists they hold. The first peer sends its entire list of items $(D, f_{D,t})$ to the second peer in the sequence.

(2) The second peer does local lookups on its inverted list to compute the intersection of the two lists. After computing the combined scores for the two terms, it sends the new result list to the next peer.

(3) Each peer repeats this as in Step (2) and adds its scores to the intersection results. The last peer then returns the k docIDs with the highest scores.

The total cost of *SA* mainly depends on the length of the shortest list, since the number of items in the intersection tends to decrease rapidly with each intersected list.

5.2. Distributed Threshold Algorithm (DTA)

Since the *Simple Algorithm* transfers the entire shortest list to the second peer, its cost is at least the length of the shortest list multiplied by the number of bytes needed to present each item. If all inverted lists for the query are large, *SA* is very inefficient. To ameliorate the problem, we introduce another algorithm called the *Distributed Threshold Algorithm* (DTA).

DTA is based on the *Threshold Algorithm* proposed in [7] and applied to P2P query processing in [23], and it decreases the total cost of query evaluation using pruning techniques. A query integrator in DTA, upon receiving a query with m terms, splits it into *m* independent subqueries and issues each one to a corresponding peer called the leader of this subquery. All subqueries execute in parallel. Each subquery scans only the prefix of the inverted list located at the leader, and continually forwards the scanned items to the peer other than the leader that has the shortest list, who will forward it to the other peers in a chain as in SA. Once a subquery obtains a new result in the intersection of the leader's prefix and all other lists, the new result is forwarded to the query integrator who maintains the current total top-k results from all subqueries. The threshold used to decide when to stop in TA is also updated regularly, and the query integrator notifies all leaders once the termination condition of TA is satisfied, or once one leader reaches the end of its list.

To control the cost of updating the threshold at the query integrator, we partition the lists into blocks of a certain size. Whenever the leader sends a block of items to the next node, it attaches the current value at the scan frontier to allow the query integrator to update the threshold. (Alternatively, this value could also be directly sent to the query integrator.) Also, by limiting the rate at which data is sent out, we can make sure that not too much data is in intermediate nodes at the time the query integrator stops the process, as this data has no impact on the final result.

DTA decreases the query execution cost greatly for queries with few terms, but as we will see performance declines rapidly for larger numbers of terms. Recall that this is due to the fact that the total cost of DTA is $\Theta(mn^{\frac{m-1}{m}}k^{\frac{1}{m}})$, where n is the number of documents in the collection (not the length of the list) and m the number of terms in the query. In our case, n will be 120 million, and thus every time we increase m there is a significant increase in the $mn^{\frac{m-1}{m}}$ term.

5.3. Bloom Circle Threshold Algorithm (BCTA)

This is our first new algorithm, and its idea is to improve DTA by integrating Bloom Filters; we call it the *Bloom Circle Threshold Algorithm* (BCTA). As before, each query is split into m independent subqueries, one for each query term. But instead of sending out the actual items in the prefix of the leader node block by block, we first circulate a Compressed Bloom Filter of each block between a select subset of the other peers, and then only send items in the prefix likely to be in the intersection. Thus, we have two phases, a Bloom Filter Circle phase and a second phase as in DTA; these are overlapped.

After the Bloom Filter of the first block of the leader's prefix arrives at the next node, it is processed as follows. Note that each block usually contains far fewer items than any of the complete lists; this implies that the optimal compressed Bloom Filter for this case uses only a single hash function, with Golomb coding of the resulting filter for good and fast compression. Thus, we can perform reverse lookups from the decompressed Bloom Filter (represented as a sorted list of hash values) into the inverted list at the receiving peer, assuming that the inverted list is organized to efficiently support such lookups. Any hash value (bit) in the Bloom Filter that does not find a match is

erased, resulting in improved compression as the thinned-out Bloom Filter is forwarded to each subsequent peers that participates in this circle. Thus, the time spent on each block is proportional to the size of the block and not the size of the inverted list at the receiving peer.

As it turns out, it is sufficient to circulate the Bloom Filters to only a subset of the peers, say the 1 to 3 peers with the shortest inverted lists other than the leader. Once the thinned-out Bloom Filter returns to the leader, only the surviving items are sent to the peers along the chain as in DTA. At the same time, of course, additional Bloom Filters are already circulating. As before, the query integrator maintains the current top-k results and eventually stops the process.

