Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks

Chunqiang Tang*
Dept. of Computer Science
Univ. of Rochester
Rochester, NY 14627-0226
sarrmor@cs.rochester.edu

Zhichen Xu
HP Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304-1126
zhichen@hpl.hp.com

Sandhya Dwarkadas
Dept. of Computer Science
Univ. of Rochester
Rochester, NY 14627-0226
sandhya@cs.rochester.edu

ABSTRACT

Content-based full-text search is a challenging problem in Peer-to-Peer (P2P) systems. Traditional approaches have either been centralized or use flooding to ensure accuracy of the results returned. In this paper, we present pSearch, a decentralized non-flooding P2P information retrieval system. pSearch distributes document indices through the P2P network based on document semantics generated by Latent Semantic Indexing (LSI). The search cost (in terms of different nodes searched and data transmitted) for a given query is thereby reduced, since the indices of semantically related documents are likely to be co-located in the network. We also describe techniques that help distribute the indices more evenly across the nodes, and further reduce the number of nodes accessed using appropriate index distribution as well as using index samples and recently processed queries to guide the search. Experiments show that pSearch can achieve performance comparable to centralized information retrieval systems by searching only a small number of nodes. For a system with 128,000 nodes and 528,543 documents (from news, magazines, etc.), pSearch searches only 19 nodes and transmits only 95.5KB data during the search, whereas the top 15 documents returned by pSearch and LSI have a 91.7% intersection.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Management, Performance, Design, Experimentation

Keywords

Peer-to-Peer System, Information Retrieval, Overlay Network

1. INTRODUCTION

According to a recent report [17], 93% of information produced worldwide is in digital form. The unique data added each year exceeds one exabyte (or 10^{18} bytes) and is estimated to grow exponentially. This trend calls for equally scalable infrastructures capable of indexing and searching rich content such as HTML, plain text, music, and image files. Peer-to-Peer (P2P) systems, on the other hand, are gaining popularity quickly due to their scalability, fault-tolerance, and self-organizing nature, raising hope for building large-scale information retrieval (IR) systems at low cost [15].

Search engines such as Google appear to be scalable for Web content, but little is known to the public about how these systems actually work. In this paper, we describe techniques to build a self-organizing search engine based on P2P technology, which naturally inherits many of the nice P2P properties—scalability, fault-tolerance, low maintenance cost, etc. The fundamentals of our system are applicable to both well-managed stable environments (e.g., Google-like search engines, data centers, and corporations) and the more dynamic P2P environment.

Although a number of P2P search techniques [16, 3, 20, 5, 22, 10] have already been proposed in recent years, with very few exceptions [6], most of them are based on simple keyword matching, ignoring advanced relevance ranking algorithms devised by the IR community through decades of refinement and evaluation [18]. Without effective ranking, queries consisting of popular words may return a superfluous number of documents that are beyond the user's capability to handle.

We focus in this paper on studying the feasibility of extending classical IR algorithms to work in the P2P environment. Some IR techniques (e.g., Google's PageRank) leverage hyperlinks to identify important Web pages. This cross-reference information, however, does not exist in many digital content. Therefore, we will start with the most popular and well-studied statistical IR algorithms, vector space model (VSM) and latent semantic indexing (LSI) [1, 7], which do not rely on the cross-reference information. VSM and LSI represent documents and queries as vectors in a Cartesian space, and measure the similarity between a query and a document as the cosine of the angle between their vector representations. According to [28, 18, 13], variants of VSM and LSI have been adopted by major search engines such as Excite. In practice, various IR techniques are combined to build pragmatic search engines. The study of how other techniques (e.g., PageRank) can complement our approach is a subject of future work.

The fundamental problem that makes search in existing P2P systems difficult is that, with respect to semantics, documents are randomly distributed. Given a query, the system either has to search a large number of nodes or the user runs a high risk of missing relevant documents. To address this problem, we introduce the notion of *semantic overlay*, a logical network where contents are organized around their semantics such that the distance (e.g., rout-

*This work was started during Chunqiang's internship at HP Labs in 2002 and was completed at University of Rochester in 2003.

SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany.

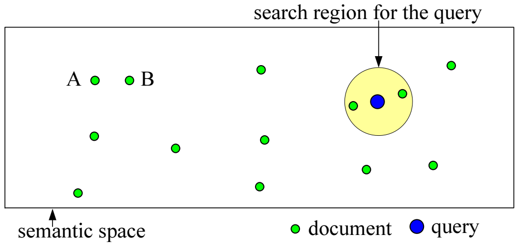

Figure 1: Search in a semantic space.

ing hops) between two documents in the network is proportional to their dissimilarity in semantics. The document semantics is produced using LSI.

Content-addressable networks (CANs) [19] provide a distributed hash table (DHT) abstraction over a Cartesian space. They allow efficient storage and retrieval of `(key, object)` pairs. An object key is a point in the Cartesian space. We use a CAN to create a semantic overlay by using the semantic vector (generated by LSI) of a document as the key to store the document index in the CAN.

Figure 1 illustrates how a semantic overlay can benefit searches. When a document's semantics is generated by LSI, each document is positioned as a point in the (semantic) Cartesian space. Documents close in the semantic space have similar contents, e.g., documents A and B. Each query can also be positioned in this semantic space. To find documents relevant to a query, we only need to compare the query against documents within a small region centered at the query, because the relevance of documents outside the region is relatively low. By doing this, the search space for the query is effectively limited and the accuracy is retained.

The basic idea of a semantic overlay is straightforward, and involves a mapping of the overlay to physical nodes in a CAN. It is, however, complicated by a number of factors. (1) We set the dimensionality of the CAN to be equal to that of LSI's semantic space, which typically ranges from 50 to 350. The "actual" dimensionality of the CAN, however, is much lower because there are not enough nodes to partition all the dimensions of a high-dimensional CAN. Along those unpartitioned dimensions, the search space is not reduced. (2) Semantic vectors are not uniformly distributed in the semantic space. A direct mapping from the semantic space to a CAN would result in unbalanced distribution of indices across the nodes. (3) Due to a problem known as the *curse of dimensionality*, it has been shown that limiting the search region in high-dimensional spaces is difficult [26].

We address these problems by leveraging the properties of the semantic space and trading accuracy for efficiency and/or storage overhead when necessary. Taking advantage of the higher importance of low-dimensional elements of semantic vectors, our *rolling-index* scheme partitions the semantic space along more dimensions by rotating the semantic vectors. Our *content-aware node bootstrapping* helps distribute the indices more evenly across nodes. Using samples of indices and recently processed queries to guide the search, our *content-directed search* algorithm substantially reduces the search region in the high-dimensional semantic space.

We have built a prototype P2P IR system called *pSearch* [24]. pSearch works by representing documents as vectors and organizing contents in a network around their vector representations. Although our experiments focus on full-text search, this method can also be applied to searching music and image files [9]. Our evaluation shows that pSearch can achieve performance comparable to centralized IR systems by searching only a small number of nodes. For a system with 128,000 nodes and 528,543 documents (from news, magazines, etc.), pSearch searches only 19 nodes and transmits only 95.5KB data during the search, whereas the top 15 documents returned by pSearch and LSI have a 91.7% intersection. Although our prototype implementation does not include some IR techniques proposed in recent years, our evaluation (in Section 7.4) suggests that pSearch has good potential to improve along with the future development of advanced IR techniques.

The remainder of the paper is organized as follows. Section 2 provides background information about IR and CAN. Section 3 gives an overview of pSearch and highlights the major challenges. Sections 4 to 6 describe our solutions to these challenges. Section 7 describes a prototype of pSearch and our experimental results. Related work is discussed in Section 8. Section 9 concludes the paper.

2. BACKGROUND

In pSearch, we use extensions to VSM and LSI [1, 7] to generate the semantic space, and use a CAN [19] to organize nodes into an overlay. In this section, we present an overview of these concepts in order to set the stage for a description of our algorithms.

2.1 Vector Space Model (VSM)

VSM represents documents and queries as *term vectors*. Each element of the vector corresponds to the importance of a term in the document or query. The weight of an element is often computed using the statistical *term frequency * inverse document frequency* (TF*IDF) scheme [1]. The intuition behind it is that two factors decide the importance of a term in a document—the frequency of the term in the document and the frequency of the term in other documents. If a term appears in a document with a high frequency, there is a good chance that the term could be used to differentiate the document from others. However, if the term also appears in many other documents, its importance should be penalized.

