SPRITE: A Learning-Based Text Retrieval System in DHT Networks

Yingguang Li[1] H.V. Jagadish[2] Kian-Lee Tan[1]
[1]National University of Singapore [2]University of Michigan
liyg@comp.nus.edu.sg, jag@umich.edu, tankl@comp.nus.edu.sg

Abstract

In this paper, we propose SPRITE (Selective PRogressive Index Tuning by Examples), a scalable system for text retrieval in a structured P2P network. Under SPRITE, each peer is responsible for a certain number of terms. However, for each document, SPRITE learns from (past) queries to select only a small set of representative terms for indexing; and these terms are progressively refined with subsequent queries. We implemented the proposed strategy, and compare its retrieval effectiveness in terms of both precision and recall against a static scheme (without learning) and a centralized system (ideal). Our experimental results show that SPRITE is nearly as effective as the centralized system, and considerably outperforms the static scheme.

1. Introduction

Peer-to-peer (P2P) computing has enabled end-users to share their data with ease. However, text data and documents are very demanding very demanding in a distributed environment as traditional centralized indexing techniques cannot readily be deployed. In a large scale distributed environment, global knowledge is usually unavailable. Moreover, it is impractical to index all terms of a document (as is done in a centralized system) in a distributed fashion. Therefore, novel techniques are needed to support text retrieval in distributed systems.

In the literature, there are several approaches to support text retrieval in P2P systems. The most straightforward approach, typically adopted in unstructured systems like Gnutella [5], is to flood a query within a certain radius of the neighborhood of the querying peer. However, such an approach is not only bandwidth inefficient but may have low recall as peers containing relevant documents may be far away and unknown within the local neighborhood searched. To reduce the communication overhead, an alternative approach is to employ *routing indexes* [2] that provide more directed search as only peers with matching query terms are searched. However, this method also operates within a certain radius in an unstructured environment, and has the same limitation of low recall.

A third approach employs a DHT-based (Distributed Hash Table) network. All terms in every document are indexed in the DHT network. In other words, each peer maintains an inverted list for the terms assigned to it by the overlay network. To process a query, all peers responsible for the query keywords are visited, and the relevant index entries are returned to the querying peer. The querying peer can then compute the similarities between the query and the documents containing those keywords to generate the ranked list. While this approach is relatively query-efficient, and is expected to have higher recall than the other approaches, the main challenge is that of building and maintaining such a distributed index. Even after stemming and stop-word elimination, the total number of terms in a document is too large: each term is likely to have been assigned to a different peer, so that a single document insertion could require updates in a large fraction of the network. Therefore, the overhead to disseminate the indexing information is too high to be of practical use. In addition, it is equally costly for the owner peer to periodically probe the indexing peers to ensure that they are still "alive". If one could somehow bring down the cost of index construction and maintenance, it appears that this third approach, with a structured overlay network, will provide the most effective retrieval system. It is exactly this challenge that we tackle in this paper.

Our proposed solution is motivated by three observations. First, a document will most often be queried using a small number of terms that characterize it. It may suffice to index a document on only these charateristic terms, and drop all others. In fact, it has been argued in [16] that if a query term p is not among the top frequent terms of a document, then adding p to the query is unlikely to materially affect the ranking of this document. Second, a term that is not used in a query has no effect on the ranking of the documents. If we can know which terms will be used in queries that seek a particular document, then we should index only those terms for the document: all other terms merely increase the index size without providing any addi-

tional accuracy. Third, users with similar interests are likely to retrieve a similar collection of documents with a similar set of queries that share some common keywords. Such a *query locality* phenomenon is not uncommon in search engine queries - analysis of Excite search engine trace [19] and Altavista search engine trace [14] showed that queries submitted to these search engines not only have significant locality, many are repeatedly issued by either the same or other users, and that multiple-word queries are common.

We note that the first and second observations suggest that it may suffice to index only a small well-chosen set of representative terms in each document. The second and third observations also hint that the query keywords may potentially contribute to the set of representative terms. Furthermore, the third observation suggests that it may be possible to learn from past queries - since similar queries share certain common keywords, past queries may be used to refine the selected representative terms.

Based on the above observations, we propose our algorithm SPRITE (*S*elective *PR*ogressive *I*ndex *T*uning by *E*xamples) that ensures a small set of representative terms are well-chosen. SPRITE also progressively learns from (past) queries to refine the set of chosen indexing terms. In this way, new terms may be injected into the system, while "obsolete" terms (as a result of changing access patterns) may be removed/replaced.

The rest of this paper is organized as follows: Section 2 discusses the related work on supporting text retrieval in a P2P network. An overview of the SPRITE architecture is described in section 3. In Section 4, we discuss how queries are processed in SPRITE. We also discuss how to integrate the text retrieval task with the overlay network routing protocols, using Chord[15] as a specific example. Whereas we have used Chord in all our examples and in our implementation, there is nothing in our central idea that depends on Chord, and the reader should be able to see how to make the necessary adaptation to a different overlay network.

In Section 5, we present the scheme to select and refine indexing terms. We have implemented the proposed strategy, and compare its retrievel effectiveness (in terms of both precision and recall) against a static scheme (without learning) and an ideal centralized system. Our experimental results, presented in Section 6, show that SPRITE is nearly as effective as the centralized system, and outperforms the static scheme. In Section 7, we discuss some possible extention works of SPRITE. Finally, we make some concluding remarks in Section 8.

