High Search Performance, Small Document Index: P2P Search Can Have Both

Yingwu Zhu
Department of Computer Science & Software Engineering
Seattle University
Seattle, USA
Email: zhuy@seattleu.edu

Haiying Shen
Department of ECE
Clemson University
Clemson, USA
Email: shenh@clemson.edu

Abstract

One primary goal in P2P networks is to provide high search performance for users to retrieve interested documents distributed over nodes. Document indexing is the key to search performance. However, it is challenging to guarantee high search performance with small document index. In this paper, we present iSearch which aims to build small document index to deliver high search performance on Gnutella-like P2P networks. The number of index terms per document is typically 4, which dramatically reduces associated cost in index storage and dissemination. iSearch explores two options to build index: top term-based indexing (TTI) and query-driven indexing (QDI). TTI bases selection of document index terms on term statistics, while QDI progressively refines document index by past queries. Our simulations show that that TTI and QDI improve search performance over random walk significantly. By dynamically adapting index based on past queries, QDI outperforms TTI greatly, by up to $2\times$ *recall improvement.*

1. Introduction

Search techniques proposed for Gnutella-like P2P systems fall into two main categories: *blind search* (without index assisting) and *index-assisted search*. Blind search includes query flooding and random walk [1]. Query flooding, while simple and failure-resilient, is unscalable by flooding queries over the overlay links. Random walk, blindly forwarding queries to randomly-chosen neighbors for answers at each step, could incur high search latency. By contrast, index-assisted search [2], [3] builds a distributed index that directs queries to the nodes which are likely to have answers for the queries, thereby enhancing search performance.

Key to index-assisted search is maintaining "good" index among nodes. Good index should be informative yet compact, providing instructive guidance on query routing while incurring small storage cost on each node and low bandwidth cost in index distribution and maintenance. This paper argues that P2P search can guarantee high performance with a really small document index, and presents iSearch that explores options for building such a small document index.

1.1. Motivation

Our work is motivated by the following observations. First, many information retrieval (IR) systems retrieve relevant documents for queries based on vector space model (VSM), where documents and queries are represented as term vectors (each term is associated with a term weight) and their relevance scores are computed using an inner product. Equation 1 calculates the relevance score for a document D and a query Q, where t is a term appearing in both D and Q, d_t is t's weight in D, and q_t is t's weight in Q. Documents with high relevance score are deemed to be relevant to the query. By VSM, terms with heavy weight (which we call *top terms* in the rest of the paper) in the document dominate the relevance score while terms with light weight contribute marginally to the relevance score. Therefore, it is wise to use top terms as document indices.

$$REL(D,Q) = \sum_{t \in D,Q} d_t \cdot q_t \quad (1)$$

However, it is costly to use all top terms in a document as its index terms since the number of top terms is not presumably small. For example, in a set of $80,008$ TREC-1, 2 AP documents [4], we extracted a term vector for each document using VSM, sorted the terms in decreasing weight order, and calculated the relative weight of each term to the biggest term weight in the document. We then averaged this normalized term weight across all documents for each term rank, and plotted the mean for each term rank in Figure 1. The y-axis is in log scale. The weight of top 50 ranked terms drops very fast and the curve for the rest of terms is "flat", showing that a set of top terms (i.e., 50) with relatively heavy weight, are central to a document. However, even taking this set of top terms as document indices still incurs nontrivial index size per document [1].

Second, user queries are generally short, mostly containing 2-3 unique terms [5]. From perspective of user queries, a document index does not need to include all its top terms. Ideally, those top terms not appearing in queries should be excluded from document indices, which

1. The average document term vector size for the TREC document collection is 179 after all words are stemmed and stop words are removed.

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

will not compromise search results but substantially reduces associated costs. Figure 2 shows CDF of a set of 50 TREC-3 ad hoc queries with respect to fraction of query terms that appear in top K ranked terms of their relevant documents in the TREC-1, 2 AP document corpus. Note that query terms are *dispersed* among top 50 ranked terms of their relevant documents. This indicates that simply choosing several top terms of a document as its indices is insufficient.

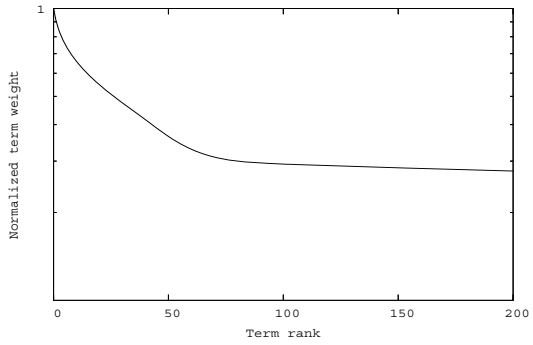


Figure 1. Ranked term weight for the TREC documents, normalized to the biggest term weight in each document.

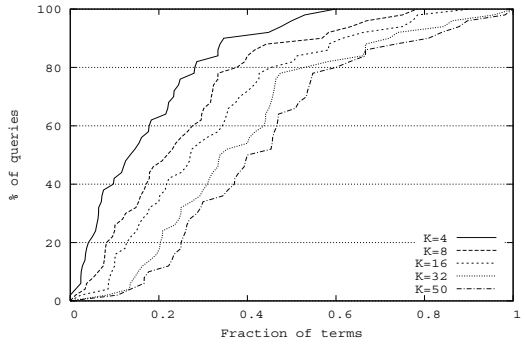


Figure 2. Consider a query Q and its relevant document set $R = \{D_1, \cdots, D_m\}$. The number of query terms intersecting with D_i's top K ranked terms is S_i ($0 \leq S_i \leq |Q|$), then the expected fraction of query terms lies in top K ranked terms of Q's relevant documents is $Frac = \frac{1}{m} \sum_{i=1}^{m} \frac{S_i}{|Q|}$. If $Frac$ is larger, indexing a document by top ranking terms is more effective. Otherwise, it is not so effective as expected.

