A Keyword-Set Search System for Peer-to-Peer Networks

by

Omprakash D Gnawali

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

© Omprakash D Gnawali, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
May 28, 2002

Certified by . . . .
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Keyword-Set Search System for Peer-to-Peer Networks

by

Omprakash D Gnawali

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Keyword-Set Search System (KSS) is a Peer-to-Peer (P2P) keyword search system that uses a distributed inverted index. The main challenge in a distributed index and search system is finding the right scheme to partition the index across the nodes in the network. The most obvious scheme would be to partition the index by keyword. A keyword partitioned index requires that the list of index entries for each keyword in a search be retrieved, so all the lists can be joined; only a few nodes need to be contacted, but each sends a potentially large amount of data. In KSS, the index is partitioned by *sets* of keywords. KSS builds an inverted index that maps each set of keywords to a list of all the documents that contain the words in the keyword-set. When a user issues a query, the keywords in the query are divided into sets of keywords. The document list for each set of keywords is then fetched from the network. The lists are intersected to compute the list of matching documents. The list of index entries for each set of words is smaller than the list of entries for each word. Thus search using KSS results in a smaller query time overhead.

Preliminary experiments using traces of real user queries show that the keyword-set approach is more efficient than a standard inverted index in terms of communication costs for query. Insert overhead for KSS grows exponentially as the size of the keyword-set used to generate the keys for index entries. The query overhead for the target application (metadata search in a music file sharing system) is reduced to the result of the query as no intermediate lists are transferred across the network for the *join* operation. Given our assumption that free disk space is plenty, and queries are more frequent than insertions in P2P systems, we believe this is a good tradeoff.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

This thesis is based on a suggestion from David Karger in the summer of 2001. To help me take this project from a basic idea to this stage, I am indebted to my advisor Frans Kaashoek, who not only gave his advice and encouragement, but also helped debug the writing of this thesis. Many thanks to Robert Morris for his help, and especially his suggestion for experiments.

I would like to thank Frank Dabek for his implementation of DHash, and the DHash append calls and for his help in KSS protoytpe implementation. I would also like to thank other members of PDOS for discussing this project with me, and Charles Blake for his willingness and success in helping me in anything I asked for.

I would like to thank my mom and dad for their encouragement.

Thanks to Madan for fixing some of my unnecessarily complicated sentences and thanks to John for buying me Burrito Max burritos.

Finally, I would like to thank Christopher Blanc, Paul Hezel and David Laemmle for bootstrapping my education.

Contents

Chapter 1

Introduction

Peer-to-peer (P2P) systems that enable music file sharing have become popular among the Internet users. Music file sharing P2P systems typically have millions of users [21]. Assuming each user contributes a few files to the network by *sharing* music files, there are tens of millions of files available in the network. The users are effectively pooling their storage and network resource to make a large number of files available to each other. A large number of files makes a music file sharing system attractive to a lot of users, but it also makes it harder to find the file that a user wants.

Keyword search system allows users to search files by specifying a few of the words from the desired metadata fields. User enters the search words, then the system communicates with other nodes and fetch the matching results. In the music file sharing systems, keyword search is done using centralized as well as distributed algorithms.

In a centralized scheme, a single node is responsible for maintaining a centralized index that maps words to files in a particular peer. All the queries are routed to the central server. Since, the server maintains the index for the entire network, the server can tell a client what files are matching and where to find those files. Napster [17] is an example of such a centralized system. Napster users connect to the Napster database of online songs, search for songs and get a pointer to the peers that are sharing those files. This architecture however creates a central point of failure. Since all the queries are being answered from a central server, the organization responsible for maintaining the central has to invest resources to make sure there is enough bandwidth and processing cycles available at the server end. The centralized nature of the service also makes systems like Napster vulnerable to denial of service attacks. These limitations of a centralized system make distributed scheme for searching the files attractive.

Gnutella [8] is a popular music file sharing system that uses a completely distributed search system. Each Gnutella node is responsible for maintaining a local index of the files it is sharing. The Gnutella nodes are organized in an overlay network over the Internet. When a Gnutella node starts, it discovers other nodes through a proxy and establishes network connection to them forming an overlay network. To search for a file in the Gnutella network, a broadcast protocol is used for searching successively deeper in the overlay network formed by the Gnutella nodes. When a node receives a query, the node runs the query in its local index and sends a reply with matching filenames. Thus, Gnutella avoids centralization, but at the expense of network flooding during a query. Broadcasting the query to all neighbors is necessary in the basic Gnutella protocol because the system does not know what files are being shared in the network ahead of time. Hence, every time a query is made, the system needs to ask other nodes if they have matching files.

One way to make a distributed search system efficient is by performing some pre-computation before queries are made. The system then uses data structures built and populated during pre-computation to do less processing and communication while answering queries. Building a search index is a popular pre-computation in a search system. When queries are made, these indices can be used to precisely locate the nodes serving the matching documents without having to communicate with all the nodes.

1.1 Distributed Inverted Indexing

An *Inverted Index* is a data structure that maps words to files and hence helps quickly find the document in which a given word appears. Building and updating an inverted index is a pre-computation step in which searchable words are extracted from a document. Once words are extracted, an associative list is created that associates each word with a document pointer (and optionally the position in which the word appears in that document). In the design presented in this thesis, the index does not contain the position for the word.

An inverted index can be built locally and served from a single server or site. Web search engines such as Google crawl the web, build the index locally and use it to quickly find the results for queries. P2P file sharing system like Napster also use a local index that gets updated when a user logs in to the Napster server and shares a file. The local index is used by the server to answer queries.

A distributed indexing system, instead of storing the entire index in one server (perhaps implemented as a server cluster), distributes the index to multiple nodes in the network. The challenge is finding the right scheme to partition the index across the nodes in the network. The simple solution is to partition the index by keywords in a document and store all index entries for a given keyword on a specific node. In this scheme, to process a query, the system needs to fetch index entries for each word in the query from the network and aggregate the results. If a user searches for *love you*, the system downloads index entries for *love* and *you* and intersects the lists. Since there are hundreds of thousands of songs with the word *love* in the title and hundreds of thousands of songs with the word *you* in the title, the system transmits a large number of index entries (potentially a few Megabytes in size) for each of these words. This is unacceptably large bandwidth for query in a P2P system because bandwidth available to most nodes in the internet is rather small [21].

Fetching index entries from multiple hosts and computing the aggregate result is similar to the *join* operation in a distributed database system. In a distributed database system, the standard optimization used to decrease the communication overhead to compute join of two lists that are in two different nodes involves transmitting the shorter of the two lists to the node that stores the longer list. Unfortunately the shorter of the lists for *love* and *you* is still a few Megabytes. Thus this optimization is unhelpful in making a P2P search efficient.

One can also send a compressed membership-set data structure such as Bloom filter to a node that has the second list. Upon receiving such a filter, the node sends the tuples that match the filter, thereby transmitting only the tuples that will be included in the final join operation [14] [16]. This work was explored by P. Reynolds and A. Vahdat in *Efficient peer-to-peer keyword searching* [20]

This thesis investigates an optimization on the simple solution that uses inverted index with a single word as a key. We present Keyword-Set Search System (KSS) that generates index entries with *sets of words* as a key.

1.2 KSS: Keyword-Set Search System

KSS, Keyword-Set Search System, is a P2P search system that uses a distributed inverted index. In KSS, the index is partitioned by a set of keywords. KSS extracts keywords from a document and builds an inverted index that maps each set of keywords to a list of all the documents that contain the words in the keyword-set. The keyword-set inverted index is computed as follows: Given the words A, B, and C in a document doc1, and a set size of two, the index entries are <AB, doc1>, <AC, doc1>, and <BC, doc1>. For a set size of three, the entry is <ABC, doc1>. All the index entries for a given keyword-set is stored on a particular node. The node is chosen by hashing the set of keywords, using consistent hashing [12] and Chord [22]. This causes the index to be evenly distributed over participating nodes.

Users search for files using keywords. The keywords in a query are divided into sets of keywords. The document list for each set of keywords is then fetched from the network. For example, if a query contains the keywords A, B and C, the system fetches index entries for AB and AC. Once all the lists have arrived, documents that appear in all the lists are the matching documents for the query. Given the index entries fetched for AB are <AB, doc1>, <AB, doc2>, and <AB, doc3>, the index entries for BC are <BC, doc1>, and <BC, doc2>, the matching documents for the query are *doc1* and *doc2*.

A KSS index is considerably larger than a standard inverted index, since there are more word sets than there are individual words. The main benefit of KSS is the low communication cost of performing a multi-word query. In a typical KSS setting, two-word queries involve no significant communication, since they are processed wholly within the node responsible for that keyword set. Queries with more than two words require that document lists be communicated and joined; but these lists are smaller than in a standard inverted index, because the number of documents that contain a set of words is smaller than the number of documents that contain each word alone. Single-word queries are processed using a standard single-word inverted index, and require no significant communication.

Thus KSS outperforms the standard distributed inverted scheme at the expense of storage overhead. Given that disk space is inexpensive and abundantly available in a P2P system, trading storage overhead for smaller query bandwidth is a good tradeoff that results in reduced network traffic during document location so more bandwidth can be used for file transfers.

1.3 Contribution

This thesis presents KSS. We describe two applications for KSS: Metadata search in a music file sharing network, and a full-text keyword search. We evaluate KSS using query traces from Gnutella network and web search. We found that KSS performs better as the number of keywords in a query increases compared to a standard inverted indexing scheme. This benefit comes at the cost of additional storage and insert overhead, compared to a standard inverted index.

1.4 Thesis Overview

Chapter 2 surveys related work. Chapter 3 presents the KSS design overview. In Chapter 4, we describe applications of KSS. In Chapter 5, we describe the system architecture of a KSS prototype implementation. In Chapter 6, we describe our evaluation model and present evaluation of KSS. Finally, we present conclusions and future work in chapter 7.

Chapter 2

Related Work

Recently a number of systems have been developed that support distributed search. Some of the relevant search algorithms in deployed P2P systems and the ones described in recent papers are surveyed in this section.