During a retrieval operation, documents are ranked according to the similarity between the document vector and the query vector, and those with the highest similarity are returned. A common measure of similarity is the cosine of the angle between vectors. Some VSM implementations normalize term vectors X to unit length ($|X| = 1$) in order to compensate for the difference in document length. Formally, given term vector $X = (x_1, x_2, \ldots, x_l)$ and $Y = (y_1, y_2, \ldots, y_l)$, the similarity between them is defined in Equation 1, where $\cos(X, Y)$ denotes the cosine of the angle between vector X and Y. Note that $|X| = 1$ and $|Y| = 1$ because they are already normalized. The similarity is therefore simply the inner product of the two vectors.

$$\cos(X, Y) = \frac{X \odot Y}{|X| \cdot |Y|} = \sum_{i=1}^{l} x_i y_i \tag{1}$$

2.2 Latent Semantic Indexing (LSI)

Literal matching schemes such as VSM suffer from synonyms and noise in documents. LSI overcomes these problems by using statistically derived conceptual indices instead of terms for retrieval. It uses singular value decomposition (SVD) [1] to transform a high-dimensional term vector (computed from VSM) into a lower-dimensional *semantic vector*, by projecting the former into a semantic subspace. Each element of a semantic vector corresponds to the importance of an abstract concept in the document or query. As in VSM, semantic vectors are normalized and their similarity is measured using Equation 1.

Let d denote the number of documents in a corpus, and t denote the number of terms in the vocabulary. VSM represents this corpus as a $t \times d$ matrix A, whose entry a_{ij} indicates the importance of

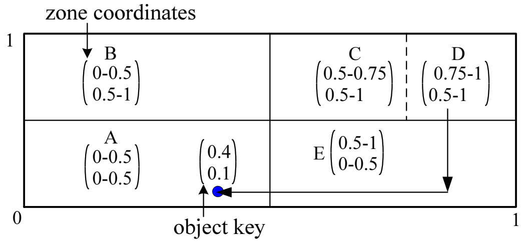

Figure 2: A 2-dimensional CAN.

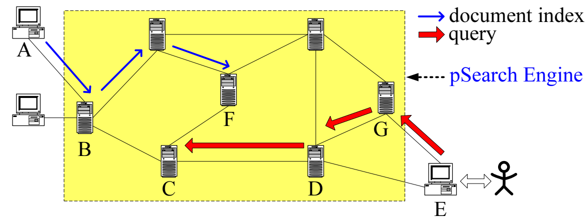

Figure 3: Overview of the pSearch system.

term i in document j. Suppose the rank of A is r. SVD decomposes A into the product of three matrices, $A = U\Sigma V^T$, where $U = (u_1, \ldots, u_r)$ is a $t \times r$ matrix, $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_r)$ is an $r \times r$ diagonal matrix, and $V = (v_1, \ldots, v_r)$ is a $d \times r$ matrix. σ_i's are A's singular values, $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$.

LSI approximates the matrix A of rank r with a matrix A_l of lower rank l by omitting all but the l largest singular values. Let $U_l = (u_1, \ldots, u_l)$, $\Sigma_l = \text{diag}(\sigma_1, \ldots, \sigma_l)$, and $V_l = (v_1, \ldots, v_l)$.

$$A_l = U_l \Sigma_l V_l^T \tag{2}$$

The rows of $V_l \Sigma_l$ are the semantic vectors for documents in the corpus. Given U_l, V_l, and Σ_l, the semantic vectors of queries, terms, or documents originally not in A can be generated by folding them into the semantic subspace [1].

By choosing an appropriate l for A_l, the important structure of the corpus is retained while the noise or variability in word usage (small σ_i) is eliminated. Former studies on LSI suggested setting l to a value between 50 and 350 and reported improvements over VSM by up to 30% in precision [7]. In addition, LSI is capable of bringing together documents that are semantically related even if they do not share terms, by learning from co-occurring word usage. For instance, a search about `car` may return relevant documents that actually use `automobile` in the text.

Raghavan [18] identified the costly SVD component to be one major scalability obstacle to LSI. In recent years, many efficient LSI variants have been proposed, e.g., concept indexing [12]. Parallel implementations of SVD also make this problem more tractable [14]. As an evidence of these advances, Excite has been reported to use a variant of LSI to index the Web [28].

In summary, LSI represents documents and queries as vectors (points) in a (semantic) Cartesian space. The similarity between a query and a document is measured as the cosine of the angle between their vector representations. Given a query as a point in the Cartesian space, the problem of finding the most relevant documents is reduced to locating the document points nearest to the query point. Therefore, the central issue in pSearch is to map the semantic space to nodes in a network and conduct efficient nearest-neighbor search in a decentralized manner.

2.3 Content-Addressable Network (CAN)

Recent overlay networks, such as CAN, Chord, Pastry, and Tapestry , offer an administration-free and fault-tolerant distributed hash table (DHT) that maps "keys" to "values". CAN partitions a d-dimensional Cartesian space into *zones* and assigns each zone to a node. An object `key` is a point in the Cartesian space and the object is stored at the node whose zone contains the point. Locating an object is reduced to routing to the node that hosts the object. Routing translates to traversing from one zone to another in the Cartesian space. A node join corresponds to randomly picking a point in the Cartesian space, routing to the zone that contains the point, and splitting the zone with its current owner.

An example CAN is shown in Figure 2. There are five nodes A-E in the overlay. Each node owns a zone in the Cartesian space. Initially C owns the entire zone at the upper-right corner. When D joins, the zone owned by C splits and part of the zone is given to D. When D wants to retrieve the object with key $(0.4, 0.1)$, it sends the request to E and E forwards the request to A.

3. OVERVIEW OF THE PSEARCH SYSTEM

In pSearch, a large number of machines are organized into a *semantic overlay* to offer the information retrieval service. Nodes in the overlay collectively form a *pSearch Engine*. Inside the Engine, nodes have completely homogeneous functions. A client intending to use pSearch connects to any Engine node to publish document indices or submit queries.

Figure 3 shows an example of how the system works. Node A publishes a document to node B inside the Engine. B builds an index for the document and routes the index in the overlay. The index is finally stored at node F based on its semantics. When a query is submitted to node E, the query is routed to node C based on the semantics of the query. C then takes the responsibility for finding relevant indices and returning them to E. In this example, C may return the index published by A and stored at F.

Not every node in a P2P system needs to be included in the pSearch Engine. We expect to construct the Engine using a subset of nodes that are stable and have good network connectivity. The boundary of the Engine is adjustable. When the load inside the Engine is high, more nodes can be recruited. A new Engine node finds its position in the overlay, takes over some indices stored at its neighbors, and starts to process queries. The entire process is completely autonomous. Indices stored on an Engine node are replicated on several of its neighbors. Should a node fail, one of its neighbors will take over its job seamlessly.

pSearch uses a CAN to organize Engine nodes into an overlay and uses an extension of LSI to answer queries. We call our algorithm pLSI. In the following, we first present a basic pLSI algorithm to outline our ideas and to highlight the challenges.

3.1 The Basic Algorithm

pLSI sets the dimensionality of the CAN to be equal to that of LSI's semantic space. The index for a document is stored in the CAN using its semantic vector as the key. Among other things, an index includes the semantic vector of a document and a reference (URL) to the document. Figure 4(a) illustrates the steps of pLSI.

1. When receiving a new document A, the Engine node generates its semantic vector V_a using LSI and uses V_a as the key to store the index in the CAN.
2. When receiving a query q, the Engine node generates its semantic vector V_q and routes the query in the overlay using V_q as the key.
3. Upon reaching the destination, the query is flooded to nodes within a radius r, determined by the similarity threshold or the number of wanted documents specified by the user.

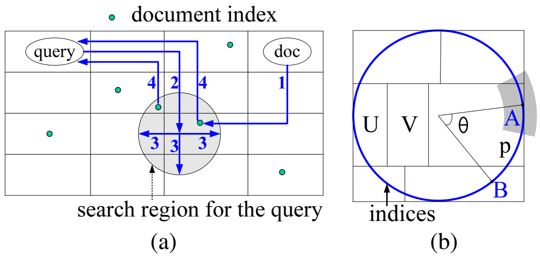

Figure 4: (a) pLSI in a 2-dimensional CAN. (b) Uneven distribution of document indices.

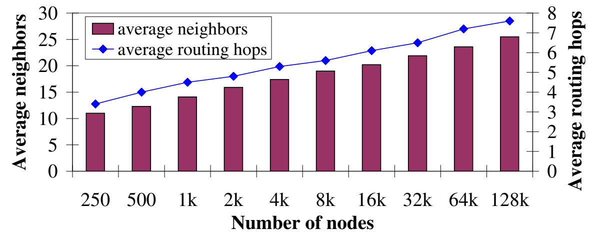

Figure 5: The average number of neighbors and routing hops for a 300-dimensional CAN.

4. All nodes that receive the query do a local search using LSI and reports the reference to the best matching documents back to the user.

Since indices of documents similar to the query (above a certain threshold) can be stored only within this radius r and we do an exhaustive search within this area, in theory, pLSI can achieve the same precision as LSI. Ideally, this radius r should be small such that only a small number of nodes are involved in a search.

