Knowl Inf Syst (2011) 26:105–125
DOI 10.1007/s10115-009-0270-7

REGULAR PAPER

A decentralized search engine for dynamic Web communities

Daze Wang · Quincy Chi Kwan Tse · Ying Zhou

Received: 13 November 2006 / Revised: 10 September 2009 / Accepted: 28 September 2009 /
Published online: 8 December 2009
© Springer-Verlag London Limited 2009

Abstract Currently, most Web search engines perform search on corpus comprising nearly entire content of the Web. The same centralized search service can be performed on a single site as well. Nonetheless, there is little research on community-wide search. This paper presents a peer-to-peer search engine ComSearch. ComSearch is designed to provide small- and middle-scale online communities—the ability to perform text search within the community. Communities are formed in a self-organizing style. P2P IR system may suffer unnecessary internal traffic in answering a multi-term query. In this paper, we propose several techniques to optimize the multi-term query process. The simulation results show that our proposed algorithms have good scalability. Compared with baseline approach, our improved algorithm can reduce the communication cost by about two orders of magnitude in the best case. We also deploy the system in a small-scale network and conduct a series of experiments to estimate the actual query response time as well as to investigate the data movement effect caused by node joining. Experimental results show that multiple data movements are quite common during network expansion. However, the percentage of multiple data movements decreases when a network is getting stable after the initial frequent joining activities. This provides possibilities for improvement on P2P data movement management.

Keywords Distributed hash table · Bloom filter · Information retrieval · Community level search · Web feed

1 Introduction

In recent years a wide range of second generation Internet-based services, such as weblogs and wikis emerged and gained rapid popularity among Internet users. Average Internet

D. Wang · Q. C. K. Tse · Y. Zhou (✉)
School of Information Technologies, The University of Sydney,
Sydney, NSW 2006, Australia
e-mail: y.zhou@usyd.edu.au; zhouy@it.usyd.edu.au

users are becoming more and more active in publishing on the Web. Many loosely organized communities have started to form around common interests.

The trend on the one hand brings huge increases in the scale of the Web; on the other hand boosts people's expectations on Web information retrieval. The traditional centralized search engine, such as Google works reasonably well for certain tasks, especially those that help people to get instant knowledge or information based on a few keywords. Such task does not require any prior knowledge since the query is performed on almost an entire copy of the Web documents. The "single point of contact to the whole world" style, which once had brought big success to search engines, may not satisfy new query requests made by experienced users. Increased scale of Web may magnify the noise level of any query performed on the entire Web. Therefore, experienced users may want to limit the search to a few Websites that they know are highly related. Currently, the option is to visit and search in each Website or to perform a site-wise search in a centralized search engine, such as Google. User will need to compare and rank the results from each site manually. The popularity of a number of specialized search engines, such as blog search (e.g., http://technorati.com/, http://www.daypop.com/), bookmark search (e.g., http://del.icio.us) and academic paper search (e.g., http://www.citeulike.org/, http://scholar.google.com/) indicates an increasing awareness of the need for specialized search. Some specialized search engines, such as technorati.com and daypop.com index only weblog and news-related pages on the Web. They select to index Web contents based on very coarse-grained styles or types (e.g., regular Web vs. news). In summary, there are fully automatic services for searching the whole Web, a portion of the Web roughly categorized by content style, and searching a single site, but there is little work done to supply search functionality to a collection of Websites, especially those interconnected weblogs or wiki. Weblogs and wikis are usually maintained by individual users, not large organizations. A prominent feature of weblogs is the large number of links coming in and going out of it [12]. Most weblogs are not isolated islands in the Web but are densely connected with other weblogs of similar topics. The densely connected weblogs soon form a new type of online community of interest to many recent researches [9,10]. Those communities are very dynamic in terms of participating members and their communication. Traditional centralized search engine design does not apply here. On the one hand, it is not possible to have an external site to centrally manage all those dynamic communities. On the other hand, an individual Web site can only manage its own data.

To address this issue, we present a self-organizing search engine, ComSearch. It is designed to provide communities like blogspace (network of weblogs with similar topics) the abilities to perform text search services covering all contents within the community from any member node. Since the index used in ComSearch is collectively managed by all members of the community, ComSearch is able to deliver flexible search services. Members form a P2P network and they can join the community in a self-organizing manner.

ComSearch is built on top of FreePastry [6], a popular structured overlay network implementation. It utilizes the property of structured overlay networks that the content, if exists, is always accessible from the host whose identifier is closest to the identifier of the content. This property allows data to be retrieved through routing of a query message instead of broadcasting to all nodes, which helps to improve the scalability in terms of the content and network size.

One big issue of P2P IR system is the high internal traffic among nodes in answering a multi-term query. This may result in delays in query response. Document clustering or classification, which requires substantial global information, is the main research effort in minimizing distributed query traffic [14]. It relies on implicit assumptions that both the entire document corpus and network topology are rather stable such that main document clustering

can be performed beforehand. This assumption cannot be held for frequently updated document corpus of online community. Hence the document clustering approach is not adopted. In this paper, we propose several techniques to optimize the multi-term AND query process in ComSearch framework. Those techniques are based on local compression using bloom filter and query routing optimization. Little or no global information is required for most techniques. We present simulation results showing that our proposed algorithms have good scalability and can improve performance of the system by about two orders of magnitude in the best case. The optimization of OR queries may require very different techniques and is not addressed here.

Another issue explored in this paper is the data movement caused by network topology change. ComSearch is designed for dynamic online communities and certain data movements are expected. We present empirical results of ComSearch deployed on a small-scale network. The empirical results are focused on actual query response times of multi-term queries and data movement issues cased by peers joining an existing network.

Our work makes the following contributions:

We present a framework that is capable of providing community-wide text search for the contents published by community members. In particular we propose to use Web feed information to provide quality and compact indices for Web content to be distributed in a P2P IR system.

We present and evaluate several algorithms for optimizing distributed multi-term queries. Those algorithms do not rely on assumed knowledge of entire document corpus and network topology.

We also investigate the data movement caused by peers joining an existing community network.

The rest of the paper is organized as follows. In Sect. 2 we present important background knowledge. In Sect. 3 we describe a few related works. Section 4 is focused on our framework and query optimization algorithms. We show the simulation results in Sect. 5. Section 6 reports the experimental results and Sect. 7 concludes the whole paper.

2 Background

In this section, we briefly describe distributed hash table (DHT) based structured P2P network, bloom filter and its parameters. We also introduce Web feed and common elements in various feed formats.

2.1 DHT-based structured P2P network

P2P networks can be classified into two categories: unstructured and structured. They have different structures and routing strategies. Unstructured P2P systems focus on data sharing and do not have well-defined rules for data placement. Early file sharing protocol, such as Gnutella (http://wiki.limewire.org/) is a good example of such network.

