Keyword Search in DHT-based Peer-to-Peer Networks

Yuh-Jzer Joung
joung@ntu.edu.tw

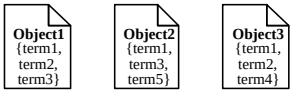
Chien-Tse Fang
r90040@im.ntu.edu.tw

Li-Wei Yang
willie@eland.com.tw

Department of Information Management
National Taiwan University
Taipei, Taiwan

Abstract

Existing techniques for keyword/attribute search in structured P2P overlays suffer from several problems: unbalanced load, hot spots, fault tolerance, storage redundancy, and unable to facilitate ranking. In this paper, we present a general keyword index and search scheme for structured P2P networks that avoids these problems, and in which object insert, delete, and search can be efficiently performed. Some experimental results are also presented to support our claim.



Key	List of Objects
term1	{Object1, Object2, Object3}
term2	{Object1, Object3}
term3	{Object1, Object2}
term4	{Object3}
term5	{Object2}
...	...

Figure 1. An inverted index of three objects.

1 Introduction

A distinguishing characteristic of DHT-based structured P2P networks(e.g., Tapestry [13], Pastry [9], Chord [11], and CAN [7]) is that search is *deterministic*: given the identifier of an object (and only through the identifier), the underlying location and routing scheme guarantees to find the object within reasonable cost. Search by identifiers, however, is useful only when we have full knowledge on what resources we want, but often times we have only partial information. So much attention has been paid to build a more flexible search service, such as keyword/attribute search, on top of the networks.

The most common way to implement keyword search in information systems is by inverted index. An *inverted index* is a set of entries of pairs (ω, O), where ω is a keyword, and O is the set of objects containing this keyword; see Figure 1. Once an inverted index is built, a set of keywords can be entered to find all objects that contain these keywords. For example, in Figure 1, by taking the intersection of the sets associated with keywords $\mathrm{term1}, \mathrm{term2}$, and $\mathrm{term3}$, we can find the object (i.e., $\mathrm{Object1}$) that has all these keywords.

To implement keyword search over a P2P network, a distributed version of inverted index can be built. A straightforward way to decentralize an inverted index is to distribute the entries so that each keyword is assigned a unique node to handle the objects that have this keyword. By incorporating into DHT networks, one can use a given keyword as key to determine the node that is responsible for the keyword, and obtains the set of objects that contain this keyword from the node. By taking appropriate join operations (e.g., intersection), one can retrieve objects with a given keyword set. This approach is used in [8, 14]. Some other systems use a similar approach, but add on more features. For example, in [2] multiple keywords are additionally treated as a single entry if they are highly correlated. This saves bandwidth and speeds up search process, but at the expense of extra storage. Correlation between keywords is also not easily available in distributed environment. In [1], each resource (object) is described by an *attribute-value tree* (AVTree), in which each link represents an attribute with a value represented by the child node. Each path from the root in an AVTree (called a *strand*) is a sequence of attribute-value pairs that can be treated as a keyword to describe the resource. Then, the concept of inverted index can be used to search resources with a given sequence of attribute-value pairs. Note that some strands are sub-paths of other strands. This means that some nodes are handling resources that have already been handled by other nodes, and some may handle a set of resources significantly larger than the others do.

The above approaches have some common problems. The first one concerns load balance. In a real world corpus, keyword frequency—the count of a keyword's occurrence in objects—varies enormously. The distribution typically follows *Zipf's law*, meaning that a few keywords occur very often while many others occur rarely (in power-law relationship). So, simply mapping each entry of an inverted index to a node in DHT networks makes the storage required for indexing at each node extremely uneven.

The second problem concerns storage redundancy. If an object σ contains keywords $\omega_1, \ldots, \omega_k$, then creating entries for each keyword (and any combination of them) means that information about the object is repeatedly stored

at k (or more) different places. (A typical object has a few to dozens of keywords in its metadata.) This redundancy also makes delete/insert an object very expensive, as it has to deal with multiple peer accesses in the network. Note that redundancy is necessary in coping with fault tolerance. However, the redundancy incurred by the above index scheme does not help solve fault tolerance naturally because the number of keywords of an object has no correlation with the failure probability of the object.

Moreover, even though an object is indexed at several places, each keyword is still handled only by a single node. Any failure to the node would then block all queries involving this keyword. The system is also vulnerable to hot spots, as nodes responsible for some popular keywords may be queried much more frequently than the others.

The last problem concerns object ranking. If the object space is huge, a query composed of a few popular keywords may yield a set with a very large number of objects. One would certainly prefer some ranking mechanism to help select relevant objects, but such feature is absent from the above work. Ranking, in general, requires some global knowledge. For example, in information retrieval, the concept of *inverse document frequency (IDF)* has been used to measure how importance a keyword is. It is defined as the logarithm of the ratio of number of documents in a collection to the number of documents containing the keyword [4]. So infrequent words have high IDF and common words such as 'mp3' have low IDF. IDF can be easily calculated when indexing service is centralized; but in a decentralized environment, the cost is high to get a good measure of it.

In this paper, we present a general keyword index and search scheme for DHT networks. The main idea is to represent each object as an r-bit vector according to its keyword set. By viewing these r-bit vectors as points in an r-dimensional hypercube, many interesting properties can be obtained:

First, the index entries of a single keyword are handled by a set of nodes. The population of this set depends on the popularity of the keyword: the more the popularity of a keyword, the more the number of nodes responsible for the keyword. As a result, the load of nodes can be balanced even though keyword distribution might follow Zipf's Law, and no node is likely to be swamped even if it handles a very popular keyword. Moreover, since a number of nodes are responsible for a single keyword, any failure of them cannot block queries involving the keyword.

Secondly, an object σ associated with a keyword set K can be efficiently and deterministically located if the set K is given. This is analogous to name/identifier search in DHT networks. As discussed earlier, DHT networks use an object's name to determine its handling node. Thereafter, locating the object is simply a message routing to the node, which can be done very efficiently in the networks. In our index scheme, we use the keyword set associating with an object to determine a unique node to index the object. When the set is known, locating the object is as efficient as name search in DHT networks. In contrast, this kind of 'pin search' is usually very expensive in existing P2P networks. Likewise, object insert and delete can also be done efficiently, as no unnecessary redundancy is introduced in our scheme to index objects.

Third, in a search operation, in addition to objects whose keyword sets match exactly with a given keyword set K, one may also wish to retrieve objects whose keyword sets *contain* K. All these objects can be easily and efficiently retrieved in our index scheme. Moreover, the larger the set K a user has specified, the more restriction a user has placed on his target objects. Accordingly, our index scheme will require fewer number of nodes to be contacted. On the other hand, when a small set of K is given, a large number of objects may satisfy the search request. In this case, a user often expects to see only a small subset of them. Our index scheme can also support this kind of operations effectively and efficiently.