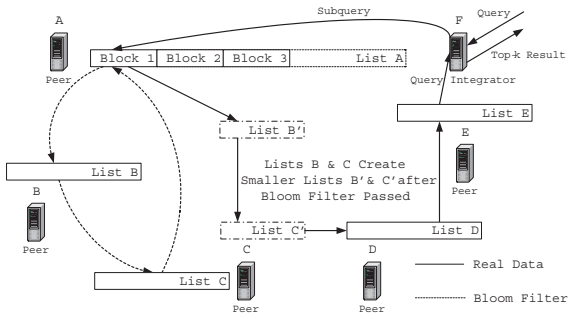

Figure 5.1. Bloom Circle Threshold Algorithm on a subquery of a 5-term query, with coordinator F, leader A, and peers labeled B to E in ascending order of the length of their inverted lists. A first circulates a Bloom Filter of each block in its prefix to B and C, and then sends any remaining items in the block through all the peers to the coordinator as in DTA.

An example of BCTA on a 5-term query is shown in Figure 5.1. There are a number of choices to be made in the implementation. We need to decide how many peers should participate in the initial Bloom Filter Circle, and how many bits to use for the domain of the hash function. A small domain results in a smaller Bloom Filter but also more false positives that result in increased communication in the second phase. In the second phase, we could decide to first send the remaining elements to those peers not participating in the Bloom Filter Circle, but this is not performance critical anymore.

5.4. Bloom Petal Threshold Algorithm (BPTA)

One problem with BCTA is that the inverted lists of the peers participating in the Bloom Filter Circle may be of very different size. To avoid too many false positives, the domain of the hash function for the Bloom Filter should be chosen somewhat larger than the longest list into which reverse lookups are performed. But doing so increases the size of the Bloom Filter, and the increased domain is only needed at the last peer in the circle. Thus, it makes sense to separate the Bloom Filter Circle into several petals, one for each peer in the circle. Each petal consists of a Bloom Filter sent to the peer, and a thinned-out and *compacted* Bloom Filter returned to the leader. By *compacted* we mean that the peer uses the fact that the thinned-out Bloom Filter is a subset of the Bloom Filter that was sent by the leader; this allows us to improve compression in the reply by removing all positions that were already zero in the received Bloom Filter from the domain.

Thus, Bloom Filters are exchanged between leader and peers in a number of petals, where in each petal we can choose the optimal domain size of the hash function based on the length of the inverted list at that peer. We call the resulting scheme the *Bloom Petal Threshold Algorithm*. Note that in principle we could use several rounds of Bloom Filter exchanges between the leader and each peer, but this does not seem to have much benefit in the asymmetric case of a small block or prefix of items being intersected with a typically much larger complete list.

5.5. Simple Bloom Petal Algorithm (SBPA)

This algorithm is actually quite similar to the approach based on Bloom Filters proposed in [17, 12], and should have similar performance. We present it last since we decided to adopt some of the communication structure of the just described BPTA algorithm into it.

The *Simple Bloom Petal Algorithm* is derived from the Simple Algorithm. Thus, instead of creating subqueries that each scan the prefix of one of the lists, we compute again the complete intersection between the shortest list and the other lists with no attempt at top-k pruning. However, instead of sending the entire shortest list, we first perform a few rounds of Bloom Filter exchanges with the other peers, by forming a number of petals with other peers, starting with the peer with the second-shortest inverted list. For each petal we choose the optimal size of the hash function domain based on the number of items involved on both sides. After all Bloom Filter steps have been performed, we start sending the surviving items in the shortest list along the path of the other peers, as in the Simple Algorithm. Thus, we use the petal structure from BPTA, but the second phase follows the simple algorithm.

6. Experimental Evaluation

In this section, we provide an experimental evaluation of the various presented algorithms. Due to space constraints, we can only present our main observations. We start out with a description of the experimental setup. We used real search queries selected from a large log of queries issued to the Excite search engine from 9:00 to 16:69 PST on December 20, 1999. We ignored queries with stopwords and with words that do not appear in our data collection (e.g., very unusual words or typos). The table below shows the percentage of queries with different numbers of terms in the complete trace.