Chord [7] and FreePastry [6] are examples of structured P2P systems. Nodes in such P2P network cannot define independently what they would store and share with the other peers in the network. Data placement and retrieval are governed by a special DHT algorithm. The term DHT includes a family of algorithms which vary in ways of organizing nodes and routing information. We briefly introduce the Pastry algorithm implemented in the FreePastry network.

Each node in a Pastry network has a unique identifier from a 128-bit circular index space. The identifier is generated by applying SHA-1 hashing algorithm on a node's IP address. Each node also maintains a small table storing the addresses and the identifiers of nodes the identifiers of which are numerically larger or smaller than its own identifier. Contents in a structured overlay network are also assigned identifiers, which are usually the hashes of the keys to the contents. For instance, important terms of an article may be used as keys. By applying secure hash algorithm (SHA-1) on a particular term, we get the identifier of the article. Contents in the network are stored in the node the identifier of which is numerically closest to the identifier of the content. Content retrieval in such a network is achieved by comparing the identifier of the content with the identifier of the node handling the query. This query is routed through the network until the node, the identifier of which is numerically closest to the identifier of the content is reached. The content is then retrieved locally and forwarded back to the requesting node.

2.2 Web feed

A Web feed is an XML document containing metadata of several Web pages from the same sites. The metadata usually include summaries of news articles or weblog posts with URLs pointing to longer versions. The summary is a good way of identifying keywords (e.g., important terms) in corresponding content (e.g., news article) for storing and retrieving purpose in a structured P2P network. URL is a natural document ID for any Web content. Other metadata, such as author, title and publishing date may also be included in the feed. The two most popular feed formats are RSS 2.0 and ATOM. Web feed is originally designed for Web content syndication. We will use RSS 2.0 as an example in this paper. In RSS 2.0, a complex type element <item>contains metadata of a news article or weblog post stored, respectively, in elements like <title>, <author>, <link>and <description>. ATOM format contains similar core elements but of different name. We will use the RSS element names in this paper.

2.3 Bloom filter

Bloom filter [3] is a hash-based data structure. A bloom filter can represent a set compactly, at the cost of a small probability of false positive. It achieves the compact representation by using a set of independent hash functions.

A bloom filter for representing a set $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$ of n elements is described by an array of m bits, initially all set to 0. The k independent hash functions $h_1, h_2, \ldots, h_k$ are then applied on each element x_i. For each element, we get k numbers over the range $\{1, \ldots, m\}$. We then turn the corresponding bit of the array to 1. The final bit array is the bloom filter representing the set S.

A bloom filter has two basic parameters: the number of hash functions k and the length of bloom filter m. We briefly discuss the configuration of these two parameters to get optimal false positive rate and network traffic.

Assume the k hash functions uniformly distribute k hash values of an element over the range $\{1, \ldots, m\}$. The probability that a specific bit is still 0 after adding an element is: $(1 - \frac{1}{m})^k$. After adding n elements into the bloom filter, the probability that a specific bit is 0 becomes $(1 - \frac{1}{m})^{kn} \approx e^{-kn/m}$. Hence the probability of false positive P_{fp} is given by

$$P_{fp} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k$$

By simple calculation, P_{fp} is minimized when $k = ln2 \cdot \frac{m}{n}$.

We can see that the network traffic is determined by two factors: the length of bloom filter m, and the desired false positive rate P_{fp}. Larger m will reduce the false positive rate, but it will result in larger bloom filter and higher network traffic. If we set the desired P_{fp} to a relatively high value, it will not only decrease the quality of results, but also transmit more incorrect final results over the network. This in turn increases overall network traffic. [12] justified that the relationship between overall network traffic and m is a logarithm function. So it is possible to find an optimal m to minimize network traffic. In Sect. 5 we present experimental results on optimal m.

3 Related works

Study by Li et al. [8] on the feasibility of P2P search engines presents two common P2P text search strategies: partition-by-document and partition-by-keyword. Partition-by-document option divides the content up among the peers. Each peer maintains a local inverted index of the documents it is responsible for. Each query must be broadcast or flooded to all peers each of which returns its most relevant document(s). The approach is mainly adopted by P2P file-sharing systems on unstructured P2P networks. The problem of this approach is that there is no guarantee that content can be located properly, even if relevant content does exist. Moreover, flooding consumes too much bandwidth. Partition-by-keyword means that responsibility for the terms that appear in the document corpus is divided among the peers. Each peer stores the posting list for the word(s) it is responsible for. A DHT would be used to map a word to the peer responsible for it. Many recent proposals [14] are based on this scheme.

Li et al. [8] justified that in naïve implementation, neither scheme is capable to support Web scale full-text search. The bottleneck here is communication cost: at Web scale a naïve partition-by-keyword implementation might require 530 MB to be sent over the whole network per query, which is impractical. ComSearch is not designed as a Web scale search engine; however, pilot experiments also suggest that scalability of such system is a serious issue. Reducing the communication cost is a key focus in this paper. Li et al. [8] proposes various techniques for reducing communication cost: caching, compression techniques, such as bloom filter, gap compression, adaptive set intersection and clustering. Theoretic compression ratios based on the size of result set for various techniques are presented. However, this paper does not propose an effective way of computing the size of bloom filter. It does not consider the practical issues, such as the impact of various ways of distributing queries and collecting back the results.

Cuenca-Acuna et al. [5] presented another P2P IR system, PlanetP that works with XML document. PlanetP uses a gossiping algorithm instead of structured overlays. It prevents flooding of queries by the use of bloom filters stored on each node. Each node has a copy of the bloom filter from all other nodes. These bloom filters summarize what data are stored in the corresponding node. For each query, the issuing node looks up their copy of the bloom filter to determine which node may have the information requested, thus minimizes flooding. Comparing with the structured overlay approach, gossiping has less maintenance cost but also worse scalability and availability.

Broder et al. [4] presented a survey of uses of bloom filter in various network applications. By investigating the nature of bloom filter, they present the bloom filter principle: "whenever a list or set is used and space is an issue, then bloom filter can be considered." Using bloom filter will introduce false positives, thus it is important to consider if false positives are

permitted in the specific application context. For partition-by-keyword P2P search engines, a multi-term query requests intermediate results to be sent from individual nodes over the network to calculate an intersection. The intermediate result sets can be compressed using bloom filter. False positives in this case represent incorrect results containing only a subset of terms. This will bring in some unnecessary traffic and may reduce the precision of final result. However, information retrieval different from database query, it does not require a perfect 100% precision. The quality of final result can be guaranteed by choosing the parameters of bloom filters carefully.

