KISS: A Simple Prefix Search Scheme in P2P Networks*

Yuh-Jzer Joung
joung@ntu.edu.tw

Li-Wei Yang
willie@eland.com.tw

Department of Information Management
National Taiwan University
Taipei, Taiwan

ABSTRACT

Prefix search is a fundamental operation in information retrieval, but it is difficult to implement in a peer-to-peer (P2P) network. Existing techniques for prefix search suffer from several problems: increased storage/index costs, unbalanced load, fault tolerance, hot spot, and lack of some ranking mechanism. In this paper, we present **KISS** (*Keytoken-based Index and Search Scheme*), a simple and novel approach for prefix search to overcome the above problems.

1. INTRODUCTION

Prefix search is a fundamental search operation in information retrieval (IR). It allows users to retrieve desired objects even though they have only partial information about the objects. For example, a query like "*comp**" can match any objects with keywords *computer*, *company*, or *competitor* that have prefix *comp*. Prefix search is often used in combination with keyword search, for example, like "*ACM SIG* proceedings*" to search proceedings from all ACM special interest groups.

Prefix search is more general than keyword search, as the latter can be viewed as a special case in prefix search. So a system that supports prefix search can easily facilitate keyword search, but not vice versa. There are, however, some techniques to extend keyword search to prefix search. The most common way is to use the *n-gram* technique [16, 7, 9]. This technique augments each keyword w with all its prefixes to describe an object associating with w. For example, in Fig. 1, object O_1 has three keywords *abc*, *abd*, and *ce*. By expanding the keywords with their prefixes, we have the following six keywords to describe the object: *a*, *ab*, *abc*, *abd*, *c*, and *ce*. Using this technique, a prefix query of "ab*" in Fig. 1 can be converted into an ordinary keyword search with query "ab" to retrieve the matching objects O_1, O_2, and O_3.

To implement keyword search, an *inverted index* data structure is typically used. The idea is to "reverse" the object-keyword mapping. That is, for a given list of pairs (σ, K) specifying the set of keywords K associating with each object σ, we build a list of pairs (w, O) specifying the set of objects O that have keyword w. For example, Fig. 1 shows the inverted index list after keyword expansion. With the inverted index, keyword search is easy to accomplish: simply look up the list for the queried keyword, and return the object set associated with it. Some set operations may also be performed if necessarily. For example, suppose one wishes to retrieve objects with three prefixes in Fig. 1: *ab**, *ad**, and *ce**. Then by looking up the entries for *ab*, *ad*, and *ce* and then performing an intersection of the corresponding object sets, the system returns the matching object O_2.

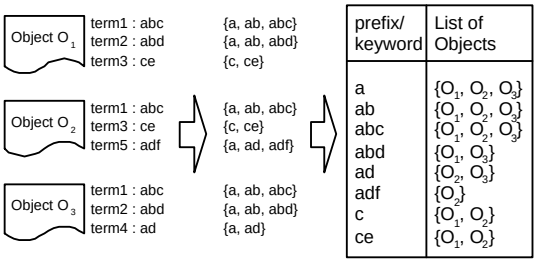

Figure 1: An inverted index of three objects.

The above approach, although commonly used in existing centralized information systems, suffers from several serious problems when implemented in a distributed environment, in particular, P2P networks. First, keyword frequency—the count of a keyword's occurrences in objects—varies enormously. This has been observed in many real world corpora. So a straightforward way of distributing keyword-object pairs to peers in a P2P network would result in a very unbalanced load. The problem is magnified when taking prefix search into account. For example, in English, many words have a common prefix, e.g., *in-*, *co-*, etc. In a preliminary study, we examined three datasets and obtained the following statistics:[1] the top 0.1% most frequent keywords already account for 43.4% of the total occurrences of the keywords, but after prefix expansion, the percentage increases to 62.6%. Therefore, simply mapping each entry in an expanded inverted index to a node makes the indexing load extremely uneven.

Secondly, the n-gram technique will expand the (original) keyword set size by approximately l-fold, where l is the av-

*This work was supported in part by the National Science Council, Taipei, Taiwan, Grants NSC 93-2213-E-002-096 and NSC 94-2213-E-002-036.