# of terms	1	2	3	4	5	6	> 6
% of queries	18.50	34.58	20.54	11.23	7.05	4.09	4.01

Table 6.1. Percentage of queries with x terms.

Since the communication cost of one-term queries in our environment is trivial, we did not include these queries in our experiments. From Table 6.1 we can see that nearly 96% of all queries have at most 6 terms. For our experiments, we sampled queries from the traces such that we have 500 queries for each number of terms from 2 to 6, and 500 queries with more than six terms, for a total of 3000 queries. The results are of course properly weighted by frequencies when we report average performance numbers for all (including single-term) queries. The document set we use consists of about 120 millon web pages crawled by the PolyBot web crawler [21] in October of 2002, with a total uncompressed size of about 1.8 TB. The total size of a highly compressed inverted index on the collection is slightly above 200 GB, and for the average query the shortest inverted list contains almost a million elements.

After mapping the docIDs to a reasonable domain, we expect that each item $(D, f_{D,t})$ can be represented in about 8 bytes under suitable compression and with negligible collision rate; thus we charge 8 bytes for each transmission of a complete item. Note that in our evaluation, we simulate all algorithms in a centralized environment and report numbers on the bandwidth consumption of the various methods.

6.1. Evaluation of All Algorithms

# of terms	shortest	longest	DTA prefix	DTA cost	SA cost
2	992	6,812	10	160	7,938
3	887	10,947	33	819	8,669
4	874	18,300	96	3,205	8,922
5	707	25,429	177	7,368	7,167
6	591	25,964	263	13,206	5,988
> 6	318	26,733	225	14,007	3,119

Table 6.2. Comparison of the Distributed Treshold Algorithm and the Simple Algorithm on top-10 queries (all numbers in KB or thousands of elements).

In Table 6.2, we compare the cost of SA and DTA. The second column shows the average length of the shortest inverted list participating in the query, which decreases with the number of query terms since with more terms there is a better chance of having at least one rare term. Conversely, the length of the longest list, shown in the third column, increases with the number of terms. The fourth column shows the number of elements in each prefix, in thousands, that are scanned before the query integrator stops execution of DTA, and the fifth column shows the bandwidth consumption of DTA in KB. Finally, the last column shows the cost of SA. Not surprisingly, DTA does much better for few query terms, while SA becomes better for 5 and more terms. Of course, DTA never scans further than the length of the shortest list, but when it does reach the end of the list, each of the subqueries in DTA has consumed about as much bandwidth as SA and it would have been smarter to just focus on the shortest list.

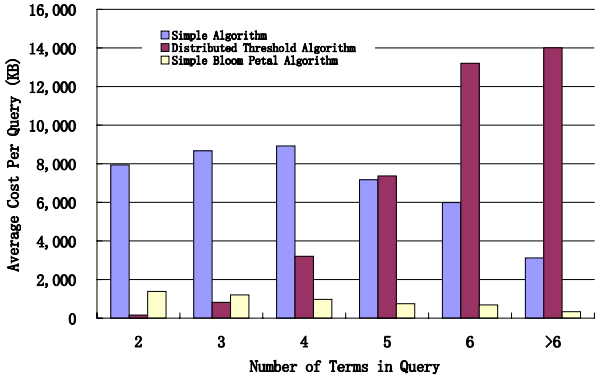

Figure 6.1. Comparison of SA, DTA, and SBPA.

We can see this result more clearly in Figure 6.1, which also shows the performance of the Simple Bloom Petal Algorithm. As we see, SBPA is much better than SA, due to its use of Bloom Filters. While SBPA also looks much better than DTA, we note that DTA is still best for 2 and 3 query terms, and once we add Bloom Filters into the DTA approach to get the BCTA and BPTA policies the comparison will become much closer again.

Before evaluating BCTA and BPTA, we performed experiments on the optimal choice of the hash domain in BCTA; details are omitted due to space constraints. We observed that, as discussed in the motivation for BPTA, there was often no clear way to optimally choose the hash domain if two or more peers participate in the Bloom Circle. However, on average the best strategy used up to three peers in the Bloom Circle and chose the hash domain as about four times the length of the longest list in the circle.