Tryfonopoulos et al. [16] proposed LibraRing, which uses the publish/subscribe method to facilitate searches. In LibraRing, a network of "super peers" forms the back bone of the routing network. Super peers are connected via a structured network overlay, utilizing the DHT for its pub/sub functionality. Each super peer is responsible for a subset of subscriptions as well as the nodes connected to it. Provider nodes (content producers) and client nodes (content consumers) join the network at any one of the super peers to access the service. Providers and clients have no knowledge of the network other than its super peer. When a client submits a query, the query is flooded among the super peers, thus limiting the flooding to a subset of the network. LibraRing is designed mainly for the pub/sub facility. The one time query function, may not scale well with a large number of super peers. Balke et al. [2] presented another system which relies on super peers and their topologies to achieve top-K distributed retrieval. The key focus of this system is to use dynamic query statistics to optimize the routine algorithm to minimize the contacting peer and the internal traffic. The local ranking algorithm reduces individual result set to the top-key documents. We do not consider the super peer structure as it is not suitable for community-wide query.

Reynolds et al. [12] proposed a P2P search infrastructure and several techniques to reduce communication cost in search. These optimization techniques include bloom filter, caching, virtual hosts and incremental results. In the work they justified that if we need to construct a bloom filter to calculate intersection of set A and B, a unique optimal length exists and is relevant to the size of both A and B. It means that for each pair of initiator node and target node, the optimal length and corresponding false positive rate of bloom filter is different. However, this may bring in extra communications in order to compute and to exchange the size of both sets. The configuration of hash functions and bloom filter length are also required to be sent with bloom filters. In our work we use a simpler technique. This technique fixes a desirable false positive rate for each bloom filter and uses a predefined optimal value to calculate the size of bloom filter based on set size. Each node can independently compute the bloom filter's size. The calculation involves only a simple multiplication. This results in faster computation at each node. It also eliminates the need of exchanging initial results size among participating nodes. The optimal value is determined by the experiment reported in Sect. 5.4.

4 System design

ComSearch takes partition by keyword approach. It is built on top of a structured overlay network, as the gossip algorithm adopted by PlanetP does not have good scalability. The node in the ComSearch network can either be a content provider of documents, for instance, a weblog, or a dedicated content syndicating host (news aggregator), or a member node which only provides storage and computation power. Hence, each node will at most have, and only have, a local copy of the documents they publish or collect. Some nodes may not have any local data.

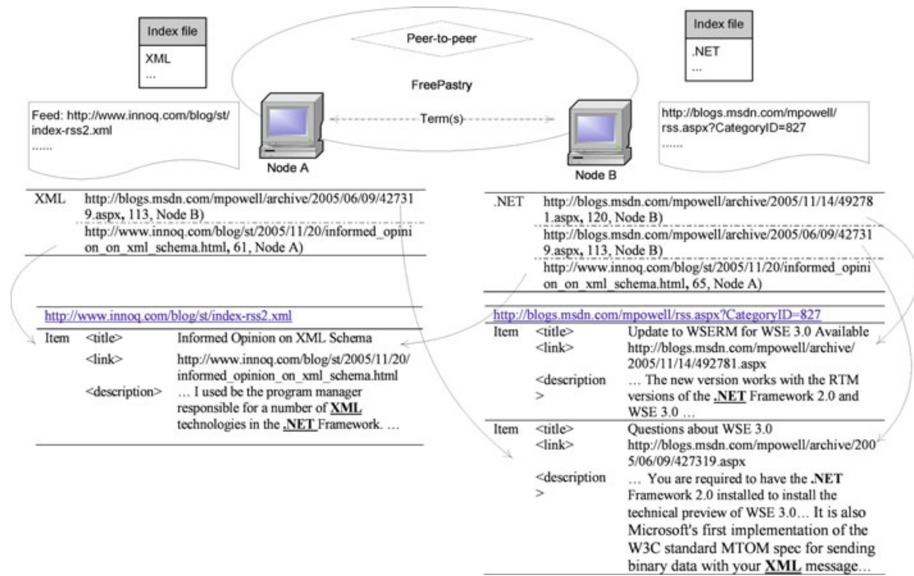

Fig. 1 ComSearch architecture

In addition to the documents, each node also keeps the inverted index files of a subset of all terms appearing in the entire corpus. Here, entire corpus means the collection of local documents in all member nodes. Each term has an object ID and is stored in the node whose node ID is closest to this object ID. Local data stores do not always match local term indices; i.e., a term index may point to a document residing on a different node in the network. Figure 1 shows the overall architecture of the ComSearch system with some sample data and inverted index to illustrate the idea. Suppose node A is locally storing feed http://www.innoq.com/blog/st/index-rss2.xml, we illustrate the result of creating indices for two terms in the item with title "Informed Opinion on XML Schema". Partial information of that item is shown in Fig. 1. The terms of interest are "XML" and ".NET". They both appear in the description of that item. Suppose the local Pastry location service on node A determines that term XML should be stored on node A and term. NET should be stored on node B. Therefore, a record of term XML is added to node A's index file. This corresponds to the record http://www.innoq.com/blog/st/2005/11/20/informed_opinion_on_xml_schema.html, 61 under entry XML. A record of term. NET is eventually passed to node B where it is supposed to store. This corresponds to the http://www.innoq.com/blog/st/2005/11/20/informed_opinion_on_xml_schema.html, 65 under entry. NET in the index file of node B. Figure 1 also shows the indexing result of two other documents: "Update to WSERM for WSE 3.0 Available" and "Questions about WSE 3.0". Both documents are stored on node B but the final indexed term records are distributed on both nodes A and B. Each node will have the complete inverted index files of any term it is supposed to store and the inverted index file of a term can only be stored in one node. For instance, in Fig. 1, node A has the complete inverted index file of term "XML", which includes information about document ("Questions about WSE 3.0") stored on node B.

Based on this data distribution scheme, query can be issued on any node of the network. Each node can independently work out the location of query terms' inverted index files and then send modified queries to all nodes that should have the term stored.

Handling single term query is very straightforward. The query will be routed from the query node to the target node that has the index file of that term. The target node then checks its index file and sends answer back to the query node. A multi-term ‘AND’ type query would involve contacting a few nodes and perform ‘join’ operation on partial results from all nodes. In this section, we will focus on presenting the multi-term query algorithms implemented in ComSearch.

4.1 Naive algorithm

We first present two direct and naïve multi-term query algorithms. Figure 2 shows a straightforward multi-term query algorithm, the “Star” algorithm. When the initial node (node I in the figure) receives a query request T1, T2, T3, it checks every term T1, T2 and T3 and then send them to corresponding nodes (node A, B, C in the figure), respectively. Afterwards each node sends its local query result A, B, C back to the initial node. Final result is generated by intersecting all the results together.