[1] The three datasets are: (1) half million of book entries from a library, (2) 300,000 news articles from Reuters, and (3) half million music records from FreeDB. The statistics is the average of them.

erage keyword length (note that some keywords may have a common prefix). We see that even for keyword search only, "distributed" inverted index already makes object insert, delete, and maintenance very expensive. This is because if an object has k keywords, then any insert/delete of the object has to access k nodes. When the keyword set size is expanded to $l \times k$, the maintenance cost increases by l-fold as well.

Third, although an object is indexed at several places, each prefix/keyword is still handled only by a single node. So any failure to the node would then block all queries involving the prefix/keyword. The system is also vulnerable to hot spots, as nodes responsible for some popular prefixes may be queried much more frequently than the others.

Finally, the above approach lacks some *ranking* mechanism to allow the system to return objects in sequence. In general, a short prefix query may result in many possible matching items. One would certainly prefer some ranking mechanism to help select relevant objects. For example, a query of "*comp**" may return objects with keywords *computer*, *company*, or *competitor*, etc. However, for a node that indexes the prefix *comp*, it has no idea the actual keyword sets the matching objects have—unless the node also maintains their object-keyword information. The latter, unfortunately, significantly increases the index cost.

In this paper, we present a simple yet effective index mechanism for prefix search in P2P networks that eliminates the above problems. We call our system **KISS** (*Keytoken-based Index and Search Scheme*), as it uses a novel technique to extract *keytokens*—character-position information—from keywords to index objects over a hypercube. Below we present the index and search scheme, followed by some experimental studies of the system. Related work and conclusions are given in Section 4 and Section 5, respectively.

2. KEYTOKEN-BASED INDEX AND SEARCH SCHEME

In this section we present KISS. We begin by presenting a very fundamental process in KISS called *tokenization*, which extracts characters and their position information in a keyword. Then we describe how to index objects in a hypercube using the character-position information extracted from their keywords. Based on this index scheme, we present search mechanisms to retrieve objects.

2.1 Tokenization

Let $\mathcal{A}$ be the set of alphabets in consideration. A *keytoken* is a pair $\langle c, i\rangle$, where $c \in \mathcal{A}$ and i a nonnegative integer. For notational simplicity, we sometimes write $\langle c, i\rangle$ as ci when no confusion is possible. Let $\mathcal{W} \subset \mathcal{A}^+$ be the set of keywords used in the system. For each keyword $w \in \mathcal{W}$, we use $w[i]$ to denote the i^{th} character of w, and $||w||$ to denote the length of w. A *tokenization* is a function τ that extracts all keytokens from a given keyword w; that is

$$\tau(w) = \Big\{ \langle w[i], i\rangle \,|\, i \le ||w|| \Big\}$$

We call $\tau(w)$ the keytoken set of w.

For example, $\tau(webdb) = \{w1, e2, b3, d4, b5\}$. Note that the character positions in a valid keytoken set (a keytoken set is *valid* if it is extracted from a word) must be continuous and start from 1; i.e., 1, 2, ... Also note that given $\mathcal{W}$, the number of all possible keytokens that can be extracted from $\mathcal{W}$ is no greater than $|\mathcal{A}| \times l_{\max}$, where $l_{\max}$ is the maximum length of a keyword. We shall use $\mathcal{T}$ to denote the set of all possible keytokens considered in the system.

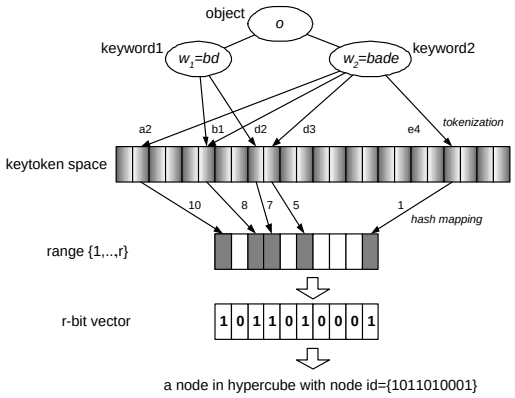

Figure 2: The KISS index scheme.

