Magnolia: An Efficient and Lightweight Keyword-based Search Service in DHT based P2P Networks

Ashish Gupta, Manan Sanghi, Peter Dinda. Fabian Bustamante
Computer Science Department, Northwestern University
Evanston, IL - 60201
Email: {ashish,manan,pdinda,fabianb}@cs.northwestern.edu

1 Introduction

DHT-based P2P systems like Chord [4], Pastry [5], Tapestry [7], Kademlia [3]) improve over unstructured P2P systems like Gnutella by providing high scalability for large number of participating nodes as well as deterministic and efficient $O(\log(n))$ lookup and routing. The overlay topology is controlled much more tightly and files are placed at precise locations as determined by hashing functions used in each system. Their lookup and search service provides for looking up and routing exactly specified file identifiers using a distributed routing architecture which can reach its target node deterministically in $O(\log(n))$ time with very high probability. This is a big improvement over flooding techniques used in traditional unstructured techniques, greatly reducing latency, node and link stress in the network. However, this performance comes at a price: their inability to support inexact searches in their original form. Indeed this is of vital important as evidenced by the extremely popular content-distribution and file sharing applications, in the domain of unstructured P2P systems e.g. Gnutella and Kazaa. In these systems users can search for vast amount of media files or documents with only partial specification of the file title or other attributes. DHT-based systems search for documents based on the documentID which is derived from hashing an entire file attribute like its title. With only partial information it is not possible to reconstruct the hash key for the file.

Some recent proposals for keyword search in DHT systems have suggested storing documents or pointers to these documents corresponding to a particular keyword to a node with nodeID corresponding to the keyID. A keyID is computed by computing h(keyword) for each keyword in a document's attribute like its title and the tuple (keyword, documentPointer) is stored at the nodeID which corresponds to this keyID. Composite keyword search can then be made possible by computing the hashes for the search keywords and visiting corresponding nodes to fetch all results which contain the keyword. The results may be processed in the network for operations like OR or AND before returning the final results to the user. While this approach provides correctness and works for simple scenarios, we argue that this approach is not aligned with the goals for which DHT-based systems are orignally intended for namely scalability, low and bounded performance, resilience which takes into account high churn in end host P2P systems, and load balancing in terms of key distribution as well as traffic in the network. Storing all the documents corresponding to a single keyword at one of the nodes can result in many problems: 1) For very large scale systems, containing billions of documents, millions of documents corresponding to a common keyword can end up on a single node, a single node thus becoming responsible for for a large number of document pointers. Depending on the distribution of keywords, distribution of these document pointers can be heavily skewed over the nodes in the network. While heterogeneity-aware proposals can alleviate this to some degree, the imbalance can be large enough not be addressed by choosing the "right" node for each keyword. 2) If a node fails (high frequency failures are common and a basic assumption in end host P2P systems), all documents corresponding to keyword(s) stored on this node are removed from the network, hampering future searches. This is especially problematic if the "popular" nodes storing documents for popular keywords fail. It has been shown before that the keyword popularity is highly skewed in searches. 3) Nodes can be swamped with searches for these popular keywords creating routing hotspots (resulting from routing large number of large number of messages to a single destination, which can take some selected routes) as well as query hotspots in the P2P network.

Unstructured P2P systems are less affected by some of these problems, especially dependence on a single node affecting reliability, hotspots or balancing of documents but at the cost of disseminating much larger and unbounded amounts of traffic and indeterministic performance. Our goal in this paper is create a DHT-based P2P architecture which is not plagued by the fore-mentioned problems while

providing low and bounded values for routing, number of nodes visited and traffic generated for a single query. We believe that a major reason for some of these problems is to use the previous DHT proposals as a basic primitive and building a search service on top of that. This limits the ability to provide bounded $\log(n)$ routing hops and scalability while dealing with these problems at the same time.

This paper proposes a simple DHT architecture Magnolia which enables keyword search in DHT-based systems. Our context in this paper is searching for media files just like in unstructured P2P systems. For example, a user searching for "mystery building" should fetch all songs containing these words e.g. Šarah Mclachlan Building a Mystery. It also supports simple Boolean operators OR and AND, case insensitive search, exact phrase search and multiple search attributes for a document e.g. Title, Author, Conference for a research paper. We believe this is a popular and useful model for searching in P2P systems as evidenced by usefulness and popularity of systems like Gnutella and Kazaa.