During a search, the only data transmitted in pLSI are the query and references to the top documents, both of which are small and are independent of corpus size. As a comparison, P2P keyword-matching systems [15] maintain an inverted list for each term [1], whose contents are the ID of documents containing this term and the frequency of this term in each document. The inverted lists for different terms are distributed to different nodes. Answering queries containing multiple terms requires intersection of the corresponding lists, and therefore, communication. The size of the lists (and correspondingly, the communication) grows proportionally with the corpus size.

pLSI relies on some global statistics to function, including the inverse document frequency and the basis of the semantic space. Distributing this information to each Engine node allows the nodes to compute semantic vectors of new documents and queries independently. It has been demonstrated that IR systems do not need precise statistics to work well, i.e., a good approximation is sufficient [11]. In pSearch, a combining tree is used to sample documents, merge statistics, and disseminate the combined statistics. The statistics are versioned. Each node maintains the two most recent versions in order to ensure that a consistent set of indices can be extracted from a single version during the process of disseminating new statistics. More details on the generation and evolution of the statistics are described in an earlier technical report [24].

3.2 Major Challenges

The basic idea of pLSI is straightforward but there are several challenges to be overcome before it can work effectively.

Dimensionality mismatch between CAN and LSI. pLSI sets the dimensionality of the CAN to be equal to that of LSI's semantic space, which typically ranges from 50 to 350. The "actual" dimensionality of the CAN, however, is much lower because there are not enough nodes to partition all the dimensions of a high-dimensional CAN. Along those unpartitioned dimensions, the search space is not reduced.

Uneven distribution of indices. There are several reasons why using semantic vectors as keys to store indices in a CAN may lead to an imbalance in the distribution of these indices across the nodes. First, semantic vectors are normalized and reside on the surface of the unit sphere $\mathcal{S}$ in the semantic space. Figure 4 (b) shows an example of a 2-dimensional CAN. pLSI only places indices on the unit sphere. The similarity $\cos\theta$ between documents A and B is proportional to their distance p on the circle, since $\cos\theta = \cos p$ for a unit sphere. The gray area is the region for searching documents close to A in semantics. Nodes U and V own two zones of the same size, but V does not store any indices. Second, even if the key space $\mathcal{S}$ is uniformly distributed across the nodes, the system can still suffer from hot spots because the indices are not uniformly distributed on $\mathcal{S}$.

Large search region. Due to a problem known as the *curse of dimensionality*, it has been shown that limiting the search region in high-dimensional spaces is difficult [26].

We address these three challenges in Sections 4 to 6, respectively.

4. RESOLVING THE DIMENSIONALITY MISMATCH BETWEEN CAN AND LSI

In our algorithm, the dimensionality of the CAN is set to that of LSI's semantic space (l). Ratnasamy et al. suggest that for an l-dimensional CAN with n nodes, each node on average needs to maintain $2l$ neighbors, and the average length of routing paths is $(l/4)(n^{1/l})$ [19]. Since l can be as high as 300, this seems to indicate a problem in that each node will have a large number of neighbors. However, their result holds only if $l < \log_2(n)$. Since at least 2^x zones will be produced by partitioning along x dimensions, partitioning along more than $\log_2(n)$ dimensions will result in more zones than there are nodes available. Therefore, when $l \geq \log_2(n)$ and zones are partitioned "evenly", only $\log_2(n)$ dimensions will be partitioned and each node has only $\log_2(n)$ neighbors. Figure 5 shows the average number of neighbors and average routing hops for a 300-dimensional CAN, which can be seen to exhibit the relationship described above. We refer to the number of actually partitioned dimensions as a CAN's *effective dimensionality*.

While the limited number of nodes avoids the problem of an excessive number of neighbors, the result is that only the low dimensions of the semantic space are partitioned, making searches less efficient. The search space along the unpartitioned dimensions is not reduced since documents with similar semantic content along these dimensions will be spread across all nodes.

This situation is illustrated in Figure 6(a). Suppose the semantic space is of four dimensions, v_0-v_3. A query Q and a document A have semantic vectors $V_q = (0.55, -0.1, 0.6, -0.57)$ and $V_a = (-0.1, 0.55, 0.57, -0.6)$, respectively. The *similarity* between V_a and V_q is 0.574 (computed from Equation 1). The majority of the *similarity* is contributed from v_2 and v_3 . For real corpora, this *similarity* is usually high enough to consider A as relevant to Q. We store the two vectors in a 4-dimensional CAN. Since there are only four nodes w-z, the semantic space is partitioned only along v_0 and v_1. Because V_a and V_q are not similar in v_0 and v_1, a search in Figure 6(a) for Q would not find A unless all nodes are probed.

Before presenting a solution to this problem, we first make some high-level observations.

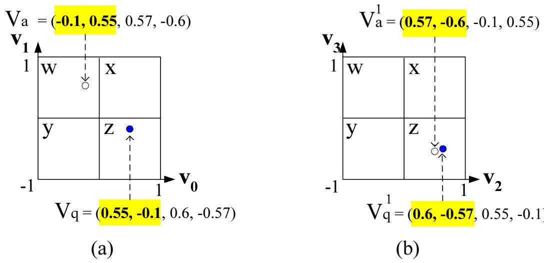

Figure 6: A rolling-index example. The position of a vector is decided by its first two elements. (a) The CAN partitions dimensions v_0 and v_1 in the original semantic space. (b) The same CAN partitions dimensions v_2 and v_3 after rotating the semantic vectors by two dimensions. The relevant document A for the query Q is found easily in node z on the rotated space.

- Although the dimensionality of the semantic space is high, in practice, the number of dimensions relevant to a particular document is much smaller. For example, concepts in chemistry are unlikely to appear in a computer science paper.
- Queries submitted to search engines are usually short, and are likely to be captured by a few concepts. As a result, only a small number of elements in the semantic vectors will contribute significantly to the *similarity*.
- SVD sorts elements in semantic vectors by decreasing importance. Figure 7(a) plots the singular values of the TREC corpus [25] (see Section 7 for more details), which corresponds to the importance of elements. The singular values largely follow a Zipf-like distribution, $\sigma_i = a \cdot i^b$, with a=190 and b=-0.3. Because of the importance of the low-dimensional elements, a significant fraction of the *similarity* is likely to be contributed by them.

Taking advantage of these facts, we propose the use of *rolling-index* to bridge the dimensionality gap and also to reduce the search space. The basic idea is to use a single CAN to partition more dimensions of the semantic space by rotating the semantic vectors.

4.1 Rolling-Index

Given a semantic vector, $V = (v_0, v_1, \ldots, v_l)$, we rotate it repeatedly by m dimensions each time to generate a series of new vectors (see Equation 3 and note that $V^0 = V$). We call these vectors *rotated semantic vectors*. We set m using Equation 4, where n is the number of nodes in the system. Equation 4 estimates the effective dimensionality of the CAN by approximating the *"average neighbors"* curve in Figure 5. Rotated vectors of different documents or queries generated with the same amount of rotation (the same i) define a *rotated space* i with $(v_{i\cdot m}, \ldots, v_{i\cdot m+m-1})$ as its m-dimensional *support subvector*.

$$V^i = (v_{i\cdot m}, \ldots, v_0, v_1, \ldots, v_{i\cdot m-1}),\ i = 0, \ldots, p-1 \quad (3)$$

$$m = 2.3 \cdot \ln(n) \quad (4)$$

Given a document A with semantic vector V_a, we store its index in p places in the CAN using V_a^i, $i = 0, \cdots, p-1$, as the keys. For a query with semantic vector V_q, we execute the pLSI algorithm in Figure 4(a) p times. Each time it uses a different V_q^i to route the query and guide searches in rotated space i. Each rotated space independently returns matching documents based on vectors on that space. Since the *similarity* between two semantic vectors is measured as their inner product (see Equation 1), rotation will not change the *similarity* measure. In a rotated space, documents close in the overlay are still close in semantics. Note that we still use full semantic vectors as the CAN keys. The *similarity* is also computed from full semantic vectors rather than support subvectors. An example of rolling-index with m=2 is shown in Figure 6(b).

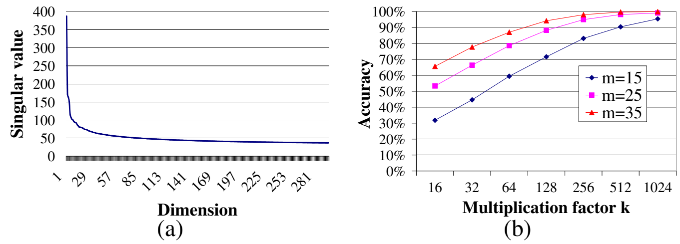

Figure 7: (a) Singular values of the TREC corpus. (b) The effect of using low-dimensional elements to identify relevant documents (documents and queries are from TREC).

4.2 Discussion

Rolling-index uses the same CAN to partition more dimensions of the semantic space, but at the increased storage cost (p times the base storage space). The low dimensions of each of multiple rotated spaces are partitioned onto a single CAN, which correspond to different dimensions in the original semantic space. Because of the importance of the low-dimensional elements, a significant fraction of documents are likely to be correctly (though not perfectly) clustered in the CAN by the first several support subvectors.