2.2 Index Scheme

Our index scheme is based on an r-dimensional hypercube $H_r(V, E)$. The hypercube can be constructed directly from a physical hypercube, or conceptually built on an underlying DHT-based P2P network $G = (V', E')$ (e.g., [19, 14]). To construct $H_r(V, E)$ over $G = (V', E')$, we simply need a mapping $g : V \to V'$ so that every logical node in the hypercube has a corresponding physical node in the network. As most DHT-based P2P networks have a mechanism to handle absence of nodes that are responsible for some identifier keys, we assume the hypercube, on which our index scheme is based, is reliable and self-organizing. In the rest of the paper by "nodes" we refer to the nodes in the hypercube. Each node has a unique r-bit binary string as its id. We use $u[i]$, $1 \le i \le r$, to denote the i^{th} bit of u (counting from the right). Below we describe how objects are indexed at the nodes in the hypercube.

Recall that $\mathcal{T}$ is the set of all possible keytokens. Let $h : \mathcal{T} \to \{1, \ldots, r\}$ be a hash function that uniformly and independently maps every keytoken in $\mathcal{T}$ to an integer in $\{1, \ldots, r\}$. We define a mapping $\mathcal{F}_h : 2^{\mathcal{T}} \to V$ as follows: $\mathcal{F}_h(T) = u$ if, and only if, $\{i \,|\, u[i] = 1, 1 \le i \le r\} = \{h(t) \,|\, t \in T\}$. In other words, $\mathcal{F}_h(T)$ is the node with a binary id whose bits are set by h according to the keytokens in T. For example, suppose $h(w1) = 3$, $h(e2) = 1$, and $h(b3) = 7$, and $r = 8$. Then the keytoken set $\{w1, e2, b3\}$ is mapped to the node 01000101.

We say that a node u is *responsible* for a keytoken set T if $\mathcal{F}_h(T) = u$. Thus, for every possible set of keytokens in the system, there is exactly one node in the hypercube responsible for the set. Note that due to hash collision, a node may be responsible for more than one set of keytokens (as $\mathcal{F}_h(T)$ might be equal to $\mathcal{F}_h(T')$ for some $T \neq T'$).

To index objects at nodes, let σ be an object and K_σ be the set of keywords associated with σ. Let $T_\sigma = \cup_{w \in K_\sigma} \tau(w)$ be the set of keytokens extracted from the keywords of σ. Then, σ is *indexed* at the node u such that $\mathcal{F}_h(T_\sigma) = u$. It should be clear that when an object σ is indexed at a node u, u needs to maintain, in addition to the keyword set of σ, the actual location information of σ (e.g., source IP, port, and file path of the object) from where σ can be retrieved. For simplicity, we sometimes say that a node u has indexed a keyword w if w belongs to a keyword set of some object indexed at u. Fig. 2 illustrates the index scheme for an object o that has two keywords *bd* and *bade*. The object is indexed at node 1011010001.

2.3 Search Strategies

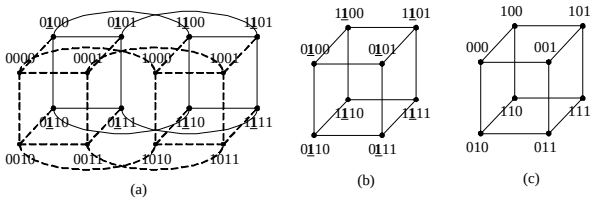

Figure 3: (a) H_4, (b) $\mathcal{H}_4(0100)$, (c) H_3.

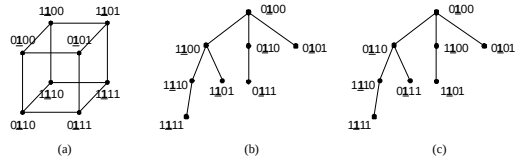

Figure 4: $\mathcal{H}_4(0100)$ (a) and its two spanning binomial trees (b) and (c).

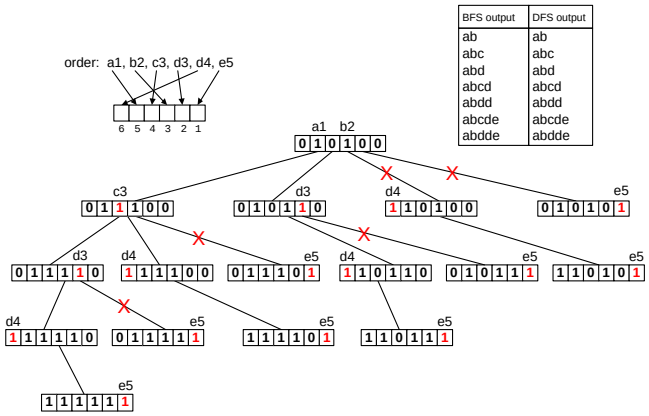

Figure 5: Search in a spanning binomial tree.