2. Related work

In structured P2P networks [15, 12], including the "loosely structured" networks [1], search on file names can be easily handled. Moreover, the *lookup* function can guarantee a term be found in $\log N$ hops, where N is the number of peers in the network. A file name can be treated as an integrated entry or a set of terms, and hashed if necessary, and then indexed in the network. However, indexing file content involves more challenging issues. Many of these have been addressed in [9], in which two major concerns are discussed: storage constraints and communication constraints. Both of these are caused by the large number of terms in a document to be indexed.

To the best of our knowledge, the most similar work to SPRITE is eSearch [16]. In eSearch, a document is indexed on the top k terms and the complete inverted list of the document is replicated and stored in k indexing peers. In the description of top term selection, the authors assume that some global statistics can be obtained. However, global statistics are expensive to obtain and tend to be inaccurate in a P2P network, where peers frequently join and leave the network, and documents are shared and unshared frequently. In SPRITE, we do not make this assumption. Term expansion is employed in eSearch. This is orthogonal to the basic scheme, and not discussed further in this paper, though term expansion could also be used with SPRITE.

In [10], the authors proposed a scheme to process content-based retrieval in hybrid P2P networks. In the hybrid network, a super peer is responsible for summarizing the contents among its normal peers. The summaries are defined as "resource descriptions". Queries are routed according to the "resource descriptions": a query is forwarded to the peers containing the relevant resources with some probability above a threshold. KSS [4] divides predefined queries into a set of combinations. Each element in the set is hashed and indexed in a structured DHT. The query term space can be very large and the combination is too complex to forecast. Besides addressing some challenges of keyword search in P2P systems, [9] proposed to combine some techniques (e.g., caching and query compression) to reduce communication cost. In [13], bloom filter is employed to compress the message size. Works based on latent semantic indexing (LSI), such as *pSearch* [18, 17], predefines the term spaces. A global knowledge is assumed to compress documents with LSI into fewer dimensions. The indexes are rotated several times and a set of important indexes are placed into an overlay of CAN [12] each time. A query is preprocessed similarly and answered as a knn search.

3. The big picture

The SPRITE system comprises a large number of computers (peers) that are organized into a structured overlay network, such as Chord, that is capable of supporting simple indexing through a distributed hash table. Each peer plays two roles: owner peer and indexing peer. An *owner peer* owns and shares certain documents. It is responsible

for maintaining each shared document it owns, locally indexing it, and selecting the *global index terms* (A global index term is a document term to be injected into SPRITE to facilitate query searching.) for it. An *indexing peer* is responsible for managing meta-data for terms assigned to it. This meta-data is primarily an inverted index of the (global index) terms managed by the peer. The information maintained in the inverted list include the documents containing the term and their respective owner peers. In addition, each indexing peer also maintains a history of past queries (rather, the keywords corresponding to the queries). To reduce the storage, each indexing peer maintains only the most recently issued queries.

There are two main services supported by SPRITE. First, a peer can share a new document with other users. In this case, the document owner has to select and publish corresponding global index terms into the SPRITE system. Second, a peer can submit a query to retrieve relevant documents through keyword search. While the query processing service is straightforward, the document sharing service is challenging. As noted above, it is too expensive to publish all the terms (even after stemming and stop-words elimination) in a document. Moreover, based on the observations in the introduction, we believe it would suffice to index only a small well-chosen set of representative terms. Thus, SPRITE publishes only a small subset of representative terms that are subsequently refined based on past queries.

More formally, let the set of documents in the network be D and the set of queries be Q, over all time. Suppose document d_i is determined to be relevant to queries q_{i1}, q_{i2}, ... q_{ik}. Let the union of the keywords in the queries be K_i. In the ideal case (with perfect knowledge into the future), document d_i is only indexed on the keywords in K_i. SPRITE attempts to do exactly this with limited knowledge: it learns a set of keywords, K_i', which approximates K_i. Terms in $K_i' - K_i$ are indexed unnecessarily; terms in $K_i - K_i'$ may cause document d_i to be misjudged as irrelevant to some query q_{ij}. Choosing K_i' wisely is at the heart of SPRITE.

Towards this end, for each document, SPRITE begins with an initial guess at the important terms. This guess can be based on user input or through automatical selection of high frequency terms in the document, or a combination of such techniques. For ease of presentation, in this paper, we shall simply pick the most frequent terms as the initial global index terms. Next, with these terms, SPRITE examines past queries that have queried these terms. (Recall that the queries are stored at indexing peers. As such, they can be obtained from the indexing peers.) Based on these queries, SPRITE identifies a new set of terms to be indexed, augmenting and replacing the initial set of index terms. This process of examining past queries, and refining the indexing terms, is repeated periodically.

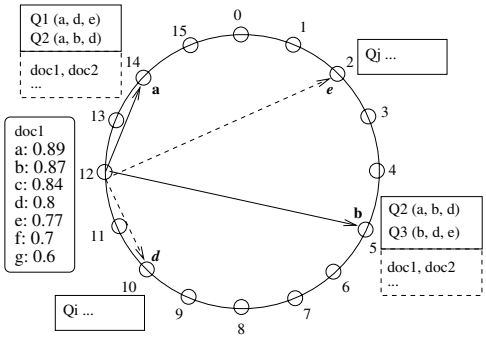


Figure 1. Indexing terms in a Chord Ring.