2.1 Napster

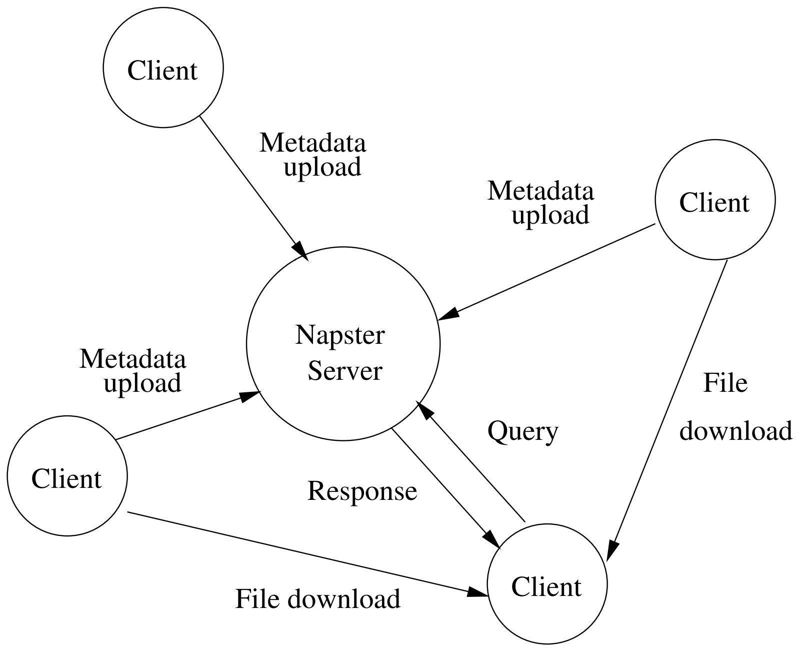


Figure 2-1: Clients connect to a central Napster server for content location. Files are downloaded from peers.

Napster [17] is system that allows people to share music. The search index is centralized; the storage and serving of files is distributed. When a node joins the system, the node sends a list of all the files to the Napster server. A user looking for a music file goes to the Napster and issues a query. The server searches its index, which is built using the metadata sent by the peers. The result for a query is a list of IP addresses of the peers that are sharing the file. The user then downloads the file directly from the peer. From file sharing perspective, Napster is a P2P system. However, from the indexing perspective, Napster is a centralized system.

2.2 Gnutella

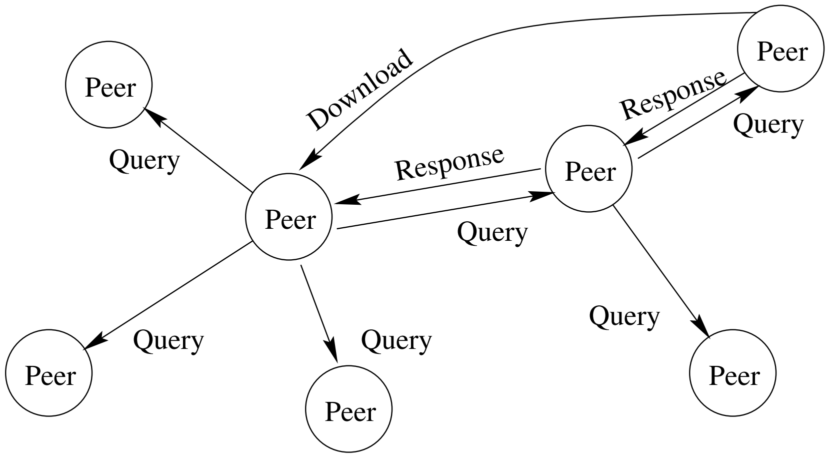


Figure 2-2: Gnutella uses a decentralized architecture document location and serving.

Gnutella is a P2P system for file sharing. Gnutella [8] search works by broadcasting the queries with non-zero TTL to all the neighboring hosts. Conceptually, this is doing a Breadth-First Search (BFS) in the network until the desired document is found. The querying node does not know which node might be sharing the desired file, so it has to broadcast the query to all its neighbors. If a file is popular, perhaps it is possible to find a node with the desired document after a few levels of search in the Gnutella network. If a file is unpopular, to find the file, one has to set the TTL of the query to a large number within the informal limit imposed by the clients in the Query Routing Protocol. Thus, the Gnutella system causes a lot of network flooding during a query. Ritter [11] presents an analysis of Gnutella search protocol suggesting that a search for a 18 byte string with a branching factor of 8 and a TTL of 8 could generate about 1.2 GB of aggregate traffic just for document location.

Gnutella developers have proposed many enchancement to the Gnutella protocol to make it more scalable, efficient and to increase search precision. The *Hash/URN Gnutella Extensions* [9] propose allowing search by using the content hash of documents, in addition to the keyword search, and transmitting SHA1 content hash along with the query results. This helps identifying the results that have different file names but are identical in content and enables search result folding by collapsing the entries for identical content, parallel downloading from multiple sources and easily cross index with foreign P2P systems that use content hash as file names. The basic Gnutella protocol matches query with the filenames. The *MetaData proposal* [23] suggests including the meta-data field along with the keywords in the query message so that the clients can search appropriate meta-data fields. This improves search precision. The *Query Routing Protocol* [3] proposes a system in which peers exchange keywords for the files they are sharing with their neighbors. In this scheme, instead of broadcasting a query to all the neighbors, peers transmit the queries only to the nodes that lead to a node with the desired file. *Ultrapeers* [2] are the nodes with with high bandwidth and processing power. Ultrapeers based Gnutella network has properties similar to the FastTrack based network we describe in section 2.3. Lv, Ratnasamy and Shenker [18] describe a a scheme that exploits the heterogeneity in the network by evolving the Gnutella topology that restricts the flow of queries to the nodes with limited bandwidth and enhances the flow of queries to the nodes with high bandwidth.

2.3 FastTrack P2P Stack

FastTrack [6] P2P Stack is a software development library for building P2P file sharing systems. It supports meta-data searching. In FastTrack, the nodes form a structured overlay of supernodes to make search more efficient. Supernodes are nodes with more

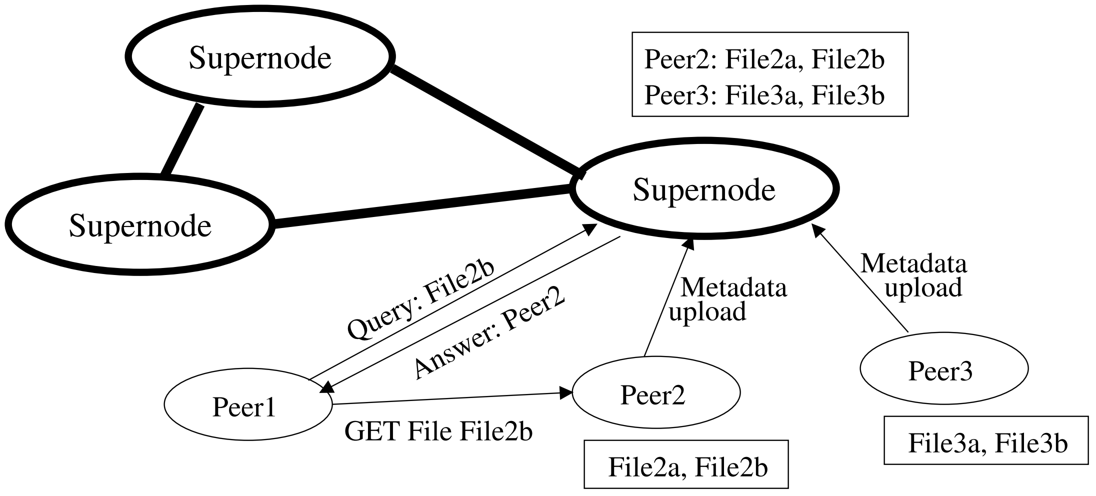


Figure 2-3: Peers connect to supernodes. Search is routed through the supernodes and downloads are done from the peers

bandwidth, disk space and processing power and have volunteered to get elected to facilitate search by caching the meta-data. The ordinary peers that do not have a lot of storage, processing or communication resource transmit the metadata of the files they are sharing to the supernodes. All the queries are also forwarded to the supernode. Gnutella-like broadcast based search is performed in a highly pruned overlay network of supernodes. The system can exist, in theory, without any supernode but this results in worse query latency.

This approach still consumes much bandwidth to maintain the index at the supernodes on behalf of the peers that are connected to the supernode. The supernodes still use a broadcast protocol for search, and the queries could be routed to peers and supernodes that have no relevant information about the query. Kazaa [13], and Grokster [10] are FastTrack applications.

2.4 JXTA Search

JXTA [24] is a P2P infrastructure with three layers of system software. The core layer consists of protocols for key mechanism of P2P networking such as node discovery and message transport. The services layer provides hooks for different P2P applications, such as, searching, and sharing. Finally the applications like file sharing applications are built on top of the services layer using its services.

The JXTA search system has single or a few search hubs which are chosen for their processing power and high bandwidth. The information providers register themselves with the search hub and upload metadata for the information they are providing to the network. The hub, upon receiving a query from a consumer, routes the queries to appropriate providers. Few details are known about the exact way searches work in JXTA.

2.5 Iterative Deepening and Directed BFS

Yang and Garcia-Molina [25] describe techniques to make queries efficient in Gnutella as well as Freenet style system. For Gnutella like Breadth First Search (BFS) flooding, they argue that iterative deepening (ID) can minimize the bandwidth consumed for each search while getting good results in a reasonable time. Instead of flooding the network up to N levels, ID requires that the nodes broadcast the queries only $N - k$ number of levels. If the result is not found, the depth of search can be iteratively increases until the result is found. ID helps save precious bandwidth because most of the searches in a Gnutella network is likely to be found within a small depth in the BFS search. If response time is critical, one can use directed BFS, which is a broadcast policy that selects a set of nodes from the list of neighbors thereby avoiding a search in all the neighbors. They also describe Directed BFS, which is a technique of broadcasting the queries to a subset of the neighbors instead of all the neighbors. The neighbors are selected based on their past performance. Directed BFS thus can be thought of as a middle ground between pure BFS and pure Depth First Search.

2.6 Local Indices

Local Indices [25] allow a node in a P2P network to answer queries on behalf of any node within r hops without forwarding a query. A node stores meta-data for all files shared by the nodes within r hops. This system allows a document that is D hops away to be found with a query with a TTL of $D - r$. This is similar to the super peer approach discussed in section 2.3 with a flexible parameter r. However there is the same kind of flooding involved for index building and querying. During index building, a node floods all the nodes within r hops. During searching, the nodes within $D - r$ hops still need to be queried.

2.7 Routing Indices

Routing indices [4] (RI) is a technique that uses a distributed index to return a list of neighbors ranked according to their *goodness* for the query so that queries can be forwarded to the nodes that are likely to have answer to the queries. A traditional inverted index stores the document pointer for each keyword. Routing index, instead of storing the document pointers for the keyword, stores data that allow a node to rank its neighbors according to their *goodness* for a query. Compound RI and Hop-count RI are examples of Routing Indices. Compound RI store the number of documents in each branch so a query can be forwarded to the branch with the highest number of documents in the relevant topic. Hop-count RI store the number of documents within a “horizon” so that a node can decide to choose the path that will return the documents with the least number of hops.