Our techniques are based on introducing a novel hashing method called Modular Hashing and new routing and lookup techniques which use Modular Hashing and take into account routing and lookup for keywords in a title. Modular hashing constructs a hash key based on individual key words in the document title rather than the treating the whole title as one unit. Modular hashing enables routing possible for a full filename even if some of the key words are specified, while maintaining constant routing state at the P2P nodes which is a constant multiple of the routing state kept in traditional DHT systems. The intermediate nodes than route each individual keyword separately using a modified routing and location protocol until it reaches the file containing the word with high probability. Our goals while providing the facility of keyword search is to provide good scalability and performance properties for the system. Specifically, Magnolia can scale to a system with millions of nodes with low and provable probabilistic bound on amount of traffic generated and number of nodes visited. It returns results with a bounded $O(\log(n))$ number of hops from the originating client and the routing state maintained per node is constant and does not depend on the number of documents in the system.

One of the advantages of our contributions is that it is very lightweight and is not a higher layer built on top the pre-determined DHT substrate for searching and indexing all the documents. It is difficult to provide good performance and scalability bounds when a service assumes another lower service as a basic primitive as the service on the upper layer then must get around any issues in the lower layer instead of focusing on fixing them. In this paper we use the context of Chord to describe the working of Magnolia. However, these techniques are DHT substrate independent and can also be integrated with other DHT approaches like Tapestry and Pastry.

2 Related Work

Some unstructured P2P systems proposals try to improve performance by limiting searches to a fraction of population. This fraction may still be large to produce satisfactory results and since their is little control over placement of keys or documents, only a random sampling of documents may be returned. Our proposal tightly bounds the numbers of nodes visited for keyword searches while providing to ability to search for rare documents also. There is control over where the document keys are placed in the network, and the proposed DHT routing architecture in deterministic number of hops and traffic generated.

In pSearch [6], an architecture for semantic based search for documents is proposed based on the popular VSM (Vector Space Model) and LSI (latent semantic indexing) models for semantic matching. It proposes distributed versions of these methods suitable for P2P systems, without any centralized dependency. It strives to provide good load balancing using heuristic mechanisms. The performance numbers in terms of traffic produced, number of hops and numbers of nodes visited are not tightly bounded and the number of nodes visited could be upto 0.4 to 1% of the total number of nodes for good accuracy, upto 10,000 nodes for a 1 million node P2P network. It can also suffer from node overload for popular keyword searches as for a single key there is no deterministic distribution of storage as well as lookup overhead. The authors suggest the possibility of replication of keys in the network to deal with any hotspots in the network.

In Bauer et al [1] suggest a distributed method to support SQL like queries over any properties of the document. Their method hashes a document property to a propID and stores all documents corresponding to the same property value on the node corresponding to propID. This would suffer from poor load balancing and resilience for popular queries as described previously. The routing also takes multiple $\log(n)$ DHT routing steps, which can be large for very large networks. It also involves transmitting large sets of intermediate final results from one node to another resulting in potentially large amount of traffic. Lack of an evaluation or analysis of performance or scalability makes the latency, resilience and scalability implications unclear.

In Garces-Erice et al [2], a distributed indexing scheme is proposed, where queries are specified as values over separate fields of a document like Title or author. Their approach is store a hierarchial index structure distributed over the P2P nodes where various possible incomplete queries for a document are generated, indexed and linked to more complete versions of that query (which completes other information in the field). This process is repeated recursively

until the index entry arrives at a complete descriptor of the document , when the document can be retrieved. Besides possibly building huge indexes for a large number of documents (since for a document, many possible incomplete queries are possible), their routing requires take much more time with multiple $\log(n)$ DHT routing steps. Also, a single descriptor is mapped to a single node in the system, resulting in node overload and resiliency problems as described previously. Also, it is not clear if it can be efficiently extended to support individual or group keyword search for a particular field e.g. a certain keyword within a title. They also use caching schemes to respond faster to queries which have been popular in the past.