When an owner peer of a document D wants to update the indexed terms of D, it polls the indexing peers with an *index update message* that contains all the global index terms of D. It is possible that a past query contains multiple global index terms of D and thus is cached by multiple indexing peers. Apparently, it involves much redundancy if such a query is sent to the owner peer by all related indexing peers. In SPRITE, every cached query is hashed also, which can be precomputed offline in fact. The *closest* term to the query can be identified among all global index terms by comparing the hash values. Only the indexing peer responsible for the closest term sends the query back. In this way, we avoid sending the same query multiple times. Note that the number of global index terms is much smaller than the number of past queries cached in the indexing peers. Therefore, the redundancy above can be removed effectively.

Figure 1 illustrates an example with peer 12 as the owner of document $doc1$. Suppose two terms a and b are selected as the most important terms in $doc1$ to be indexed initially. These terms are published to the appropriate indexing peers, say peer 14 and peer 5 for a and b respectively. Now suppose Peer 14 receives two queries, $Q1$ and $Q2$, on term a, and peer 5 has two queries, $Q3$ and $Q4$, on term b. In the next learning period, peer 12 sends messages to peer 14 and peer 5 for past queries on terms a and b respectively. Upon receiving the four queries, peer 12 calculates the similarity between the queries and document $doc1$ and then chooses another set of terms to be published further. In this example, terms d and e are chosen and added into the index. It is worth noting that even though term c has a higher rank (more frequent) than d and e for $doc1$, yet it is not indexed because it has not been used in any query for $doc1$ thus far. One may worry that c may have been specified as a search term in many queries, none of which returned $doc1$, and regarding which peer 12 is thus completely unaware. However, we note that peer 12 can be unaware of such queries only if they do not involve any of $\{a, b, d, e\}$. $doc1$ will not be relevant to any such query with high probability, since it only specifies one of multiple frequent terms in $doc1$.

Next, let's look at the information retrieval service. A query is processed in searching and retrieval phases. The searching phase is more complicated and important since it decides the quality of the answers. Given a query, all indexing peers responsible for the query terms are visited, and the related indices are obtained by the querying peer. Besides the term frequency, the document length and the counted document frequency are also returned along with each index entry. The term frequency and document length can be combined as a normalized term frequency. Next, at the querying peer, index entries for the same document are consolidated and used to calculate the similarity between the document and the query. Finally a ranked list is constructed and a desired number of documents are returned to users as answers. The retrieval phase is simply a downloading action to the relevant documents, so we do not discuss it further in this paper.

Note that we do not have the precise document frequency of a term (i.e., the number of documents containing the term). Instead we use as surrogate the *indexed document frequency*, which is the number of documents for which this term has been chosen as a global index term. The difference between these two frequencies is the set of documents in which the term occurs but has not been chosen for the global index. The indexed document frequency for each term is easily available at its indexing peer. Semantically, one can see that indexed document frequency serves the same purpose as, and can even be argued to be more appropriate than, regular document frequency. This intuition is borne out by the retrieval quality results we present in Section 6.

4. Query processing

Consider a query peer that issues a keyword search, say comprising n terms. The query peer first hashes on each keyword to determine the indexing peer responsible, and retrieves the corresponding inverted list entries. Using these, it can determine the similarity between the query and potentially relevant documents.

In traditional IR techniques, every term in a document is assigned a certain weight based on some statistics. One of the most popular formulas is $TF{\cdot}IDF$. The weight of term k in document i is:

$$w_{ik} = t_{ik} \times \log \frac{N}{n_k}.$$

Here, t_{ik} is the frequency of term k in document i normalized by the document length, N is the total number of documents in the entire corpus and n_k is the document frequency, or number of documents containing term k.

In a structured P2P network, t_{ik} is available as part of the metadata in the inverted list. The number of documents containing term k, n'_k, can be counted by the querying peer once the list is retrieved. However, this *indexed document frequency* is smaller than n_k in the case of SPRITE because the term may appear in some documents but is not selected as a global index term because it is lowly ranked among other terms in these documents. N, unfortunately, cannot be accurately determined in a P2P context: peers join and leave the network and documents may be shared and unshared at will. However, N is usually much larger than n_k, except for the terms in the stop word list, which are filtered away anyway. As long as N is the same for all the peers in calculating term weights, only the absolute IDF values will be affected and so does the similarity. Thus, it will not affect the relative positions of documents in the final ranked list. Therefore, we can simply use a sufficiently large N.

Given the individual term weights, we use the similarity formula proposed in [8] (the second method):

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} w_{Q,j} \times w_{i,j}}{\sqrt{number\ of\ terms\ in\ D_i}}$$

where $w_{Q,j}$ is the weight of the jth term in query Q, and $w_{i,j}$ is the weight of the jth term in document D_i. Note that the number of terms in D_i is available in the metadata of the inverted list retrieved. This formula simplifies the normalization (compared to the original similarity formula) and reduces the computation cost. Its performance is shown to be almost the same as the original formula in [8].

A document D_i containing a specified query term t_j may not have chosen t_j to be a global index term. In this case, w_{ij} is erroneously assumed to be zero rather than positive, and the value of $sim(Q, D_i)$ computed is decreased. In the next section, we will show how to choose index terms such that if the true value of w_{ij} is large, then t_j is chosen as a global index term for document D_i with high probability. If the true value of w_{ij} is small, then approximating it to zero introduces only a small error in the score computation and may make no difference to whether D_i is included in the ranked list for Q.