2.8 Discover Query Routing

The resource discovery system called *Discover* [15] uses content routers to route the queries to the information providers that can answer a query. The information providers register themselves with the content routers with content labels, a compact representation of the content they serve. Content labels are predicates that are true of all the documents a provider is sharing or all the documents reachable through a router. The labels are propagated throughout the network. The content routing system consists of a hierarchical network with information providers at the leaves and content routers as mediating nodes. A client issues a query to a content router. The router forwards the the query to other content routers for which the content label matches the query and finally reach the information provider.

2.9 Distributed Join using Bloom Filters

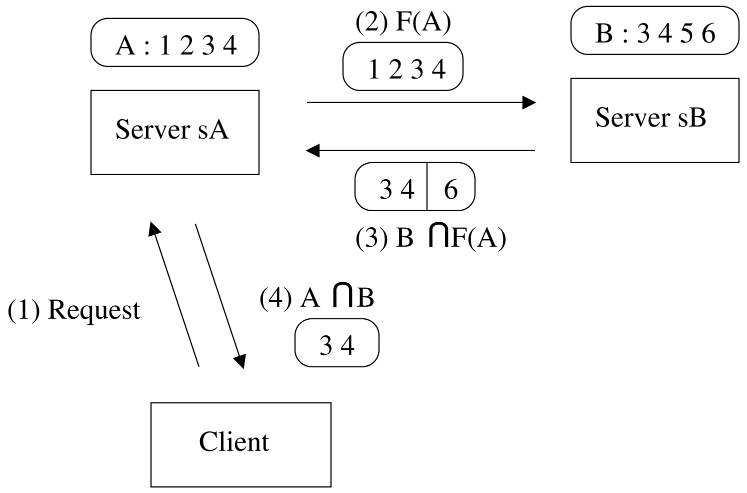


Figure 2-4: Using Bloom filters to compute the match for an *AND* query.

Techniques used for computing joins in a distributed databases can be used in P2P networks for efficient keyword searching [20]. This technique maintains a list of files for each keyword. The lists are stored in a distributed hash table implemented in a P2P system that maps keys to locations. To compute the result of a query consisting of more than one keyword, such as A *and* B, the keywords are sent to a node N_a that has a list of files for one of the keywords "A" in the query. The node N_a then sends a Bloom filter based on its list of documents for A, $F(A)$ to a node N_b that contains the list for the other keyword B. The filter is significantly smaller than the list itself. N_b upon receiving a bloom filter intersects the filter with its list of documents for B and sends the intersection, $B \cap F(A)$, back to the node N_a. The node N_a then computes $A \cap (B \cap F(A))$ which is the result for the query A *and* B. This technique can be extended to answer queries with more than two keywords.

Chapter 3

KSS Design Overview

KSS, Keyword-Set Search System, is a P2P search system that uses a distributed inverted index. A naive distributed inverted index scheme partitions the index by using each keyword in the document as a key for the index and maps that key to a particular node in the network. In this scheme, the list of index entries tend to be large and thus incur a lot of communication overhead during a query when the index entries are fetched.

KSS index is partitioned by a set of keywords (as opposed to a single keyword). The list of index entries for each set of words is smaller (since the list contains only the files that match all words in the keyword-set) and thus results in a smaller query time overhead. The downside to this scheme is, it has much higher insert and storage overhead because more index entries are generated and transmitted across the network when a document is inserted into the network.

Given that the disk space is inexpensive and abundantly available in a P2P system, trading storage overhead for smaller query overhead is a good tradeoff that results in reduced network traffic during document location. Studies have shown that 66% of the peers share no files [1]. These *free-riding* nodes issue no *insert* request and are only interested in querying and downloading the files. Thus, there are more nodes issuing query requests than the nodes issuing insert requests in the network. Thus, having a higher insert overhead only affects a small number of nodes that share files while most of nodes that only issue queries enjoy the low query overhead. Another study

[21] shows that peers that are connected to the internet with a high speed connection such as Cable, Dual ISDN, and DSL are more likely to share a file, while users with slower connection to the internet (Modem, ISDN) account for disproportionally large number of queries in the system. Thus, in a system that uses KSS, a large number of peers, and especially the slow peers that are mostly interested in searching for files and downloading them, will incur low indexing overhead and the peers with fast internet connection (because they tend to share more files than average), will incur most of the indexing overhead. We think this is a desirable property of an indexing scheme for a P2P network which tends to be heterogenous in network bandwidth available to the peers.

KSS works as follows: when a user *shares* a file, KSS generates the index entries for the file for each set of words in the file, hashes the keyword-set to form the key for the index entry, maps the keys to the nodes in the network using consistent hashing and Chord.

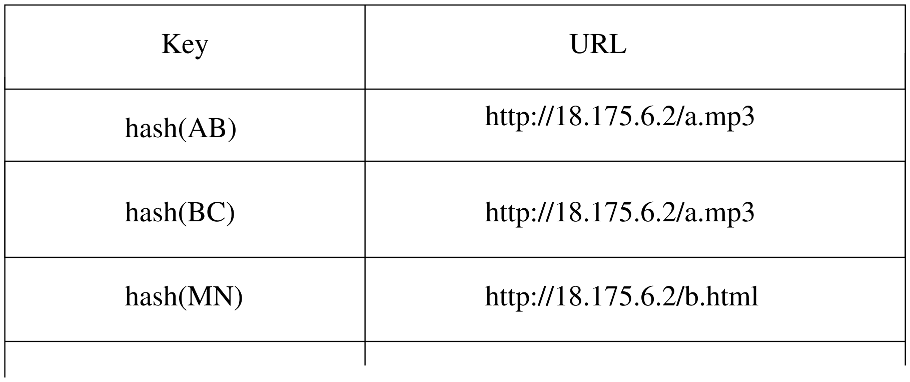
To find the list of documents matching a query, the system finds the nodes storing the index entries for each keyword set by hashing the keywords from the query, fetches each list, and intersects the results to find a list of documents that contain all the keywords in the query.

In this chapter, we describe the KSS distributed index with an example. Then we discuss the properties of KSS that determine its efficiency.

3.1 KSS distributed index

KSS uses a distributed inverted index to answer queries efficiently with minimal communication overhead. Each entry of the index contains (1) the hash of the searchable keywords or set of keywords as the index key, and (2) a pointer to the document. Document pointer contains enough information to fetch the file. URL and CFS [5] file path are examples of document pointers. Figure 3-1 shows a logical representation of the index.

To index a document, KSS extracts all the keywords from the document. These



Key	URL
hash(AB)	http://18.175.6.2/a.mp3
hash(BC)	http://18.175.6.2/a.mp3
hash(MN)	http://18.175.6.2/b.html

Figure 3-1: Structure of the KSS index

are the keywords that a user will later use to search the document. KSS then sorts the list of keywords, deletes the duplicate words and forms an ordered set called *allwords*.

KSS then computes the index entries from the set *allwords*. The size of the keyword-set is a configuration parameter to the KSS algorithm. If the size of keyword-set is two, KSS generates index entries for each pair of keyword in *allwords*. If the size of keyword-set is three, KSS generates index entries for each combination of three words from *allwords*. Given that the size of keyword-set is k, KSS generates index entries for all k-word combination from *allwords*. The number of index entries for a n-word document is given by $C(n,k) = \frac{n!}{(n-k)!k!}$. The key for an index entry is formed by hashing the keyword-set in the entry. Thus each key is computed by hashing k keywords. Conceptually this is the same as forming the keys by hashing the elements of k-way set product of *allwords*. The key for each index entry is then mapped to nodes in the P2P network using Chord. The entries are sent across the network to the node. The node appends the entries to its local list of entries.

To answer a query that contains k keywords, the peer computes a key to the index by hashing the keywords in the query. The key is again mapped to a node in the P2P network using Chord. KSS then fetches the index entries for the computed key from the node. To answer a query that contains more than k keywords, the peer picks multiple subsets that have k words from the query, fetchs the index entries for each subset, then finally intersects the lists to compute the final result.

3.2 KSS Example

The number of keywords to be used for forming the index keys is a configurable parameter in the KSS scheme. For certain applications it might be necessary to have all the combination of all the subset sizes of the searchable words. This enables the system to answer a query with any number of keywords by forwarding the query to only one node. In other applications, it might be enough to have a keyword-pair scheme in which all the possible pairs of words are indexed. In this section we will present an example that indexes every pair of words from the document and uses that index to reply to user queries.

KSS reads each word in a document identified by documentID (typically a SHA1 content hash) into a set *allwords*. Conceptually, KSS duplicates the set and calculates the cross product of the two sets. KSS generates index entry of the form <keyword-pair, documentID> for each element of the set formed by cross multiplication $allwords \times allwords$. KSS then inserts the index entries in the distributed hash table using the hash of the keyword-pair in the index entry as the key.

Let A, B, C, and D be the words in a document identified by *docID*. KSS creates index entries for each of the six combinations *(AB, AC, AD, BC, BD, CD)*. For a set of size two, $C(n, 2)$ gives the number of unique entries in the cross product of sets of n unique keywords. Following are the unique index entries generated for the example word set:

<AB, docID>	<AC, docID>
<AD, docID>	<BC, docID>
<BD, docID>	<CD, docID>

To perform a query with two words, the words are sorted and hashed and the resulting key is used to lookup the list of documents containing the pair of words. For example, a query for words B and A would be answered by finding a list of documents for the entry AB in the distributed index.

To answer a query with more than two words, KSS picks two of the query words arbitrarily, and computes the hash for the concatenation of those words and finds the node that stores a list of documents for the given pair of keyword. The entire query is then sent to the node which first finds a list of index entries for the keyword-pair. The node then finds the node that is responsible for storing the entries for the rest of the query words and forwards its list to the node. The node that receives the list and the query does a lookup in its local list of entries, joins the received list with the local list and forwards the resulting list to a node that is responsible for remaining words in the query. This process continues until all the search words in the query are cosumed.

For example, if a query for the words A, B, C, and D is received, the system locates the node N_{ab} that stores a list of index entries for AB. The entire query $A\ B\ C\ D$ is sent to the node N_{ab}. N_{ab} locally searches its list for index entries with the key *Hash(AB)*. N_{ab} then forwards its list of entries for AB to N_{cd}, the node responsible for storing the entries for *CD*. Upon receiving the list of entries for AB and the remaining words from the query, N_{CD}, does a local lookup for the list of entries for *CD* and joins the results to compute a list for *AB and CD*. This final list of documents that match all the four words in the query is returned to the querying node. Alternative design that is efficient for queries (with high insert overhead) would have been to use keyword-set of size four, in which case KSS creates an index entry for the key formed by hashing *ABCD*. In this setting, KSS can answer a query for documents containting A, B, C, and D by forwrding the query to only one node, the node that is responsible for the key formed by hashing *ABCD*.