Figure 3 shows another naïve algorithm taking serial approach. Upon receiving a query request, node I first determines if there are locally stored terms in the request. If present, then a local result is obtained from the database. The remaining terms, if any, are routed to a node (node A in the figure) that is responsible for the first term in the query. At node A this procedure is repeated. At last every node receives the local result of its next node and performs an intersection with its own local result, then route the outcome to its previous node. Finally,

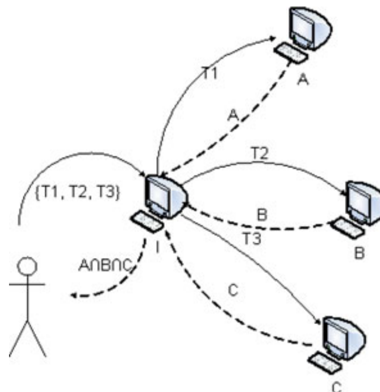

Fig. 2 Query with Star algorithm

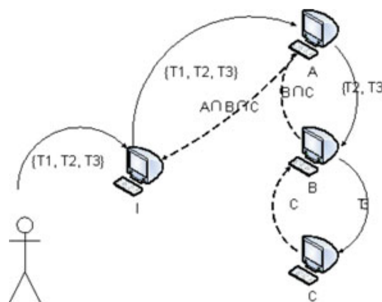

Fig. 3 Query with Serial algorithm

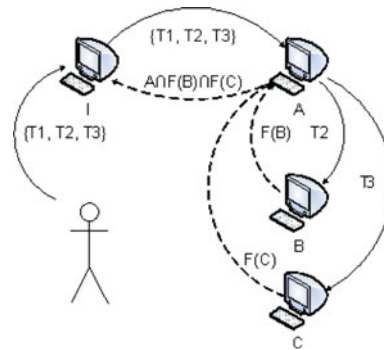

Fig. 4 Query with StarWithBF algorithm

node I will receive the intersection of all the results. Comparing with the Star algorithm, the Serial algorithm can cut down the overall network traffic by sending intersection instead of the whole result sets.

4.2 Naïve algorithms with bloom filter

We then apply bloom filter on the two algorithms to minimize network transmission in the query process. For convenience, we name the two algorithms with bloom filter "StarWithBF" and "SerialWithBF", respectively. The parameter configuration of these bloom filters is discussed in Sect. 2.3.

The StarWithBF algorithm is slightly modified from the Star algorithm. In Star algorithm, the node that receives a query (e.g., node I in Fig. 4) is responsible for aggregating all results from different nodes. In StarWithBF, we request the node that has information of the first query term T1 (e.g., node A in Fig. 4) to do the aggregation. This can save the cost of sending inverted index file of T1, which may be quite large, over the network if the querying node (I) does not have information for any query term. If node I happens to have at least one query term then StarWithBF becomes the same with Star algorithm.

Here, the "results" from other nodes are actually bloom filters ($F(C)$ denotes the bloom filter of result set C) of their local result sets. After receiving all the bloom filters, node A intersects its own local result with the bloom filters, and then returns the outcome $A \cap F(B) \cap F(C)$ to node I.

The SerialWithBF algorithm as illustrated in Fig. 5 is very similar to the Serial algorithm. The only difference is that nodes send bloom filters instead of the whole result set. Both StarWithBF and SerialWithBF will return result that contains certain false positives. SerialWithBF may produce more false positives than StarWithBF as it has nested bloom filter operations.

4.3 A revised serial algorithm with bloom filter

The performance of Serial and SerialWithBF algorithm is optimized if the bloom filter is always sent from a node with a smaller result set to a node with a larger one. If we need to get the intersection of nodes A, B and C and we have $N_C \leq N_B \leq N_A$, where N_A represents the size of result set at node A; the network traffic is minimized if we send $F(C)$ to B and then send $F(B \cap F(C))$ to A. However, original SerialWithBF algorithm sends queries in the same order as terms appear in a query. In this section we propose a revised Serial algorithm

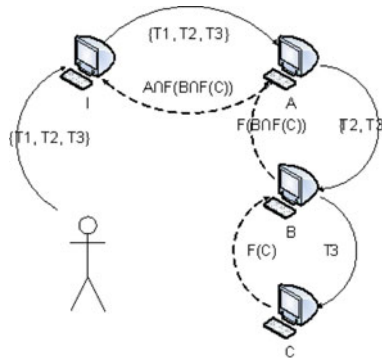

Fig. 5 Query with SerialWithBF algorithm

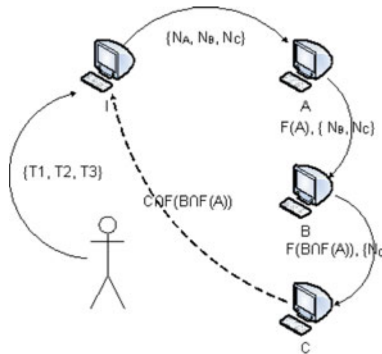

Fig. 6 Query with ImpSerial algorithm

with bloom filter. We call it “ImpSerial” in short. ImpSerial is inspired by the bloom filter intersection technique proposed in Reynolds et al. [12].

Figure 6 shows how the ImpSerial algorithm works. The algorithm can work with arbitrary number of terms, but for simplicity we use a 3-term query here as an example. The algorithm starts by sending query requests to the corresponding nodes. Instead of instantly returning query results or bloom filters, the nodes answer the queries by returning the sizes of their result sets. The initial node (node I) is responsible for receiving size information N_A, N_B, N_C from all nodes involved in the query process.

These values are then sorted at node I in numerical order. After that, node I generates a list containing sorted nodes and sends it to the node that has the smallest result set. Suppose we have $N_A \leq N_B \leq N_C$, node A will be the first node. Node A then removes itself from the list and sends the list with the bloom filter of its result set to the next node on the list, that is, node B. Node B calculates the intersection of B and F(A), and then send the bloom filter of $B \cap F(A)$ to node C.

At the last node (node C) the list becomes empty. Here, we have two options. Node C may calculate the intersection of C and $F(B \cap F(A))$ and return it to node I directly. This option will bring in some false positives and will eliminate weight information of terms. Alternatively, node C can send $C \cap F(B \cap F(A))$ back to B and then to A. By sending result back to previous nodes and calculating $A \cap (B \cap (C \cap F(B \cap F(A))))$ we can eliminate false positives and retain weight information; but the network traffic will increase a lot. In ComSearch we

rank the query results using date information, so option one presents a trade-off between precision and network traffic. Furthermore, the precision can be improved to an acceptable range by reducing the false positive rate of bloom filter. Hence, option one is implemented for the algorithm.

4.4 Caching

Two types of caching techniques can be used in ComSearch: server result caching and client result caching. In the server result caching, every node caches its local query results to avoid frequent local database query operations. Pushing the cache to the client side can save both network traffic and local database cost [16]. ComSearch adopts client side caching option. It is implemented by requesting each node to cache the bloom filters they received from other nodes.

It is also viable to cache the query results instead of bloom filters. This option can avoid false positives and can retain weight information. However, bloom filters are much smaller than the actual query results and more information can be saved in a fixed sized cache, thus increasing cache hit rate.