Before leaving this section, we mention an alternative approach to compute the similarity between a query and a document. Instead of the querying peer performing the computation, we can push the task to the indexing peers. This approach is adopted in [16]. Here, for each term of a document indexed, all the terms of the entire document are also stored as meta-data. In this way, the indexing peers can determine the similarity between a keyword and the documents containing the term to produce the ranked list. However, the indexing peers have to return their locally produced ranked lists to the querying peer eventually and the querying peer needs to merge the ranked lists into one. Many similarity calculations and ranked list sorting are performed repeatedly and redundantly. Therefore, we choose to assign the entire task to the querying peers.

5. Index construction and tuning

When an owner peer shares a document D, it indexes some representative terms in the system. This involves two stages. First, some initial terms in D are chosen and injected into the system. Next, the second stage is performed periodically to tune the index progressively. Essentially, at each run, more terms are selected from D based on the historical queries. The index terms for D are then refined by inserting new terms and removing noisy terms. To control the number of terms to be maintained, we limit the maximum number of terms to be indexed to a small value (say, 30). We will present the two steps below. Before that, we describe the metadata maintained at each peer.

5.1. Metadata in SPRITE

Recall that in SPRITE, each peer plays two roles: owner peer and indexing peer. Every indexing peer maintains two types of information: (a) A number of terms and the corresponding inverted lists, i.e. the documents that contain those terms. For each indexed term, the indexing peer also needs to store the owner peer's IP address, the owner document ID, the term frequency in the document and the document length. These metadata are used in query processing. (b) A set of queries, σ. Each query essentially comprises a set of keywords. Note that a query is only maintained at peers whose indexing terms contain at least one query term. These queries are used in the learning process.

At every owner peer, for each term in a document, two values are stored: (1) $qScore$, the similarity between the document and the most similar historical query (maintained at an indexing peer) containing this term (to be discussed shortly); and (2) QF (query frequency), the number of historical queries containing this term.

5.2. Initial term selection

When an owner peer first shares a document, we need to select an initial representative set of global index terms. The initial important terms of a document can be selected systematically or input by users. As a first cut, we adopt the following approach. First, we summarize the terms in a document and filter them with a stop-word-list to remove frequent but meaningless terms, such as "the" and "is". Second, we apply the stemming algorithm to unify terms by removing the suffix, such as "ed" and "ing". These two methods are well studied in the text retrieval community. The top F most frequent terms are then chosen as the initial terms. Note that at this point, only local information is available, so initial term selection solely relies on term frequency in the owner peer.

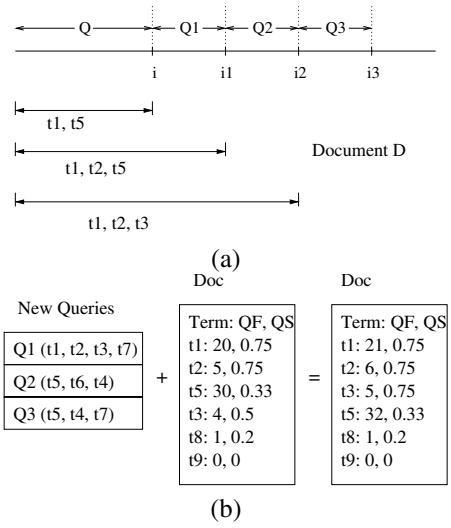


Figure 2. The learning phase in SPRITE.

5.3. Tuning indexing terms

The learning stage is invoked periodically. We shall first present the basic idea with a naive implementation, and then discuss an efficient scheme. We use Figure 2(a) to illustrate the learning stage. In the figure, we show several iterations of learning: at iteration i, an owner peer bases its learning on the historical query set Q; at iteration i_1, the peer learns from a larger set of queries, $Q \cup Q1$; and so on. At iteration i, it is able to identify two terms, say t_1 and t_5. At iteration i_1, it identifies two new terms t_2 and t_6. However, suppose that we are limited to indexing only 3 terms, and it turns out that t_6 is the lowest ranked among the 4 terms, thus t_6 is removed. In iteration i_2, a new term t_3 replaces an obsolete term t_5.

The query set used for learning is determined by the current set of indexed terms. Essentially, for each indexing term, the indexing peer is polled to retrieve the query metadata of that term. The query set is then the union of all queries over all the indexing terms. The crux of the learning scheme lies in selecting the useful terms of a document from the query set. Now, from the query set, we can gather two important pieces of information. First, we can determine how similar is the document to the past queries. We define the query score, $qScore$, as follows:

$$qScore(Q, D) = \frac{|Q \cap D|}{|Q|}$$

Intuitively, if a query is very similar to a document, then it indicates that the terms in the query can represent the key meaning of the document. In other words, the document is likely to be relevant to that query. Careful readers may question why we have not used the conventional formula to measure the similarity between a query and a document. If the conventional formula is employed, the role

of a query and document are interchanged: the document is treated as a query and the queries are treated as the document corpus. This is because we are now selecting similar queries for a document. In the conventional formula, the more documents a term occurs in, the less important the term is, which is not true in our scenario. When choosing descriptive queries, a term occurring in many queries as well as in the document indicates that the term is more descriptive of the document. Therefore, $qScore$ can represent the similarity between a query and a document better than the conventional formula.

Second, for each term t in the query set ϑ, we can determine how frequently it appears in ϑ. This is denoted as $QF(t, \vartheta)$, the query frequency of t in ϑ. This essentially tells us how common the query term is. Intuitively, if a term occurs frequently in many queries, it may be potentially useful to index it.