In the above search operation, when the number of matching objects is large and a user only expects to see a portion of them, our index scheme facilitates a variety of ways to help users rank the matching objects. Specifically, objects in our index scheme are easily distinguished by the number of keywords they associate. For example, let K be a set of keywords. Our index scheme can easily locate objects that are associated with exactly the set K of keywords, objects that are associated with K plus one more keyword, K plus two more keywords, and so on. Moreover, within each category, e.g., K plus one more keyword, objects can further be distinguished by the extra keyword they have, e.g., K plus a specific keyword σ_1, K plus a specific keyword σ_2, and so on.

This interesting feature allows upper level applications to retrieve objects in the order they wish. For example, an application might prefer more specific objects to be retrieved first. In this case, when a search request with a keyword set K is issued, our index scheme can return objects containing this keyword set K in the order by giving preference to those with more extra number of keywords. On the other hand, if an application prefers more general objects, then our index scheme can give preference to those with fewer number of extra keywords. Furthermore, our index scheme may also sample some objects in each category described above, e.g., objects that have an extra keyword σ_1, an extra keyword σ_2, ..., two extra keywords σ_1, σ_2, two extra keywords σ_1, σ_3, ..., and so on; and then return these sample objects along with their extra keyword(s) to help users refine their queries. Note that no global knowledge is required to implement this ranking mechanism.

The rest of the paper is organized as follows. Section 2 presents our system model and formalize the keyword search problem; Section 3 presents the keyword index and search scheme; and Section 4 presents some experimental results. Related work and conclusions are offered in Section 5.

2 System Model

We envisage a P2P application system as a four-layer structure shown in Figure 2. Typically, applications such as file sharing, document retrieval, storage sharing, and service discovery, are placed on top of a P2P overlay, which in

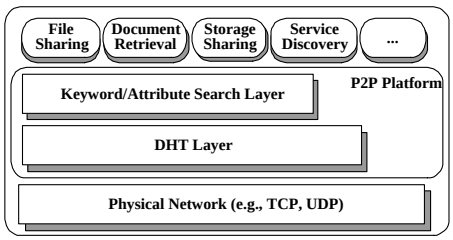


Figure 2. System Architecture.

turn is built on a physical network. Here we have inserted a *keyword/attribute search layer* in between the application layer and the P2P overlay to facilitate object retrieval.

To provide guaranteed search—that is, if an object is residing somewhere in the network, then it can be located with reasonable cost—our keyword/attribute search layer is built on a DHT network. To make the layer as general as possible, we do not assume any specific DHT overlay, but rather a generalized structure on which our keyword/attribute search layer can be linked to. The following section presents such a generalized DHT overlay.

2.1 A generalized Model of DHT Networks

A DHT network (or simply a *DHT*) is an overlay built upon a physical network. The overlay can be modeled by a directed graph $G = (V, E)$, where V is the set of nodes in the network, and E is the set of links between nodes. Each node u is assigned an a-bit unique identification. When no ambiguity is possible, we simply use u for the identification. An edge (u, v) in E means that u knows a direct way to send a message to v.

A set $\mathcal{O}$ of objects is shared among the network. Each object $\sigma \in \mathcal{O}$ also has a unique ID, and has a set of replicas spread in the network. For every node u storing a copy of σ, we use (σ, u) to denote *reference* to the replica. The reference (σ, u) typically consists of the IP/port of u and the physical address of σ within the node. A reference to σ must be obtained in order to access a copy of σ. So references serve as some index to objects, and locating an object is equivalent to locating a reference to the object. In DHTs, a reference of an object is not necessarily stored at the same node that stores the physical copy of the object. The set of references maintained at a node v is denoted by $Refs_v$.

Associated with the overlay G is a distributed object location and routing (DOLR) scheme for accessing objects in the network. A DOLR scheme involves a mapping $\mathcal{L} : \mathcal{O} \rightarrow \{0, 1, \ldots, 2^a - 1\}$, and one routing mechanism. The mapping $\mathcal{L}$ deterministically and uniformly maps each object $\sigma \in \mathcal{O}$ (by its ID) to exactly one node—represented as an a-bit binary string—to handle the object. The routing mechanism determines, for every two nodes u and v in V, at least one (u, v)-path in G.

Three operations are supported by the DOLR scheme for accessing objects: $Insert$, $Delete$, and $Read$. When node u publishes a copy of object σ, it invokes the operation $Insert(x, \sigma, u)$ to place the reference (σ, u) of the copy to a node x whose ID equal to $\mathcal{L}(\sigma)$. The operation proceeds as follows: First, the node x is determined. Then, an insert request for (σ, u) is forwarded hop by hop to x along a (u, x)-path in the overlay. When the insert request arrives at x, the reference (σ, u) is added to $Refs_x$. Accordingly, $Delete$ and $Read$ operations can be derived based on the $Insert$ operation.

The above DOLR scheme requires every potential node ID in $\{0, 1, \ldots, 2^a - 1\}$ be assumed by a unique node in the network. In practice, nodes may join and leave the network dynamically. Furthermore, to avoid ID collision the size of the node ID space (i.e., 2^a) is typically much larger than the actual number of nodes that may participate in the network. So not all potential node IDs have a mapping to actual nodes, and the DOLR scheme must have a *surrogate routing* mechanism to handle absence of nodes. That is, if a node v is absent, then the scheme will find an existing node $S(v)$ in V to play the role of v so that every message to v will be automatically routed to $S(v)$. As a result, we may assume that the overlay is reliable and self-organizing.

2.2 The Keyword Search Problem

Insert, delete, and read operations in the DOLR scheme require the target object ID be known. Some applications may need to locate an object with only *keywords (attributes)*. To provide keyword search service on the overlay, we assume that each object $\sigma \in \mathcal{O}$ is associated with a set K_σ of keywords. For any object σ, we say that a keyword set K *can describe* σ if $K \subseteq K_\sigma$.

For each set K of keywords, we define a set $\mathcal{O}_K$ of objects, where $\mathcal{O}_K = \{\sigma \,|\, \sigma \in \mathcal{O}, K \subseteq K_\sigma\}$. That is, $\mathcal{O}_K$ is the set of objects that can be described by K. The size $|\mathcal{O}_K|$ is called the *keyword frequency* of K.

To provide keyword search service, we need to design a distributed index scheme so that an object can be located by specifying a few keywords in a query. Two functions can be identified in the service:

Pin Search: Given keyword set K, the service should return the set $\{\sigma \,|\, K_\sigma = K\}$ of objects that are associated with exactly the keyword set K.

Superset Search: Given keyword set K and some threshold t, the service should return a set of $\min(t, |\mathcal{O}_K|)$ objects that can be described by K.