Third, user queries have significant locality: 30-40% queries are repeated queries and query repetition frequency follows a Zipf distribution [6]. Also, most users submit queries consisting of terms from a small lexicon, resulting in similar queries or common terms in their queries. Finally, like-minded groups of users on the Internet search for the same sorts of things [7]. Users with common interests could submit same or similar queries.

To summarize, the first observation shows that top terms (top ranked by term weight) are good index term candidates for documents. The second observation suggest that it be overkill to index a document by using a complete list of its top terms. However, a small index composed of several "pre-determined" top terms does not suffice to characterize a document for different queries as query terms are indeed dispersed among the full top term lists of the relevant documents pertinent to the queries. For different queries, the set of index terms in a document should be different. That is, index terms of a document should be dynamically adjusted for different queries, if it is not mission impossible. The last two observations imply that past queries can assist in document indexing due to query locality and similarity. Exploiting past queries opens the door to dynamically adjusting index terms.

Based on the above observations, we propose iSearch which exploits both term statistics and query locality to build document index. iSearch aims to deliver high search performance while using a very small number of dynamically-chosen top terms as a document index, e.g., 4 representative terms.

In particular, we make the following contributions:

- We explore two options for building document indices: top term-based indexing (TTI) and query-driven indexing (QDI). We point out that a very small set of top terms are capable of indexing a document without compromising search results.
- We design iSearch which uses past queries to progressively shape document indices and exploits attenuated bloom filters [8] to maintain distributed indices at each node. iSearch w/ QDI is responsive to change of query locality by dynamically adapting document indices.
- We evaluate the performance of iSearch using TTI and QDI respectively, and compare their performance against random walk and SETS [11]. Our simulations show that both TTI and QDI dramatically improve search performance over random walk. QDI outperforms SETS when searching a small fraction of nodes. Moreover, QDI significantly outperforms TTI, by up to $2\times$ recall improvement.

1.2. Caching vs. iSearch

Caching has been shown effective to Zipf distributed queries. We argue that caching is orthogonal to iSearch. Caching alone has three main limitations which can be addressed by iSearch: (1) Search systems exploiting cached query results provide no guarantee of data freshness and high recall. (2) As the number of objects exponentially increases and the object size increasingly grows, caching needs to address a number of issues such as cache storage, cache placement, cache replacement, and so forth. (3) Although

caching is effective to repeated queries, it remains open whether caching is cost-effective for similar queries.

Roadmap. The remainder of the paper is structured as follows. Section 2 presents design of iSearch. Section 3 describes experimental setup and provides experimental results. Section 4 gives a brief review of related work. We conclude the paper in Section 5.

2. Design

2.1. Design Goals

iSearch has the following design goals:

- **Retaining the overlay structure.** iSearch does not alter the simple yet robust overlay structure of Gnutella-like P2P networks, and relies on continuously refined distributed indexes to improve search performance.
- **Small index size.** iSearch uses a very small number of index terms (i.e., about 4-5 terms) per document without compromising search results while reducing index storage and maintenance cost.
- **High search performance.** iSearch provides high recall for queries while contacting a relatively small percentage of nodes per query processing.

For the sake of clarity, in the paper we use *top terms* to represent top ranked terms ordered by term weight in a document. In other words, top terms in a document are those terms whose weight (assigned by VSM) is over some threshold value, and they are central to the document.

2.2. Choosing Document Index Terms

Central to iSearch is to determine a small set of index terms for each document. iSearch explores two options: top term-based indexing (TTI) and query-driven indexing (QDI). In practice, iSearch starts with TTI, and then operates with QDI which progressively refines document indices by past queries. In what follows, we discuss these two options in turn.

For each document D on peer P, P uses VSM to extract a term vector where each term is associated with a weight. Term t's weight is assigned by using "dampened" tf scheme [9] in the form of $d_t = 1+\log f_t$, where f_t is t's term frequency in D. The main advantage for the "dampened" tf term weighting is that it does not require global term statistics, which is important especially for dynamic P2P environments.

TTI bases choice of index terms solely on term statistics. By TTI, P sorts D's term vector in decreasing weight order and chooses top K ranked terms (or top K terms) as D's index terms. K is very small, typically of about 4-5. However, as discussed earlier, simply choosing several top terms as document indices is not optimal as query terms are dispersed among the top term lists of their relevant documents. On the other hand, using the full top term list of a document as index terms is inefficient, as a large index size increases index storage and distribution cost.

The selection of index terms by QDI takes into account both term statistics and past queries. To work with QDI, each peer P maintains a *query cache* which stores past queries. Each cache entry includes query ID, query term vector, and timestamp. Query ID is generated by hashing the set of query terms sorted in alphabetical order. Most user queries consist of 2-3 terms, and thus the cache entry is expected to cost modest storage. P learns past queries in two ways: (1) When P sees a new query Q during query routing, P caches Q; (2) P exchanges with its neighbor nodes periodically for past queries by gossip messages. The query cache is actually split into two partitions: *Learned Partition* (LP) and *To-be-learned Partition* (TP). TP holds queries that have not been used for document index shaping while LP contains those that have been used for document index shaping. Past queries are evicted from LP based on their timestamps.