In order to demonstrate that the low-dimensional elements are often sufficient to identify relevant documents, we conducted an experiment with the TREC corpus. We first retrieved the 15 most relevant documents for each TREC query based on the similarity of the 300-dimensional semantic vectors. The results form set A. In each rotated space, we then retrieved $k \cdot 15$ documents for each query solely based on the similarity of the m-dimensional *support subvector* of that space (rather than the full semantic vector as in our rolling-index algorithm). Here, k is a constant multiplication factor. The results for the first four rotated spaces form set B. Figure 7(b) reports the average *accuracy* of set B with respect to set A, where accuracy $= \frac{|A \cap B|}{|A|} \times 100\%$. When $m = 25$ and $k = 128$, the size of B is only 1.3% of the corpus size, but it already covers almost 90% of the relevant documents in set A. In Section 6, we will introduce an algorithm that selectively searches documents in a big set B to achieve a high accuracy at low cost.

On the other hand, although the low-dimensional elements are statistically of higher importance for the entire corpus, for an individual document discussing unpopular concepts, some high-dimensional elements may carry heavy weight. For queries about these concepts, rolling-index will be less effective. A solution to this problem is *selective rotation*. Given a document, in addition to storing its index in the first p rotated spaces, we also store the index in the rotated spaces whose corresponding support subvectors cover the heavily-weighted elements that are not covered by the first p spaces. Likewise, given a query, it also searches some extra rotated spaces whose corresponding support subvectors cover the heavily-weighted elements of the query.

The dimensionality of the semantic space increases as the size of the corpus increases, which may also make rolling-index less effective. We propose using hierarchical document clustering to partition the document space into clusters and map each cluster on top of the same CAN [24]. We leave an evaluation of these improvements for future work.

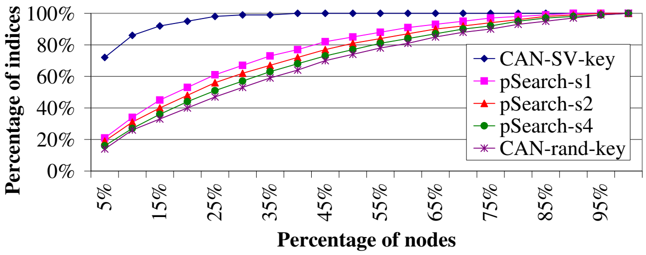

Figure 8: The effect of content-aware node bootstrapping.

5. BALANCING INDEX DISTRIBUTION

To cope with the uneven distribution of indices, we propose *content-aware node bootstrapping* to force the distribution of nodes in the CAN to follow the distribution of indices.

At node join, the node randomly picks a document that it is going to publish and computes the semantic vector of the document. This semantic vector is randomly rotated to a space i ($0 \leq i < p$) and the rotated semantic vector (instead of a random point suggested by [19]) is used as the point toward which the join request is routed. The node whose coordinates contains the rotated semantic vector splits in the middle along the lowest unpartitioned dimension and hands over half of its zone to the new node.

This bootstrapping process has three effects.

- *More balanced index distribution.* A larger number of nodes will be used in areas in the semantic space that have dense document population.
- *Index locality.* Assuming the documents published by a node have similar semantics, on space i, indices of a node's documents are likely to be published on itself or its neighbors.
- *Query locality.* Assuming the documents published by a node are good indications of the user's interests. Queries submitted by the user would usually result in neighboring nodes to where the query is submitted.

Figure 8 evaluates this load-balancing technique by distributing the TREC corpus to a 10,000-node 300-dimensional CAN. Nodes are sorted in decreasing order according to the number of indices they store. We draw the node percentage on the X-axis. The Y-axis gives the percentage of indices owned by corresponding nodes. The *CAN-rand-key* series are the original CAN proposal where nodes and indices are randomly populated. This serves as the baseline for comparison. The *CAN-SV-key* series use random points for node bootstrapping but indices are stored using semantic vectors as keys. *pSearch-*s_p uses our content-aware bootstrapping, and indices are stored under semantic vectors. Here p is the number of rotated semantic spaces. Note that even the load for *CAN-rand-key* is not completely balanced due to the randomness. With a larger corpus, the load is expected to become more balanced.

As can be seen from this figure, without load balancing, 5% of the nodes store 72% of the indices (the curve on the top). The load-balancing technique is effective even if only a single rotated space is used. Increasing the number of rotated spaces can further balance the index distribution, because indices on different rotated spaces compliment each other.

This bootstrapping process with multiple rotated spaces does not adversely affect the neighbor distribution and routing performance of the overlay. The CAN evaluated in Figure 5 uses this bootstrapping with four rotated spaces.

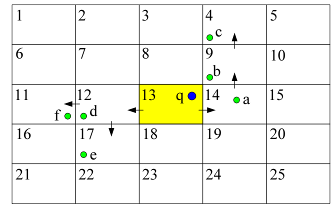

Figure 9: An example of content-directed search.

6. REDUCING THE SEARCH SPACE

Rolling-index clusters indices in the overlay based on their semantics, making it possible to find the most relevant documents to a query by searching a fraction of the nodes in the overlay. However, existing *centralized* index structures [26] used to limit search space for multidimensional data usually work well for low-dimensional data, but the search space grows quickly as the dimensionality of the data increases. This is known as the *curse of dimensionality*.

Weber et al. listed several interesting observations about high-dimensional spaces [26]. We summarize two relevant observations here. (1) High-dimensional data spaces are sparsely populated. (2) The distance between a query and its nearest neighbor grows steadily with the dimensionality of the space.

6.1 Content-Directed Search

Because of the large nearest-neighbor distance, a naive nearest-neighbor search will not work well unless a large number of nodes are searched. To solve this problem, we use the contents (indices) stored on nodes and the recently processed queries to guide searches to the "right" nodes. In the sparse high-dimensional semantic space, documents usually form tight clusters (see the top curve in Figure 8). If a relevant document is found, this document is likely to be surrounded by other relevant documents.

We illustrate the basic ideas with an example in Figure 9. 1-25 are IDs of nodes in the two-dimensional CAN. a-f are semantic vectors of documents and q is the semantic vector of a query. The user wants to retrieve three documents relevant to q. Based on Euclidean distance, a, b, and c should be the search results.

The search starts at node 13, whose coordinates contains q. No document is found on node 13. We maintain a queue $\mathcal{N}$ that contains the candidate nodes yet to search. After searching node 13, its routing neighbors are added into this queue, $\mathcal{N} = \{8, 12, 14, 18\}$. In the background, each node samples indices stored on its neighbors. These samples are used to decide the search order for nodes in $\mathcal{N}$. In this example, the samples from node 14 are similar to the query. We search node 14 next and find document a. Node 14's neighbors are added into $\mathcal{N}$, $\mathcal{N} = \{8, 12, 18, \mathbf{9}, \mathbf{15}, \mathbf{19}\}$. Likewise, we choose node 9 to search because its samples are similar to the query. Document b is found and node 9's neighbors are added into $\mathcal{N}$, $\mathcal{N} = \{8, 12, 18, 15, 19, \mathbf{4}, \mathbf{10}\}$, which leads us to search node 4 and find document c. After that, nodes 12, 11, and 17 are searched in turn but no better matching documents are found. At this point, the chance of finding documents better than a, b and c is low. The search is terminated.

Although the example used above assumes that nodes are searched sequentially, the speed of the search process can be improved by accessing nodes in parallel. The tradeoff is in terms of the number of nodes searched unnecessarily.

This search algorithm takes advantage of the processing power of a large number of nodes to pre-process (sample) the semantic space in the background to enable efficient search. The use of sampled *full* semantic vectors to direct searches also removes some inaccuracies due to the limited clustering capability of the support subvectors used in rolling-index.

6.2 Description of the Search Algorithm

We proceed to give a more formal description of our search algorithm. We use $X[Z]$ ($X[Z,Y]$) to denote an attribute X of node Z (with Y as an additional parameter). For instance, $D[Z]$ is the set of semantic vectors in indices stored on node Z, and $Q[Z]$ is the set of semantic vectors of queries recently processed by Z. A node Z stores indices and processes queries for multiple rotated spaces. $D^i[Z]$ and $Q^i[Z]$ refer to only those parts relevant to space i.