In practice, superset search can be designated as *cumulative*, where the results returned by consecutive searches with the same keyword set must be different. This is typically used in large information system such as Google in which users can 'browse' through large matching object sets steps by steps.

3 The Keyword Index and Search Scheme

Our distributed index scheme is built on a logical structure—an r-dimensional hypercube vector space—over a DHT network. To present this index scheme, we first introduce the hypercube, and then the relationship between the logical structure and the underlying DHT network.

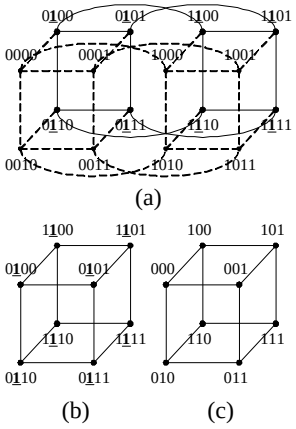


Figure 3. (a) H_4**, (b)** $\mathcal{H}_4(0100)$**, (c)** H_3**.**

3.1 An r-Dimensional Hypercube Vector Space

An r-dimensional hypercube $H_r(V, E)$ has 2^r nodes. Each node u in V is represented by a unique r-bit binary string. We use $u[i]$ to denote the i^{th} bit of u (counting from the right). For every two nodes u, v in V and every integer i in $\{0, 1, \ldots, r-1\}$, there exists an (undirected) edge (u, v) in E if, and only if, u differs from v only at the i^{th} bit. We say that the (u, v)-edge crosses the i^{th} dimension, and u is v's neighbor in the i^{th} dimension, and vice versa.

For each node $u \in V$, we define a set $One(u)$ of integers as follows: $One(u) = \{i \,|\, u[i] = 1, 0 \leq i \leq r-1\}$; that is, the positions at which u has bit one. Similarly, we define $Zero(u) = \{i \,|\, u[i] = 0, 0 \leq i \leq r-1\}$. For example, if $v = 010100$, then $One(v) = \{2, 4\}$ and $Zero(v) = \{0, 1, 3, 5\}$. Let u, v be two r-bit vectors. We say that v *contains* u if and only if $u[i] \Rightarrow v[i], \forall 0 \leq i < r$; that is, $One(u) \subseteq One(v)$.

The following two definitions will be used in subsequent sections.

Definition* 3.1** *Let $H_r = (V, E)$ be a hypercube and $u \in V$ be a node. A* ***subhypercube induced by *u, denoted by $\mathcal{H}_r(u)$, is a subgraph $G = (U, F)$ of H_r such that every node $w \in V$ is in U if and only if w contains u, and every edge $e \in E$ is in F if and only if its two end points are in U.*

By the definition, all nodes w in U have bit '1' at each position in $One(u)$. If we remove those bits, then each node w becomes an $(r - |One(u)|)$-bit string; that is, a $|Zero(u)|$-bit string. Furthermore, in the resulting graph every two nodes have an edge if and only if they differ in exactly one bit. That is, the resulting graph is a $|Zero(u)|$-dimensional hypercube. So $\mathcal{H}_r(u)$ is *isomorphic* to a $|Zero(u)|$-dimensional hypercube. Figure 3 illustrates $\mathcal{H}_4(0100)$ induced by node 0100 in H_4, which is isomorphic to H_3.

Broadcast can be done very efficiently in hypercubes through the use of spanning binomial trees [3]. The following gives the definition.

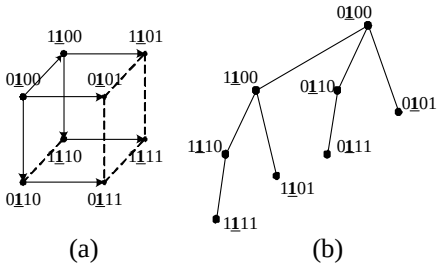


Figure 4. (a) $\mathcal{H}_4(0100)$**, (b)** $SBT_{\mathcal{H}_4}(0100)$**.**

Definition* 3.2** *Let $H_r = (V, E)$ be a hypercube of dimension r and $u \in V$ be a node. The* ***spanning binomial tree *rooted at u, denoted by $SBT(u)$, is a tree consisting of the nodes in V and the edges defined by the following: For any node $v \in V$, let p be the dimension satisfying $v[p] \oplus u[p] = 1$ and $v[i] \oplus u[i] = 0$, $\forall i < p$ ($p = -1$, if $u = v$). Let $Z_v = \{p-1, p-2, \ldots, 1, 0\}$ ($Z_v = \emptyset$ if $p \leq 0$). Then the parent of v in $SBT(u)$ is*

$$\begin{cases} v[r-1]\ldots v[p+1]\overline{v[p]}v[p-1]\ldots v[0] & \text{if } p \neq -1 \\ \emptyset & \text{if } p = -1 \end{cases}$$

and its children are

$$\begin{cases} v[r-1]\ldots \overline{v[p]}\ldots \overline{v[j]}\ldots v[0], \forall j \in Z_v & \text{if } p \neq -1 \\ v[r-1]\ldots \overline{v[j]}\ldots v[0], \forall j \in \{r-1, \ldots, 1, 0\} & \text{if } p = -1 \end{cases}$$

By the definition, a spanning binomial tree $SBT(u)$ of H_r has depth r, and a node v in the m^{th} level of $SBT(u)$ (where root u is at level 0) has Hamming distance m from u. (The *Hamming distance* between any two r-bit binary strings u and v is $Hamming(u, v) = \sum_{i=0}^{r-1}(u[i] \bigoplus v[i])$.) That is, in a spanning binomial tree, every node that is i depth from the root has exactly i bits different from the root in their binary representation. This important property will later be used in our keyword search scheme.

Recall that a subhypercube induced by a node u in H_r, i.e., $\mathcal{H}_r(u)$, is isomorphic to a $|Zero(u)|$-dimensional hypercube. We can similarly define spanning binomial trees on $\mathcal{H}_r(u)$ by masking the bits occurring at the positions in $One(u)$ for every node in $\mathcal{H}_r(u)$. In the paper we will need the spanning binomial tree rooted at u on $\mathcal{H}_r(u)$. We call this tree the *spanning binomial tree induced by* u, and denote it by $SBT_{\mathcal{H}_r}(u)$. Figure 4 illustrates $SBT_{\mathcal{H}_4}(0100)$.