Each peer independently and periodically shapes document indices using queries in TP. After shaping document indices, queries in TP will be moved into LP which may cause query eviction. The basic idea behind QDI is that it *re-ranks* document terms based on queries in TP and selects the resulting top K ranked terms as new document index terms.

In order to re-rank terms for a document D, QDI needs to calculate a score for each term $t \in D$ using queries in TP. In fact, only top X terms (ranked by term weight) are considered in the score calculation for D. X, on one extreme point, can be the number of unique terms in D. QDI uses a term weight threshold to exclude those terms with light weight from score calculation and re-ranking since these terms contribute marginally to relevance score as shown in Equation 1. Excluding trivial terms from the score calculation is beneficial: (1) Reducing calculation cost, and (2) Minimizing interference of trivial terms in document indices. For example, if a trivial term $t \in D$ appears in many queries, the resulting score for t may be high enough to be chosen as D's index term. Based on Equation 1, if no top terms of D are in these queries, D is unlikely to be relevant to these queries due to low relevance score. So, it is unwise to use t as D's index term.

Assume the set of queries in TP is $\psi = \{Q_1, \cdots, Q_m\}$. The score for term $t \in D$ is computed by

$$Score(t, \psi) = max(Sim_t(D, Q_i)) \cdot \log QF(t, \psi) \qquad (2)$$

Where $QF(t, \psi)$ is t's query frequency representing number of queries in ψ that contain t, and $Sim_t(D, Q_i)$ measures similarity of D to Q_i with respect to t.

$$Sim_t(D, Q_i) = \begin{cases} \frac{|D \cap Q_i|}{|Q_i|} & if \quad t \in Q_i \\ 0 & otherwise \end{cases}$$

Intuitively, if a large fraction of query Q_i's terms fall in D's top term list (which contains X terms with heaviest weight), then D is very likely to be relevant to Q_i according to Equation 1. Consequently, the top term t, if appearing in Q_i, is a good index term candidate for D. This accounts for the first part of Equation 2's right component. Moreover, if many queries contain a top term t, then t is a good index term candidate for D. This accounts for the second part of Equation 2's right component, though we use $\log$ to avoid overemphasizing impact of t's query frequency.

A nice property of Equation 2 is that the score calculation is *accumulative*. That is, we do not need to keep and go through the whole history of past queries to update a term's score upon every new set of queries in TP. This is two-fold: (1) Queries in LP can be freely evicted, and (2) A term's score is continuously updated upon a new set of queries in TP. Thus, QDI works in an iteration-based manner: A new iteration starts with an empty TP. Then, the peer populates its TP using queries learned from query routing and gossip messages. At the end of the iteration, QDI is performed to update term scores for each document and then queries in TP are demoted to LP. Note that the iterations across the nodes are unnecessarily synchronized. Each peer can independently trigger QDI process.

To assist in updating term scores, each peer needs to maintain per-term data structures for each local document D, which includes $HistSim(t \in D) = max(Sim_t(D, Q_i))$ (the maximum value of similarity of D to queries in past iterations, with initial value of zero) and $HistQF(t \in D)$ (the query frequency in past iterations, with initial value of zero). Each peer also maintains a per-document data structure called *Term Score Ranking List* (TSRL) which is initially empty. TSRL contains tuple $< t, HistQF(t \in D), HistSim(t \in D), score >$, sorted in decreasing order of $score$. Algorithm 1 shows term score update for document D at peer P upon a new set ψ of recent queries. QDI differs from TTI in that it re-ranks a document's top terms by their scores instead of term weight and chooses the top K ranked terms with highest scores in its RTSL as document index terms.

Algorithm 1 P.updateScore(D, ψ)

```
1: R← D's TSRL
2: L← D's top term list by pruning those terms with light weight
3: for each t∈L do
4:    qf = (t∈R) ? HistQF(t ∈ D) : 0
5:    sim = (t∈R) ? HistSim(t ∈ D) : 0
6:    for each Q_i∈ψ do
7:       if t ∈ Q_i then
8:          qf++
9:          if t ∉ R then
10:            sim = Sim_t(D, Q_i)
11:            score = sim × log qf
12:            insert < t, sim, qf, score > into R
13:         else
14:            if Sim_t(D, Q_i) > sim then
15:               sim = Sim_t(D, Q_i)
16:            end if
17:            score = sim × log qf
18:            update t's tuple in R with < t, sim, qf, score >
19:         end if
20:      end if
21:   end for
22: end for
23: sort R in decreasing order of score
24: D's TSRL ← R
```

2.3. Building Distributed Document Index

iSearch distributes document indices among peers using attenuated bloom filters (ABF). Each peer P maintains an ABF for each of its neighbor nodes. An ABF with depth d is a multi-level of d identically-sized bloom filters where the i-th level bloom filter holds document index information at nodes i hops away from P. Consider an ABF associated with P's neighbor node R. The 1-st level bloom filter contains the document index terms at R, which is one hop way along the neighbor link $P \rightarrow R$. The 2-nd level bloom filter summarizes the document index terms at nodes two hops away along the neighbor link $P \rightarrow R$ (i.e., R's neighbor nodes excluding P). The 3-rd level bloom filter aggregates the document index terms at nodes three hops away along the neighbor link $P \rightarrow R$, and so on.