Now, given a set of queries ϑ, the similarity of term j in query i to document D ($t_{ij} \in D$) is defined in the following formula:

$$Score(t_{ij}, D) = qScore(Q_i, D) \cdot \log QF(t_{ij}, \vartheta).$$

The formula indicates that a term is representative to a document if (1) the query containing it is similar to the document; and (2) the term in the document is frequent among the queries. Intuitively, it is insufficient to consider (1) alone since it does not factor in the frequency of the ocurrences of the terms in a query (and hence fails to consider similar queries). It is insufficient to consider (2) alone because a document is relevant to a query if there are more matching terms from the query. Thus, a combination of the two is necessary. In combining the two, we have used a logarithm of the QF to give higher weight to the contribution of $qScore$. The reason for reducing the effect of QF is because the qualities of the queries are different. Expert users usually have good domain knowledge and issue high quality queries. Such queries are very useful in differentiating the requested documents from others. On the other hand, poor queries always include terms that are too general to distinguish the requested documents.

Based on the ranking by (this combined) $Score$, we pick the high scoring terms to be indexed. Now, a straightforward optimization is for the owner peer to store the query sets whenever they are retrieved, so that each iteration only needs to pull back the incremental query set. Even so, this algorithm is expensive in terms of both storage cost and computation cost. The owner peer has to keep all the past queries and check all of them in each iteration of learning. We propose an algorithm that can compute $Score$ for all terms based on only the incremental query set between iterations (without having to recompute from the entire historical query set). See Algorithm 1.

Let the query set between the current iteration and the last iteration be Q'. Here, the owner peer only needs to store some statistics for the past queries (upto the last iteration, but excluding queries in Q') along with the documents instead of the entire set of queries. For each term in a shared document, only its query frequency and the largest query score in the history are maintained. Then every new query in Q' is processed. If the term occurs in the query, we calculate the query score for this term and count its query frequency in Q'. If the query score is larger than the one saved for past queries, we update it for this term. The query frequency of this term is the sum of the one for past queries and the one in Q'. A new similarity between the term and the document is calculated with the two parameters. We then insert the term with its new similarity into a list sorted by similarity. If the term exists in the list, then we simply update its similarity value. After all the new queries are processed for a term, the largest query score of the term is stored in the statistics and the query frequency of the term is increased also. Given two sets, S_1 and S_2, it is obvious that $max(S_1 \cup S_2) = max(max(S_1), max(S_2))$. So, the query score used is the largest for a term. QF is simply a count function and is thus cumulative. With the same two factors, the multiplication is the same, so the results of Algorithm 1 is equivalent to the naive scheme described earlier (that reprocesses all the queries in each learning iteration). Clearly, since Algorithm 1 exploits incremental computation (i.e., only need to compute for queries that arrive between the last iteration and the current iteration), it is very efficient.

Algorithm 1: The optimized learning algorithm.

```
1  Q' is current query set;
2  RL is a rank list, which is empty initially;
3  for each t in the document D_k do
4      Let qf be the query frequency of t stored for the past queries;
5      Let qf' = QF(t, Q');
6      for each Q_i ∈ Q∪Q' do
7          if t ∈ Q_i then
8              Let qs be the largest query score associated with t in
               the past queries;
9              qs' = qScore(Q_i, D_k);
10             if qs < qs' then qs = qs';
11             Let s = qs · log (qf + qf');
12             if t is not in RL then
13                 Insert (s, t) into RL;
14             else
15                 if the existing similarity is smaller than s then
16                 Replace the existing similarity with s;
17 Choose top T ranked terms for this document;
```

A learning example with Algorithm 1 is discussed in figure 2(b). A document, Doc, is limited to be indexed with three terms. At time i, $t1$, $t2$ and $t5$ are indexed (shown in the left Doc). Their similarities to the documents for the past queries are: 0.75*log 20=0.975, 0.75*log 5=0.524 and 0.33*log 30=0.492 respectively. Three queries are pulled

back in the learning process: {Q1, Q2, Q3}. Then the query frequency and the largest query score are updated accordingly (shown in the right Doc). We recalculate the similarities and obtain a new ranked list. The new score of $t3$ is 0.75*log 5=0.524 and the new score of $t5$ is 0.33*log 32=0.501. Thus, $t3$ is indexed and $t5$ is removed from the distributed index for Doc.

6. Performance study

In this section, we evaluate the performance of SPRITE. As reference, we use a centralized text retrieval system and the basic eSearch system [16]. The centralized system acts as an ideal distributed system with perfect global knowledge, including the exact document frequency and total number of documents in the corpus. (We used a classic $TF \cdot IDF$ scheme in the centralized system). Hence, it is expected to be superior. By comparing against it, we will be able to see how close SPRITE is to an optimal solution. The basic eSearch system indexes a fixed number of most frequent terms in a document. It is the best distributed search system currently known. The comparison against eSearch demonstrates the gain that can be derived from adaptivity/learning.

We preprocessed the documents in the standard way: removing the terms in the stop-word-list, and then stemming is applied to the remaining terms. The default stop-word-list in Lucene is used for this purpose. We used the two standard metrics for text search: precision and recall. If the top K documents are returned for a query, K' of them are relevant to the query and there are R relevant documents in the entire corpus, then the precision is defined as K'/K and the recall as K'/R. All precision and recall results presented later are in terms of the ratio of a specific system over the centralized system.