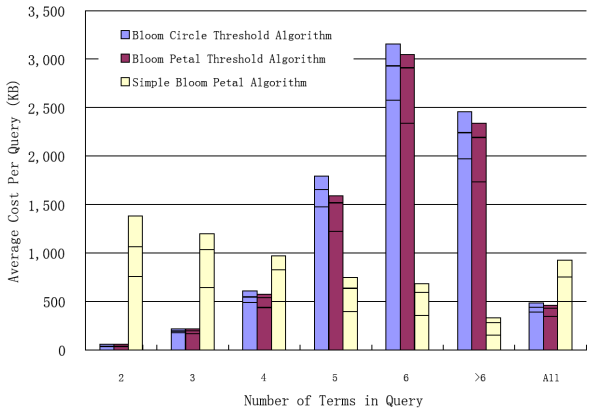

Figure 6.2. Comparison of BCTA, BPTA, and SBPA.

Figure 6.2 compares the total costs of BCTA, BPTA, and SBCA. Note that the Bloom Petal Threshold Algorithm consistently outperforms the Bloom Circle Threshold Algorithm due to the better choice of hash domains, though only by a moderate amount. Both methods also now outperform the Simple Bloom Petal Algorithm (which is comparable in performance to the Bloom Filter

method in [12]) for queries with up to 4 terms. We also show the costs of the different phases of the algorithms, indicating that the first Bloom Filter steps in BCTA and BPTA typically dominate the total cost, as one would expect. Finally, when averaged over the entire query distribution, which contains many short queries, the methods based on top-k pruning outperform SBPA by a factor of 2.

6.2. Selecting the Best Approach for Each Query

As we have seen, BPTA appears to perform best for queries with up to 4 terms while SBPA is better for longer queries. This motivates the simple policy for selecting the best method for each query shown in Table 6.3, which also specifies the best setting of the number of Bloom Filter petals and the size of the hash domains for the Bloom Filters. (A value of 4 bits means that the domain is chosen as 2^4 times the length of the longer inverted list.)

# of terms	algorithm	BF steps	BF bits
2	BPTA	1	5
3	BPTA	2	4
4	BPTA	3	4
5	SBPA	4	2
6	SBPA	5	2
> 6	SBPA	5	2

Table 6.3. Selecting the best algorithms and parameters.

# of terms	shortest list (K)	algorithm	BF steps	BF bits
2	≤ 30	SBPA	1	3
2	> 30	BPTA	1	5
3	≤ 100	SBPA	2	2
3	> 100	BPTA	2	4
4	≤ 300	SBPA	3	2
4	> 300	BPTA	3	4
5	$\leq 4,000$	SBPA	4	2
5	$> 4,000$	BPTA	4	5
6	All	SBPA	5	2
> 6	All	SBPA	5	2

Table 6.4. An improved selection policy.

A second policy proposed in Table 6.4 also takes the length of the shortest inverted list into account. In Figure 6.3 we compare the two policies to an optimal but infeasible policy that always chooses the right algorithm and set of parameters for each query. We see that both policies come quite close to the optimal, with the second policy doing slightly better as it takes the length of the shortest inverted list into account. The optimal approach requires about 191 KB of bandwidth for the average query, while the second policy has a cost of about 218 KB. Some improvements in the policy might be possible for 4 and 5 terms, where there is still a gap between the optimal and the feasible policies.

In Figure 6.4 we compare the frequency of queries with a particular number of terms with their total impact on the average cost. As we see, while most queries are short,

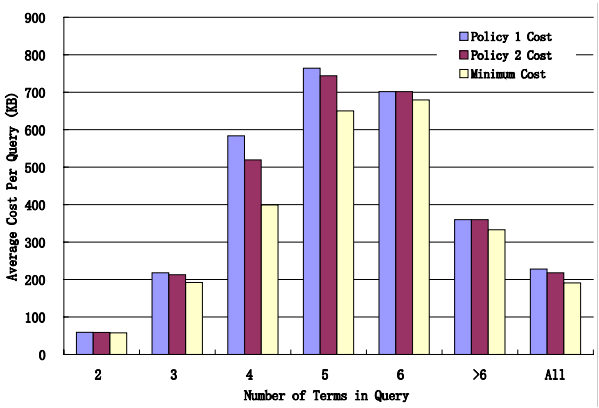

Figure 6.3. Comparison of different selection policies.

queries with 4 or more terms are responsible for a disproportionate amount of the total cost.