Now we present how peers collectively distribute document indices using ABFs. We here use QDI to choose document index terms. As described in Algorithm 2, each peer P generates a bloom filter based on its local documents, and sends the bloom filter to all its neighbors with level parameter of 1. Each peer receives from all its neighbor nodes messages each containing a bloom filter and level value, and stores the received bloom filters on the corresponding ABFs (lines 1-4 in Algorithm 3). If the level value l is less than the depth of ABFs, the peer merges corresponding bloom filters to produce a new bloom filter and sends the new bloom filter to its corresponding neighbor node with level value of $l+1$ (lines 8-17).

Algorithm 2 P.genAndSendLocalBF(int K)

```
1: BloomFilter bf ← an initialized bloom filter with all bits set to zero
2: for each document D ∈ P do
3:    X ← top K ranked terms in D's TSRL
4:    for each term t ∈ X do
5:       bf.insert(t)   // insert t into the bloom filter
6:    end for
7: end for
8: level = 1
9: Message M = {bf|level}   // | is a concatenation operator
10: for each neighbor N_i do
11:    send M to N_i
12: end for
```

To summarize, the process of building the distributed document index is essentially an outward propagation of a local bloom filter to nodes within a radius of d hops from the source peer of the bloom filter. An ABF is an aggregate of such bloom filters from different peers. As higher-level bloom filters in ABFs are constructed from union of lower-level bloom filters, the false positive rate increases at high-level bloom filters. As a result, lower-level bloom filters have a narrower but more accurate view of document indices

Algorithm 3 P.createABFs(int $level$)

Require: d is depth of ABFs
Require: $\{N_1, \cdots, N_m\}$ are P's neighbors
for $i = 1$ to m **do**
  receive $M = \{bf|level\}$ from N_i
  $ABF_{N_i}[level] = bf$ *// store bf for N_i at this level of ABF*
end for
if $level \geq d$ **then**
  return
end if
for $i = 1$ to m **do**
  BloomFilter $bf \leftarrow$ initialized with all bits set to zero
  for $j = 1$ to m **do**
    if $i \neq j$ **then**
      $bf = bf \cup ABF_{N_j}[level]$ *// $\cup$ is a merge operator for bloom filters*
    end if
  end for
  Message $M = \{bf|level + 1\}$
  send M to N_i
end for

nearby while higher-level bloom filters have a broader but rougher view of document indices at remote nodes.

As a final point, the small index size achieved by iSearch is beneficial when coupled with ABFs: Fewer index terms per document decrease either bloom filter size (i.e., thus reducing index storage and distribution cost) or the false positive rate in ABFs (i.e., lowering the probability of forwarding a query toward a "deceptive" direction during query routing, and thus improving search performance).

2.4. Using Document Index to Route Queries

As mentioned earlier, each peer associates an ABF with each of its neighbor links. Upon receiving a query, the peer first evaluates the query against its local documents. If any documents are deemed to be relevant to the query (according to some relevance threshold), the peer forwards the responses to the querying node. If the TTL of the query message is above zero, the peer chooses a neighbor as the query's next hop.

The next hop for the query is chosen by examining a set of ABFs level-by-level on those neighbors to which the peer has not forwarded the query before. The peer checks the first level of the ABFs. If a bloom filter covers all the query terms, it is likely that the desired documents are one hop away and the query is forwarded to the corresponding neighbor. If no bloom filter matches the query terms, the peer looks for a match in the 2nd-level of each ABF. As before, if a match is found, the query is forwarded to the corresponding neighbor. Otherwise, the peer examines the next levels of the ABFs. If no matching neighbor is found after checking all the levels of the set of ABFs, the peer selects a neighbor whose first-level bloom filter has the highest hit ratio of the query terms, as the next hop.

During query forwarding, bookkeeping techniques are used to sidestep redundant paths. Each query is assigned with a globally unique identifier GUID by its initiator node, and each peer keeps track of the neighbors to which it has forwarded the query with the same GUID. The next hop selection algorithm disregards the neighbors to which the query has been forwarded before. If a peer has already sent the query to all its neighbors, to ensure forward progress, the peer flushes the bookkeeping state for this query and forwards the query to a randomly-chosen neighbor.

2.5. Maintaining Distributed Document Index

The process of maintaining document indices is similar to that of building document indices, except that it is performed in a bandwidth-efficient manner. The purpose of maintaining document indices is to keep distributed document indices up-to-date.

Consider ABFs with depth of d. When a peer updates index terms of its local documents, the peer generates a new local bloom filter, calculates bit differences from its old local bloom filter, and propagates the bit differences in a form of *diff compression* outwards to every node within d hops from itself. In addition to the ABFs associated with each outgoing neighbor link, each peer also keeps a copy of ABFs that are maintained by each of its neighbors in *reverse direction* of the neighbor links. When document index terms are changed, the peer determines the changed bits in its local bloom filter and the copy of ABFs in reverse directions of its neighbor links. It then sends these bits out to each neighbor in a form of diff compression. Upon receiving such a message, each neighbor finalizes the bit changes in its corresponding ABF and computes the bit changes to be made in each of its own neighbors' ABFs. The bit changes are propagated as well. This process repeats until the bit changes reach nodes d hops away from the source of the index update [8].