3 System Model and Problem Formulation

3.1 The DHT System

By DHT-based Peer to Peer systems we refer to systems like Chord, Pastry, Tapestry, Kademlia, Viceroy which have several well defined properties. Some of their important properties are: 1. Each document maps to a unique node in the system. Their mapping methods also achieve good load balancing of keys over the entire set of nodes. 2. They provide $O(\log(n))$ routing and lookup time for any object in the system. 3. Routing state kept at each node is constant and independent of the number of nodes or documents stored in the system Their basic approach is to assign unique IDs to nodes as well as documents via hashing a node attribute (e.g. IP address) and a document attribute (e.g. title) to a unique key. Their routing design then provides to locate the node holding a document given its key. These systems mainly differ in the way they route a request to the final node. The routing design also affects their ability to handle high instability inherent in end host P2P systems.

In this paper we use the example of Chord to illustrate how our methods can be incorporated into an existing DHT system. The next section gives a very brief description of routing in Chord, the part essential for understanding our later discussions.

3.1.1 A brief look at Chord

Figure 1 shows how routing works in Chord. For efficient routing each Chord node maintains a finger table where entry i points to a successor node $n + 2^i$. For a node n to perform a lookup for key k, the highest node n' which lies between n and k is identified. If such a node exists, the lookup is repeated from n', otherwise the successor of n is returned which is maintained in a separate finger table. The figure shows how a key corresponding to 33 would be routed in Chord.

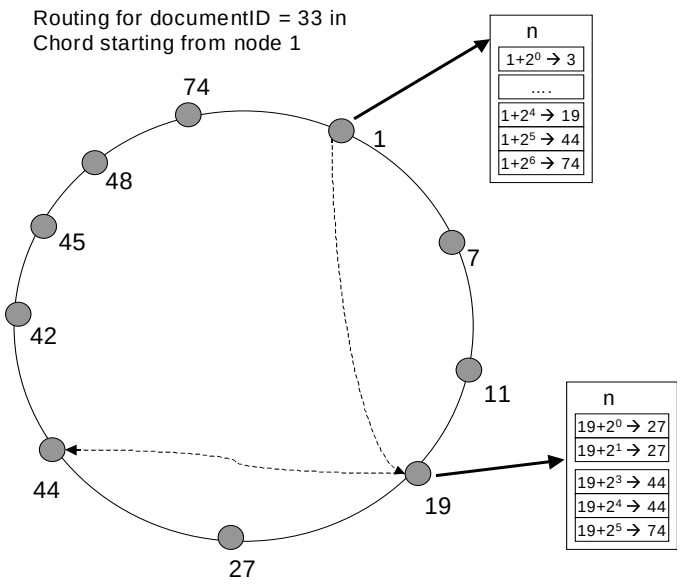

Figure 1. Routing in Chord using Finger Tables. Each node also maintains a successor list which is not shown in the figure

3.2 Document Model and Problem Description

In this paper, we consider a document storage and sharing application where different users store files and other users can retrieve them by searching for them in the P2P system. This is similar in functionality to Gnutella or Kazaa. The system is used to store files in a distributed and resilient fashion which can be later retrieved given its identification. Keeping in mind the popular model prevalent today, in this paper we assume the identification to be of the form of a filename which describes its title. Our approach is also extendible to multiple attributes of a file e.g. MP3 files have ID tag information containing title, artist, album and genre of each song. Users can search based on partial knowledge of any field. In our initial discussions we mainly focus on searching on one attribute i.e. the title to explain our approach. Later we describe how it can be easily extended to support multiple tags.

A title w is assumed to be composed of multiple keywords $w_1.w_2.w_3...w_n$ where the keywords are separated by some delimiter like space, hyphen or underscore. There is no requirement any specific delimiter but only require that there be some way to identify keywords in the title. The design of our system imposes an upper bound on the number of keywords. The length of each keyword is unrestricted. The upper bound is an adjustable parameter which can be varied by adjusting other parameters as we describe below.