3.3 Storing metadata in the index entry

To make queries faster, we can store additional information in the index entry in the field *metadata* to avoid having to forward the list of index entries from one node to another multiple times to compute the final list of documents matching a query. Figure 3-2 shows the structure of the index with the field *metadata*.

If we are building index entries for metadata search application, we can put the

Key	Metadata	URL
hash(AB)	X Y Z	http://18.175.6.2/a.mp3
hash(BC)	X Y Z	http://18.175.6.2/a.mp3
hash(MN)	O P Q R	http://18.175.6.2/b.html

Figure 3-2: Structure of the KSS index with a field for storing metadata

entire metadata field in the index entry. This enables us to answer queries by transferring only one list, the result list, across the network.

We now give an example to demonstrate how this change makes searches more efficient. Let a query be $A\ B\ X\ Y\ Z$. KSS hashes the first two keywords, AB, to form a key. The key is used to lookup a node N_{ab} that is responsible for storing the index entries for the keywords AB. The query $A\ B\ X\ Y\ Z$ is then forwarded to N_{ab}. N_{ab} upon receiving that query, does a local lookup for index entries with the key $hash(AB)$ and with the metadata field containing the words X, Y, and Z. It then returns the list of matching documents to the querying node. Thus there is only one list transferred across the network – the result of the query. This is in contrast to the method of obtaining matching documents by repeatedly forwarding the lists from one node to another performing a *join* at each hop as each word in the query is processed.

We can use a variant of this optimization in a full-text search application. If a document is small or insert overhead is not of a concern, the entire document can be put in the *metadata* field. Otherwise only the parts of the document that contain the words users will likely use in the queries should be put in the *metadata* field. While indexing a large text document, the first paragraph, for example, is a good candidate for the *metadata* field. To answer a query, a node can do a lookup in the metadata field for words from the query in its list of matching entries and return the list of matching index entries.

3.4 KSS Properties

In this section, we discuss various cost properties that determine the efficiency of the KSS system.

3.4.1 Insert Overhead

Insert Overhead is the number of bytes transmitted when a document is inserted in the system. This is the same as the total network usage when a client initiates a file-share action, for example, in a music file sharing system. When a user asks the system to share a file, the system generates index entries which are inserted in the distributed index. The larger the number of index entries generated for each document, the more the required Insert Overhead. If we generate index entries for a document with n keywords using all the possible subsets of all the possible keyword-set sizes of the keywords as keys, the overhead will be exponential in number of keywords (2^n). For keyword-pair scheme, the overhead required is bounded by $C(n, 2)$. The degenerate case creates index entry with hash of each word in the keyword set as key which has a small insert overhead.

Let A, B and C be the keywords in a document to be indexed. KSS can create three index entries with hashes A, B and C as keys resulting in a small insert overhead. KSS can also create index entries with all the possible subsets (A, B, C, AB, BC, AC, ABC) resulting in a high insert overhead. We can configure KSS to create index entries for some subset of all the possible combinations in the keyword set, in which case the insert overhead will be somewhere in between.

Let n be the number of words in a document. Let k be the size of the keyword-set. Let *size* be the size of an index entry. The total indexing related insert overhead required to *share* the document is bounded by $C(n, k) * size$.

Query Overhead

Query Overhead is a measure of bytes transmitted when a user searches for a file in the system. Ideally a search system transmits only a subset of the query keywords and the matching results that the user is interested in. The goal in a real search system is to approximate this theoritical minimum as much as possible. Transmitting the search keywords is not a real issue because most of the queries have only a few words. It could be a concern if those words need to be transmitted to a lot of hosts. The real issue is sending the intermediate result list in the system from one host to another while result aggregration is still being done. An *and* query, for example, requires that two lists corresponding to the two words be retrieved. This could result in a huge waste of overhead if the final result set is a tiny fraction of the lists. In this system we minimize the query overhead needed for an *and* search by directly fetching a result set for the entire *and*clause rather than fetching result set for each clause and joining them.

Let A, B and C are the keywords in the user query. A standard distributed inverted index approach makes a call to the node responsible for the word A. The node then forwards its list for A to the node responsible for B which intersects the list for A with its list for B and forwards the resulting list A *and* B to the node responsible for C. The node then intersects the list for A *and* B with its local list for C to compute the result for the query A *and* B *and* C which is sent to the querying node. Thus drawback of this approach is the huge waste of network bandwidth, especially when most of documents in the intermediate lists being transferred from one node to another for join operation contain the documents with all three words.

Let q be the number of words in a query. Let k be the size of the keyword-set that was used to build the index while documents were inserted in the network. At least $\frac{q}{k}$ lists of index entries are fetched across the network because at each hop, k words from the query are used to form a key to find the node storing index entries for the next k words in the query. Thus, there are $\frac{q}{k}$ hops, hence about $\frac{q}{k}$ lists need to be fetched from across the network.

Storage Overhead

Storage Overhead is a measure of number of bytes that need to be stored in the disks of peers. Storage Overhead directly correlates with the Insert Overhead. Higher storage is needed for larger number of entries in the list and also for larger size per entry.

KSS Overhead with metadata

The index entries are larger with the addition of the *metadata* field. Since the number of index entries generated when a document is inserted in the network is the same; now with a larger size for each entry, more bytes would be transmitted across the network. Thus the insert and storage overhead for an indexing scheme that uses the metadata field is higher than that of the unoptimized KSS scheme.

When a query contains more than k keywords, after picking k words to form the key, KSS forwards the query keywords to the node responsible for that key. The node upon receiving the query keywords and the index key formed by hashing the k query keywords, does a local search for index entries with the specified key and also with the query keywords in the field *metadata*. The result list for a query is returned to the querying node without having to forward the list from one node to another for intersection. Thus KSS sends the query to only one node in the network, and fetches the list of matching index entries from the it. The query overhead for KSS scheme that stores the metadata field in the index entry is equal to the size of the matching list of the queries. Assuming that the list of the matching documents is not stored at the querying node itself, it is not possible to do the query with a lower overhead because the matching list of the documents must be returned to the querying node regardless of the query algorithm.

Chapter 4

Using KSS

In this chapter, we present two major applications of the KSS system : Meta-data search system, and full-text search system. We also discuss how to configure KSS to adapt it to specific application needs.

4.1 Meta-data Search

In a music file sharing system, users want to search the meta-data fields such as, song title, artist's name, band's name, and description. For meta-data search application, KSS is typically configured to use a key-word pair, i.e, two words per keyword-set that is used to form the key for the index entry. Thus configured, KSS builds a distributed inverted index with an entry for each pair of words that occurs in any meta-data field—the key for the index entry is the pair of words, ordered alphabetically. The text from that particular metadata field and a pointer to the audio file are also stored in each entry.

KSS indexes a new document by adding the documents's identifier, such as URL or content hash, to each entry corresponding to a pair of terms that the document meta-data contains. This requires a number of insertions quadratic in the number of meta-data keywords. We choose to place in the index entry not just the document's identifier, but also the entire meta-data string for that document: this makes searches on more than two keywords more efficient.

To perform a search for a two-word query, we use DHash to find the node that stores the index entries for the key formed by hashing those two words, and return the list of entries with the matching keys.

To perform a query with more than two words, we pick two of the query words arbitrarily, and use DHash to find the node with the entries for that pair. That node can, locally, inspect the index entries with matching keys to find out which ones have the query keywords in the *metadata* field (this is why we chose to store all the meta-data, rather than just the document ID, in the entry). Nodes can build standard non-distributed indexing structures to make searches of their local collection of index entries efficient.

KSS creates the index as follows (each field is identified with a unique meta-data field ID, metaID):

```
word[0..n] = meta-data field
for (i = 0; i < n; i++)
 for (j = i+1; j < n; j++)
  set_add(keywords,
    concat(sort(word[i], word[j])))
for (i = 0; i < keywords.size; i++)
 push(index_entries, <hash(keywords[i]+metaID),
                        meta-data field, documentID>)
```

For each meta-data field, the indexer reads the words into a set *word*. Conceptually, the indexer duplicates the set and calculates the cross product of the two sets. Then, the indexer generates index entries of the form <hash(keyword-pair + metaID), meta-data field, documentID> for each element of the set formed by cross multiplication *word* × *word*. The indexer inserts the set *index_entries* in the distributed hash table using the hash of {keyword-pair+metaID} as the key.

For example, if the title of the song is *When tomorrow comes*, the following entries are generated:

- < hash({comes tomorrow} + titleID), when tomorrow comes, URL >

- $<$ hash({comes when} + titleID), when tomorrow comes, URL $>$
- $<$ hash({tomorrow when} + titleID), when tomorrow comes, URL $>$

In order to support single word searches, KSS also builds similar entries with the keys formed by hashing: {when}, {tomorrow}, and {comes}.

A client performing a search forms keyword-pairs from the query and computes the hash of <word-pair+metaID>, where metaID specifies the meta-data field to be searched. The client then passes all keywords in the query to the node responsible for the $search - key$. The node that receives the call, locally finds the entries for $search - key$ using a local inverted index. It only selects the entries that match all keywords from the query in its *metadata* field. This list is returned to the searching client which displays the results to the user. The file selected by the user is then fetched using the documentID which can be a URL or content hash to be used in a file system such as CFS.

To answer a query KSS sends only one message: a message to the node that is responsible for the $search - key$. To get this performance, the algorithm replicates the meta-data fields on every node that is responsible for some pair of words in the meta-data field. Since we assume that meta-data fields are small, storage is plenty, and communication is expensive, we consider this a good tradeoff for MP3 meta-data search application.

4.2 Full-text indexing

The KSS algorithm as described in the Meta-data search application uses large amounts of storage if it were used for full-text indexing because that would require putting the entire document in the *metadata* field of the index. The storage costs can be greatly reduced, at a small cost in the algorithm's ability to retrieve all relevant documents.

In order to support full text indexing for documents, KSS considers only pairs of words that are within distance w of each other. For example, if w is set as five and the sentence is *FreeBSD has been used to power some of the most popular Internet web sites*, only the words appearing within five words from each other are used to generate the index entries, such as *most* and *popular*, while *used* and *sites* are not paired up. The algorithm for indexing meta-data is a special case of this general technique, with unlimited w.

The size of the index and the resulting quality of search depends on the size of the window w. A large window makes the algorithm generate many combinations, thereby increasing the size of the index. A large window also consumes a lot of insert bandwidth as there are more keywords-pairs to be indexed. On the positive side, with a larger window, queries with words that are farther apart in the document can still match the document. Thus w should be chosen based on the desired tradeoff of space consumption, amount of network traffic, and thoroughness of retrieval.