3. Evaluation

3.1. Experimental Setup

The document set used in simulations is from TREC-1,2-AP which contains AP Newswire documents in TREC CDs 1 and 2. We extracted the documents with text and valid author fields from the document set, resulting in $80,008$ documents and $1,880$ authors. Assuming that each author corresponds to a node and her associated documents are stored on the corresponding node, we mapped the $1,880$ authors to a random graph of $1,880$ nodes generated by GT-ITM [10], and distributed their documents to the corresponding nodes. The average node degree is 8 and the minimum node degree is 4. The mean, 1st-percentile, and 99th-percentile of the number of documents per node are 42.5, 1, and 417, respectively. The document term vector was derived from the text field using VSM. The terms in the document vectors were stemmed and stop words were removed. The document term vector on average has 179 unique terms.

We used a set of 50 queries form TREC-3 ad hoc topics with query number form 151 to 200. The query vector was derived from the title field using VSM. The terms were stemmed and stop words were removed as well. The query term vector on average has 3.5 unique terms. The 50 queries each comes with a query relevant judgment file which contains a set of already identified relevant documents by TREC-3 ad hoc query assessors. Since we only used $80,008$ documents with valid author and text fields, we removed the other documents from the accompanying relevant judgment files.

As noted by Xie et al. [6], users queries have significant locality and query repetition frequency follows a Zipf distribution. Based on the original 50 TREC-3 queries, we generated various sets of queries where the frequency of the i-th popular query is proportional to $\frac{1}{i^{\alpha}}$. Each set contains 492 queries.

Moreover, as mentioned earlier, users with common interests could submit similar queries [7]. Thus, from the original 50 TREC-3 queries we generated a new set of similar queries. The generation of similar queries follows two principles: (1) Similar queries share some common terms, and thus share some common relevant documents. (2) The term popularity distribution and relevant document distribution in the new query set are consistent with those in the original query set. That is, popular query terms in the original query set should appear frequently in the new set, and if an original query has many relevant documents, the new similar queries derived from it should also have many relevant documents.

For an original query $Q = \{t_1, \cdots, t_m\}$ consisting of m unique terms, a derived similar query $Q' = \{t'_1, \cdots, t'_m\}$ shares some common terms, determined by $C = \frac{|Q \cap Q'|}{|Q|}$. C controls similarity of the derived query to the original query and is 0.7 in our simulations. The shared common query terms for Q' is randomly chosen from Q.

The rest of query terms for Q' is selected as follows. Let $Z = Q - Q'$. We need to find a replacement term from the document set for each term in Z and these replacement terms constitute the rest of query terms for Q'. Term replacement is based on term importance — that is, the replacement term is "equally" or "similarly" important as the original term in the document collection. The term importance is computed by $I(t) = Freq(t) \cdot DF(t)$, where $Freq(t)$ denotes the t's term frequency in the document set and $DF(t)$ represents the number of documents containing t. Let the term importance difference for two terms t_1 and t_2 be $Diff(t_1, t_2) = |I(t_1) - I(t_2)|$. The less the difference is, the more similar they are. For each term $t \in Z$, the replacement term is randomly chosen from a set of 5 terms which have the least importance difference to t. For each original TREC-3 query, we generated 9 similar queries. Consequently, the number of synthetic queries is 500, including 50 original queries and 450 new queries.

Now turn our attention to how to determine relevant documents for each derived query. Intuitively, the derived query should share some relevant documents with the original query and have some different relevant documents. For each original query Q, we calculated document relevance to the query using Equation 1 for all documents and put the top $1,000$ ranked documents (by relevance score) into its rank list RL_Q. Similarly, we calculated the rank list $RL_{Q'}$ for each of its derived queries, say, Q'. For each document $D_i \in RL_{Q'}$, if D_i is a relevant document to Q, then D_i is also a relevant document to Q'. For each relevant document $D_j \in RL_Q$ (assume D_j's rank position is m in RL_Q), if $Dj \notin RL_{Q'}$, then the document in $RL_{Q'}$ with rank position of m is deemed to be relevant to the derived query.

Our simulations study two indexing schemes: TTI and QDI. iSearch simulator can be configured to use either of the two schemes. The default number of index terms per document is 4 unless otherwise specified. The default depth of ABFs is 3, which has been shown the best fit in our simulations. The bloom filter size is 1KB.We used random walk as the baseline system, comparing against iSearch. It is worth pointing out that using random graphs as the overlay topology in our experiements is to avoid bias against random walk in performance comparison. In addition, we present a comparison with SETS [11]. SETS is a search system using a topic-driven query routing protocol on a topic-segmented overlay built from Gnutella-like P2P systems. A topic segment in SETS contains nodes with similar content. The topic-segmented overlay is constructed by performing node clustering at a single designated node, and each cluster corresponds to a topic segment (nodes within a topic segment are connected by local links while nodes belonging to different topic segments are connected by long-distance links). Given a query, SETS first computes R topic segments which are most relevant to the query and then routes the query to these segments for relevant documents.

Two main performance metrics are used in our experiments. (1) *Recall*. It is used to quantify quality of search results, and is defined as the number of retrieved relevant documents divided by the total number of relevant documents. (2) *Query processing cost*. It is defined as the fraction of nodes which are involved in a query processing. A low query processing cost increases system scalability since system resource consumption is proportional to the number of nodes visited by a query.