A user query is defined by $Q = q_1$ OP q_2 OP q_3 ... OP q_k where q_i is one of these a) a keyword b) an exact phrase P consisting of multiple keywords. Exact phrase searching enables ordered AND searches for multiple keywords. Example "building a" will match "Building a Mystery" but not "A building without windows". OP defines a boolean operator OR or AND. We also want to support queries for multiple attributes as discussed in the last section. A multi-attribute query would be of the form $Q_{multi} = attribute1 : q_1 OP ... OP q_k \; attribute2 : q_1 OP ... OP q_k$ ..... Our goal is to design a system which can retrieve all documents pointers with high probability which satisfy a query Q with the following goals:

1. Performance: The system should be able to return replies within $O(\log(n))$ hops of the originating client node.

2. Scalability: The number of nodes visited and traffic generated per query should be low and bounded. Moreover, these should be adjustable depending on the scale of the system and other requirements.

3. Load Balancing: For popular keyword searches, a single node should not be swamped by traffic but should be spread out over a spread of nodes. In case of node failures, the search capabilities for any of the keywords should not be affected as no node stores all document pointers for a single keyword. The document pointers and documents should be well balanced in the system irrespective of nature of heterogeneity in the system.

4. Routing State: The amount of routing state kept per node should be constant and not depend on the number of documents in the system.

While our model does not support complex SQL like queries, we believe this model is flexible enough to support a wide range of useful queries and has been designed keeping in mind facilities provided by traditional flooding based systems like Gnutella. Moreover, this functionality is provided without much overhead in terms of per node state, traffic generated or routing and lookup performance.

4 Modular Hashing

In this section, we describe modular hashing which is an important component towards enabling keyword based routing in DHT systems. Traditionally in a DHT-based system the entire file descriptor like its title is mapped to a unique m bit key using popular hashing functions like MD5 or SHA1. This key is then use to store and subsequently look up the document. Obviously, we cannot generate the original m bit key for a document from any partial knowledge of file attributes, hence the inherent problem with keyword lookup in DHT systems.

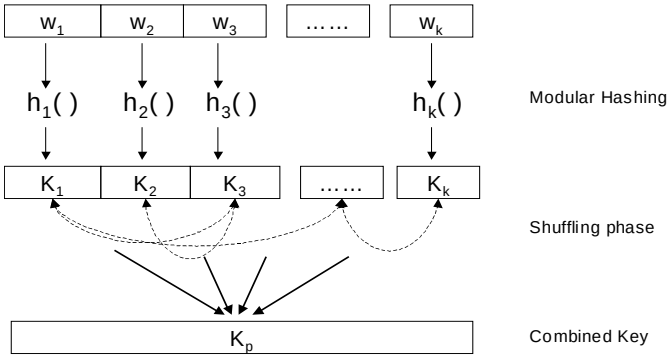

Figure 2. Modular Hashing is used to apply individual hash functions to each keyword which are then shuffled and then combined to obtain the final documentID.

Modular Hashing aims to alleviate this problem. We originally discussed it in context of change detection for high speed network traffic where hash based data structures called sketches were used to store a highly summarized view of very large traffic streams. It is used to hash the document title to get its documentID which is used while storing a document. It is illustrated in Figure **??**. Modular Hashing uses n different hash functions $H_1, H_2, H_3, ...H_k$ each of which maps an arbitrary input to a different m' bit key. Each of these hash functions is applied separately in order on each of the keywords w_1 to w_k to produce keys K_1 to K_k , each of length m' bits. These n m' bit keys are then shuffled to create a random ordering which are then combined to form a m bit key K_p called the Pointer Key. Shuffling helps to get good load balancing properties for storing a set of document title which have lot of common keywords. This is analyzed in more detail in Section 6. The number of keywords allowed per title would be $\frac{m}{m'}$. Modular hashing allows capturing of information about each keyword separately which is then later used in the routing and lookup phase to allow keyword search. The routing and lookup architecture is described in the next section (Section 5).

Another m bit key K_s is also generated independently of modular hashing using the entire title as input. This key is called the Storage Key. The nodeID corresponding to K_p is used to store a pointer to the node corresponding to K_s which actually stores the document. The reason for two different keys is explained below.