In this section we present search methods in KISS. We first observe that, due to our index scheme, *pin search*—given a keyword set K, locating the object that is associated exactly with K—is straightforward. This is because given K, we simply need to extract the keytoken set $T = \cup_{w \in K} \tau(w)$, and then find the node responsible for T, that is, the node $\mathcal{F}_h(T)$. The node $\mathcal{F}_h(T)$, given its identifier, can be easily located using the routing scheme in the underlying DHT network. From the node, by a local search of its index table, it can return the actual location of the object.

For example, in Fig. 2, to search objects with keyword set $\{bd, bade\}$, we first obtain the keytoken set $T = \{a2, b1, d2, d3, e4\}$, and then find the node $\mathcal{F}_h(T) = 1011010001$. Then we can issue a query to node 1011010001 to retrieve the objects. Pin search is particularly useful when one wishes to locate an object for maintenance (e.g., updating its index record), or when one has a precise description about his target object.

We now turn our attention to prefix search, where given a prefix w, we need to retrieve objects that contain a keyword w' such that w is a prefix of w'. We shall use '$\preceq$' to denote the prefix relation. So $w \preceq w'$ means that w is a prefix of w'. We begin by noting that if a node u is responsible for the prefix w, i.e., $\mathcal{F}_h(\tau(w)) = u$, then for every w' such that $w \preceq w'$, every node that may index w' must have an identifier v satisfying the following condition: $u[i] = 1 \Rightarrow v[i] = 1$, $1 \leq i \leq r$. So, to search objects whose keyword sets contain prefix w, we need only to search nodes that have bit '1' at the positions that u also has, i.e., at the positions $\{h(t) \,|\, t \in \tau(w)\}$. These nodes, in fact, form a "subhypercube" of H_r, as defined below:

Definition 2.1. *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 $v \in V$ is in U if and only if $u[i] = 1 \Rightarrow v[i] = 1$, and every edge $e \in E$ is in F if and only if its two end points are in U.*

Fig. 3 illustrates $\mathcal{H}_4(0100)$ induced by node 0100 in H_4, which is isomorphic to H_3.

So, given a prefix w, nodes which may index a keyword w' with prefix w must be in the subhypercube induced by $\mathcal{F}_h(\tau(w))$. For example, in Fig. 2, since $\mathcal{F}_h(\{b1, d2\}) = 0011000000$, all nodes that may index a keyword beginning with bd must have an identifier like xx11xxxxxx, and all such nodes are in the subhypercube induced by node 0011000000.

To explore nodes in a hypercube, we recall that nodes in a hypercube can be traversed via a *spanning binomial tree* [10]. There are many ways to construct a spanning binomial tree. For example, Fig. 4 illustrates two spanning binomial trees of the hypercube induced by node 0100. For our purpose, we need a spanning binomial tree that can assist ranking and even reduce the search space. For example, consider Fig. 5, which shows a spanning binomial tree of the subhypercube induced by node 010100. The root node 010100 is responsible for the keytoken set $\{a1, b2\}$. Nodes 011100, 010110, 110100, and 010101 are the first level nodes in the tree. We identify the keytoken that sets a corresponding bit to one on top of the bit. For example, keytoken $c3$ sets bit 4 to one. We note that the tree is arranged so that if we explore the tree in a breadth-first style (BFS), we can first locate objects with keyword ab, then with keyword abc, then with keyword abd, and so on. In contrast, a depth-first search (DFS) will first locate objects with keyword ab, then with abc, then with abd, and so on. It can be seen that, with a few exceptions, keywords searched in BFS are sorted in length and then in alphabetic order, while DFS retrieves keywords sorted in alphabetic order and then in length (i.e., in lexicographical order).