3.2. Experimental Results

We summarize our results before presenting the details: (1) Both TTI and QDI outperform random walk significantly. In addition, QDI gains higher recall than SETS when searching a small percentage of nodes. (2) With the presence of query locality and similar queries, QDI improves recall dramatically by progressively refining document indices,

with up to 2× recall improvement over TTI. (3) iSearch uses a very small set of index terms per document (typically of 4), without impairing search performance while reducing index storage and maintenance cost. (4) QDI is responsive to change of query locality. As query locality changes, iSearch may dilute impact of past queries and weigh more on recent queries to better determine index terms, thereby improving search performance.

3.2.1. Impact of Query Locality. In the first set of experiments, we used three sets of Zipf distributed queries with different values of α, i.e., 0.3, 0.7, and 1.0. Each set contains 492 queries. As shown in Figure 2, query terms are dispersed in top terms of their relevant documents, and are unnecessarily on top K ranked terms. We intentionally made queries which have less query terms intersecting with top 4 ranked terms of their relevant documents more frequent in Zipf distribution. This is biased against TTI since TTI always chooses top K ranked terms as index terms for a document. We randomly split each set of queries into two equally-sized groups: The first group of queries are used for index training, and the results are collected based on the second group. Figure 3 shows results for $\alpha = 0.3, 0.7$, and 1.0 respectively. Note that the number of index terms per document is 4. Three main observations can be made: (1) iSearch outperforms random walk significantly; (2) iSearch w/ QDI outperforms iSearch w/ TTI, by up to 2× recall improvement when visiting less than 50% nodes. (3) Different Zipf distributions have little impact on iSearch's performance. This makes a distinction between iSearch and caching, as caching is more effective as query locality increases (i.e., as α increases).

Figure 4 shows comparison with SETS. We observed similar characteristics with different α values. SETS outperforms TTI due to its node clustering by which queries can be efficiently forwarded to relevant node groups for answers; QDI greatly outperforms SETS when 50% of nodes or less are visited for a query. This is because QDI's precise document indices can guide queries toward the most relevant nodes for answers. In the rest of the paper, we focus our evaluation on iSearch with different workloads.

The second set of experiments study the impact of number of index terms per document. Figure 5 shows results for $\alpha = 0.7$. We observed similar characteristics for α=0.3 and 1.0. The x-axis denotes number of indexing terms per document. The number in parenthesis represents the percentage of nodes contacted by iSearch for a query processing. The figure shows two representative cases: 10% and 50%, Contacting less than 50% nodes shows similar characteristics as 10% while contacting more than 50% nodes shows similar characteristics as 50%. Two main observations can be made: (1) The number of index terms per document has little impact on iSearch w/ QDI except that the number of index term is 3. This is two fold. First, the average number of

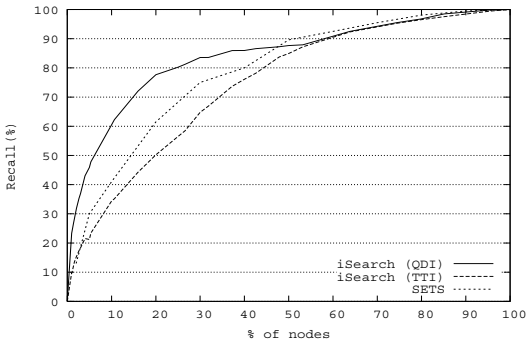


Figure 4. Performance comparison with SETS. The number of index terms per document in iSearch is 4 and α= 0.7

query terms is 3.5, so a number of 3 index terms per document is insufficient. Second, iSearch w/ QDI is capable of well defining index terms for documents as long as the number of index terms per document is sufficient (i.e., 4). In the remainder of the paper, we focus our experiments on the number of index terms of 4 per document. (2) When visiting a small number of nodes ($< 50\%$), iSearch w/TTI improves recall as the number of index terms increases. This is consistent with Figure 2 that query terms are dispersed among top term lists of the relevant documents, and more top terms as index terms are able to provide more hints for iSearch due to more complete indices. Simply choosing several top terms as document indices is insufficient. While visiting a sufficiently large number of nodes (50% or more), the number of index terms has little impact on iSearch w/ TTI. This is because the sufficiently large number of visited nodes compensates the incomplete document indices by fewer terms.

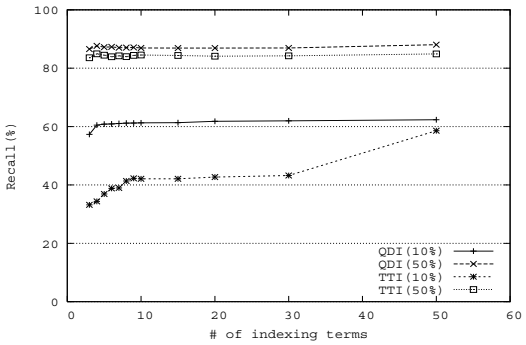
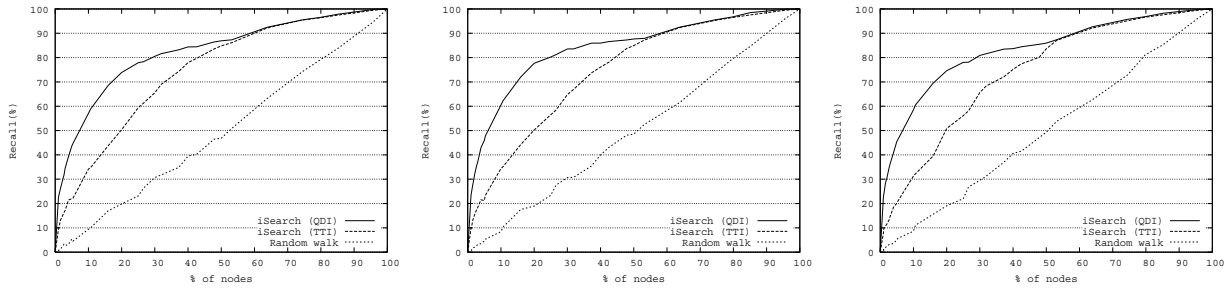


Figure 5. Recall yielded with respect to the number of index terms per document.