We implemented Chord as designed in [15]. All terms are hashed using MD5 hash function. Our study is based on simulation, and all experiments are conducted on a dual-Pentium4 3.0GH CPU PC with 1GB RAM.

6.1. Data set and query set

To evaluate SPRITE, we need queries to be "similar" (share some keywords and relevant documents) for SPRITE to learn from. Unfortunately, benchmarks are usually created to exercise a maximum of functionality with as few queries as possible. Hence, there is little similarity between queries. To deal with this, we implemented a query generator to generate queries from a real dataset and its corresponding queries. We used the TREC9 dataset and its queries [7] as the base dataset. This dataset contains 348565 documents and 63 queries and their corresponding relevant documents (identified by experts). Our generator is designed based on two reasonable properties: (a) queries with similar relevant documents as answers ought to share some common keywords; and (b) the term distribution and result distribution should follow those of the original query set. The first property ensures that the system can build an effective index with the training queries and the testing queries can benefit from the learning process. The second property ensures fairness: popular terms in the original query set should occur frequently in the generated query set. If an original query has many answers (the documents), then the new queries derived from it should have many answers as well. In the centralized system (the benchmark), the relevant document distribution in the ranked list of a new query should be similar to that of its original query. The query generator comprises the following two phases.

Phase 1: Term Selection. In phase 1, for each query in the original dataset, we generate k new queries. (In our study, we set k to 9.) Let $Q = \{q_1, q_2, ...q_n\}$ be an original query. A new query $Q' = \{q'_1, q'_2, ...q'_m\}$ is composed of two sets: Q' = $Q'_1 \cup Q'_2$. The terms in Q'_1 are from the Q: $Q'_1 \subset Q$. Each term in Q'_2 is randomly selected from the term space, which contains all terms appearing in all documents. Thus, while Q'_1 inherits some terms from Q, Q'_2 targets different aspects of the documents to introduce some noisy terms to model a more realistic scenario. For the new query Q', we need to identify a set of documents R' as its relevant results (see phase 2).

We define a tunable parameter, $O = \frac{|Q'_1|}{|Q|}$, to control the overlap between the original and new queries, where $|Q|$ denotes the number of terms in Q. The threshold, O, determines the percentage of terms in the original query that is retained in the new queries. Tuning this factor will change the overlap between the original query and new queries. The actual terms in Q'_1 are randomly picked from Q.

In order to select terms of type Q'_2, we pick terms from the entire corpus that are "equally" important as the terms that have been dropped from Q. The importance depends on the distribution of the term: the number of term occurrence and the number of documents containing the term. We define a simple metric to measure the distribution of a term in a corpus.

$$Distribution(t_i) = Freq(t_i) \times Num(t_i)$$

Here, $Freq(t_i)$ is the total term frequency of term t_i in all documents and $Num(t_i)$ is the number of documents containing term t_i. The two factors are used to measure the importance of the term. The reason we do not use the conventional term weight formula $TF \cdot IDF$ is that it can only represent the weight of a term in a document. $Distribution(t_i)$ focuses more on the distribution of a term in the corpus. Given a term in $Q - Q'_1$, we find the top S similar terms and choose one of them randomly to replace

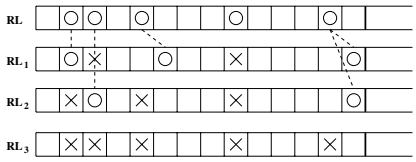


Figure 3. Defining relevant documents.

the old term. Here, the difference between two terms t_i and t_j is measured by $|Distribution(t_i) - Distribution(t_j)|$ (the smaller the value is, the more similar they are). In our study, S is set to 5. All terms in Q'_2 are selected randomly from the replaced terms in $Q - Q'_1$.

Phase 2: Identifying Relevant Documents. In phase 2, the relevant documents of the generated queries are defined based on the relevant documents of the original queries. We now define some documents as relevant answers to the new queries. A new query ought to share some relevant documents with the original query and have some new relevant documents for itself. With the centralized system, we can calculate the ranked list, RL for the original query Q, and RL' for a new query Q', over all the documents. The top E documents in the ranked lists are considered when defining relevant documents for Q'. Some relevant documents will never be returned to users because their ranks are very low and users are usually interested in a small number of results only. Thus, they will not affect the precision or recall and are not considered when defining relevant documents for the new queries. For each such document in RL' and relevant to Q, we define it as relevant to Q' and mark the relevant document in RL with the most similar rank. Then, for each unmarked relevant document in RL, the document in RL' with the same rank is defined as relevant to Q'. An example is shown in Figure 3. Here, RL is the ranked list to an original query Q, and RL_1, RL_2 and RL_3 are the new ranked lists for new queries Q_1, Q_2 and Q_3 derived from Q. Circles are the original relevant documents to Q and crosses are newly defined relevant documents. The left most document has the highest rank. In this example, $E = 14$ (In the experiments, $E = 1000$). For Q_1, 3 original relevant documents (marked with circles) are in its top E ranked list. Three documents in RL with the most similar ranks are marked (indicated by the dashed lines). For the remaining two relevant documents, two documents in RL_1 with the same ranks are defined as relevant to Q_1 (marked with crosses). In this way, the distribution of the new relevant documents is similar to the distribution of the original relevant documents.