For a node Z, we use a single vector $U^i[Z]$ computed from Equations 5 and 6 to summarize the indices stored on Z and the queries recently processed by Z. Equation 6 normalizes H such that $U^i[Z]$ is of unit length. $U^i[Z]$ is the centroid (center of mass) of indices in $D^i[Z]$ and queries in $Q^i[Z]$.

$$H = \sum_{d \in D^i[Z]} d + \sum_{c \in Q^i[Z]} c \tag{5}$$

$$U^i[Z] = \frac{H}{|H|} \tag{6}$$

In the background, Z requests each of its neighbors P to return k_c samples of semantic vectors that are in indices stored on P and have the highest similarity to the summary vector $U^i[Z]$. Z also requests k_r random samples from P. In our current implementation, $k_c = 0.8s$ and $k_r = 0.2s$, where s is a constant. These returned semantic vectors are stored on Z as a sample set $S^i[Z,P]$, which is used as an estimation of indices stored on P. When Z is under search for a query q (with rotated semantic vectors V_q^i), Z uses Equation 7 to estimate the highest similarity between the query vector V_q^i and semantic vectors in indices stored on P, based on the sample set $S^i[Z,P]$.

$$e^i[P, V_q^i] = \max_{d \in S^i[Z,P]} \cos(d, V_q^i) \tag{7}$$

The metric in Equation 7 is used to direct the search on each rotated space. It chooses the nodes whose sampled indices have high similarity to the query (i.e., high $e^i[P, V_q^i]$ value) to search first and stops when no better matching document is found during the most recent T node visits. Let Z_q^i denote the node whose coordinates contains the rotated query vector V_q^i. Node Z_q^0 (on space 0) acts as the coordination center during the search. It maintains a queue of candidate nodes yet to search ($\mathcal{N}$) and a queue of indices of identified relevant documents ($\mathcal{R}$). On each rotated space, the search starts from node Z_q^i. A node under search returns the estimated similarity ($e^i[P, V_q^i]$) of its neighbors P and the indices of the best matching documents found locally to node Z_q^0. Z_q^0 adds the returned indices and the neighbor nodes P into the index queue $\mathcal{R}$ and the node queue $\mathcal{N}$, respectively. Z_q^0 makes the decision on which node(s) in $\mathcal{N}$ to search next based on the estimated similarity ($e^i[P, V_q^i]$) returned from the searched nodes.

Recall that the search starts from node Z_q^i on each rotated space. For a node Z that is visited during the search, there exists a path that leads the search to Z from Z_q^i and all nodes on the path are also visited during the search. Let $r[Z, Z_q^i]$ denote the hops of the path (or the number of nodes on the path). The quit threshold T is dynamically computed from

$$T = \max(5, F - 5 * i) * 0.8^w \tag{8}$$

$$w = \min_{Z \in \mathcal{N}} r[Z, Z_q^i] \tag{9}$$

where quit bound F is a constant set by the user or a default system value, i is the ID of the rotated space that is under search, and w is the smallest hops to reach nodes still in the node queue $\mathcal{N}$ from node Z_q^i (note that, once searched, a node is removed from $\mathcal{N}$).

Two components decide the quit threshold T. The first component, $\max(5, F - 5 * i)$, decreases as the space ID i increases. The intuition is that on lower spaces we want to search more nodes because of the importance of the low-dimensional elements. To guarantee that at least some searches are performed on high spaces, this component is no less than 5. The second component, 0.8^w, monotonically decreases during the search. The intuition is that the quit threshold should get tighter after the near neighbors of Z_q^i have already been searched.

Multiple nodes are searched concurrently under three rules. (1) Nodes Z_q^i ($i = 0, \ldots, p-1$) are searched in parallel. (2) The direct routing neighbors of Z_q^0 are always searched and are searched in parallel, because of the importance of the low-dimensional elements. (3) In addition to the first two rules, in each round we select the top b nodes from the node queue $\mathcal{N}$ to search in parallel. b is decided using Equation 10, where T is the quit threshold in Equation 8, and d is a dynamic "concurrency factor". Designing an algorithm to automatically fine tune d is a subject of future work.

$$b = \min(d, T/2) \tag{10}$$

In addition, at the cost of extra storage space, our current implementation allows a node to replicate its neighbors' indices and to process queries on their behalf, in order to reduce the number of visited nodes and data transmitted during a search. In the future, the neighboring-content sampling process can be extended to implement *selective* index replication. A node can use Equations 5 and 6 to compute a vector to represent itself and only replicate its neighbors' indices whose similarity to this vector is beyond a threshold. Selective replication has the potential to reduce the amount of replicated indices while achieving performance similar to simple replication. We leave an evaluation of this for future work.

7. EXPERIMENTAL RESULTS

We built a pSearch prototype to validate our algorithms. We implemented an overlay simulator based on CAN. [1] Cornell's SMART system (v11.0) [2] implements VSM. We extended SMART with several modules and tools to implement LSI. We used LAS2 in the SVDPACK [23] package to compute the SVD of large sparse matrices. We then linked SMART with the CAN simulator and implemented the pLSI algorithms to build a pSearch prototype. We validated the correctness of our LSI implementation using the MED, ADI, and CRAN corpora [2]. The precision is consistent with that reported in the literature [7].

7.1 Experimental Setup

We experimented with the Text Retrieval Conference (TREC-7 and TREC-8) corpus [25], one of the largest corpora available in the public domain and widely used in IR researches. It includes 528,543 documents from news, magazines, etc., with a total size of about 2GB. Topics 351-450 are used as queries.

We used SMART to index the TREC corpus. The entire document or query content is indexed with the "atz" term weighting scheme [2]. The SMART stop word list is used as is. The SMART

[1] In practice, we can employ eCAN [27], a hierarchical version of CAN, to improve the routing performance, while retaining CAN's Cartesian space abstraction.

[2] "atz" differs from the classical "atc" [2] in the vector normalization step. During normalization, "atz" divides each element of the term vector by $\sqrt{\sum_{i=0}^{20} \frac{tf_i^2}{i+1}}$, where tf_i is the i-th heaviest-weight element in the vector before normalization.

	description	default value
n	number of nodes in the system	10,000
l	dimensionality of LSI and CAN	300
p	number of rotated semantic spaces	4
m	rotated dimensions	use Equation 4
s	size of the sample set $S^i[Z, P]$	50
F	quit bound in Equation 8	24
k	number of returned documents for a query	15
g	number of warm-up queries (size of $Q^i[Z]$)	0
d	the concurrent-search factor in Equation 10	1

Table 1: Parameters varied in experiments.

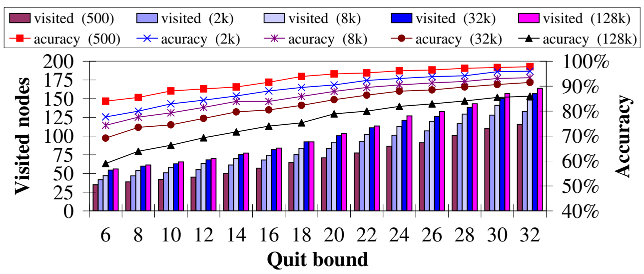

Figure 10: The effect of varying system size.

stemmer is used without modification to strip word endings. The indexing process takes about 20 minutes to complete on a 1.7GHz Pentium IV machine with 1GB of memory. We randomly sampled 15% of the indexed documents to generate a term-by-document matrix. Terms appearing in only one sampled document are not included in the matrix. This leaves us with 79,316 sampled documents and 83,098 indexed terms in the matrix. We applied SVD to this matrix to compute the basis of the semantic space. The SVD computation takes about 30 minutes. Using this basis, we project *all* 528,543 documents into the semantic space, computing a semantic vector for each document. This takes about 7 minutes. Note that most of the above process can be done by pSearch Engine nodes concurrently. (Our LSI configuration largely follows [8].)

The main metrics we use are the number of *visited nodes* during a search and the *accuracy* of search results. System resource consumption, which is proportional to the number of visited nodes, is analyzed in Section 7.5. For each configuration, we use LSI to retrieve a fixed number of documents for a query. The returned documents form set A. We refer to documents in A as *relevant documents*. We then use pLSI to retrieve the *same* number of documents for the same query. The returned documents form set B. The accuracy is defined as follows.

$$\text{Accuracy} = \frac{|A \cap B|}{|A|} \times 100\% \qquad (11)$$

The *accuracy* metric compares pLSI against the centralized LSI baseline. A discussion on pLSI's absolute performance can be found in Section 7.4.

7.2 Evaluation of Accuracy vs. Number of Nodes Visited

Table 1 shows the parameters we vary in our experiments and their default values. Unless otherwise noted, our experiments use these default values without index replication. Our default baseline uses rolling-index with four rotated semantic spaces. While rolling-index does help reduce the number of visited nodes for a

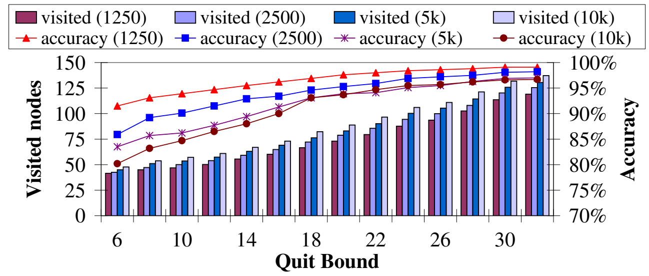

Figure 11: The effect of simultaneously varying system and corpus size.

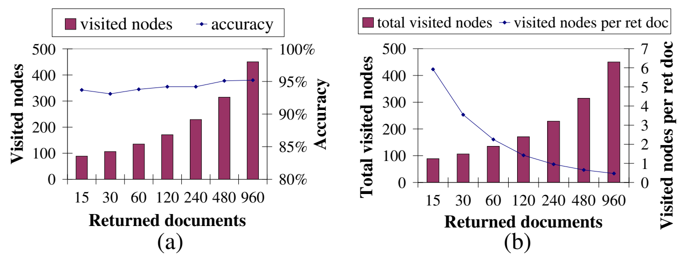

Figure 12: The effect of varying the number of returned documents for a 10k-node system. (a) Visited nodes and accuracy. (b) Visited nodes per returned document.

given accuracy, this number can still be high. Our baseline therefore combines rolling-index with content-directed search using a default of 50 sample indices from neighboring nodes.