Another important feature to observe is that not all of the nodes need to be searched. For example, the node 110100 cannot possibly index any object. To see this, recall that the character positions in a valid keytoken set must be continuous and start from 1. For a keytoken set T, we define a function pos to project the positions in T; that is, $pos(T) = \{i \,|\, \langle x, i\rangle \in T\}$. For a given node u in our index hypercube, let T_u be the maximal keytoken set responsible by u. Then, u cannot index a keyword w of length l if $pos(T_u)$ does not contain $\{1, 2, \ldots, l\}$. The node 110100 is responsible for the keytoken set $\{a1, b2, d4\}$. However, since it does not contain a keytoken with position 3, it cannot index a keyword of length greater than two. In fact, the node cannot even index any object, as any object with a keytoken set $\{a1, b2\}$ will be indexed at node 010100, not at 110100. Likewise, its child node 110101, which is responsible for the keytoken set $\{a1, b2, d4, e5\}$, cannot index any object, either. Such kind of nodes that cannot possibly index any object are called *null* nodes. They allow us to reduce the hypercube search space.

To construct a spanning binomial tree like Fig. 5, we need some ordering on keytokens in $\mathcal{T}$. Assume the alphabets in $\mathcal{A}$ is totally ordered by a relation '$\leq$'. Then, we define a

total order '$\leq$' over keytokens in $\mathcal{T}$ as follows:

$$\langle a, i\rangle \leq \langle b, j\rangle \text{ iff } i < j \vee (i = j \wedge a \leq b)$$

For example, let $\mathcal{A}$ be the English letters with alphabetical ordering. Then $a1 \leq b2 \leq c2$.

The total order allows us to determine which node to choose first when "spanning" a binomial tree from a given root node. The complete definition is somewhat complex. Due to space limitation, it will be given in the full paper.

2.4 Guided Depth-First Search

The hypercube search space can be further reduced by noting that some keytokens are highly correlated. For example, in English, many words begin with $de*$, so $d1$ has high probability to occur with $e2$ simultaneously. Similarly, $t4, i5, o6$, and $n7$ are highly correlated (representing words $???tion$). To investigate correlation between keytokens, we analyzed four large databases from the real world (see Footnote 1) and all found the correlation feature among their keytokens. This property allows us to *cluster* highly correlated keytokens so that we can first explore them when traversing within a spanning binomial tree. The correlation allows us to have a higher probability to retrieve objects than exploring other keytokens, and therefore to reduce search costs.

Based on this, we develop a search strategy called **Guided Depth-First Search (GDFS)** to explore spanning binomial trees. When a node u is to traverse its children nodes in a depth-first style, it first asks its children nodes in the next level to see how many matched objects they have. Then node u collects the results, sorts them in a descending order, and then uses the order to decide which child node to traverse first. Each internal node in the tree uses the scheme recursively to traverse its subtree. Note that keytoken correlation is relatively stable. So the next-level information of a node can be cached at the node and need not be inquired in every search operation.

2.5 Summary

We summarize some important features of KISS. First, we observe that, even though we have extracted keytokens from the keyword set of an object and use the keytoken set to determine an index node for the object, the index node is still determined uniquely by an object's keyword set. So even if several objects all contain some popular keywords, and the keywords share some common prefixes with other keywords, the keytoken sets of these objects are still likely to differ in some way, and so the objects are likely to be indexed by more than one node. The more the popularity of the prefixes/keywords, the more the number of objects containing these prefixes/keywords, and so the more the number of nodes to index the objects. So indexing load can be balanced even if prefix/keyword frequency follows Zipf's law with sharp slope. Moreover, since there are a number of nodes to index a prefix/keyword, no single node failure can deny all queries involving the prefix/keyword.

Secondly, 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 take only one lookup operation in the P2P overlay, as opposed to about $k \times l$ operations needed by the distributed inverted index in combination with the n-gram technique introduced in Section 1, where k is the number of keywords an object has, and l is the average keyword length. Replication certainly helps increase fault tolerance (at the cost of extra storage and consistency maintenance), but this is up to the applications. If one wishes, 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 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.

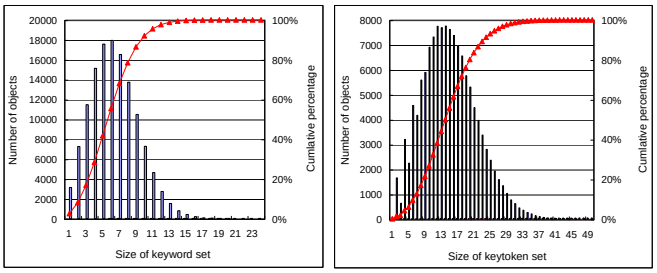

Figure 6: The distribution of the sizes of keyword set (left) and keytoken set (right).