When indexing a large collection (of documents) comparable to the Web, it is not feasible to store the entire document along in the *metadata* field. Instead, we use a search protocol that intersects the result for each keyword-pair in the query to come up with a list of documents containing all the keywords in the search. For example, if a query is for documents matching *A B C*, the node fetches results for AB and AC and later intersect the two lists to come up with a list of documents that have all the three words: A, B and C.

To answer a query for multi-word-query A B C using the standard inverted keyword scheme, one would need to fetch a large document list for the keywords A, B and C. KSS queries transfer less data, since fewer documents contain the pairs AB or AC than contain any of A, B, or C.

If the index is generated using the keyword set size of two, and a query has more than two search words, KSS algorithm described so far will make many requests, thereby saturating its bandwidth available to other nodes. For example, search for *A B C D*, will fetch the lists for AB, BC, and CD. We can, however, employ the following optimization (recursive search) if there are a lot of search terms in order to distribute query effort to other peers at the expense of latency.

First, compute the hash to find the node responsible for any one pair of keywords, and send the query to the KSS layer that runs on that node. The KSS layer fetches the list for the pair of words from local DHash system, and issues another search query with the remaining pairs. The KSS layer performs these searches recursively until all the search terms are consumed. The KSS layer then intersects the obtained result with its local result set, and sends the final set to the querying node.

This approach takes longer to reply to queries. The query time can be shortened by parallelizing the search. The level of parallelism can be determined by any peers or as a policy by the node where the search originates. For example, a node that gets a search request for (A, B, C, D, E) can make search calls with (A, B, C) and (C, D, E), thereby aproximately halving the query time.

4.3 KSS Configuration

As evident from the two examples presented in this chapter and the description of the KSS costs in chapter 3.4, KSS parameters must be customized to match the requirement and expectation for a specific application. In this section we describe the issues related to customizing KSS for specific applications.

4.3.1 Contents of Index entry

In KSS, there is no reason to limit the index key to the hash of keyword pairs. In order to support one-word queries, KSS must generate standard single-word inverted indices. In fact, KSS can generate index entries with a subset of any or all size from the list of words in the search field. For example, if a subset of size five is used, KSS can answer queries with five words by forwarding the key formed by the words in the query to a single node. This speed comes at the cost of an increased number of index entries that need to be distributed in the P2P network. This is an issue of trading higher insert bandwidth for smaller query bandwidth. Using a large subset increases the insert bandwidth, but in a system where each query has a lot of keywords, this will minimize the query bandwidth. Using a subset of size one will consume a large query bandwidth for any search with multiple keywords.

4.3.2 Search across multiple meta-data fields

If space used by the index is not an issue, we can support even more powerful search in the full keyword search application with a small modification in the described protocol. In order to index the keywords for a document, KSS concatenates each keyword with its meta-data field and generate the index entries rather than generating entries for each pair of keywords for each meta-data field separately. Suppose an MP3 file with author *Antonio Vivaldi* and title *Four Seasons* is to be indexed with this enhancement. KSS generates index entries from the following list of four words *author:antonio author:vivaldi title:four title:seasons* at the same time. With this approach, a search for files with Vivaldi as a composer and with the word four in the title can be searched by creating the key corresponding to the words: *author:vivaldi:title:four* and doing a one lookup for that key instead of intersecting the list for result from author search with *Vivaldi* and title search with *four*. This scheme takes a lot more space than indexing each meta-data field separately. Indexing each meta-data field separately is then seen to be a form of windowing that we use if that space blowup is too much.

4.3.3 Storage requirement

In the KSS adaptation described in section 4.1, KSS stores the entire search field in the index entry. In the full text search system described in section 4.2, KSS did not store any content of the search field. The decision to include or exclude the search field can be seen as a tradeoff between query bandwidth and storage requirement. If it is not prohibitive to store the entire search field (meta-data fields, documents), KSS can answer any query in one lookup. If index entries do not include any content of the search field, KSS is forced to intersect the documentIDs from multiple searches to come up with a list of documents common to all the results, thereby using higher query bandwidth in a system with a limiting space constraint. A reasonable compromise might be to store words within the window in each index entry so that multiple keyword queries with words appearing within a window can still be answered by forwarding the query to one node.

4.3.4 Document ranking

KSS described for meta-data and full-text search do not have a document ranking system. KSS displays the results in the order they appear in the result lists, and that order is determined by the local indexing scheme used by each node to maintain its share of index and the intersection algorithm used by the querying node. Since a user is not likely to browse through hundreds of documents returned from search, it is important that KSS is able to display highly relevant documents as the top results from the search. One way to support that would be to include a ranking score in each index entry and sort the result set based on the ranking score before displaying the list to the user. The score might depend on the distance between the word pairs and might be weighted depending on where they appear. Words appearing in the title of a document or the first sentence of a paragraph could be given a higher weight than the ones appearing in the middle of a paragraph.

Chapter 5

System Architecture

KSS can be implemented in any P2P platform that supports Distributed Hash (DHash) Table interface. Examples include Chord, CAN [19] and Tapestry [26]. In this chapter, we will describe an example system using the Chord system.

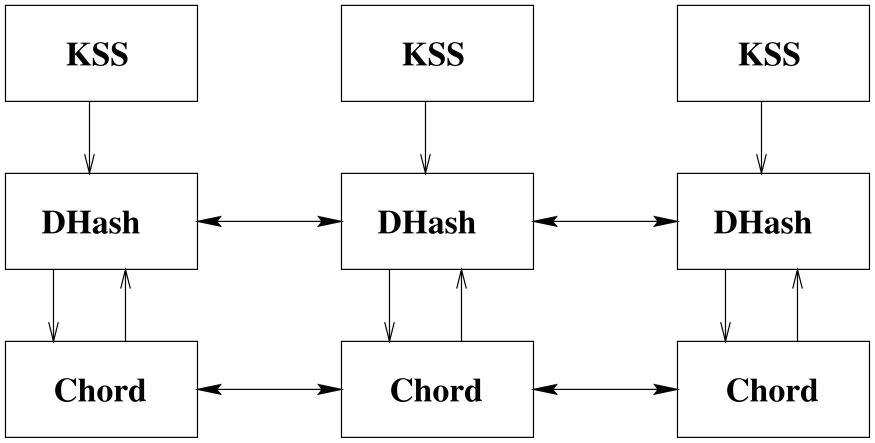


Figure 5-1: KSS system architecture. Each peer has KSS, DHash and Chord layers. Peers communicate with each other using asynchronous RPC.

5.1 The Chord Layer

Each Chord node is assigned a unique node identifier (ID) obtained by hashing the node's IP address. As in consistent hashing [12], the ID's are logically arranged in a circular identifier space. Identifier for a key is obtained by hashing the key. Key, k is assigned to the first node with ID greater than or equal to k in the ID space. This node is called the successor node of the key k. If a new node joins the ring, only some of the keys from its successor needs to be moved to the new node. If a node leaves the ring, all the keys assigned to the leaving node will get assigned to its successor. Rest of the key-mapping in the ring stays the same. Thus, the biggest advantage of the consistent hashing as compared to other key mapping schemes is that it allows nodes to join or leave with a small number of keys remapping.

Each node in the system that uses consistent hashing maintains a successor pointer. Locating the node responsible for a key just by using the successor pointer is slow. The time it takes to get to the node responsible for a key is proportional to the total number of nodes in the system. In order to make the lookup faster, Chord uses a data structure called finger table. The i^th entry in the finger table of node n contains the identity of the first node that succeeds n by at least 2^{i-1} on the ID circle. Thus every node knows the identities of nodes at power-of-two intervals on the ID circle from its position. To find the node that is responsible for key k, we need to find the successor for k. To find the successor for k, the query is routed closer and closer to the successor using the finger table, and when the key falls between a node and its successor, the successor of the current node is the successor for the key k. The iterative lookup process at least halves the distance to the successor for k at each iteration. Thus an average number of messages for each lookup in a Chord system is O(log N).

5.2 The DHash Layer

The DHash layer implements a distributed hash table for the Chord system. DHash provides a simple get-put API that lets a P2P application to *put* a data item in the nodes in the P2P network and *get* data given their ID from the network. DHash works by associating the keys to data items in the nodes. A get/put call first uses Chord to map the ID to a node, and does a get/put in its local database using the ID as the key.

The following is a summary of DHash API:

- `put(key, data)`: Send the data to the key's successor for storage.
- `get(key)`: Fetches and returns the block associated with the specified Chord key.
- `append(key, data)`: If there is no entry for the given key, insert a new entry in the hash table with the given key. If an entry for the key already exists, add a new entry with the given key and data value.

We now explain why the following properties of DHash are important to the design of KSS and other P2P applications using this interface.

5.2.1 Availability

Dhash replicates the content in its local database to a configurable number of other nodes that are nearby in the identifier space. Since there are multiple nodes that store the content for a key, even with some of the nodes joining and leaving the network, the data is still likely available in the network.

5.2.2 Caching

Dhash caches index blocks along the lookup path to avoid overloading servers that are responsible for answering queries with popular words. As a result of caching along the lookup path, the queries can also be answered in shorter amount of time than a full lookup.

5.2.3 Load Balance

Chord spreads the blocks uniformly in the identifier space. This gives good load balance if all the nodes are homogeneous. Virtual servers, the number of which is proportional to the network and storage capacity, are used to make sure that a peer with less bandwidth and disk space don't bear as much responsibility as a peer with a lot of bandwidth and disk space to spare.

We refer readers to [5] for a detailed analysis of these features.

5.3 KSS Layer

The Keyword-Set Search layer is written using the DHash API. When a client inserts a document, appropriate index postings are generated and routed to the nodes in the P2P network using the DHash `Append` call. When a client requests a search, KSS makes a DHash `Get` call to fetch the lists for subset of query words. The document lists are then intersected to find the documents that contain all the words in the query.

KSS provides the following API to client application:

- `insert(document)`: Extract the keywords from the document, generate index entries, and store them in the network.
- `search(query)`: find the document list for disjoint set of keywords in the query, and return the intersected list of documents.

5.4 Implementation Status

KSS prototype was implemented in 1000 lines of C++ using the DHash API described in section 5.2. KSS is linked with the Chord and DHash libraries and runs as a user level process. The KSS prototype that we implemented uses the set size of two, i.e, uses the keyword-pair for indexing. The prototype has a command line interface that supports the following commands:

- `upload <filename>` : Uploads an mp3 file to the network and stores the file using the DHash with SHA1 content hash as the key. Extracts meta-data from the mp3 file, generates the index entries and stores them in the network.