Caching is implemented for the SerialWithBF algorithm. When node A needs to get the bloom filter from node B, it will first look up its cache for the bloom filter. If there is a hit, it is not necessary for node B to send F(B) to A. This improves the overall performance. If a query contains multiple terms, the bloom filter needed for answering the query and cached locally may both be intersections of some terms. For instance, we may use a bloom filter $F(B \cap F(C))$ to answer the query, a locally cached $F(B \cap F(D))$ does not consider as a cache hit. We would expect lower hit rate in multi-term query compared with those from single-term query.

Research on query logs has shown that the popularity of terms appearing in a large collection of queries roughly follows a Zipf distribution [13]. Hence caching the most popular terms may result only in better performance. ComSearch implements such caching mechanism. Its cache is maintained by a least-recently-used (LRU) based algorithm. LRU removes a cached term if it is not requested for a predefined period of time.

5 Simulation and results

We use simulation to evaluate the performance of the various multi-term query algorithms. In this section, we present the details of simulation design, datasets, environment and results.

5.1 Simulation design

We model our simulator as a Markovian simulation. The simulator contains a given number of nodes. A corpus (dataset) is randomly divided among the nodes. The simulator was build using Java on top of FreePastry framework. Apache Lucene package is used for setting up the inverted index and Hsqldb, an open source java database engine (http://hsqldb.org/), is used as the local database management system for each node.

The simulator can accept queries on any arbitrary node, and use a pre-configured algorithm to get query results. The query result set, the number of hits, the number of nodes contacted

in the query and the overall network traffic incurred in the query are returned to the node that issues the query request.

The number of nodes contacted in a query is calculated by counting all the nodes involved in the query. The count includes the initial node that issues the query. The maximum number of nodes being contacted is (number of terms + 1) assuming that all terms are distributed in different nodes.

The overall network traffic is estimated by cumulating the length of all messages routed over the network. Every ASCII char is counted as 1 byte long. The length of bloom filter is counted as $|\frac{m}{8}|$ plus the length of head information.

5.2 System measurements

We use three metrics to measure the performance of ComSearch: precision of query results, network traffic per query and number of nodes contacted. Recall is not measured because bloom filters and cache do not introduce any false negatives. Thus, the recall of all algorithms will be constant.

Among our five algorithms, the basic Star algorithm and the basic Serial algorithm do not bring in any false positives or false negatives, so the query result returned from basic algorithms is used as a base line for comparison. The precision of a given algorithm is calculated as

$$P = \frac{\text{number of results of basic algorithms}}{\text{number of results of improved algorithms}}$$

Network traffic, or number of bytes sent over the network per query, is considered as the most important metric of performance in this paper. Reducing network traffic will decrease the overall query latency.

Number of nodes contacted is mainly measured in cache experiment. Cache algorithms are expected to reduce the number of nodes being contacted. This will in turn reduce the network traffic and local CPU time.

5.3 Simulation environment

We gathered 14,000 RSS feeds from various online RSS and RDF aggregators as the experimental corpus. These feeds are mainly for news articles. They can be viewed as the base content of an online news community. Feeds are randomly assigned to nodes as their local contents. Feed may contain various number of items. Each item is a summary of a news article Web page and it represents a document in ComSearch. There are 34,998 distinctive documents in the whole system. After initial text preprocessing, there are 677,933 terms in the entire corpus. The number of distinctive terms is 30,776.

To emulate realistic queries, we find the top 50 most popular terms in the corpus used in each experiment, and use those to generate queries of given number of terms. We do not test rare terms as they would result in small term postings and much less internal traffic. However, the proposed approach works for all sorts of terms. The frequency of terms in all generated queries in an experiment follows the same Zipf distribution as that in the corpus. For example, if a term T_A has a probability of 0.15 to occur in the whole corpus, then in our generated queries T_A also has a probability of 0.15 to be selected. The simulator runs on a PC with 2 Pentium 4 Xeon 2.8 GHz CPUs and 2G RAM.

5.4 Bloom filter setting

Bloom filters may reduce network traffic with slight sacrifice on query accuracy. The configuration of bloom filters is essential for the performance. In this section, we report the test performance of StarWithBF, SerialWithBF and ImpSerial algorithms under various parameter configurations. The purpose is to find an optimal value to use in all simulations. For each algorithm, 100 queries are submitted and average precision and network traffic are calculated. The same query process are repeated under various configurations of bloom filter parameter $\frac{m}{n}$, where $\frac{m}{n}$ represents the number of bits per element (m represents the total number of bits and n is the number of documents in an intermediate result set). The number of hash functions is calculated by formula $k = ln2 \cdot \frac{m}{n}$.

We start this experiment with a dataset of 2,000 feeds. The dataset is then expanded to 6,000, 10,000 and 14,000 feeds to see if the optimal value is consistent while the system scales. The queries used are all 3-term ones.

Figure 7 shows the average network traffic per query as a function of the bloom filter length for 14,000 feeds. As expected, the function is similar to a logarithm curve. As the length of bloom filter goes up, the false positive rate decreases. However, as bloom filter length continues growing, the size of bloom filters becomes a dominating part in the overall network traffic. Through our experiment we can see that the optimal value is 16 bits per element. The logarithm feature and optimal number is consistent over all dataset sizes.

Figure 8 and Table 1 shows the relationship between precision and bloom filter length. We can see that setting the parameter to 16 bits per element will minimize network traffic, but the precision of SerialWithBF algorithm and ImpSerial algorithm is not very good. If we use the value 32 bits per element to construct the bloom filters, in ImpSerial algorithm, there is only 5% increase in network traffic while the precision is increased from 45% to 100%. The same trade-off cannot be achieved in SerialWithBF algorithm. Moving from 16 bits/element to 32 bits/element, both precision and network traffic go up for about 30%. This indicates that the growth of bloom filter length has a smaller impact on the overall traffic of ImpSerial algorithm than those of the other two algorithms. For ImpSerial algorithm, we may opt to use 32 bits per element bloom filter to achieve both desirable precision and near optimal network traffic. However, for the other two, 16 is the optimal value.