Finally, objects in KISS are easily distinguished by the *characters* and the *positions* of them in the keyword sets they associate. In particular, given a prefix w, one can easily locate nodes that index a word w concatenated with one more specific character, with two more specific characters, and so on. So by exploring different search style, KISS can easily facilitate some kind of ranking on objects, e.g., by sorting the matching keywords in lexicographical order or in alphabetical order.

3. EXPERIMENTAL RESULTS

In this section we present experimental results for evaluating the performance of KISS. We use the the website records collected at PChome (http://www.pchome.com.tw, a local portal in Taiwan) as our dataset. The dataset consists of 131,180 website records in Chinese, where each record contains the following six fields: ID, Title, URL, Category, Description, and Keyword. We chose the dataset because Chinese is character-based and so prefix search is particularly important.

Each record is treated as an object to be indexed in KISS, and the set of words in the Keyword field is treated as the keyword set associating with the object. We decompose each keyword in the keyword set into keytokens as described in Section 2.1. The distribution of keyword set sizes is shown in Fig. 6, left. On average, each object is associated with 6.3 keywords, and there are a total of 117,320 distinct keywords. The distribution of keytoken set sizes after tokenization is also shown in Fig. 6. On average each object is associated with 15 keytokens, and so the average size of keytoken set increases to about 2.4 times of the size of a keyword set.

Before the experiments, we need to determine the dimensionality r of the hypercube in KISS. It can be seen that if r is too small, then many different keytokens are likely to be hashed to the same bit position, and so the collision will be high. On the other hand, a large r may result in a huge but "sparse" index hypercube, as there will be 2^r nodes, but many of them have little index load. This makes the search costs considerably high. So selecting an appropriate r is crucial to the performance of the system. A detailed analysis of the choice of r will be provided in the full paper. In the experiment we set r to 16.

In the first experiment we measure load distribution of KISS. We built a hypercube of dimension $r = 16$, and assigned 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 percent-

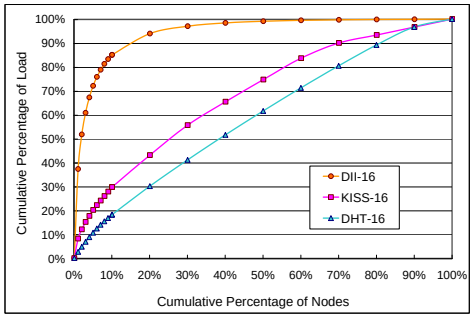

Figure 7: Load distribution.

age of objects each node handles. The results are shown in Fig. 7 on the line marked as "KISS-16". For comparison, we observe that most P2P networks use some hash functions (e.g., SHA-1) to distribute objects to nodes, and the results are generally considered as "well balanced". So we also draw the load distribution that simply uses a hash function to distribute objects to nodes in the hypercube. The line is marked as "DHT-16" as a benchmark. Note that the scheme is used only for reference; it cannot support any search operation.

In addition, we also studied the load distribution of the distributed inverted index (DII) scheme discussed in Section 1. For this, we expand each keyword w with $||w|| - 1$ additional keywords taken from all possible prefixes of w. Then we built an inverted index list of the keywords (see Fig. 1). We then hash all the keywords to nodes to determine which node is responsible for indexing which keyword. The node that is responsible for a keyword then maintains the list of objects that have the keyword. Note that in this scheme an object may be indexed by more than one node. We measure a node's load by the number of objects it indexes as a percentage of the sum of the numbers of objects indexed by all nodes. The load distribution of this scheme is also drawn in Fig. 7 by the line "DII-16".

From the figure we see that DII results in an extremely unbalanced load. The top 10% nodes account for 85% of the system's total loads! In contrast, KISS is significantly better than DII in terms of load balance. However, there is some room between KISS and DHT. This is because in KISS, nodes that have few bit-1's are likely to index no objects. Future work will consider how to let the nodes share loads with others.