- `search <query words>` : Searches for documents for the given query. Displays an enumerated list of matching documents to the users.
- `download < resultID >` : Downloads the file from the network. *resultID* is the integer ID from the list of matching documents.

Chapter 6

Experimental Findings

The main assumptions behind the KSS algorithm are: (1) query overhead to do standard inverted list intersection is prohibitive in a distributed system; and (2) P2P systems have so much storage to spare that we can save query overhead by using keyword-sets; and (3) all or at least most of the documents relevant to a multi-word full-text query have those words appearing near each other. We attempt to validate these assumptions with some simple experiments. In this chapter we present preliminary findings from our experiments.

6.1 Efficiency in Full-text search

In this section we attempt to measure the efficiency of the KSS algorithm for full-text search of documents, the most challenging application for KSS.

6.1.1 Analysis Methodology

In order to analyze KSS costs and efficiency for full-text search, we ran a web crawler that visited the web pages on the LCS website and downloaded the text and HTML files recursively up to thirteen levels. Our crawler downloaded about 121,000 HTML and text pages that occupied 1.1 GB of disk space. We then simulated inserting of a document using KSS by running a Perl script on the downloaded files to clean HTML

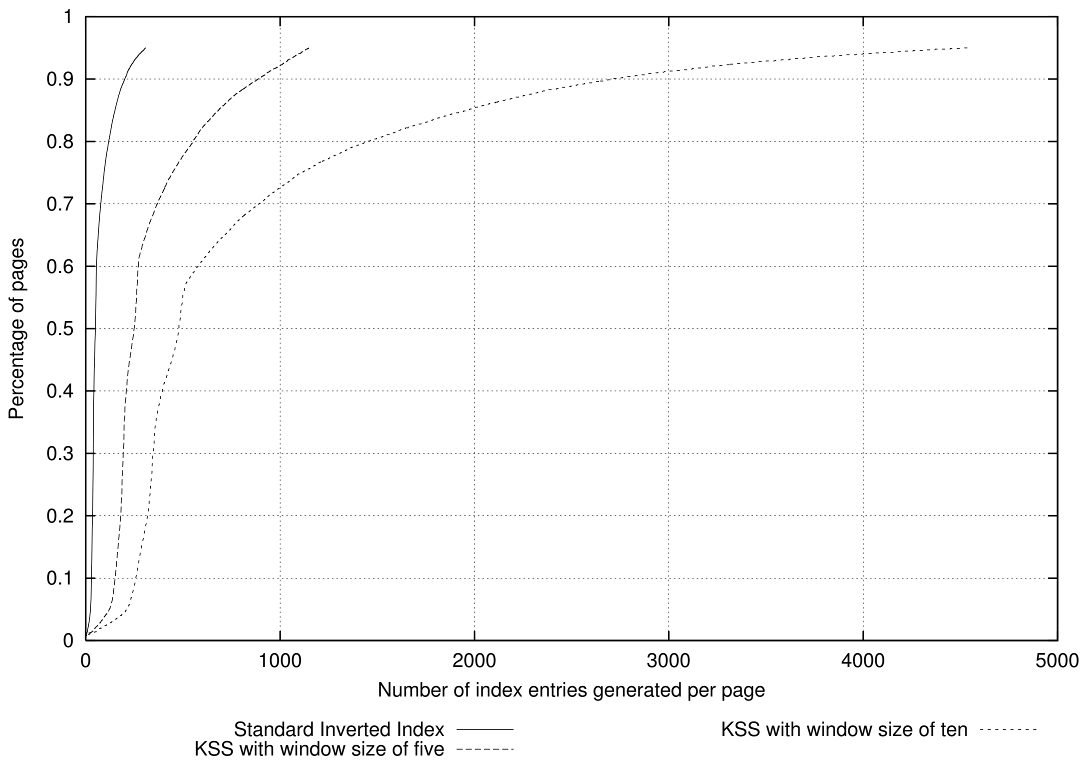


Figure 6-1: Cumulative distribution of the number of documents for which the given number of index entries in x-axis are generated using the standard inverted indexing scheme.

tags, and extract plain text. The extracted text consumed about 588 MB of storage. We then ran the KSS algorithm on each text file to create index entries and write them to a file, *index* file. Each line in the index file represents an index entry, and contains the SHA1 hash of the keyword-set, and the SHA1 content hash to be used as a document pointer. For example, the KSS index entry for the words *lcs* and *research* in a document called *doc1* would look like: <SHA1(lcsresearch), SHA1(content of doc1)>.

We analyzed costs for a query by doing a search in the *index* file, extracting appropriate index entries for the search keywords, and measuring the size of the lists. Search keywords for this experiment were obtained from the log of searches done by the actual users on the LCS website.

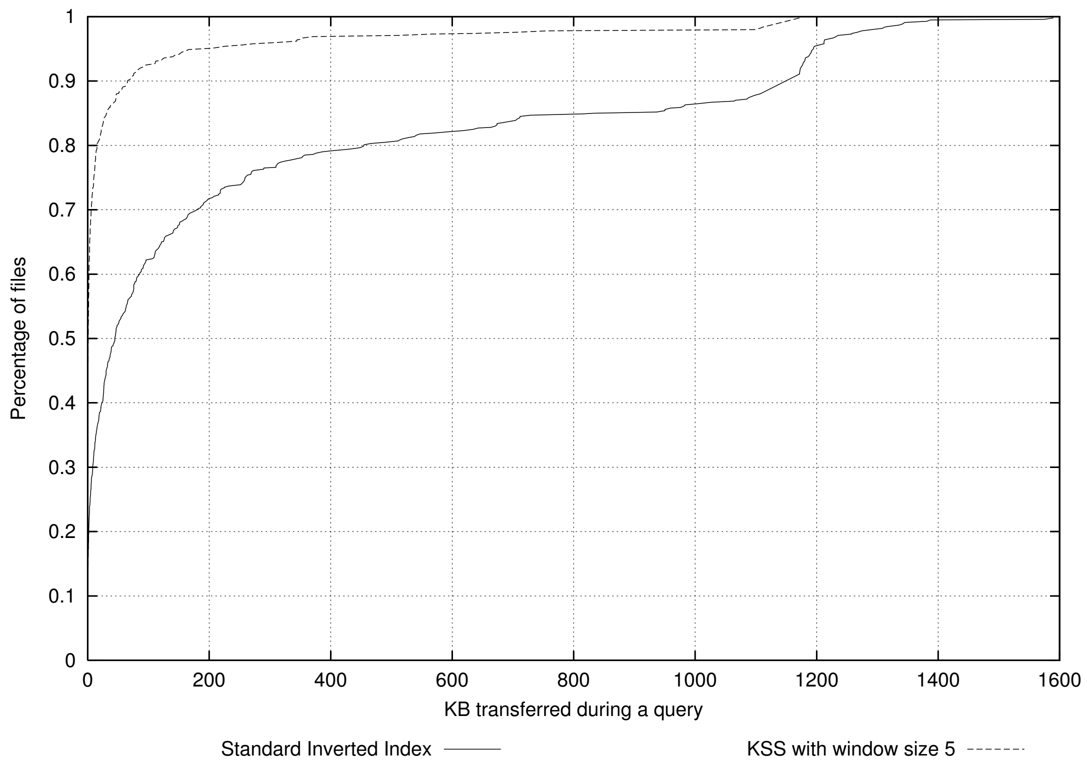


Figure 6-2: Cumulative distribution of number of queries that result in given number of bytes in x-axis to be transmitted across the network during a search using the KSS with window size of five and the standard inverted scheme.

6.1.2 Insert Overhead

Inserting a document in a network results in index entries for that document being transmitted to appropriate nodes in the P2P network. To analyze the insert and storage overhead for a document *insert* operation, we ran a Perl script that extracts the text from the document. We then ran the KSS indexing algorithm with a different window sizes and the standard inverted indexing algorithm for the same set of files.

We counted the number of entries generated for each downloaded file and plotted the cumulative distribution of number of index entries generated as a percentage of the number of files. Figure 6-1 presents a distribution of the number of index entries generated when each document is inserted in the system using KSS with window size of five, ten and using the standard inverted indexing scheme. Multiplying the number of entries in these distribution graphs by the size of each of entry (40 bytes) converts x-axis into a measure of bytes transmitted across the network during a document

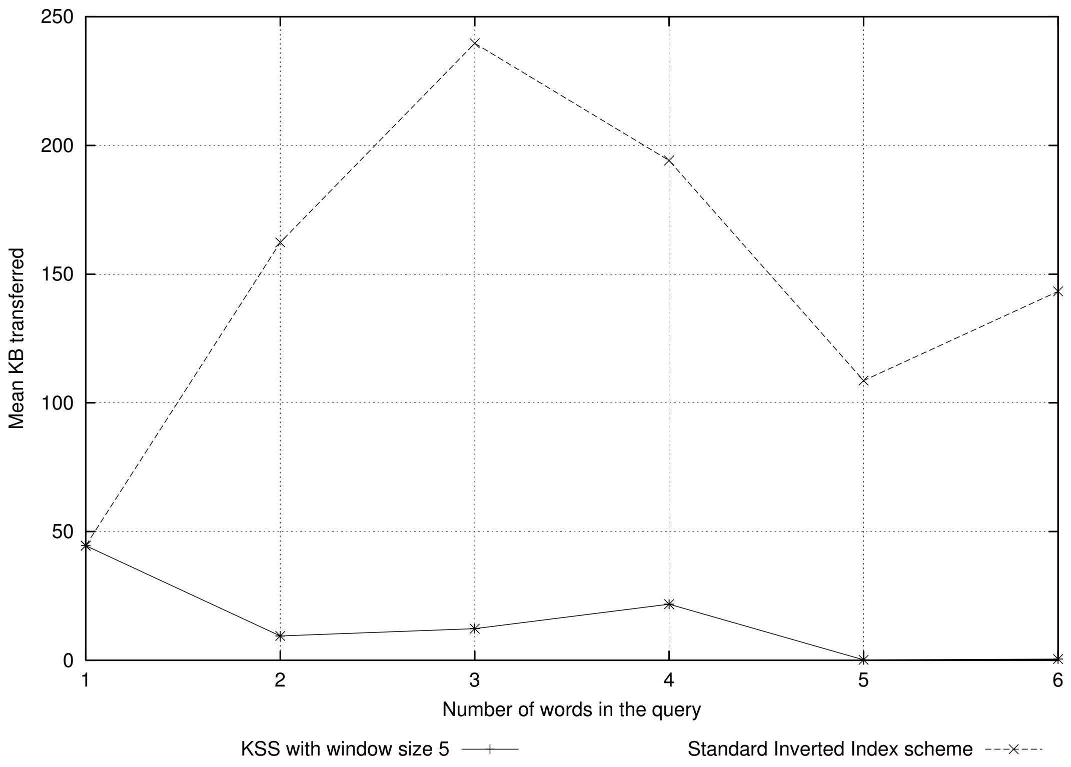


Figure 6-3: Mean data transferred in KB (y-axis) when searching using the standard inverted index compared to KSS with window size of five, for a range of query words (x-axis).

insert.