3.2 Map to a DHT Network

Our index scheme is based on an r-dimensional hypercube $H_r(V, E)$. The hypercube can be constructed directly from a physical hypercube (e.g. HyperCuP [10]), or conceptually built on an underlying DHT network $G = (V', E')$. To construct $H_r(V, E)$ over $G = (V', E')$, we simply need a mapping $g : V \rightarrow V'$ so that every logical node in the hypercube has a corresponding physical node in the network. Depending on the applications, the dimensionality r of the hypercube may be smaller or larger than a—the size of the

node identification in the DHT layer. When r is larger, we have more logical nodes than physical nodes. To balance load, we will need g to be a hash function that can uniformly map the nodes in V to V'. When r is smaller than a, only a portion of the physical nodes in the DHT layer will actually be responsible for indexing objects. This allows some leeway in selecting indexing nodes. In fact, many researches have observed that nodes in P2P networks are not homogeneous: some are more stable/powerful than the others. So we may select stable/powerful nodes to serve as indexing nodes in the hypercube.

3.3 An Index Scheme over the Hypercube

We now present an index scheme on a hypercube $H_r(V, E)$. Throughout the rest of the section, unless stated otherwise, all nodes refer to the nodes in the hypercube. Let $\mathcal{W}$ be the set of all keywords considered in the system. Let $h : \mathcal{W} \rightarrow \{0, 1, \ldots, r-1\}$ be a uniform hash function that maps every keyword in $\mathcal{W}$ to an integer in $\{0, 1, \ldots, r-1\}$. We define a mapping $\mathcal{F}_h : 2^{\mathcal{W}} \rightarrow V$ as follows: $\mathcal{F}_h(K) = u$ if, and only if, $One(u) = \{i \mid h(\sigma) = i, \sigma \in K\}$. In other words, $\mathcal{F}_h(K)$ is the node with a binary number whose bits are set by the hash function h according to the keywords in K. We say that u is *responsible* for K if $\mathcal{F}_h(K) = u$. Thus, for every possible set of keywords in the system, there is a node in the hypercube responsible for the set. Note that a node may be responsible for more than one set of keywords (as $\mathcal{F}_h(K)$ might be equal to $\mathcal{F}_h(K')$ for some K and K'). We use $\mathcal{R}_u$ to denote the set of keyword sets for which u is responsible; that is, $\mathcal{R}_u = \{K \subseteq \mathcal{W} \mid \mathcal{F}_h(K) = u\}$.

To build the index scheme, for each object σ that is associated with keyword set K_σ, we let the node $\mathcal{F}_h(K_\sigma)$ in the hypercube maintain an entry $\langle K_\sigma, \sigma \rangle$ in its index table. We say that σ is *indexed* at the node, and we use $\mathcal{O}_u$ to denote the set of objects that are indexed at u; that is, $\mathcal{O}_u = \{\sigma \in \mathcal{O} \mid K_\sigma \in \mathcal{R}_u\}$.

Given a keyword set K, we can locate a copy of object associated with K by first finding the node in H_r that is responsible for K. The node is determined by $\mathcal{F}_h(K)$. Once the node is located, its index table can be searched to obtain the ID of an object σ that is associated with the keyword set K. Then, a call $Read(\sigma)$ to the underlying DHT network will invoke the DOLR scheme to return a copy of σ. So pin search is directly supported by the scheme.

Although pin search is quite effective in locating a particular object, one would often want results based on the superset of keywords they submit in a query set K, i.e., the superset search. To locate copies of those objects, we need to find all those nodes that are responsible for a superset of K. Recall that a subhypercube $\mathcal{H}_r(u)$ of H_r induced by u consists of all nodes w in V that contain u (that is, $u[i] \Rightarrow w[i]$). So every node in the subhypercube is responsible for a keyword set that is a superset of K, and every node in H_r that is responsible for a superset of K is also in the subhypercube. This property allows us to search for only the subhypercube if we wish to find out any other object that can also be described by K.

Moreover, when searching the subhypercube, we can explore the spanning binomial tree rooted at u, i.e., the tree $SBT_{\mathcal{H}_r}(u)$. Recall that a node v at depth i in the tree has Hamming distance i from the root. For every keyword set $K_v \in \mathcal{R}_v$ and $K_u \in \mathcal{R}_u$, if $K_u \subseteq K_v$, then $|K_v| - |K_u| \geq i$; that is, K_v contains at least i more keywords than K_u. This means that if we search the tree $SBT_{\mathcal{H}_r}(u)$ in a breadth-first style, then we can locate objects whose associated keyword sets gradually enlarge during the search, thereby allowing the upper-level applications to retrieve relevant objects more effectively.

The following two lemmas summarize the above properties. Their proofs are straightforward.

Lemma 3.1 *Let $u \in V$ be a node in $H_r = (V, E)$, and $K \subseteq \mathcal{W}$ be a keyword set for which u is responsible. Then, all objects that can be described by K are indexed at some node in the subhypercube $\mathcal{H}_r(u)$ induced by u.*

Lemma 3.2 *Let $u \in V$ be a node in $H_r = (V, E)$, and let $\mathcal{H}_r(u)$ be a subhypercube induced by u and $SBT_{\mathcal{H}_r}(u)$ be the spanning binomial tree rooted at u in the subhypercube. For every node v in the tree, if v is at depth d, then for every keyword set $K_u \in \mathcal{R}_u$, every keyword set $K_v \in \mathcal{R}_v$ such that $K_u \subseteq K_v$ has at least d more keywords than K_u.*

Finally, when performing a search, a typical scenario is that a user starts by specifying a set of keywords, browses through some returned objects, and then adds more keywords to refine the search. The following lemma says that the second query has a search space within the first one. An implication of the lemma is that we can cache some information about the nodes visited in earlier queries for search refinement, so as to save bandwidth.

Lemma 3.3 *Let $K_1, K_2 \subseteq \mathcal{W}$ be two keyword sets. If $K_1 \subseteq K_2$, then $\mathcal{H}_r(\mathcal{F}_h(K_2))$ is a subhypercube of $\mathcal{H}_r(\mathcal{F}_h(K_1))$.*

In the following we present detailed access operations to the index scheme. Each node u in the hypercube H_r maintains an index table Tbl_u of entries of the format: $\langle keyword\_set, object\_id \rangle$. An entry $\langle K, \sigma \rangle$ in the table means that there is an object σ in the network that is associated with keyword set K. The set of entries $\langle K, \sigma_1 \rangle$, ..., $\langle K, \sigma_n \rangle$ with same keyword set can obviously be combined into a single entry $\langle K, \{\sigma_1, \ldots, \sigma_n\} \rangle$.