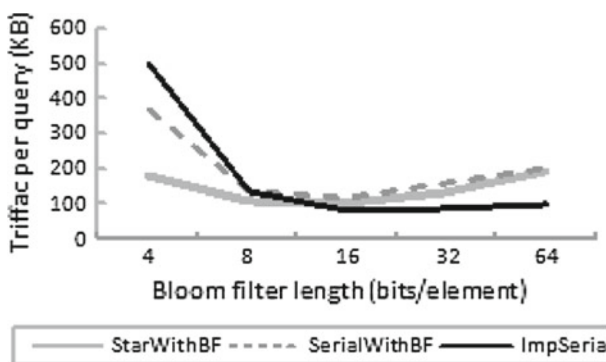

Fig. 7 Network traffic as a function of BF length

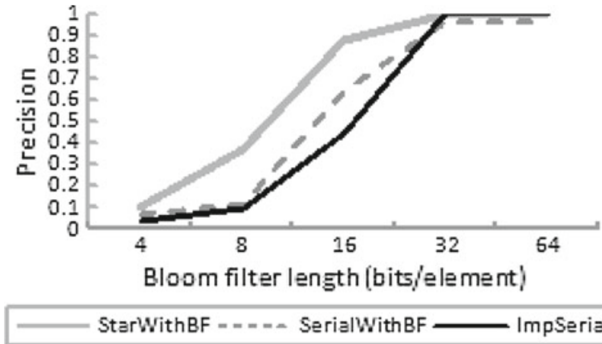

Fig. 8 Precision as a function of BF length

Table 1 Performance of various bloom filter lengths

	16 bits/element		32 bits/element	
	Traffic (Bytes)	Precision	Traffic (Bytes)	Precision
StarWithBF	104380.2	0.88269	131796.1	1
SerialWithBF	117301.1	0.636831	162090.4	0.96
ImpSerial	84278.89	0.454518	89171.6	1

5.5 Scalability in terms of network size and corpus size

This simulation seeks to evaluate the scalability of our proposed algorithms. We start with 10 nodes each containing 100 feeds in average. We issue 100 2-term queries on Star, Serial, StarWithBF, SerialWithBF and ImpSerial algorithms and compare the average network traffic per query. After that we change the queries to 100 3-term ones and do the same process. We continue with 4-term, 7-term and 10-term queries and then scale the system to 20, 40, 60, 80, 100, 120 and 140 nodes. Every node in the experiment contains 100 feeds in average. The bloom filters in this experiment are configured to 16 bits per element in length and 11 hash functions are used. While the size of the system grows linearly, we expect the average network traffic of our improved algorithms grows sub-linearly. Due to space limit, we show only results of 4-term query here. Figure 9 shows the internal traffic as a function of network sizes.

We also test the scalability in terms of corpus size. We fix the number of nodes at 100 but increase the size of whole corpus gradually from 1,000 feeds to 2,000, 4,000, 6,000, 8,000, 10,000, 12,000 and 14,000 feeds. Figure 10 shows the internal traffic as a function of corpus sizes.

The results for other multi-term queries have similar features. The only difference lies in the actual traffic per query. The results clearly show that the Serial algorithm has better scalability than the Star algorithm. This is because it cuts down duplicated transmission to some extent. However, scalability of both basic algorithms falls far behind the scalability of the three algorithms that implement bloom filter. Figures 11 and 12 further compare the scalability of the three. Again, only 4-term query result is show here. For 2-term queries the three algorithms achieve similar performance. For other groups, the performance of StarWithBF algorithm is very unstable as exemplified by some sharp ups and downs in Fig. 11.

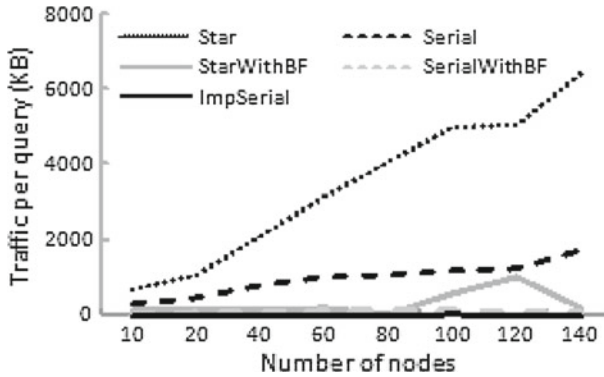

Fig. 9 Traffic per query as a function of network size

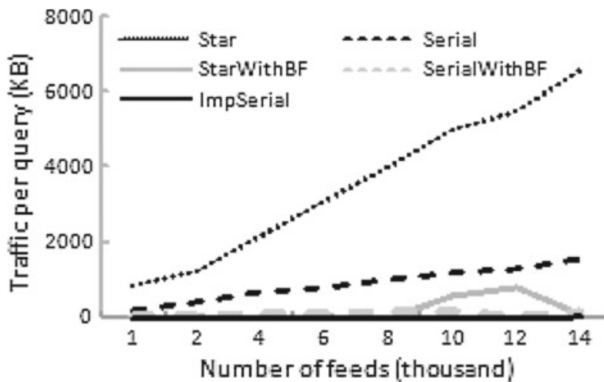

Fig. 10 Traffic per query as a function of corpus size

The internal traffic of StarWithBF relies heavily on the sizes of result sets of involving nodes. Generally SerialWithBF algorithm has better scalability than StarWithBF; but it is not as good as ImpSerial algorithm. The ImpSerial algorithm has a stable performance. The advantage over other algorithms is very obvious in 4-term, 7-term and 10-term queries experiments. Its performance is nearest to the theoretical optimal bloom filter compression ratio. When the query returns 0 result, which happens quite often for long queries, the ImpSerial algorithm can avoid unnecessary traffic by checking the number of partial hits on each participating node in advance.

5.6 Caching performance

We implement caching technique over the Serial algorithm and issue 100 2-term queries to compare the performance of the original algorithm and the new one with caching technique. The ImpSerial algorithm is used as a reference baseline. The same process is repeated for 3-term, 4-term, 7-term and 10-term queries. The experiment runs twice using 50 and 100 nodes, respectively. The corpus used in this experiment contains 10,000 feeds.

From Fig. 13 and Table 2 we can see that caching can decrease both the network traffic and the number of nodes contacted per query. In 2-term and 3-term queries the difference is not so obvious because these queries are issued at the beginning of the simulation and the cache