7.2.1 Effect of varying system size, corpus size, and number of returned documents

Figure 10 shows the effect on the number of visited nodes and the accuracy of the search results when varying the number of nodes. The results are averaged over 100 queries. The only non-default parameter is s (the number of samples). With 500 nodes, we set s to 150. We decrease s by a factor of two each time the number of nodes quadruples because the total number of documents is fixed and each node can sample a larger percentage of its neighbors' contents when the indices are spread over a larger number of nodes.

From the figure, we can observe that as the system size increases exponentially, the number of visited nodes only increases moderately. This is because the number of neighbors of a node is one deciding factor for the number of visited nodes, which increases logarithmically with the system size. Even for the 32k-node system, pSearch can achieve an accuracy of 90% by visiting just 139 nodes. For the 128k-node system, the accuracy is about 86%. We will show how to improve it by varying other parameters. Second, as we relax the quit bound F that controls the accuracy, the accuracy improves only slowly with the increase in the number of visited nodes, suggesting that search results can be returned to users incrementally without waiting for it to reach the final quit bound.

Unlike Figure 10 that only varies the number of nodes, Figure 11 varies the number of nodes and the size of the corpus proportionally, i.e., full TREC corpus for 10k nodes, half TREC corpus for 5k nodes, etc. The search cost in this figure increases moderately as system size and corpus size scale.

Figure 12(a) shows the effect of varying the number of returned documents for each query. Although it seems that the number of visited nodes grows quickly while the accuracy remains the same,

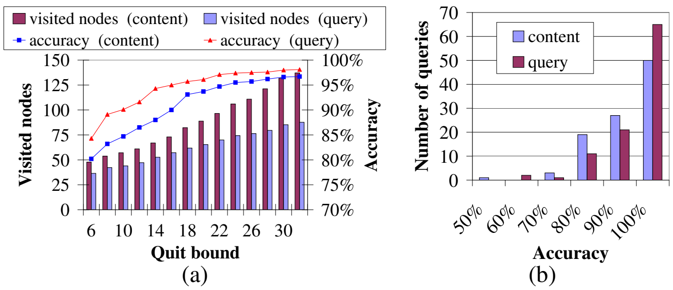

Figure 13: Comparing content-directed search heuristics for a 10k-node system. (a) Accuracy and visited nodes. (b) Accuracy histogram.

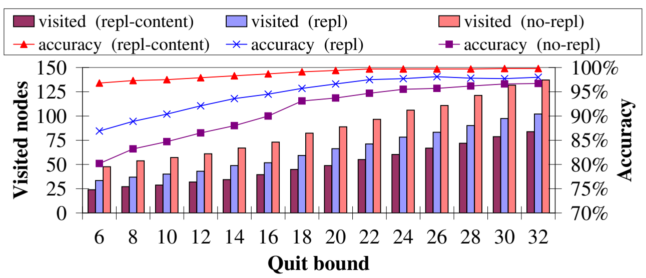

Figure 14: The effect of replication on a 10k-node system.

the average number of nodes that need to be searched to return one relevant document decreases drastically as we increase the number of returned documents. This is illustrated in Figure 12(b). When the user requests 15 documents, on average 5.9 nodes need to be searched to find one relevant document. When the number of returned documents increases to 960, on average only 0.47 nodes need to be searched to find one relevant document. It has been reported that a significant percentage of users only view the top 10 search results [13]. We believe that using 15 as the default number of returned documents is appropriate.

7.2.2 Effect of varying content-directed search heuristics

We next evaluate the heuristics in the content-directed search. The results for a 10k-node system are shown in Figure 13(a). For the *content* series, g=0 and p=4; for the *query* series, g=5000 and p=2. The *content* heuristic uses only the node contents to direct searches. The *query* heuristic warms up the system by processing 500,000 queries before measuring the performance. It then uses both node contents and past queries to direct searches. When queries have locality, learning from past history can increase the accuracy by up to 5.9% while reducing the number of visited nodes, due to a more selective sampling process for neighboring contents.

Figure 13(b) shows the accuracy histogram of the 100 queries measured in the experiment. x% on the X axis means the number of queries whose accuracy is between x% and (x+10)%. Compared with the *content* heuristic, the *query* heuristic improves the accuracy of about 15% of the queries from $[90\% - 100\%)$ to 100%.

7.2.3 Effect of replication

All the above results are achieved without replication. Replication can improve both the accuracy and the efficiency (see Figure 14). The *no-repl* series with no replication serve as the baseline. In the *repl* series, each node replicates its direct neighbors' contents. In the *repl-content* series, besides replicating neighbor

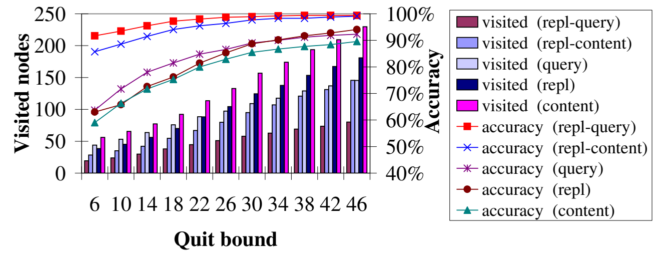

Figure 15: Performance of a 128k-node system.

P's contents, a node Z also replicates the samples that P keeps for P's neighbors. Like the *content* series in Figure 13, we set g=0 when doing the sampling. This figure suggests that when replication is used, a small number of rotated spaces and a tight quit bound already achieves a good accuracy. For instance, it visits only 24 nodes in a 10k-node system to achieve an accuracy of 96.8%. This is because search of the replicated nodes is avoided and searches are directed to the right places more accurately.

7.2.4 Results on a larger system size

Figure 15 presents the performance of a 128k-node system. The configurations for the *content* and *query* series are the same as those in Figure 13. The *repl* and *repl-content* are the same as those in Figure 14. *repl-query* differs from *repl-content* in that it uses the *query* heuristic for sampling, with g=5000 and p=2. As can been seen in the figure, the accuracy of the *content* series can approach close to 90% by relaxing the quit bound while only 0.2% of the nodes are visited. Combining replication and the *query* heuristic can achieve an accuracy of 91.7% by visiting only 19 nodes, or an accuracy of 98% by visiting 45 nodes.

It must be acknowledged that this example distributes a relatively small corpus over a large system. However, the results in Figure 11 show that pLSI can retain good performance as the corpus size and system size scale proportionally. As the number of indices per node increases, the chance to find more relevant documents on a single node should also increase, which would actually reduce the number of visited nodes.

7.3 Analysis of Result Source Distribution

We now try to understand the distribution of retrieved documents across the nodes in the system. The percentage of relevant documents retrieved from each rotated space is shown in Figure 16(a). The left Y-axis is the percentage of relevant documents found on each rotated space. The right Y-axis is the percentage of nodes visited in each rotated space out of all visited nodes (rather than the total number of nodes in the system). For the *content* heuristic, about 75.8% of relevant documents are found on the first rotated space. For the *query* heuristic, this number goes up to 92.3%.

The important message here is that although the optimal dimensionality of LSI is between 50-350, a large fraction of documents can be correctly (though not perfectly) clustered in the overlay by the low-dimensional elements. This property, which is a result of SVD (see Figure 7), makes search in the semantic space much easier than that in other high-dimensional spaces. Using the sampled *full* semantic vectors to direct searches, the *content-directed search* algorithm also helps remove some inaccuracies due to the limited clustering capability of the low-dimensional elements.

Figure 16(b) shows the percentage of relevant documents found and nodes visited at different hops away from Z_q^i. Here the hop count corresponds to $r[Z, Z_q^i]$. $r[Z, Z_q^i]$ measures how many steps it takes to reach Z from Z_q^i during the search, which could be big-

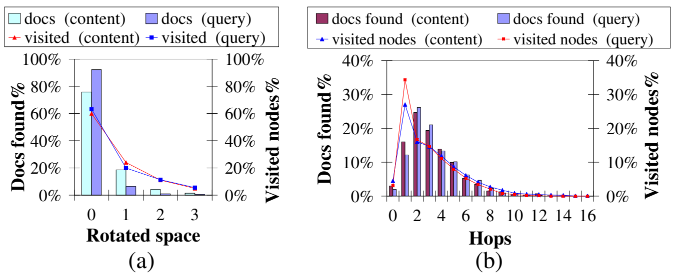

Figure 16: Distribution of documents found for a 10k-node system. (a) Rotated spaces. (b) Hop counts.

ger than the shortest route between Z and Z_q^i. In this figure, the *visited nodes* curve fits well with the *docs found* curve, meaning that our content-directed heuristics are effective at directing searches to the right places even if it is already several hops away from Z_q^i. The *query* heuristic has a longer tail than the *content* heuristic, showing that the *query* heuristic is more accurate at directing searches.