Insert

When a node u in the DHT publishes a copy of object σ, it uses the function $\mathcal{L}$ to determine the node (i.e., $\mathcal{L}(\sigma)$) in the DHT network that handles references of σ. Then, u invokes the operation $Insert(\mathcal{L}(\sigma), \sigma, u)$ in the underlying DOLR scheme to place the reference $\langle \sigma, u \rangle$ of the copy to the node $\mathcal{L}(\sigma)$. When the node $\mathcal{L}(\sigma)$ adds the reference to its reference list, the insert operation ends if a copy of σ already exists. If no copy of σ is there, then an index of the object is created and inserted into the hypercube. To do so, let K_σ be the set of keywords associated with σ. Node $\mathcal{L}(\sigma)$ computes the node $\mathcal{F}_h(K_\sigma)$ in the hypercube that is responsible for K_σ. The node $\mathcal{F}_h(K_\sigma)$ corresponds to the physical

node $g(\mathcal{F}_h(K_\sigma))$ in the DHT network. Then, node $\mathcal{L}(\sigma)$ invokes the operation $Insert(g(\mathcal{F}_h(K_\sigma)), K_\sigma, \sigma)$ in the underlying DOLR scheme to place an index entry $\langle K_\sigma, \sigma \rangle$ at node $g(\mathcal{F}_h(K_\sigma))$.

Delete

If a node u in the DHT wishes to delete a copy of object σ it has previously published, it uses the same procedure as in the insert operation to locate the node that handles the reference $\langle \sigma, u \rangle$. Then, u invokes the operation $Delete(\mathcal{L}(\sigma), \sigma, u)$ in the underlying DOLR scheme to remove the reference from node $\mathcal{L}(\sigma)$. The delete operation ends if there is other copies of σ. Otherwise, no copy of σ exists in the network, so the index of the object should also be deleted from the hypercube. The index delete process is similar to the insert process. Node $\mathcal{L}(\sigma)$ invokes the operation $Delete(g(\mathcal{F}_h(K_\sigma)), K_\sigma, \sigma)$ in the underlying DOLR scheme to delete the index entry $\langle K_\sigma, \sigma \rangle$ from node $g(\mathcal{F}_h(K_\sigma))$.

Superset Search

A superset search operation composed of a keyword set K and some threshold t must return a set of $\min(t, |\mathcal{O}_K|)$ objects that can be described by K. By Lemma 3.1 the search space can be limited within the subhypercube $\mathcal{H}_r(\mathcal{F}_h(K))$. Although any node in the subhypercube may potentially index some objects that can be described by K, there are subtle difference between the objects. By Lemma 3.2, if a node x is d steps away from the root in the spanning binomial tree $SBT_{\mathcal{H}_r}(\mathcal{F}_h(K))$, then every object indexed at x that can be described by K is associated with a keyword set that contains at least d more keywords than K has, and so is likely to be less general (and thus more specific on a certain subject) than the objects with exactly the keyword set K. Depending on the applications, we can explore the spanning binomial tree in a breadth-first style from either *top down*, or *bottom up*. The former returns search results by giving preference to more general objects, while the latter to more specific objects. Here we present the first approach; the other alternative can be done with only a slight modification.

Let $v = \mathcal{F}_h(K)$ be the node that is responsible for K. A node u that initiates a superset search request sends v a $T\_QUERY(K, t, u, -, -)$, where K is the queried keyword set, t is a threshold, and u is a direct contact to the node u collecting the results; the other two fields are not used in the initial request, but will be used later on during the operation. ('T' in T_QUERY stands for "superset search with Top-down approach".) When v receives the request, it examines its index table to find all entries $\langle K', O \rangle$ with $K' \supseteq K$, and then sends the object IDs (up to t count) in the entries directly to u. The search operation ends if t object IDs have been returned. Otherwise, v records the remaining number of object IDs to be collected in a counter c. Node v also initializes a queue U of pairs (x, d), $0 \leq d \leq r-1$, where x is a direct contact to a node x in $SBT_{\mathcal{H}_r}(v)$, and d is an index for computing the children of x in $SBT_{\mathcal{H}_r}(v)$. The queue U is initialized as follows: For each i in $Zero(v)$, (y, i) is added to U, where y is v's neighbor in the i^{th} dimension.

Then, the search operation proceeds by the steps below:

1. If U is not empty, the first pair (w, d) is removed from U, and v sends w a message $T\_QUERY(K, c, u, d, v)$. Node v then proceeds to Step 3 to wait for the result. If U is empty, the search operation terminates.

2. For each node w receiving a message $T\_QUERY(K, c, u, d, v)$, w first examines its index table to find all entries $\langle K', O \rangle$ with $K' \supseteq K$, and then sends the object IDs (up to c count) in the entries directly to u. Let c_1 be the number of object IDs returned. The search operation can stop if $c_1 \geq c$, and in this case w sends v a message T_STOP to inform v of this. Otherwise, w prepares a list L consisting of the following pairs: $\{(x, i) \mid i < d \wedge i \in Zero(w)$ and x is w's neighbor in the i^{th} dimension $\}$. Then w sends a message $T\_CONT(c_1, L)$ to v.

3. Node v terminates the search operation on receiving a message T_STOP from w. Otherwise, it waits until w sends it a message $T\_CONT(c_1, L)$. Then, v adds the pairs in L to U, sets c to $c - c_1$, and goes to Step 1 to continue the operation.

As noted in Section 2.2, superset search can be made cumulative to let consecutive searches browse through a large matching set steps by steps. In the above operation, cumulative superset search can be easily implemented by letting the root node $v = \mathcal{F}_h(K)$ keep the queue U for subsequent queries until the search has completed.

3.4 Remarks

We make some comments on the performance of the index scheme. First, we observe that the indexing node of an object is determined uniquely by its keyword set. So even if objects $\sigma_1, \ldots, \sigma_n$ all contain some popular keywords $\omega_1, \ldots, \omega_k$, it is likely that the keyword sets of these objects still differ in some way, and so the objects are to be indexed by more than one node. The more the popularity of the keywords, the more the number of objects contain these keywords, and so the more the number of nodes are to index the objects. So storage load can be balanced even if keyword frequency follows Zipf's law. Moreover, since there are a number of nodes to index a keyword, no single node failure can block all queries involving the keyword.

Second, because the storage load for indexing a popular keyword (or a keyword set) is distributed to a number of nodes, the query load to the keyword can also be distributed to the nodes as well, so as to avoid hot spots. To see this, suppose a class S of objects has some common keywords. As commented above, if the set S is large, the objects will likely to have some other keywords to distinguish them. If a user knows more about the additional keywords, our index scheme can quickly locate the subclass of objects he is looking for. However, if a user has little knowledge about

S, he is likely to use just some popular keywords to search. This kind of simple queries, in fact, play a major part in user query behavior [8], and so are inevitable to any information system. A consequence of this to our index scheme is raising a potential hot spot to the nodes handling exactly some very popular keyword sets. Note that in this circumstances it is reasonable to assume that the user needs only a portion of S. This leaves much leeway to upper-level applications in resolving queries. For example, *query expansion* [5, 6] can be used to expand keyword sets. Moreover, the applications can add some keywords, based on, say, the user's preference or his past logs, to help him locate his interest. This customization not only improves search quality, but also alleviates the potential hot spot. Nevertheless, many DHT overlays have their techniques in dealing with hot spots, and they can be assumed by the underlying DHT layer in our system to cope with the problem. For example, in Tapestry [13], if some node has been queried very often, then the results will be cached along the path to the querying nodes, thereby preventing the hot node from being swamped.