Following table summarizes the insert overhead:

Algorithm	Number of index entries	Total index size
Standard inverted index	12.1 million	480 MB
KSS with window size of five	92.5 million	4 GB
KSS with window size of ten	169.6 million	7 GB

Since KSS generates entries for permutations of words within a window rather than just for each word, the insert overhead is much higher for the KSS system than the standard inverted index scheme.

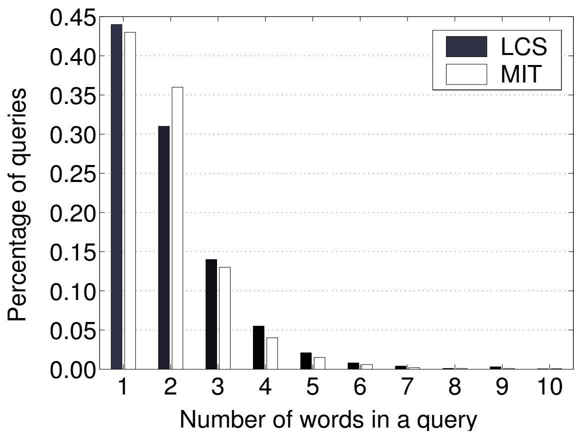


Figure 6-4: Percentage of queries (y-axis) with a given number of words (x-axis).

6.1.3 Query Overhead

Searching for documents matching a query involves fetching index entries for each keyword-pair in the query and intersecting the lists. In order to measure the query overhead in a KSS system, we measured the size of the list of matching index entries in the *index* file for each keyword-pair. We then multiplied the size of the list by the size of an entry (40 bytes) and obtained the number of bytes that would be transmitted across the network during the search. We use index entries generated with keyword-set size of two (keyword-pair) to answer queries with multiple words. To answer queries with one keyword we fall back to the standard inverted scheme and use index entries generated for each keyword.

We randomly selected 600 queries from the LCS website search trace, and measured the number of matching entries for each keyword-pair in the query and converted that to Kilobytes.

To measure the efficiency of the KSS scheme over the standard inverted index scheme, we did a measurement of number of bytes that would be transmitted across a network to answer user queries for each scheme. For KSS, we measured the size of the matching list for each keyword-pair from the query. For example, to measure the KSS overhead for the query $A\ B\ C\ D$, we measured the size of the lists for AB (which is sent to the node responsbile for CD), and the size of the list for AB *and* CD (which is the result of the query and is sent to the querying node). We then obtained the number of bytes by multiplying the size of the lists by the size of each entry. For standard inverted scheme, we computed the size of the list for A (which is sent to the node responsible for B), the size of the list for A *and* B (which is sent to the node responsible for C, the size of the list for A *and* B *and* C (which is sent to the node responsible for D), and the size of the list for A *and* B *and* C *and* D (which is the ressult of the query and is sent to the querying node). We then converted the size of the lists to bytes. Figure 6-2 shows the result of this experiment. Figure 6-2 shows that the query overhead for 90% of the queries in the KSS scheme is less than 100 KB, while under the standard inverted index scheme, only about 55% of the queries are answered by transferring less than 100 KB. KSS answers queries by intersecting the short lists of keyword pairs while the standard inverted index scheme joins the longer lists for each keyword; thus KSS is able to answer most of the queries with smaller overhead.

From the randomly selected 600 queries, we extracted one, two, three, four, five and six word queries and ran KSS and single inverted index schemes on each of the six sets of queries separately. We measured the size of the matching lists for each algorithm, converted that to Kilobytes, and plotted the mean number of Kiloytes that need to be transmitted across the network for each set of queries in figure 6-3. For each set of queries, KSS is able to answer queries by transferring significantly smaller number of bytes. We also observe that the queries with more than three words become more and more specific with more words and hence there are fewer bytes transferred across the network because of the fewer matching index entries.

6.1.4 Number of words in a query

Analysis of search traces from the MIT Lab for Computer Science (LCS) web site at http://www.lcs.mit.edu/ (5K queries), and the main MIT web site at http://web.mit.edu/ (81K queries) shows that about 44% of queries contain a single keyword, and 55% contain either two or three keywords. (see Figure 6-4). Two word queries (35% of all queries) can be answered by forwarding the query to only one node (instead of two nodes in the standard inverted index scheme); for three word queries (20% of all

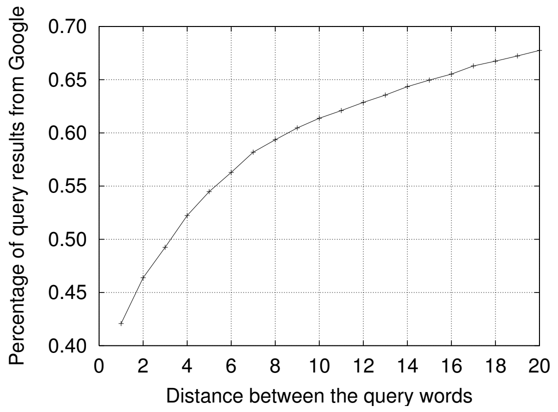


Figure 6-5: Cumulative distribution of the number of hits (y-axis) with a given distance (in words) between the query words (x-axis).

queries) can be answered by contacting forwarding query to two nodes (instead of three nodes in the standard inverted index scheme).

6.1.5 Search accuracy

The windowing version of KSS assumes that a multi-word query has the words appearing near each other (within the window width) in the text. If this is not true, KSS fails to retrieve the document. To find out how often the words in multi-word queries are near each other in a match that is considered good by a traditional search technique, we submitted the 5K queries from the LCS website search trace to Google google:web. We then calculated the distances between the query words in the top ten pages returned by Google. From figure 6-5, we can see that with a window size of 10 words, KSS would retrieve almost 60% of the documents judged most relevant by Google in multi-word queries. Our scheme could not do better than this because Google uses metrics such as text layout properties and page rank in addition to word proximity to rank the results.

6.1.6 Improvement over single inverted index

KSS is more efficient than the standard inverted index scheme for queries with multiple keywords. KSS degenerates to a standard inverted scheme with single word queries. However, the insert overhead is higher than the standard inverted scheme. If most of the queries have only one word, then the efficiency during multi-word queries is not enough to compensate for the huge insert overhead. We have shown that 44% of the queries contain a single keyword. In this case, we can not use the efficient KSS scheme to reduce query overhead as we fall back to the standard inverted index scheme. However, 55% of the queries contain two or three keywords, and we are able to use KSS to reduce query overhead on these queries. The results are not as precise as the matches returned by Google.

6.2 Efficiency in Meta-data search

In this section we attempt to measure the efficiency of the KSS algorithm for meta-data search of music files, the target application for KSS.

6.2.1 Analysis methodology

In order to analyze KSS costs and efficiency for meta-data search, we downloaded the free Compact Disc Database (CDDB) database called FreeDB [7]. The database contains information about each audio CD, such as CD title, song title and length, and artist names. The database contained information about 3.64 million song titles in about 291000 albums. We then ran the KSS algorithm with set size of two and the standard inverted index algorithm to compute the index entries for all the titles.

In order to measure the communication overhead during the search, we replayed queries from the Gnutella network. 600 random queries were selected from a log of about one million queries. Then KSS was used to find the appropriate entries in the *index* file for each keyword-pair for the selected queries.

6.2.2 Insert Overhead

KSS index entries for a meta-data search application contain the entire meta-data in the index entry. Since the length of the title is variable, the index entries are also

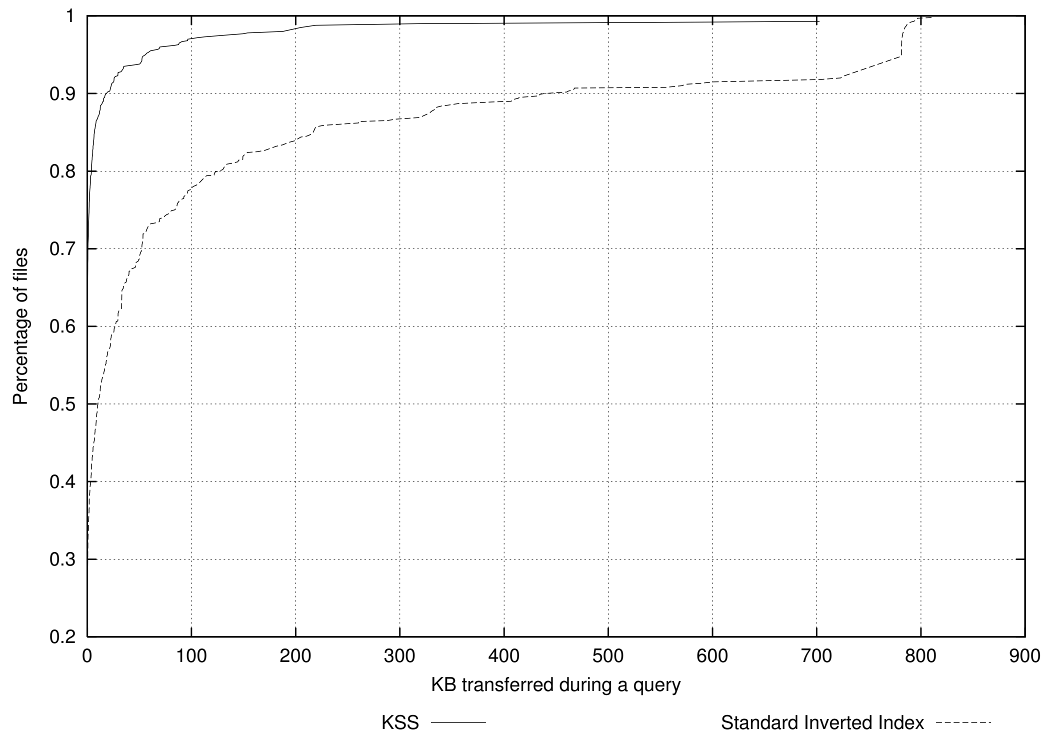


Figure 6-6: Cumulative distribution of number of queries for the given number of bytes in x-axis that need to be transmitted across the network during a search.

of variable length. Fixed size index entry with padding for smaller titles results in a simpler implementation at the expense of insert and query costs. In our experiment, the average size of each entry was about 75 bytes. KSS generated about 11 index entries for each music file shared by the user. Thus, when a user initiates a *share* action for a music file, KSS generates index entries for that file, and transmits about 750 bytes of application level data (index entries) across the network.