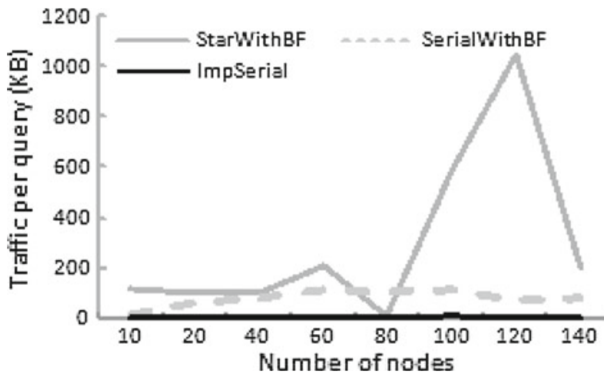

Fig. 11 Scalability of BF algorithms with respect to network size

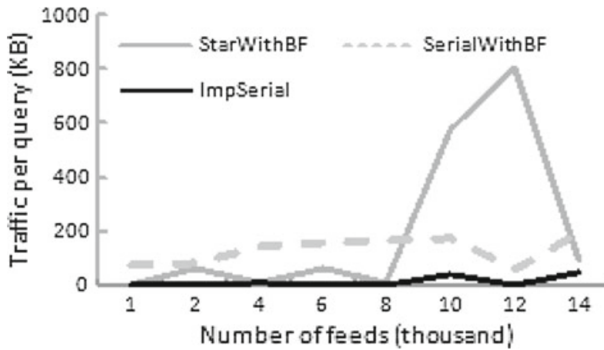

Fig. 12 Scalability of BF algorithms with respect to corpus size

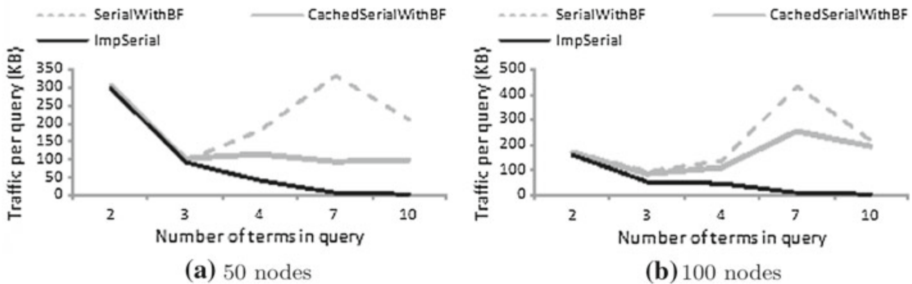

Fig. 13 Comparing caching with other algorithms. **a** 50 Nodes, **b** 100 nodes

is in filling up stage. In 4-term, 7-term and 10-term queries there is a remarkable reduction in the both metrics. Queries are randomly assembled from the list of terms appearing in the data corpus. It is very likely that 10 randomly assembled terms may not return any results. This explains the traffic reduction observed in SerialWithBF on 10-term query. However, it should be noted that the average network traffic of ImpSerial algorithm is still much smaller than that of CachedSerialWithBF.

Table 2 Average number of nodes contacted in a query

		2-term	3-term	4-term	7-term	10-term
50 Nodes	SerialWithBF	2.98	3.94	4.85	7.72	10.7
	Cached	2.96	3.64	3.93	4.6	5.56
100 Nodes	SerialWithBF	2.93	3.78	4.69	7.13	9.38
	Cached	2.86	3.34	3.49	3.41	4.28

6 Experimental results

In this section we present the experimental results of running ComSearch on a network of up to 30 PCs connected by 10 Mbps Ethernet. We use the experiment first to investigate the base case query response time under real network environment. We also investigate the data movement caused by node joining an existing network.

6.1 Query response time

We are interested to learn the possible maximum response time in a real network. The queries we submitted to ComSearch are combination of most common terms in the entire document corpus. Such queries do not necessarily represent the query a Web user may issue. However, they do represent worst case response time where all terms in a query have large intermediate result sets. Figure 14 plots the mean response times for 1 and 4 common term queries against the number of nodes and number of feeds for basic Serial algorithm. Dashed lines represent mean response times for 1-term query while solid lines represent mean response times for 4-term query. There is not much difference in response time for queries performed on 10, 20 or 30 nodes. However, with respect to the feed number, the response times increase in a linear scale with the increase of the corpus size. This is consistent with the observation

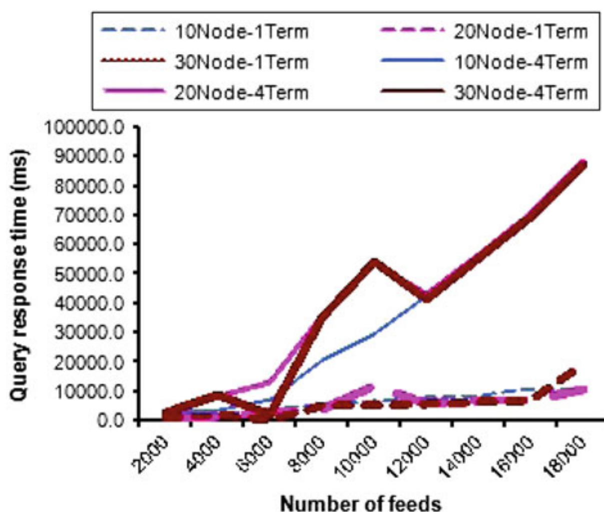

Fig. 14 Query response versus number of nodes versus number of feeds

from the simulation. Basic Serial algorithm does not use any optimization (bloom filter or caching), the recorded actual response time is higher than those of real world search engine. It would be much smaller for ImpSerial algorithm. Also, as shown in the simulation results, the internal traffic does not increase with the increase of network size or corpus size for the ImpSerial algorithm and two other algorithms with bloom filter. We do not expect the actual query response time to be increased as well.

6.2 Node joining

To investigate the data movement caused by node joining, we build an initial network consisting of 10 nodes each that stores a certain amount of data. New node will join the system continuously until the network has a maximum of 30 nodes. The data brought by new node will be indexed and routed to its destinations in a similar way as described in Sect. 4. More importantly, some indices previously stored in existing peers need to be relocated to the new node. We will focus on the movement of existing indices in this experiment; hence we assume that all new nodes do not bring new data to the network.

Figure 15 shows the relationship between the percentage of records moved and the number of existing nodes. The theoretic data movement based on the ideal situation that all nodes store equal amount of records is plotted in dashed lines. There are reasons to believe that the relationship is hyperbolic as the amount of data at each node should follow a $1/x$ relationship, where x represents the number of nodes in the system. Therefore, the trend line (solid) had been plotted as a hyperbolic function. Since the experiment starts from 10 nodes, measurements for node numbers 1–10 are not taken. This hyperbolic relationship would be difficult to show with the measurements taken. The decrease of the theoretical values from 10 nodes to 29 nodes is only 5.6, while the spread of the values is almost 10. Data movement is sensitive to a number of factors including the order of joining, the original dataset and the new datasets. The large spread of the values is caused by those unmeasured uncertainties.

Figure 16 plots the cumulative percentage of total multiple movements against the percentage of number of nodes joined. It is obvious that a large amount of record has been moved multiple times during the network expansion. Of the total amount of records moved to expand the network from 10 nodes to 20 nodes (200% expansion), approximately 30% of all the record movements are multiple movements. For a 300% expansion (expending from 10 nodes to 30 nodes), almost 55% are multiple movements. It is also observed that almost 95% of all the multiple movements have been completed after approximately 80% of