Third, each object is indexed by only one node, regardless of how many keywords it has. So, unlike distributed inverted index, the scheme does not introduce extra cost to index an object. Object insert, delete, and pin search therefore takes only one lookup operation in the DHT overlay, as opposed to k operations usually required by distributed inverted index, where k is the number of keywords an object has. Replication certainly helps increase fault tolerance (at the cost of extra storage and consistency maintenance), but this is up to the applications. Note that replication here refers to the index information that is used for keyword search. Object replication is orthogonal to index replication, and has already been assumed in our DHT model (see Section 2.1). If one wishes, (index) replication can be done in two ways. One is to deal with it directly in the index layer, for example, by building a secondary hypercube. The other is, again, to assume this function as part of the underlying DHT overlay, as many existing DHT overlays already have their techniques for replication and fault tolerance.

Fourth, when the hypercube is conceptually built from a DHT, each node in the hypercube has a direct mapping to a physical node in the DHT. So every message sent between two nodes in the hypercube during insert, delete, and search operations is easily translated to be a message sent between two physical nodes in the DHT. So no routing information in the hypercube is necessary for the operations; the underlying DHT can take the responsibility for locating any destination node. Still, when search in a spanning binomial tree, a node may need to contact its immediately children, which are actually its neighbors in the hypercube. So caching neighboring information, as well as search results in the hypercube, does help reducing communication cost and boosting performance in the search. We will address the cache issue in the experiment in Section 4.

Finally, we observe that the index scheme is decomposable: instead of using a single large hypercube to index objects, we can divide the entire keyword set into smaller, disjoint subsets, and then use a hypercube for each subset to index objects. This is useful when objects have multiple attributes, among which some of them are less frequently used in search than the others. we can use several smaller vectors to describe an object. A large index vector results in a large dimension of indexing hypercube, which in turn increases search complexity (see the section below). Decomposing keyword sets therefore increases search performance.

3.5 Complexity Analysis

In this section we analyze the search cost in our scheme. Pin search is clearly quite effectively: it takes only one message transmission for query and another one for returning the result. Insert and Delete operations are also quite efficient: each takes only one message to update a node's index table, as each object is indexed at only one node.

For superset search that is composed of a keyword set $K \subseteq \mathcal{W}$ and is initiated by a node $u \in V$, the cost of the operation consists of three parts: (i) transmitting the query from u to the node v that plays the logical node $\mathcal{F}_h(K)$ in the hypercube layer; (ii) search in the subhypercube $\mathcal{H}_r(\mathcal{F}_h(K))$; and (iii) transmitting the IDs of all objects in $\mathcal{O}_K$ to node u by the nodes that receive the search request.

When searching in the subhypercube, query messages are sent sequentially from the root $\mathcal{F}_h(K)$ to the other nodes so that the operation can be terminated when it returns enough matching objects. The size of the subhypercube $\mathcal{H}_r(\mathcal{F}_h(K))$ is $2^{r-|One(\mathcal{F}_h(K))|}$. So if local computing time is negligible compared to the message transmission time, the operation requires $2^{r-|One(\mathcal{F}_h(K))|}$ message transmission time in the DHT overlay, and costs at most $2 \cdot 2^{r-|One(\mathcal{F}_h(K))|}$ messages. One can also speed the search process by sending query messages simultaneously to the nodes in the spanning binomial tree $SBT_{\mathcal{H}_r}(\mathcal{F}_h(K))$ that are at the same level. In this case, the time complexity can be sped up to $r - |One(\mathcal{F}_h(K))|$.

The above analysis depends on the parameter $|One(\mathcal{F}_h(K))|$, which can be calculated as follows. Recall that $One(\mathcal{F}_h(K))$ is defined to be the set $\{i \,|\, h(\sigma) = i, \sigma \in K\}$, where h is a hash function that uniformly and independently maps every keyword in $\mathcal{W}$ to an integer in $\{0, 1, \ldots, r-1\}$. Let $|K| = m$. Then the probability that $|One(\mathcal{F}_h(K))| = j$, $j \in \{1, 2, \ldots, \min(r, m)\}$, is equivalent to the probability that m distinct balls are distributed into r distinct buckets such that exactly j out of the r buckets are nonempty. So

$$\begin{aligned} & \mathbf{P}(|One(\mathcal{F}_h(K))| = j) \\ = \; & \tbinom{r}{j} \sum_{i=0}^{j} (-1)^i \tbinom{j}{i} \big(1 - \tfrac{i+r-j}{r}\big)^m \end{aligned} \tag{1}$$

So the expected value of $|One(\mathcal{F}_h(K))|$ is

$$\begin{aligned} & \mathbf{E}(|One(\mathcal{F}_h(K))|) \\ = \; & \sum_{j=1}^{j=\min(r,m)} j \cdot \tbinom{r}{j} \sum_{i=0}^{j} (-1)^i \tbinom{j}{i} \big(1 - \tfrac{i+r-j}{r}\big)^m \end{aligned}$$

4 Experimental Results

We conduct some experiments to see the performance of the hypercube index scheme. The object set we use consists

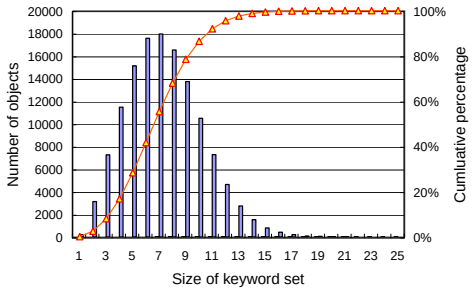


Figure 5. The distribution of keyword set sizes.

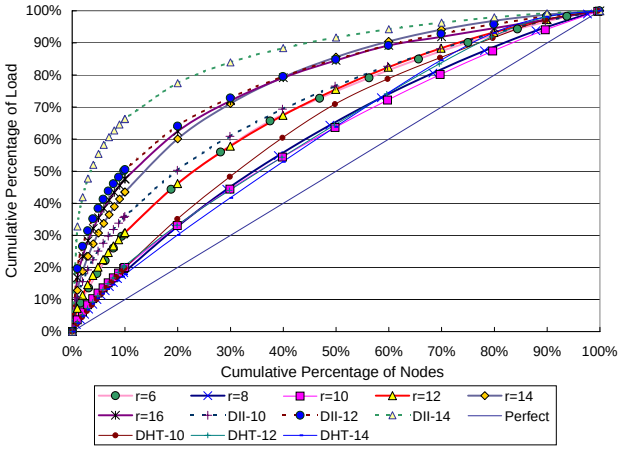


Figure 6. Load distribution.

of 131,180 records from the website directory of PCHome (http://www.pchome.com.tw), the largest local portal site in Taiwan. Each website record is maintained manually by experienced editors, and contains the following six fields: ID, Title, URL, Category, Description, and Keyword. Table 1 shows some examples.