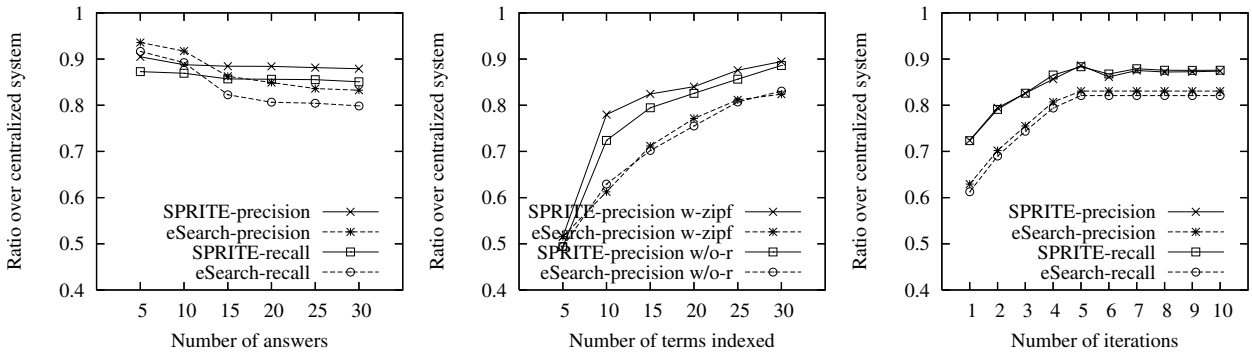
6.2. Experimental setup

We started with 63 queries from the TREC9 dataset, so we eventually have 630 queries with the overlap ratio O = 70% after the query generation. We split these queries into 2 equal groups: a training set and a testing set. The queries are randomly assigned to the groups. For each query in the training set, the keywords are inserted into SPRITE. Next, we insert the metadata of the documents into the system as follows. For each document to be inserted, 5 most frequent terms are initially indexed. Following the 5 initial terms, 3 iterations of learning are executed by the owner peer of a document. In each iteration, 5 new terms are indexed. So the total number of terms indexed equal to 20. Once all the documents have been indexed, we run the queries in the testing set. For each query, we retrieve top 20 answers and determine its precision and recall. For eSearch, we set the number of indexed terms as 20. In the above description, the parameters used (e.g., 5 initial terms), are the default settings. Unless otherwise stated, we assume the default settings.

6.3. Experimental results

First, we compare the precision and recall between SPRITE and eSearch when the number of answers varies. As shown in Figure 4(a), the eSearch system outperforms SPRITE when the number of answers is small (5-10); but SPRITE gives better performance when the number of answers is larger (15-30). Both eSearch and SPRITE are not as good as the centralized system, which is the price for indexing 20 terms only. Some relevant documents are missed due to some unindexed terms. We also observe that SPRITE's precision of 89% and recall of 87% are relatively constant with respect to the centralized scheme. The eSearch system degrades much faster when the number of answers is larger. The terms indexed in SPRITE are more representative for the documents because it is able to learn from past queries. Therefore, SPRITE can perform constantly well when the number of answers increases; the most frequent terms indexed in eSearch can only benefit a small fraction of documents in the collection.

Next, we vary the number of terms indexed. Figure 4(b) shows the results of two sets of queries: "w/o-r" (without repeats), where every query appears exactly once and "w-zipf" (with Zipfian distribution, whose slope is set to 0.5), where the frequency of a query is roughly inversely proportional to the popularity of the query. The "w/o-r" query set is an extreme case that is biased against SPRITE. Most queries are repeated as we mentioned previously and the phenomenon is shown in [19] and [14]. SRPITE can obtain the least knowledge from the past queries in this case. Note that when 5 terms are initially indexed, no learning process is involved, so the two systems have the same performance. First, we observe that SPRITE outperforms eSearch with the same number of terms indexed. In fact, the gain over eSearch is larger with fewer terms indexed (except when



(a) Varying number of answers. (b) Varying number of index terms. (c) Change on query pattern.

Figure 4. Comparison between SPRITE and basic eSearch on Effectiveness

the number of terms is 5). Second, SPRITE can achieve similar performance as eSearch with fewer terms. For example, the performance of SPRITE with only 20 terms indexed is nearly the same as that of eSearch with 30 terms indexed. This is very important and useful in a P2P system since indexing fewer terms means lower cost for inserting the global index terms initially as well as for maintaining the index subsequently. This also suggests that many frequent terms indexed by eSearch do not contribute to answering queries. Instead, SPRITE successfully removed these redundant terms. Lastly, under the circumstance that either queries do not even repeat ("w/o-r") or queries are issued in a very skewed distribution ("w-zipf"), SPRITE always outperforms eSearch. SPRITE can sufficiently learn the key meanings of a document from similar queries or identical queries. We observe similar trend for recalls and do not present the results due to the space limitation.

Finally, we study SPRITE's robustness to changes in query access patterns, e.g., users may be interested in one collection of documents in a period and then in another collection later. Figure 4(c) depicts the precision and recall when query pattern changes. The query set is evenly partitioned into two groups such that all new queries and their corresponding original query are in the same group. In the first 5 learning iterations, queries in one group are processed and evaluated. In the next 5 iterations, the other group of queries are processed and evaluated. Thus, in the first 5 iterations, none of the queries in the second group is known to the system. In this set of experiments, we set the maximum number of terms to index to 30, after which the number of indexed terms remains unchanged. Instead, we apply term replacement (as described in Algorithm 1) only. This is also the reason the performance of eSearch remain unchanged after iteration 6. SPRITE always outperforms eSearch as usual when the number of indexed terms increases in the first 5 iterations. From the 6th iteration, new queries are issued in the system. As can be seen from the results, SPRITE adapts to the changes very quickly. The precision and recall decrease a little bit at the beginning of the new queries arrival, but are still better than those of eSearch. After just one iteration, SPRITE recovers and gives good performance in a stable status. The reasons are twofold: The first 5 iterations mainly polish the indices of related documents based on the first group of queries. They have very little effect on the later queries and their relevant documents. When the new queries are issued (in the 6th iteration), the terms indexed (based on the first group of queries) are unable to provide adequate relevant documents. However, SPRITE's learning capability ensures that the indices are carefully tuned to meet the new set of queries in the following iterations.