Following table summarizes the insert cost:

Algorithm	Number of index entries	Total index size
Standard inverted index	12.9 million	844 MB
KSS with keyword-pair	38.3 million	2.7 GB

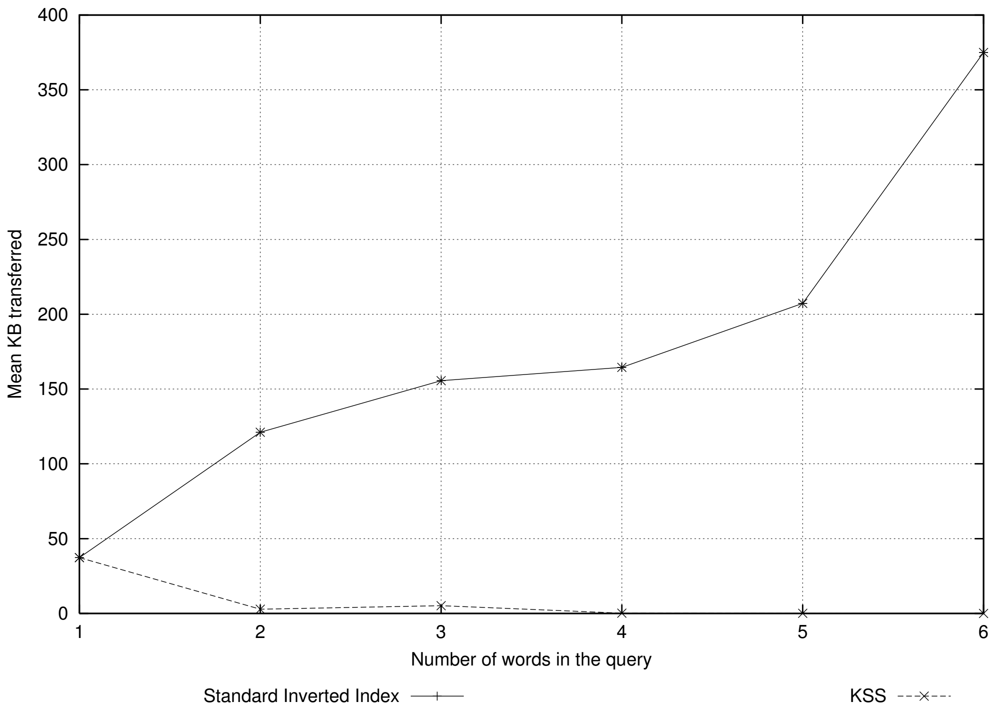


Figure 6-7: Mean data transferred in KB (y-axis) when searching using the standard inverted index compared to KSS with keyword-pair, for a range of query words (x-axis).

6.2.3 Query Overhead

When a user enters a query with keywords for the title of a song, the system that uses the standard inverted scheme fetches the index entries for each keyword and intersects the results repeatedly till the result is found. For query *A B C*, we computed the size of the list for *A* (which is sent to the node responsible for *B*), the size of the list for *A and B* (which is sent to the node responsible for *C*, the size of the list for *A and B and C* (which is the result and is sent back to the querying node). For meta-data search application, KSS stores the entire metadata in the index entry. KSS sends the query to the node responsible for any one pair of query words. The node upon receiving such a request, does a lookup in its local list for entries with the query words in the entry. Thus, the only list that is transferred across the network is the match for the query. Figure 6-6 shows the distribution of number of bytes transferred for each query using the standard inverted index scheme as compared to the KSS system. Figure 6-6 shows that the query overhead for 90% of the queries in the KSS scheme is less than 25 KB, while under the standard inverted index scheme, only about 55% of the queries are answered by transferring less than 25 KB. KSS forwards the query to a node responsible for one of the keyword-pairs in the query. That node locally searchs for index entries for the given key and with song titles in the metadata field of the index entry. Thus KSS is able to answer the queries with smaller overhead. The standard inverted index scheme, on the other hand, joins the lists for each keyword resulting in a high query overhead.

We also ran the search experiment with single inverted index scheme and KSS scheme for queries with one, two, three, four, five and six words separately, measured the distribution, and plotted the mean number of Kilobytes that need to be transmitted across the network for each set of queries in figure 6-7. The figure shows that the overhead for the standard inverted index scheme keeps increasing with the increasing number of words in a query. This is because there are more lists to be transferred across the network for each search. Thus, even though the queries are becoming more specific with more words in the query, the initial few lists to be transferred for join are still large, and hence account for the increasing overhead for the standard inverted index sheme. In metadata search application, the query overhead for KSS is equal to the matching list of index entries for the reasons explained in section 4.1. With more words in a query, there are fewer matching songs. This explains the decreasing query overhead for KSS with a larger number of keywords. KSS transfers significantly fewer bytes to answer queries than the standard inverted index scheme.

6.2.4 Number of words in a query

Analysis of search traces from Gnutella (15 million queries) shows that more 37% of the queries have more than two words, and about 50% of the queries have more than two words. (see Figure 6-8).

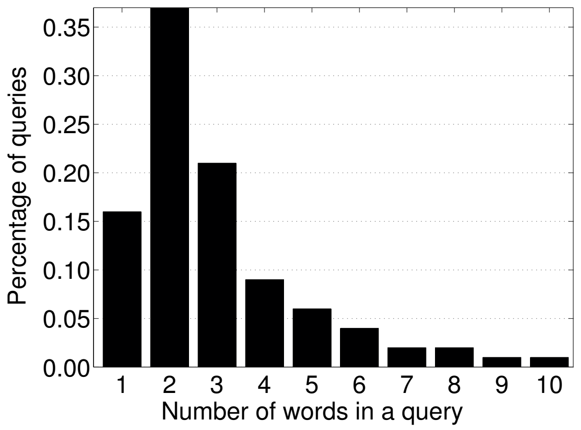


Figure 6-8: Percentage of queries (y-axis) with a given number of words (x-axis).

6.2.5 Improvement over single inverted index

The insert overhead for KSS is considerably higher than that of the single inverted index scheme. However, KSS transmits only one list across the network during a meta-data query – the result list. This is in contrast to the large intermediate lists that are transmitted across multiple nodes for join operations in the standard inverted index scheme. KSS benefits are more pronounced for queries with higher number of words. We have shown that majority of the queries in a music file sharing system tend to have multiple keywords. Thus, KSS performs better than the single inverted index scheme for metadata search.

Chapter 7

Future Work and Conclusions

In this thesis, we proposed KSS, a keyword search system for a P2P network. KSS can be used with music file sharing systems to help users efficiently search for music. KSS can also be used for full-text search on a collection of documents. Insert overhead for KSS grows exponentially with the size of the keyword-set while query overhead for the target application (metadata search in a music file sharing system) is reduced to the result of a query as no intermediate lists are transferred across the network for the *join* operation.

In full-text search application, KSS results are not as precise as the results returned by the Google search engine. We can improve accuracy by including information (such as font size, word position in the paragraph, distance between the words) on the words being using to form an index key. This still will not achieve the accuracy of Google because KSS, as described in this thesis, does not have the notion of documents being linked from another documents. Hence we can not use a ranking function like page rank that uses the link structure of the web to compute highly relevant matches.

We focussed our attention only to the cost of a query and index building. For a complete analysis, we need to analyze the system level costs (Chord, DHash) to determine the overall insert and query overehead in a KSS system. In all our preliminary experiments, we have made an assumption that KSS will be able to find documents in a relatively short amount of time. For a real system, it is important to know how quickly the system can find documents and this depends on the P2P message routing latencies.

As the KSS system described in this thesis grows old, the problem of *stale* index entries becomes serious. This results in a lot of index entries pointing to documents that are no longer in the network. Users with malicious intent might insert an excessive number of index entries in the distributed index (making the index large) or insert entries that point to documents that do not exist. Thus an index could become unusable. A possible solution is rebuilding the index periodically, dropping the invalid entries from the index.

During the project, a prototype of a KSS system was built. The current version of KSS software has a command line interface to upload documents, download documents, and search for documents using query keywords. Ultimately, a statically linked application with a GTK interface should be built and an alternative web-proxy based UI should be provided to the users.

Bibliography

[1] E. Adar and B. Huberman. Free riding on gnutella, 2000.

[2] Anurag Singla and Christopher Rohrs. Ultrapeers: Another Step Towards Gnutella Scalability, December 2001.

[3] Christopher Rohrs. Query Routing for the Gnutella Network, December 2001. http://rfc-gnutella.sourceforge.net/Proposals/QRP/query_routing.htm.

[4] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. Technical Report 2001-48, Stanford University, CS Department, November 2001.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

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

[7] FreeDB. http://www.freedb.org/.

[8] Gnutella. http://gnutella.wego.com/.

[9] Gordon Mohr. Hash/URN Gnutella Extensions (HUGE) v0.92, October 2001.

[10] Grokster. http://www.grokster.com/.

[11] Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.darkridge.com/ jpr5/gnutella.html/.

[12] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, TX, May 1997.

[13] Kazaa. http://www.kazaa.com/.

[14] L F Mackert and G M Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 149–159, Kyoto, Japan, 1986.

[15] Mark A. Sheldon and Andrzej Duda and Ron Weiss and David K. Gifford. Discover: a resource discovery system based on content routing. *Computer Networks and ISDN Systems*, 27(6):953–972, 1995.

[16] James K Mullin. Optimal semijoins for distributed database systems. In *IEEE Transactions on Software Engineering*, pages 558–560, May 1990.

[17] Napster. http://www.napster.com/.

[18] Scott Shenker Qin Lv, Sylvia Ratnasamy. Can heterogeneity make gnutella scalable? In *First International Workshop on Peer-to-Peer Systems (IPTPS) 2002*, Cambridge, MA, March 2002.

[19] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of SIGCOMM 2001*, San Diego, August 2001.

[20] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. Technical Report 2002, Duke University, CS Department, February 2002.

[21] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[22] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[23] Sumeet Thadani. Meta Information Searches on the Gnutella Network, 2001. http://rfc-gnutella.sourceforge.net/Proposals/MetaData/meta_information_searches.htm.

[24] Steve Waterhouse. Jxta search: Distributed search for distributed networks. http://search.jxta.org/JXTAsearch.pdf.

[25] Beverly Yang and Hector Garcia-Molina. Efficient search in peer-to-peer networks. Technical Report 2001-47, Stanford University, CS Department, November 2001.

[26] Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area locat ion and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.