Each record is treated as an object to be indexed in our hypercube scheme, and the set of words in the Keyword field is treated as the keyword set associating with the object. The distribution of keyword set sizes is shown in Figure 5. On average, each object is associated with 7.3 keywords.

In the first experiment, we study how the dimensionality r of the hypercube affects load distribution in the hypercube. For each given r, we build a hypercube of dimension r, and assign each object in our data set a node in the hypercube responsible for indexing the object. Then we rank the load of nodes from heavy to light, and determine the percentage of objects each node handles. The results are shown in Figure 6 for $r = 6$ to 16. A perfectly balanced of load would be a straight line with a slope 1. This line is also shown in the chart for reference. Another reference lines in the charts are obtained by simply hashing the objects directly to the nodes. These lines are referred to as 'DHT-r' in the charts. A typical DHT network hashes objects (by their names) to determine their handling nodes, as well as to balance load. So the reference lines provide a guideline to see if our index scheme can achieve the load balance of regular DHT networks. From Figure 6, we see that the 'DHT' lines deviate slightly from the 'Perfect' line. That is, although the direct hashing scheme is considered quite balanced in load, it is unlikely to achieve a perfect scenario.

We see from the figure that the load distribution improves when r increases from 6 to 10—at which the scheme achieves the load balance of DHT, but becomes worse when r increases from 10 to 16. We draw eight more charts in Figure 7 to further explore the relationship between r and the load distribution. In each chart, there is a node distribution in the hypercube with respect to the number of bit one in their IDs. More precisely, each point (x, y) in the line, $x \in \{0, 1, \ldots, r\}$, represents the percentage (the y value) of nodes that have an ID u such that $|One(u)| = x$. As expected, the line follows normal distribution with mean $r/2$. The other line is object distribution. Each point (x, y) in the line represents the percentage of objects indexed at those nodes u such that $|One(u)| = x$. Note that the node distribution curve is always centered in each chart, while the object distribution curves has their top positioned at roughly the same x-value (as the data set is fixed). So when r increases, the object distribution curve appears to move (left) toward the node distribution curve, then separates apart.

Intuitively, indexing load will be balanced if an object distribution can approach the node distribution. From the charts we see that the two distributions are most close to each other when r is around 10. That explains why the curve $r = 10$ in Figure 6 is closer to the perfect load line than the others.

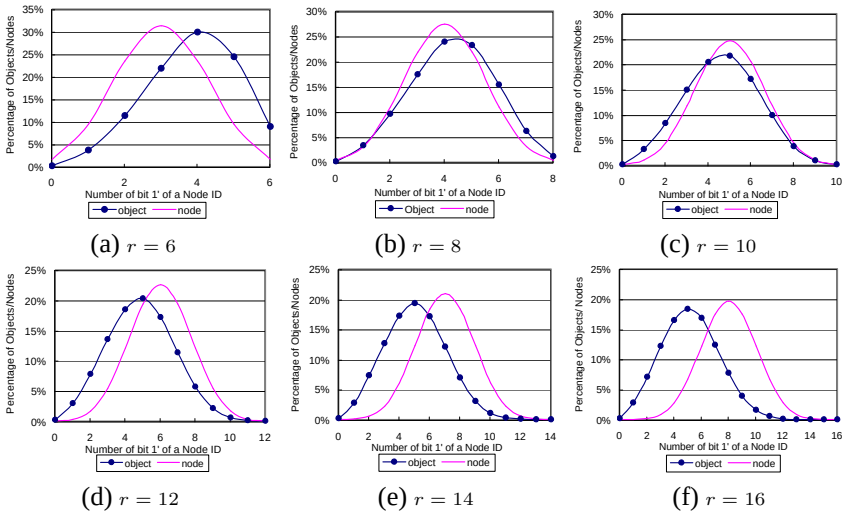
One would be more interesting to see how r can be determined without experiment. In fact, the above observation does highlight us some clue in choosing r. We see that the object distribution curve in Figure 7 is determined by the keyword set sizes distribution in Figure 5. So if a distribution of keyword set sizes is known, then by using Equation (1), we can calculate an appropriate r to let object distribution approach node distribution in Figure 7, thereby to balance the index load.

For comparison, we have also drawn load distribution of the distributed inverted scheme in Figure 6. To draw the distribution, for each keyword used in the data set, we hash the keyword to determine a node in the hypercube to handle the keyword, and then insert all objects (by their references) containing the keyword to the node. The load distribution of this scheme is referred to as 'DII-r', where r is the hypercube dimension. To avoid clouding the chart, here we only show $r = 10, 12$, and 14. From the figure we see that the scheme results in very unbalanced load as compared to ours.

The second experiment studies query (superset search) performance of our index scheme. To conduct the experiment, we build a hypercube of dimension r, and index the data set in the hypercube. We then issue some queries to the hypercube, and measure the number of nodes need to be contacted to resolve the queries. For the queries, we use

ID	Title	URL	Category	Description	Keyword
11	Hinet	http://www.hinet.net	0818013020	Largest ISP in Taiwan	ISP, telecommunication, network, download
18491	TVBS News	http://www.tvbs.com.tw	0318201207	Providing daily news, entertainment news, and news search	TVBS, news

Table 1. Two website records from PCHome.



(a) $r = 6$ (b) $r = 8$ (c) $r = 10$

(d) $r = 12$ (e) $r = 14$ (f) $r = 16$

Figure 7. Object vs. node distribution.

query logs collected at PCHome in some two-week period. Each log records the set of keywords of a query, the time of the query, and the originator (IP address) of the query; but here only the keyword set and the time information will be used. For each size m, we sample some popular keyword sets of size m from the query logs, and use the keyword sets as queries to the hypercube index scheme. The results for $r = 8, 10$, and 12 and $m = 1, 2, \ldots 5$ are shown in Figure 8.

Observe that if the node $\mathcal{F}_h(K)$ responsible for keyword set K has j bits of '1' (i.e., $|One(\mathcal{F}_h(K))| = j$), then a query of keyword set K in the worst case may have to search $2^{r-|One(\mathcal{F}_h(K))|}$ nodes in the hypercube (see Section 3.5). That is, $2^{-|One(\mathcal{F}_h(K))|}$ of the total nodes in the hypercube. If $|K| = m$, and m is relatively small as compared to r, then $One(\mathcal{F}_h(K))$ has high probability to be m. So the percentage of nodes that could be searched is approximately 2^{-m} at 100% recall rate. So from Figure 8 we see that if all matching objects need to be returned, then approximately 2^{-m} of nodes need to be contacted in both $r = 10$ and 12. For $r = 8$, as the value is small, the percentage is higher than 2^{-m} for all $m > 1$. From the charts we also see that the search space depends approximately linearly on the recall rate. This is because the indexing load in our scheme is distributed evenly among nodes.