7.4 Sensitivity to System Parameters

The final set of experiments evaluate the sensitivity of pLSI to the underlying VSM baseline and system parameters.

Term weighting schemes. LSI is a proposal to improve VSM and is built on top of VSM. VSM produces the sampled term-by-document matrix from which the basis of the semantic space is computed. Therefore, LSI's precision and recall are tied to the underlying VSM baseline [8]. Our VSM baseline is SMART 11.0, which unfortunately does not include some important IR techniques proposed in recent years, e.g., Okapi term weighting and automatic relevance feedback (we are in the process of implementing them). Consequently, our LSI baseline is inferior to the top systems reported in recent TREC's.

We believe that the on-going efforts in the IR community to improve the performance of the baseline are orthogonal to our efforts to make the decentralized implementation close to the centralized baseline. Figure 17(a) supports our claim by showing pLSI's performance with different underlying VSM baselines (see [2] for the term weighting schemes in parentheses). Regardless of the *absolute* performance of the underlying VSM baselines, pLSI consistently achieves high *relative* accuracies at reasonable cost, showing great promise that pLSI can improve along with future developments of advanced VSM baselines.

Query length. The sensitivity to the number of query terms is reported in Figure 17(b). Each TREC query consists of three parts—title, description, and narrative. The *all* series use all three parts as the query, which is our default configuration; the *title+desc* series use the title and description; the *title* series use the title only. On average, a query in the *all*, *title+desc*, and *title* series contains 21, 7.8, and 2.4 terms, respectively. Overall, pLSI's *relative* accuracy is not very sensitive to the number of query terms.

Parallel search. In Figure 17(c) we vary the concurrent-search factor d (the numbers in parentheses), which specifies the number of nodes searched in parallel. The baseline is the *content* heuristic (d=1) in Figure 13(a). As can be seen from the figure, increasing the search concurrency only increases the number of visited nodes moderately, while speeding up the search process by nearly a factor of d. With an algorithm to fine-tune d dynamically, we expect to achieve even better speedup at lower cost.

Rotated dimensions. The sensitivity to the number of dimensions m by which each space is rotated is shown in Figure 17(d). pLSI is not sensitive to m so long as it is larger than a certain threshold (e.g., 10). This is because the first 40 important dimensions of the semantic space are already partitioned by the four rotated spaces in use, and the *content-directed search* algorithm helps remove some inaccuracies due to the use of rolling-index.

Dimensionality of the semantic space. Figure 17(e) shows the effect of changing the dimensionality of the semantic space l (the numbers in parentheses). pLSI is not very sensitive to l. The suggested value for l is 50-350, but we expect it to increase as the corpus grows. This insensitivity to l suggests from another angle that pLSI has good potential to scale with corpus size.

Sample size. Lastly, we vary s, the size of the sample set that is used to guide the search (see Figure 17(f)). To leave room for the accuracy to improve as s varies, we set the quit bound F and the number of rotated spaces p differently for different configurations: for *content*, $F = 20$ and $p = 4$; for *query*, $F = 10$ and $p = 2$; for *repl-content* and *repl-query*, $F = 6$ and $p = 2$. From the figure, we can see that the *query* heuristics are insensitive to the number of samples due to its more accurate index sampling process. The *content* heuristics are more sensitive to s when s is small.

7.5 Analysis of System Resource Usage

In this section, we analyze the storage and network resource consumption based on the experimental results. When publishing the index of a document, the data transmitted (B_d) and storage cost (S) are given by

$$B_d = S_I \cdot p \cdot h + S_I \cdot R \tag{12}$$

$$S = S_I \cdot p \cdot R \tag{13}$$

where S_I is the size of the index, p is the number of rotated semantic spaces, h is the average routing hops in the CAN, and R is the number of replicas of the index. The total data transmitted for processing a query is given by

$$B_q = S_Q \cdot p \cdot h + v \cdot (S_Q + S_R) \tag{14}$$

where S_Q is the size of the query message, v is the number of visited nodes, S_R is the size of returned data from a visited node.

We set the variables in the above equations based on the most pessimistic results for the 128k-node system: S_I=S_Q=1.5KB, p=4, h=8, R=1 (no replication) or R=25 (with replication), v = 230, and S_R = 1KB. The index and query are 1.5KB because they consist of a 300-dimensional semantic vector (1200B) and some small metadata. The data returned from each visited node contain the similarity and reference to the top documents, and the estimated similarity ($e^i[P, V_q^i]$) of the neighbors, all of which are independent of corpus size, query length, and document length.

For this pessimistic setting, the data transmitted for processing a query is 632KB. Without replication, the data transmitted for publishing one index is 49.5KB and and the storage cost is 6KB; with replication, these numbers are 85.5KB and 159KB, respectively.

One big advantage of pSearch over P2P keyword-matching systems [15] and Gnutella-style query-flooding systems is that its bandwidth consumption for processing a query is independent of corpus size, query length, and document length. The factors that decide the resource usage are either constant or increase slowly as the system scales, e.g., the routing hops h and the number of visited nodes v. On the other hand, it will take Gnutella-style systems to transmit 192MB data to just flood a query to 128k nodes.

On each visited node, computing the relevant documents can be done efficiently. When indices are in memory, one node in our pSearch prototype can process a query against 0.9 million indices per second. When indices are on disk, it can process a query against 0.1 million indices per second.

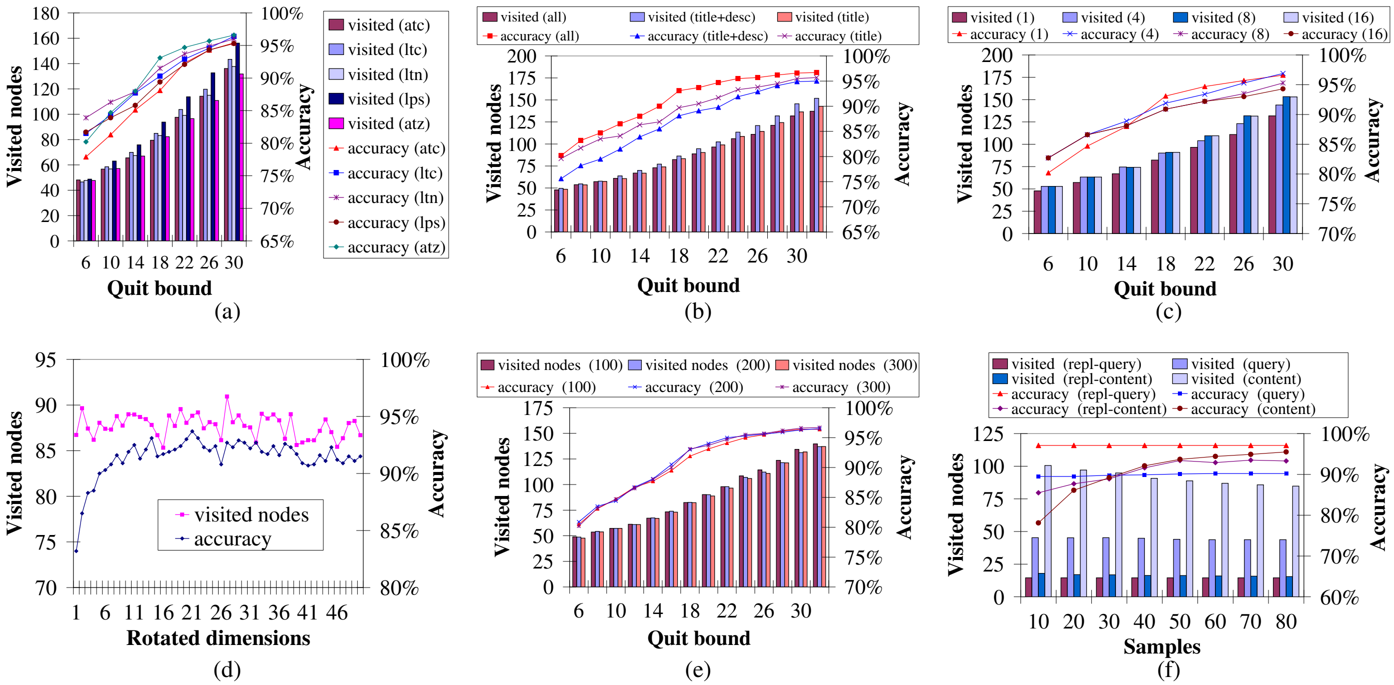

Figure 17: Sensitivity to system parameters for a 10k-node system. (a) Term weighting schemes. (b) Query length. (c) Parallel search. (d) Rotated dimensions. (e) Dimensionality of the semantic space. (f) Sample size.