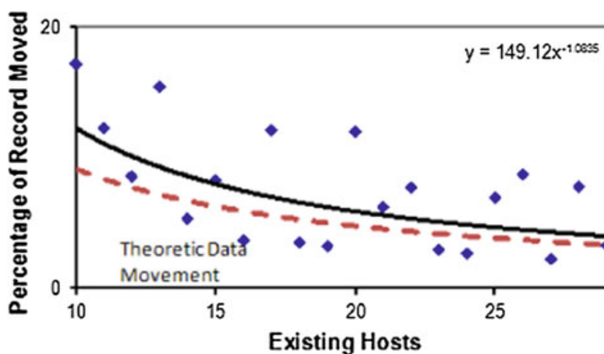

Fig. 15 Percentage of records moved versus number of existing hosts

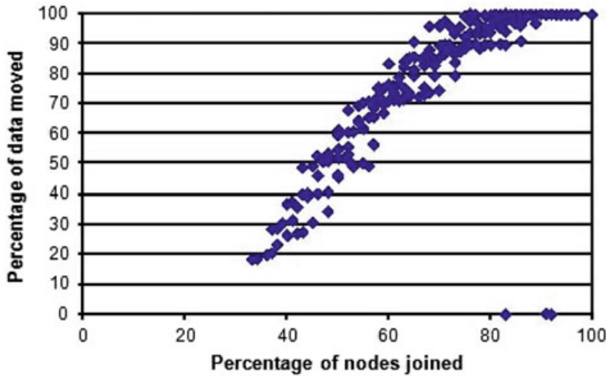

Fig. 16 Percentage of total node joins versus percentage of data moved more than once

the final number of nodes joined the network. For example, for a 10-node network, if there are a total of 1,000 multiple movements, then after the 8th node joins the network (80% of the 10-node network), about 950 multiple movements are completed. The observation on multiple movements is a good indicator of an alternative solution of node joining. In our experiment, data are instantly moved to its new target node when the topology of network changes. Alternatively, a link can be placed in the new target node pointing back to data's original position. This will increase the number of hops during query but will save network traffic caused by multiple movements of the same data item. Data can be moved to its final target node periodically or when there is clear sign of a relatively stable network.

Node departure is another common activity in a self-organized peer to peer network. In normal case, when a node departs it will invalidate entries related with its local data in other nodes and send its term postings to neighboring nodes. There are also unusual cases when a node may depart without doing all the regular departure preparations. The whole network may have to perform some re-indexing if there is no data replication. However, these are beyond the scope of this paper.

7 Conclusion and future work

In this paper, we proposed a self-organizing decentralized searching engine ComSearch mainly designed for dynamic online communities. We presented several techniques to optimize multi-term query process. We applied bloom filter and caching to reduce the overhead of multi-term AND queries. To evaluate our approach, we first built a simulator implementing the techniques; experiments with various system configuration and scale have been done on the simulator. We found the optimal length of bloom filter in our system through experiments. Results also show that our proposed techniques and algorithms have good scalability while the system scale grows. Comparing with the naïve query algorithms, our revised algorithm can reduce overall network traffic for about two orders of magnitude in the best case. Considering the decentralized and self-organizing nature of our system, ComSearch would be attractive for online communities who want flexible searching services. We also deployed the system in a small-scale network and conducted a series of experiments to estimate the actual query response time as well as to investigate the data movement effect of node joining. Experimental results show that multiple data movements are quite common during network

expansion. However, the percentage of multiple data movements decreases when a network is becoming stable. This provides incentives and possibilities for improvement on P2P data movement management.

The experiment results on node joining and data movement prompt several interesting data placement issues that deserve further study. We will investigate data placement strategies involving link structure and replication and study the cost involved in node joining and departure stage under various strategies.

References

1. Anagnostopoulos A, Broder A, Punera K (2008) Effective and efficient classification on a search-engine model. Knowl Inf Syst 16(2):129–154
2. Balke WT, Nejdl W, Siberski W, Thaden U (2005) Progressive distributed top-k retrieval in peer-to-peer networks. Poceedings of the 21st international conference on data engineering. Tokyo, Japan
3. Bloom B (1970) Space/time tradeoffs in hash coding with allowable errors. Commun ACM 13(7): 422–426
4. Broder A, Mitzenmacher M (2003) Network applications of bloom filters: a survey. Internet Math 1(4):485–509
5. Cuenca-Acuna FM, Peery C, Martin RP, Nguyen TD (2003) Planet P: infrastructure support for P2P information sharing. Proceedings of the 12th international symposium on high-performance distributed computing, 22–24 June 2003
6. Druschel P, Engineer E, Gil R, Hu YC, Iyer S, Ladd A (2006) FreePastry. http://freepastry.rice.edu/
7. Ion S, Robert M, David LN, David RK, Kaashoek MF, Frank D, Hari B (2001) Chord: a scalable peer-to-peer lookup protocol for internet applications. Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications. San Diego, California, United States, 2001
8. Li J, Loo B, Hellerstein J, Kaashoek F, Karger D, Morris R (2003) On the feasibility of peer-to-peer web indexing and search. Proceedings of the 2nd international workshop on peer-to-peer systems, Berkeley, California, 2003
9. Lento T, Welser HT, Gu L, Smith M (2006) The ties that blog: examining the relationship between social ties and continued participation in the Wallop weblogging system. Proceedings of the 3rd annual workshop on the weblogging ecosystems: aggregation, analysis and dynamics, WWW2006. Edinburgh, May 23, 2006
10. Lin Y, Sundaram H, Chi Y, Tatemura J, Tseng B (2006) Discovery of Blog communities based on mutual awareness. Proceedings of the 3rd annual workshop on the weblogging ecosystems: aggregation, analysis and dynamics, WWW2006. Edinburgh, May 23, 2006
11. Ng P, Ng V (2008) RRSi: indexing XML data for proximity twig queries. Knowl Inf Syst 17(2):193–216
12. Reynolds P, Vahdat A (2003) Efficient peer-to-peer keyword searching. Proceedings of middleware 2003. Rio de Janeiro, Brazil
13. Searls D, Sifry D (2003) Building with blogs. Linux J 107:65–73
14. Silverstein C, Marais H, Henzinger M, Moricz M (1999) Analysis of a very large web search engine query log. ACM SIGIR Forum 33(1):6–12
15. Tang C, Xu Z, Mahalingam M (2003) pSearch: information retrieval in structured overlays. ACM SIGCOMM Comput Commun Rev 33(1):89–94
16. Tryfonopoulos C, Idreos S, Koubarakis M (2005) LibraRing: an architecture for distributed digital libraries based on DHTs. Proceedings on the 9th European conference on research and advanced techonology for digital libraries. 18–25 Sept 2005
17. Yuan C, Chen Y, Zhang Z (2003) Evaluation of edge caching/offloading for dynamic content delivery. Proceedings of the WWW2003 Budapest, Hungary, 20–24 May 2003

Author Biographies

Daze Wang received a B.E. degree from University of Science and Technology in China., and a MSc. Degree from The university of Sydney. The work was done when he was doing Master of Science at School of Information Technologies, the University of Sydney.

Quincy Chi Kwan Tse is currently a PhD student at the University of Sydney supported by NICTA. He is currently researching in the area of intelligent transportation systems, specifically looking at vehicular communications for safety and road management systems.

Ying Zhou is currently a lecturer at school of Information Technologies, the university of Sydney, Australia. Her research interests include Web information retrieval, Web community and recommendation systems.