We also study Zipf distributions of 492 TREC-3 ad hoc queries with different query popularity from the aforementioned query sets. That is, the queries with more query terms in top 4 ranked terms of their relevant documents



(a) (b) (c)

Figure 3. Performance comparison between iSearch and random walk. The number of index terms per document in iSearch is 4. (a) α= 0.3. (b) α= 0.7. (c) α= 1.0

are more frequent. This favors iSearch w/ TTI. Again, the set of 492 queries are randomly split into two groups: The first group is used for shaping indices, and the results are collected based on the second group. Figure 6 shows results for α= 0.7. The results for other values of α are similar, and omitted here due to space constraints. When contacting 10% nodes or less, iSearch w/ QDI and iSearch w/ TTI have similar performance. This is because a large percentage of queries have their terms falling among top 4 ranked terms of their relevant documents. The top 4 ranked terms in many documents are very good indices. However, when visiting more than 10% nodes, iSearch w/ QDI dramatically outperforms iSearch w/ TTI. The is resulted from more accurate indices shaped by QDI.

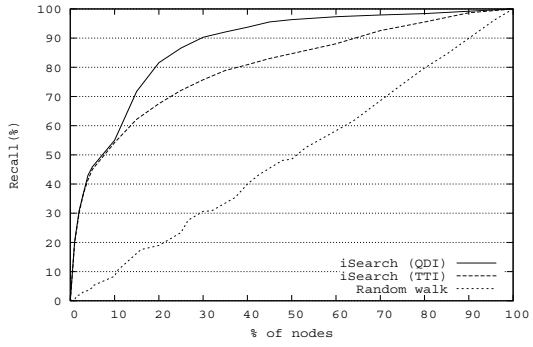


Figure 6. Performance comparison for α= 0.7 with respect to % of nodes visited by each query. The number of index terms per document is 4.

3.2.2. Impact of Similar Queries. In the first set of experiments for similar query workload, we randomly split the 500 queries into two groups: The first group is used for index shaping, and the results are collected based on the second group. Figure 7 shows the results. Two main observations are made: (1) iSearch w/ QDI and iSearch w/ TTI substantially improve recall over random walk. (2) iSearch w/ QDI outperforms iSearch w/ TTI dramatically, by up to 2× recall improvement.

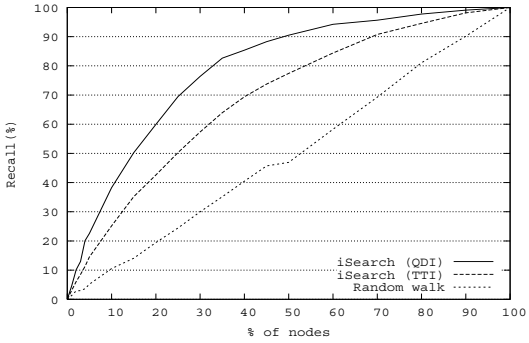


Figure 7. Performance comparison for similar queries with respect to % of nodes visited by each query. The number of index terms per document is 4.

The second set of experiments evaluate the performance of iSearch w/ QDI as query locality changes. We first randomly split the 50 original TREC-3 ad hoc queries into two identically-sized groups (G_1 and G_2), and then added their derived similar queries into same groups. This resulted in two equally-size groups each with 250 queries. As the original 50 TREC-3 queries are different from each other, we expect that the two groups G_1 and G_2 constitute different query localities. Each group was further randomly divided into two equally-sized sets, resulting in four equally-sized sets: G_{11}, G_{12}, G_{21}, and G_{22}, each with 125 queries. Then, the iSearch simulator ran four iterations, each fed with a set (in the order of $G_{11}, G_{12}, G_{21}, G_{22}$). iSearch starts with TTI for the first query set, and then works with QDI for the successive sets. All the past queries in previous iterations are used by QDI to refine document indices. Figure 8 shows recall achieved for various percentages of nodes visited by iSearch. Again, QDI outperforms TTI greatly (comparing recall at iteration 2 with iteration 1). More importantly, iSearch w/ QDI is responsive to change of query locality (e.g., the performance dip at iteration 3 is due to the fact that past queries in iterations 1 and 2 have different

query locality form those in iterations 3 and 4, providing some inaccurate or obsolete information about document indices), and improves recall significantly at iteration 4. This demonstrates iSearch's ability in adapting document indices by progressively exploiting past queries.

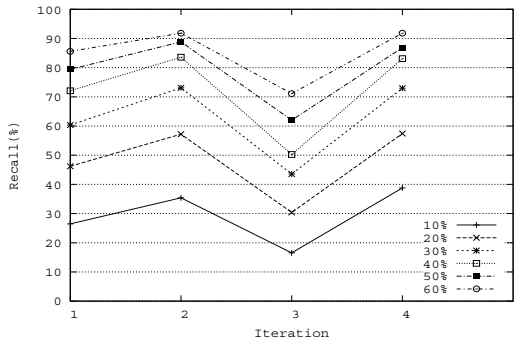


Figure 8. Performance of iSearch for four query sets. The number of index term per document is 4. The i-th data point represents the average recall yielded for the queries in i-th query set.