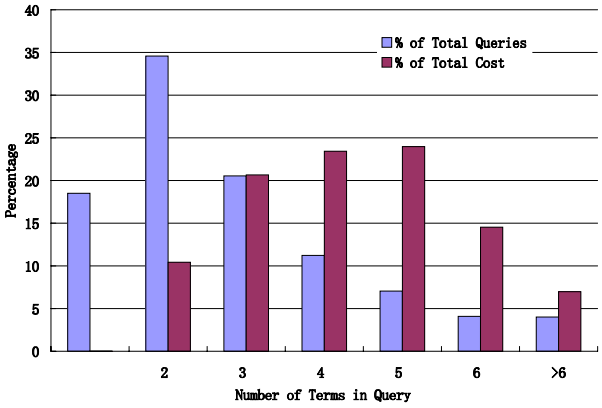

Figure 6.4. Average cost versus frequency of queries with different numbers of terms.

Finally, in Figure 6.5, we provide an estimate on how these numbers would change for a collection with 3 billion web pages, i.e., 25 times our test set. The cost of SBPA would increase approximately linearly, while the behavior of BPTA is more complicated but much better than linear in the collection size for 2 and 3 query terms. Details are omitted due to space constraints, but we observe an average cost of over 3 MB per query that would be unacceptable in most scenarios.

7. Concluding Remarks

While our results show a reasonable cost for query processing with up to 120 million web pages, there are still a number of challenges that need to be overcome before large-scale P2P web search engines can become a reality. We note first that some additional optimizations in the Bloom filter implementation, plus some optimizations in the pruning schemes such as earlier termination and variable scan speeds depending on the relative list lengths, should result in additional benefits in the range of maybe $20 - 30\%$. Some extra benefits would result from the use of caching techniques [2] or hybrid organizations such as

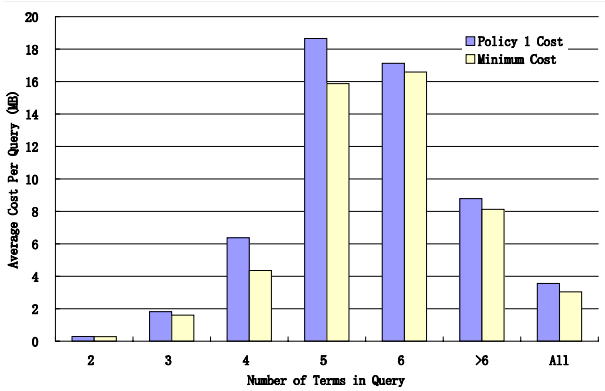

Figure 6.5. Estimated cost of policies on 3 billion pages.

[24]. However, the cost of query processing on 3 billion pages would likely remain high.

Another challenge is how to support ranking functions that use the distance between query terms in the document, as commonly done in current web search engines. We have some initial promising ideas in this direction that we plan to explore. Finally, there are a number of challenges other than query processing, such as index updates and dynamic changes in the network topology that a real system would have to deal with. Even if large-scale P2P search becomes feasible, it is not clear that it would be preferable to a centralized architecture. On the other hand, we hope that some of the same techniques will turn out to be useful for real search problems on more moderate (but still large) widely distributed textual collections.

Acknowledgements: We thank Xiaohui Long for help in preparing the data and query sets. This work was supported by NSF CAREER Award CCR-0093400, NSF ITR Award CNS-0325777, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

References

[1] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *9th Symp. on String Processing and Information Retrieval*, 2002.

[2] 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.

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

[4] 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.

[5] 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, Jan. 2000.

[6] 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.

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

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

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

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

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

[12] 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.

[13] 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 Conf.*, 2003.

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

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

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

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

[18] 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, Nov. 2001.

[19] 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, Aug. 2002.

[20] 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, Dec. 2002.

[21] 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.

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

[23] 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 *Int. Workshop on the Web and Databases (WebDB)*, 2003.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.