In the second experiment, we study prefix search performance in KISS. We measure the number of nodes need to be visited with respect to a given recall rate. To issue the queries, we use query logs collected at PChome, and randomly sample some of them as our experimental queries. There are not many prefix queries, though. Therefore, we use the following method to convert real-life keyword queries to prefix queries: For each sampled query q, let K_p be the set of keywords issued in the query. If K_p is a singleton, we extract the prefix of length m from the keyword as our prefix query, where m is a simulation parameter to be used later. If, however, K_p contains more than one keyword, say, $w_1, \ldots, w_k$, then we averagely take prefixes of them, say $w'_1 \preceq w_1, \ldots, w'_k \preceq w_k$, so that the total length of the prefixes $w'_1, \ldots, w'_k$ is m. Nevertheless, a majority of the queries are single-keyword queries.

We vary prefix length $m = 2, 3, 4, 5, 6$, and for each m we use 500 sample prefix queries of the length to evaluate search performance of KISS under the following three search strategies: BFS, DFS, and GDFS+tree pruning (recall that a spanning binomial tree can be pruned to eliminate search paths from null nodes). The results are shown in Fig. 8.

From the results we see that, in general, the longer the queried prefix length, the smaller the number of nodes need to be visited. We note that each line is composed of some segments. For example, the result of $m = 2$ in BFS has a curve from (0,0) to about (92%,25%), then to (100%,50%). To explain this, observe that the size of a spanning binomial tree induced by a node u is 2^{r-n}, where n is the number of bit-1's u has. Let T_q be the keytoken set extracted from a query q. Then the search space of q is determined by the number of different values $h(t)$ maps to, $t \in T_q$ (see Section 2.2). If $|\{h(t) \,|\, t \in T_q\}| = n$, then 2^{-n} of the total nodes need to be searched in the worst case. For a prefix query q of length m, where m is small, it has high probability that T_q (which consists of m keytokens) will be mapped to a node with m bit-1's. So the search space is 2^{-m} of the network size. Since index load in KISS is sort of balanced, the number of nodes visited grows in proportion to the recall rate. For example, if $m = 2$, then in the worst case 25% of nodes need to be visited to reach 100% recall rate. However, due to hash collision, some prefix of length 2 may be mapped to a root node with only one bit-1. For these queries, the search space vs. recall-rate line grows from $(0, 0)$ to (50%,100%). By combining the two possible cases for $m = 2$, we have the two-segment curve in Fig. 8. The other cases for different m are similar.

We also note that on average GDFS with tree pruning performs much better than DFS, which performs much better than BFS. It is not surprised to see that GDFS with tree pruning outperforms both DFS and BFS (see Section 2.4). To see why DFS outperforms BFS, recall that a node with fewer bit-1's is less likely to index an object as compared to a node with more bit-1's. So when m is small, nodes at the bottom of a spanning binomial tree to be searched tend to index more objects than nodes at the upper levels. So BFS causes more "empty" nodes to be visited in the early stage of search than DFS does. So BFS requires more nodes to be visited than DFS with respect to the same recall rate.

4. RELATED WORK

Search has been one of the key issues in fully decentralized P2P networks. When the network is Gnutella-like unstructured, each node often locally maintains objects it shares to the network. Since query messages are passed around nodes to check if they have the desired targets, a variety of searches (including, of course, prefix search) can be offered. However, since search is basically a blind process traversing around the network, in general, search space size grows in proportion to recall rate. In the worst case, the entire network needs to be searched to retrieve an object. So search is unscalable. The work of [12, 4] focuses specifically on keyword search in unstructured P2P networks.

Structured P2P networks like DHTs basically support only exact name match, as objects are given a unique identifier obtained by hashing their names to determine their locations in the network. Keyword search must be built on top of the overlay to enhance search functionality. Several mechanisms have been proposed for keyword search in DHTs (e.g., [15, 18, 6, 11]), but all of them, except [11] on which our work is built, use inverted index as the primary data structure. Moreover, they do not consider prefix search.

In addition to keyword search, range queries in structured P2P networks have recently attracted much attention (e.g., [2, 17, 8, 1, 13, 5]). Range queries concern search of objects with keys in a given range. They are useful, for example, when one wishes to search computing resources