A question is naturally raised: Can we further improve the performance of iSearch w/ QDI by weighing more on recent queries and diluting effect of past queries in earlier iterations as query locality changes? We use exponentially moving average (EMA) to emphasize importance of more recent queries, by putting more weight (e.g., 0.9) on term scores calculated in the last iteration. Figure 9 shows EMA improves recall, especially when iSearch w/ QDI visits a small number of nodes for query processing. While EMA may not be the optimal solution, we believe that it is beneficial for iSearch w/ QDI to weigh more on more recent queries to adjust index terms as query locality changes.

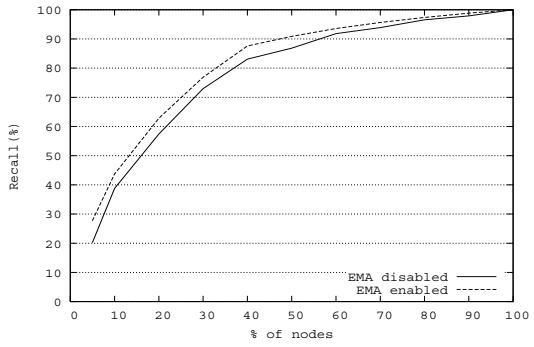


Figure 9. Performance impact of EMA on iSearch w/ QDI for the query set at the 4-th iteration with respect to percentage of nodes visited per query processing. The number of index term per document is 4.

4. Related Work

Search on Gnutella-like P2P networks can be categorized into two groups: *blind search* (without index assisting) and *index-assisted search*. Query flooding and random walk [1] fall into the former group. Index-assisted search is centered on building good document indices to improve search. Crespo et al. [2] proposed three document index schemes to assist search, including compound routing indices, hop-count routing indices and exponential routing indices. PlanetP [3] uses bloom filters to summarize content on each node and floods the bloom filters to the entire network. However, search performance enhancement comes at the price of storage cost and summary distribution bandwidth cost. iSearch borrows the concept of attenuated bloom filters from [8] to distribute document indices within certain radius, thus reducing cost in index storage and distribution bandwidth.

Systems such as [12], [11], [13] take a different approach by organizing nodes into groups to improve search performance. iSearch differs from such systems in that it retains the simple overlay structures of Gnutella-like P2P networks. In particular, GES [12] and SETS [11] use the concept of node vector to summarize the content on each node. The node vector is derived based on term statistics of the documents on a node. iSearch w/ TTI shares similar wisdom that top terms are good index term candidates and provide good hints for queries.

5. Conclusions

In this paper, we present iSearch which performs text retrievals on Gnutella-like P2P networks with high recall by using a very small index size per document. iSearch explores two options to build such document indices: TTI and QDI. We show that both TTI and QDI improve search performance over random walk significantly. Due to the presence of query locality and similar queries submitted by Internet users, as well as the fact that query terms are dispersed among top terms of their relevant documents, QDI builds more precise document indices, thereby outperforming TTI dramatically, by up to $2\times$ recall improvement. Our simulations also show that iSearch w/ QDI is very responsive to change of query locality, by quickly adapting document indices.

Acknowledgment

The authors would like to thank reviewers for their helpful comments. This research was supported in part by U.S. NSF grants CNS-0834592, CNS-0832109 and CNS-0917056.

References

[1] Q. Lv, P. Cao, and E. Cohen, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of 16th*

ACM Annual International Conference on Supercomputing (ICS), (New York, NY), pp. 84–95, June 2002.

[2] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, (Vienna, Austria), pp. 23–32, July 2002.

[3] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen, "PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities," in *Proceedings of 12th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, (Seattle, WA), June 2003.

[4] "Text retrieval conference (trec)." http://trec.nist.org.

[5] 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 *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS)*, (Berkeley, CA), pp. 207–215, Feb. 2003.

[6] Y. Xie and D. O'Hallaron, "Locality in search engine queries and its implications for caching," in *Proceedings of INFOCOM*, 2002.

[7] K. Patch, "Net scan finds like-minded users." http://www.trnmag.comi/ftories/2003/050703/Net_scan_finds_like-minded_users_050703.html.

[8] S. C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proceedings of IEEE INFOCOM*, vol. 3, (New York, NY), pp. 1248–1257, June 2002.

[9] H. Schutze and C. Silverstein, "A comparison of projections for efficient document clustering," in *Proceedings of ACM SIGIR*, (Philadelphia, PA), pp. 74–81, July 1997.

[10] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proceedings of IEEE INFOCOM*, vol. 2, (San Francisco, CA), pp. 594–602, IEEE, Mar. 1996.

[11] M. Bawa, G. Manku, and P. Raghavan, "SETS: Search enhanced by topic segmentation," in *Proceedings of The 26th Annual International ACM SIGIR Conference*, (Toronto, Canada), pp. 306–313, July 2003.

[12] Y. Zhu and Y. Hu, "Enhancing search performance on gnutella-like P2P systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 12, pp. 1482–1495, 2006.

[13] K. Spripanidkulchai, B. Maggs, and H. Zhang, "Efficient content location using interest-based locality in peer-to-peer systems," in *Proceedings of IEEE INFOCOM*, vol. 3, (San Francisco, CA), pp. 2166–2176, Mar. 2003.