7.6 Summary of Experimental Results

We have quantified the efficiency and accuracy of pLSI by experimenting with one of the largest corpora available in the public domain. The following are our major findings. (1) pLSI can achieve a good accuracy at reasonable cost with respect to bandwidth and number of nodes searched, and this performance is scalable with respect to system size, corpus size, and number of returned documents. (2) Rolling-index needs only a small number of rotated spaces to work effectively, limiting the space overhead as well as the number of visited nodes. (3) The content-directed heuristics are effective at directing searches, and learning from past history can be beneficial when the queries have locality. (4) Replication improves performance, but at the cost of extra storage. (5) pLSI's performance is not very sensitive to its major parameters. In particular, its absolute performance shows good potential to improve along with future developments of advanced IR techniques.

8. RELATED WORK

Centralized indexing systems such as Napster suffer from single point of failure and performance bottlenecks at the index server. On the other hand, flooding-based techniques such as Gnutella send a query or index to every node in the system, consuming huge amounts of network bandwidth and CPU cycles.

To minimize the number of nodes a query probes, heuristic-based approaches direct a search to only a fraction of the node population. These approaches can be further divided into three categories: random walk, employing summarization, and organizing nodes with similar contents or sharing interests into groups.

Lv et al. [16] study search and replication strategies in unstructured P2P networks. They find that random walk and expanding-ring search are more efficient than flooding.

Several projects employ Bloom filters to summarize contents in the network. Rhea and Kubiatowicz [20] describe a method that uses Bloom filters to summarize neighbors' contents. A query is only forwarded to neighbors that have relevant documents with high probability. PlanetP [6] uses a Bloom filter to summarize contents on each node and floods the summaries to the entire system. Instead of using Bloom filters, Crespo and Garcia-Molina [5] introduce the notion of Routing Indices that give a promising "direction" toward relevant documents.

Schwartz [21] describes a method that organizes nodes with similar contents into a group. A search starts with random walk but proceeds more deterministically once it hits in a group with matching contents. Motivated by research in data mining, Cohen et al. [3] use guide-rules to organize nodes into an associative network. Sripanidkulchai et al. [22] extend an existing P2P network by linking a node to other nodes that satisfy previous queries.

Replication has also been explored to improve search efficiency. FastTrack [10] designates high-bandwidth nodes as super-nodes. Each super-node replicates the indices of several other nodes. Cohen et al. [4, 16] find that setting the number of object replicas to the square root of the searching rate for an object minimizes the expected search size on successful queries.

Some systems [15] hash each term into an ID and store indices in a DHT using term ID as the key. These systems need to intersect the inverted lists [1] of terms to find documents that contain multiple query terms. This cost grows proportionally with corpus size.

Except for PlanetP, all the above systems use simple keyword matching, ignoring the advanced relevance ranking algorithms devised by the IR community. We believe that searches are not just to find documents that contain certain keywords, but to find the most relevant documents. More importantly, pSearch has the advantage of being able to use semantics to limit the search space.

Compared with our P2P architecture, distributed IR systems such as GlOSS [11] usually employ a centralized or hierarchical index to route queries among a small number of distributed sites.

9. CONCLUSION

As one of the first attempts at building self-organizing, large-scale P2P IR systems, we have described several techniques to extend the LSI algorithm to work in a decentralized environment. We have quantified the efficiency of pLSI with respect to bandwidth and number of nodes searched, and the extent to which pLSI can retain LSI's efficacy, by experimenting with one of the largest corpora available in the public domain. We made the following contributions in this paper.

- pSearch is the first system that organizes contents around their semantics in a P2P network. This makes it possible to achieve accuracy comparable to centralized IR systems while visiting a small number of nodes and transmitting a small amount of data.
- We proposed the use of *rolling-index* to resolve the dimensionality mismatch between the semantic space and a CAN, taking advantage of the higher importance of low-dimensional elements of semantic vectors. This helps reduce the number of visited nodes by partitioning the semantic space along more dimensions.
- We employed *content-aware node bootstrapping* to balance the load, which also achieves index and query locality. This helps distribute document indices evenly across nodes.
- We employed *content-directed search* (using index samples and recently processed queries) to guide searches to the right places in the high-dimensional semantic space. This helps further reduce the number of visited nodes.

Although our experience with the TREC, MED, ADI, and CRAN corpora shows great promise, more experiments are needed to study whether pLSI can be applied to a much larger corpus that has hundreds of millions or even billions of documents. Among our enhancements to pLSI, content-aware node bootstrapping is expected to scale well with corpus size. Intuitively, we expect the performance of content-directed search to improve as corpus size increases, because relevant documents will be found at closer distances. The rolling-index may be affected adversely as corpus size increases due to the enlarged dimensionality gap between LSI and CAN. Our proposals to improve rolling-index, selective rotating and hierarchical clustering, are yet to be evaluated.

Our future work includes implementing other IR techniques and efficient LSI variants (e.g., concept index) in pSearch and comparing another search algorithm pVSM [24] with pLSI. We also plan to experiment with a large HTML corpus crawled from the Web.

Acknowledgments

We thank Deqing Chen, Wenrui Zhao, Chi Zhang, Boon Ang, Magnus Karlsson, Christos Karamanolis, the anonymous reviewers, and our shepherd Robert Morris for their valuable feedback. We thank Mallik Mahalingam for his contribution during the initial stages of this project, and John Sontag and Dejan Milojicic for their support. Chunqiang and Sandhya were supported in part by NSF grants CCR-9988361, CCR-0219848, ECS-0225413, and EIA-0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; and by the U.S. Dept. of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460, and by equipment or financial grants from Compaq, IBM, Intel, and Sun.

10. REFERENCES

[1] M. Berry, Z. Drmac, and E. Jessup. Matrices, Vector Spaces, and Information Retrieval. *SIAM Review*, 41(2):335–362, 1999.

[2] C. Buckley. Implementation of the smart information retrieval system. Technical Report TR85-686, Department of Computer Science, Cornell University, Ithaca, NY 14853, May 1985.

[3] E. Cohen, A. Fiat, and H. Kaplan. Associative Search in Peer to Peer Networks: Harnessing Latent Semantics. In *IEEE INFOCOM'03*, April 2003.

[4] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *ACM SIGCOMM'02*, 2002.

[5] A. Crespo and H. García-Molina. Routing Indices for Peer-to-peer Systems. In *ICDCS'02*, July 2002.

[6] F. M. Cuenca-Acuna and T. D. Nguyen. Text-Based Content Search and Retrieval in ad hoc P2P Communities. In *the International Workshop on Peer-to-Peer Computing*, May 2002.

[7] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[8] S. Dumais. Using LSI for information filtering: TREC-3 experiments. In *the Third Text REtrieval Conference (TREC3)*, 1995.

[9] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[10] FastTrack. http://www.fasttrack.nu.

[11] L. Gravano, H. García-Molina, and A. Tomasic. GlOSS: text-source discovery over the Internet. *ACM Transactions on Database Systems*, 24(2), 1999.

[12] G. Karypis and E.-H. S. Han. Concept indexing: A fast dimensionality reduction algorithm with applications to document retrieval and categorization. In *CIKM'00*, November 2000.

[13] R. Lempel and S. Moran. Optimizing Result Prefetching in Web Search Engines with Segmented Indices. In *VLDB'01*, 2001.

[14] T. A. Letsche and M. W. Berry. Large-Scale Information Retrieval with Latent Semantic Indexing. *Information Sciences*, 100(1-4):105–137, 1997.

[15] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS'03*, February 2003.

[16] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *ICS'02*, June 2002.

[17] C. D. Prete, J. T. McArthur, R. L. Villars, I. L. Nathan Redmond, and D. Reinsel. Industry developments and models, Disruptive Innovation in Enterprise Computing: storage. *IDC*, February 2003.

[18] P. Raghavan. Information Retrieval Algorithms: A survey. In *the 8th SIAM Symposium on Discrete Algorithms (SODA)*, January 1997.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM'01*, August 2001.

[20] S. Rhea and J. Kubiatowicz. Probabilistic Location and Routing. In *IEEE INFOCOM'02*, June 2002.

[21] M. Schwartz. A Scalable, Non-Hierarchical Resource Discovery Mechanism Based on Probabilistic Protocols. Technical Report CU-CS-474-90, University of Colorado, 1990.

[22] K. Sripanidkulchai, B. Maggs, and H. Zhang. Enabling Efficient Content Location and Retrieval in Peer-to-Peer Systems by Exploiting Locality in Interests. *ACM SIGCOMM Computer Communication Review*, 32(1), January 2002.

[23] SVDPACK. http://www.netlib.org/svdpack.

[24] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information Retrieval in Structured Overlays. In *HotNets-I*, October 2002. Expanded version available as HP technical report HPL-2002-198, "PeerSearch: Efficient Information Retrieval in Peer- to-Peer Networks".

[25] Text Retrieval Conference (TREC). http://trec.nist.gov.

[26] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98*, pages 194–205, August 1998.

[27] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays using Global Soft-State. In *ICDCS'03*, May 2003.

[28] J. D. Zakis and Z. J. Pudlowski. The World Wide Web as Universal Medium for Scholarly Publication, Information Retrieval and Interchange. *Global Journal of Engineering Education*, 1(3), 1997.