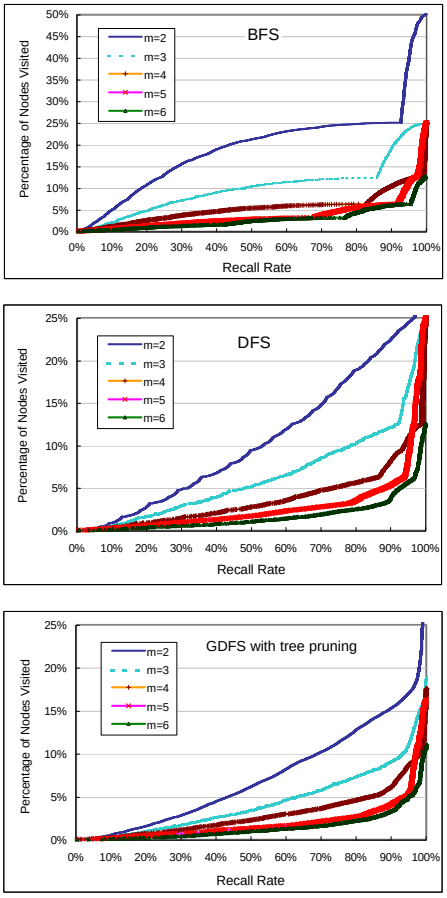

Figure 8: (colored) Query performance of (a) BFS (b) DFS, and (C) GDFS. Note that BFS is on a larger scale than the other two.

with CPU clock ranging in between 2-4GHz. Range queries are related to prefix search, as the latter can be translated into a range query covering all objects with a given prefix. There is subtle difference, however. Range queries often concern one attribute at a time, and the research focus is on how to efficiently retrieve adjacent keys as well as to balance load when key distribution may skew. Multi-attribute range queries [3], e.g., "search for computing resources with 2-4GHz CPU clock, at least 1GB RAM, and disk space 80-200GB," are considered more complicated, if one does not wish to tackle the problem one dimension at a time and then take a join operation of the results. In contrast, prefix/keyword search naturally allows multiple words to be issued and, unlike distributed inverted index, our hypercube index scheme is able to tackle a query without the need of a join operation among some potentially large object sets.

5. CONCLUSIONS

We have presented a simple and novel keytoken-based index and search scheme, KISS, for prefix search in structured P2P networks. The idea is to tokenize every keyword associated with an object, and then hash the keytoken set into an r-bit vector. This r-bit vector represents a node in an r-dimensional hypercube. By mapping the conceptual hypercube to the underlying DHT network, we can locate the object using only one lookup operation in the network. Therefore, object insert, delete, and maintenance can be done very efficiently and effectively. Moreover, the index scheme allows nodes to evenly distribute their loads in indexing popular keywords and prefixes. It also allows us to retrieve words that grow in length incrementally and alphabetically—a key feature for implementing prefix search. This feature also supports some form of ranking as, by applying different exploring strategies, different ordering of objects can be retrieved. In contrast, a traditional approach that uses distributed inverted index in combination with n-gram technique suffers from extremely unbalanced load, high maintenance costs, and unable to support ranking.

6. REFERENCES

[1] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. In PODC 2004, pages 115–124.

[2] B. Awerbuch and C. Scheideler. Peer-to-peer systems for prefix search. In PODC 2003, pages 123–132.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In SIGCOMM 2004, pages 353–366.

[4] H. Cai and J. Wang. Foreseer: a novel, locality-aware peer-to-peer system architecture for keyword searches. In Middleware 2004, LNCS 3231, pages 38–58.

[5] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In P2P 2005.

[6] P. Ganesan, Q. Sun, and H. Garcia-Molina. Adlib: A self-tuning index for dynamic peer-to-peer systems. In ICDE'05, pages 256–257.

[7] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. *Information retrieval: data structures and algorithms*, pages 66–82, 1992.

[8] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In CIDR 2003, 2003.

[9] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In IPTPS 2002, pages 242–259.

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

[11] Y.-J. Joung, C.-T. Fang, and L.-W. Yang. Keyword search in DHT-based peer-to-peer networks. In ICDCS 2005, pages 339–348.

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

[13] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: Prefix hash tree. In PODC '04, pages 368–368.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In SIGCOMM 2001, pages 161–172.

[15] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In Middleware 2003, pages 21–40.

[16] G. Salton. *Automatic Text Processing. The Transformation, Analysis and Retrieval of Information by Computer*. Reading, MA: Addison-Wesley, 1988.

[17] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[18] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen. Making Peer-to-Peer Keyword Searching Feasible Using Multi-level Partitioning. In IPTPS 2004, pages 151–161.

[19] 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 SIGCOMM 2001, pages 149–160.