7. Discussion and future work

SPRITE offers an effective learning mechanism for supporting text retrieval in DHT-based P2P networks. In this section, we discuss how some other features, many of these are orthogonal to our work, can be easily incorporated into SPRITE to further enhance its capabilities and robustness.

First, peers can join and leave the network when some queries are being processed. There are two methods to prevent query failure from peer failure. If a peer responsible for a term is "down" and a query containing the term is issued during this period, then the term can be discarded when calculating the ranked list in the querying peer. However, such a query will not be answered accurately if the term happens to be dominant. A second approach to reduce the "damage" from peer failure is replication. In SPRITE, we can replicate the indexes of a peer in its successor peers periodically. With these two schemes, peer failure will have little impact in SPRITE. This is because the probability is very small that the index of a document containing a dominant query term is not replicated when the query is issued and the original

peer responsible for the term fails, even if the index replicating period is long. In fact, SPRITE has the additional advantage that only a small number of terms are replicated.

Second, it is possible for a peer to become overloaded. There are two scenarios of unbalanced load in SPRITE. (a) An indexing peer indexes some popular terms (i.e., terms that appear in many documents), which means many owner peers will frequently poll the indexing peer to maintain the index and/or check if the peer is still active. As such, the indexing peer will be busy with these requests. We note that a popular term has a very high document frequency, which leads to a small IDF. Therefore, the term will contribute little in the similarity calculation. Thus, a simple solution is to advise the document owner peers that the term has a high document frequency. The document owner peers can then discard the term and pick an analogously important term to index. The overhead is very small since it only requires one communication. (b) During query processing, popular terms/documents are queried by many users. As such, the indexing and owner peers may become a bottleneck. Some techniques such as *LAR* [6] and *Range-partition* [3] can be easily adapted. Popular documents can be replicated in other peers to reduce the load of the owner peer. A hot term can be cached in peers responsible for the terms that always appear with it in some queries as in *LAR*. When routing/processing a query, the indexing peer also needs to check the cached indexes. If an entry is found in the cache, then the peer responsible for the hot term will not be contacted. If a peer is responsible for indexing many terms, then it can invite an underloaded peer to share the range it is responsible for as in *Range-partition*. The invited peer passes over its original partition to its successor and shares a range with the overloaded peer.

Finally, a common technique used in distributed information retrieval is query expansion where extra terms are added to a query. In [11], various query expansion methods for distributed information retrieval are discussed. Since cooperation among peers is not as close as in a distributed system (with a small number of servers), local context analysis technique can be employed in SPRITE. In local context analysis, global information is not required. Nouns are extracted and the co-occurrence of nouns in a document is analyzed. Queries are enriched accordingly.

8. Conclusion

This paper presents the design and evaluation of SPRITE, a P2P keyword search system, with the following features. First, only a small number of selected terms of a document are indexed. This is extremely important in a P2P system, not only for index construction and update, but also because periodic checking on distributed indexes is required. Second, SPRITE uses progressive learning to refine the set of selected index terms. Our extensive simulation study showed that SPRITE can achieve performance similar to a centralized system in terms of precision and recall, and considerably outperforms a static index term selection approach.

Acknowledgement: Kian-Lee Tan and Yingguang Li are partially supported by a university research grant R-252-000-237-112. H.V. Jagadish is supported in part by NSF grant IIS-0219513.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *SODA'03*, 2003.
[2] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS'02*, July, 2002.
[3] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB'04*, 2004.
[4] O. D. Gnawali. A keyword-set search system for peer-to-peer networks.pdf. In *Master thesis. Massachusetts Institute of Technology*, 2002.
[5] Gnutella Development Home Page. http://gnutella.wego.com/.
[6] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *ICDCS'04*, 2004.
[7] W. Hersh, C. Buckley, T. Leone, and D. Hickam. Ohsumed: An interactive retrieval evaluation and new large test collection for research. In *SIGIR'94*, 1994.
[8] D. L. Lee, H. Chuang, and K. Seamons. Document ranking and the vector-space model. *IEEE Software*, 1997.
[9] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS*, 2003.
[10] J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *CIKM'03*, 2003.
[11] P. Ogilvie and J. Callan. The effectiveness of query expansion for distributed information retrieval. In *CIKM'01*, 2001.
[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM'01*, 2001.
[13] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the International Middleware Conference*, June, 2003.
[14] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large alta vista query log. In *Digital System Research Center, Technical Report 1998-014*, Oct, 1998.
[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, 2001.
[16] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *NSDI'04*, 2004.
[17] C. Tang, S. Dwarkadas, and Z. Xu. On scaling latent semantic indexing for large peer-to-peer systems. In *SIGIR'04*, 2004.
[18] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM'03*, 2003.
[19] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *InfoComm'02*, July, 2002.