We commented in Section 3.4 that cache may help boost performance of the system. In the third experiment we study query performance of the index scheme in the presence of cache. The setting is similar to the previous one, except that we install a cache at each node. We use a simple FIFO scheme to manage the cache. The capacity of the cache is $\alpha \times \frac{|\mathcal{O}|}{2^r}$, where α is some parameter, and $\frac{|\mathcal{O}|}{2^r}$ is the average index size per node. For a fixed recall rate, we measure the percentage of nodes need to be contacted with respect to α. The results are shown in Figure 9. Each line in the figure represents the effect of cache for a fixed r and recall rate. The X-axis represents the α parameter—the cache capacity relative to the average index size. For $r = 10$, the average index size per node is $131,180/2^{10}$, which is about 128, and for $r = 12$ the average index size is about 32. A point in the line represents the average percent of nodes need to be contacted per query under the specified recall rate. The average is obtained by supplying all queries (about 178,000) given at some day to PCHome to the hypercube. We see that with only a small fraction of the index size as cache, the performance improved is already huge. With only 1/6 of the index size as cache, less than 1% of nodes need to be contacted even if we demand 100% recall rate for both $r = 10$ and 12.[1]

5 Related Work and Conclusions

In contrast to the distributed inverted index approach, our hypercube index scheme uses a clustering approach to group objects based on their keyword sets. In our scheme, objects are clustered together based on their keyword sets. If two objects σ_1 and σ_2 respectively have keyword sets $K \cup K_1$ and $K \cup K_1 \cup K_2$ for any K, K_1, and K_2, then σ_1 is placed in the cluster C with keyword set K logically no farther to C than σ_2 is. Note that with respect to keyword set K, we can say that σ_2 is more specific than σ_1,

[1] A statistic of the query logs we obtained from PCHome shows that, on average, the ten most popular queries account more than 60% of the total queries per day. This also explains why cache will be so effective in our system.

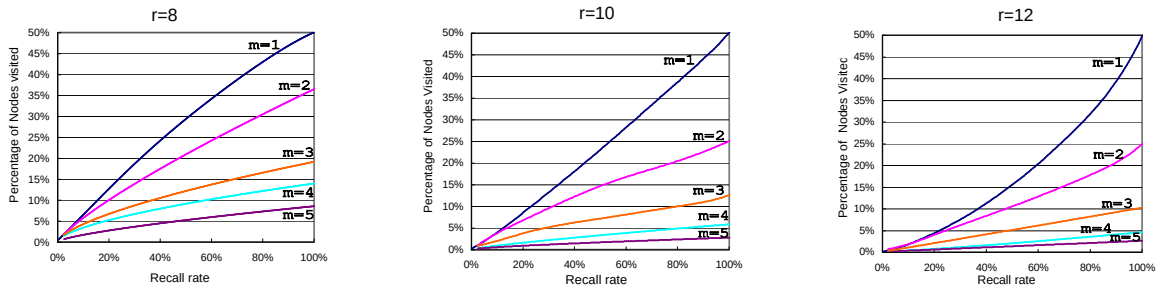


Figure 8. Query performance—cacheless.

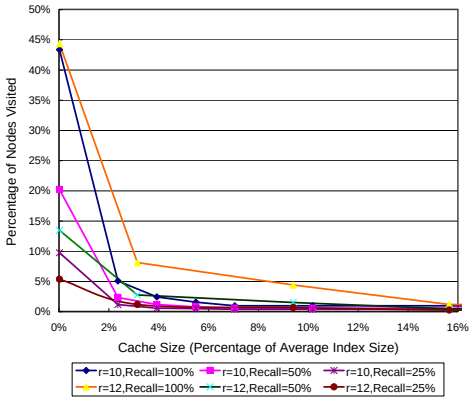


Figure 9. Query performance—with cache.

but whether σ_2 is semantically closer to K is up to applications, although our index scheme allows σ_2 to be retrieved before σ_1, or vice versa. Our goal here is not to build a text-search documents retrieval system, but a general purpose keyword/attribute search layer on top of which various applications—including documents retrieval—can be built. Content-based full-text search in large peer-to-peer distributed information systems has been investigated in pSearch [12] by gracefully incorporating information retrieval techniques into CAN (Content-Addressable Network) [7]. A typical implementation of pSearch requires a CAN of dimension ranges from 50 to 350 [12], which is an overkill for searching objects with only a few to dozens of keywords in their metadata, such as multimedia documents, computing resources, and web services. We believe that a good keyword/attribute search layer must offer a deterministic yet flexible search so that all objects matching some specified attributes can be precisely located, and returned in a way the applications wish, with minimum knowledge on the global state. Such function is particularly useful in resource and service discovery, and in multimedia documents sharing; and these are the target of our work.

References

[1] M. Balazinska, H. Balakrishnan, and D. R. Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proc. Pervasive 2002*, *LNCS 2414*, pp. 195–210.

[2] O. D. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, MIT, June 2002.

[3] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. on Computers*, 38(9):1249–1268, September 1989.

[4] K. S. Jones. Index term weighting. *Information Storage and Retrieval*, 9(11):619–633, 1973.

[5] M. Mitra, A. Singhal, and C. Buckley. Improving automatic query expansion. In *Proc. SIGIR '98*, pp. 206–214.

[6] K. Nakauchi, Y. Ishikawa, H. Morikawa, and T. Aoyama. Peer-to-peer keyword search using keyword relationship. In *Proc. GP2PC 2003*, pp. 359–366.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. SIGCOMM 2001*, pp. 161–172.

[8] P. Reynolds and A. Vahdat. Effi cient peer-to-peer keyword searching. In *Proc. Middleware 2003*, *LNCS 2672*, pp. 21–40.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, *LNCS 2218*, pp. 329–350.

[10] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP: Hypercubes, ontologies and effi cient search on p2p networks. In *Proc. AP2PC 2002*, *LNCS 2530*, pp. 112–124.

[11] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM 2001*, pp. 149–160.

[12] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. SIGCOMM 2003*, pp. 175–186.

[13] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing.

Technical Report UCB/CSD-01-1141, University of California, Berkeley, April 2001.

[14] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. D. Joseph, and J. Kubiatowics. Approximate object location and spam fi ltering on peer-to-peer systems. In *Proc. Middleware 2003*, *LNCS 2672*, pp. 1–20.