Using Modular hashing, when we have m' bits per keyword, the collision rate per keyword can be high if m' is small. There are two things to consider here: First, our fi-

nal key space is still m bits. However, for each keyword, if the keyword space is y bits i.e. there are 2^y possible values for each keyword and these y bits map to a m' bit key , the collision rate per m' bit key (number of keywords corresponding to a unique m' bit key) is approximately $\frac{y}{m'}$. For example if $m' = 12$ and size of keyspace for a keyword is 2^{20}, approximately 256 keywords would map to a particular m' bit key. But these is not the collision rate for the entire title, since each title is made up of several keywords, with each keyword mapping to a separate m' bit key. Also m' is tunable and this collision rate can be decreased to arbitrary levels.

Amount of query traffic is usually much less compared to traffic generated from actual fetching of the entire file. While looking up document titles for containing particular keywords,using Modular Hashing many nodes may be visited which may be potential candidates for the keyword searches. For a single keyword, the number of nodes visited depends on m'. For a n node network, number of nodes having a common value of a particular m' bit key (if their nodeID is also seen as a combination of k m' bit keys) would be approximately $\frac{n}{m'}$. So even in the worst case if more than one node correspond to a potential node containing a keyword search query, this increased query traffic has no effect on the more intensive file fetch traffic since the file is stored at the successor node of K_s. The file fetch traffic is well balanced amongst all the nodes in the network. We provide an upper bound for the routing traffic in Section 6. Note that multiple searches for the same title will result in multiple queries and fetches from the same node in traditional DHT systems also as the same title will be mapped to the same key. Various passive and proactive replication strategies have been proposed to provide load balancing for this. However discussion of load balancing and replication techniques for file storage and retrieval in DHT-based P2P systems is orthogonal to our discussion and out of scope of this paper.

5 Routing and Lookup

In this section we describe how a DHT-based routing system which would work in conjunction with Modular Hashing to obtain efficient and scalable keyword lookups. As discussed before, Magnolia is not a search layer built above a standard DHT-substrate like some other approaches but is a modification of a DHT routing methods to allow for keyword searches. This allows us to obtain tight constraints for performance and also makes search mechanism very lightweight.

We describe the routing schemes in context of routing in Chord, as it is easier to describe. However, other DHT systems like Plaxton-mesh based systems like Tapestry can also be modified to support keyword searches. As described

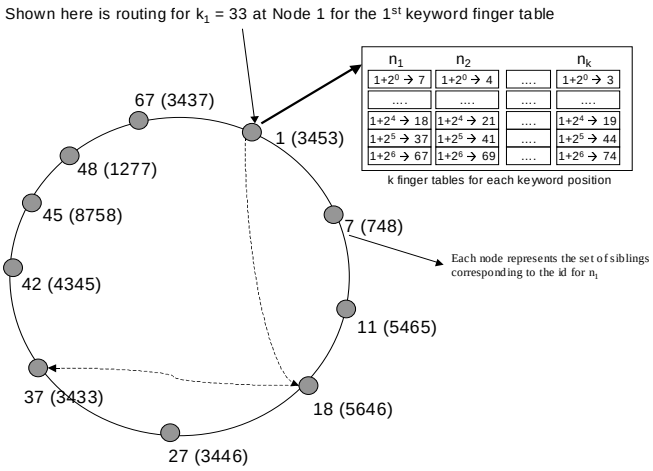

Figure 3. For searching a single keyword, its m' bit key is routed using k different m' bit finger tables in parallel. Only routing with the first finger table is shown for clarity. Each node is labeled with n_1 as well as n in parenthesis. Here each node represents a set of nodes which might have identical values for the m' bit key in the 1st position of their nodeID (n_1).

earlier in section 3.1.1, Chord routes a document key to the nodeID which most immediately succeeds the document ID. Chord achieves scalable $O(\log n)$ routing by maintaining a finger list, containing addresses of nodes corresponding to nodeIDs $n + 2^i$ where n is the current nodeID and i is the key space. This produces an exponentially spaced list of nodeIDs per node and allows locating the nodeID corresponding to any documentID within $O(\log n)$ steps. It also maintains a immediate successor list for correctness purposes.

For keyword-based routing, we want the routing substrate to route to all potential nodes containing a document which has a keyword in the query with a upper bound on number of hops as well as number of nodes visited. First we describe what is the relation between a documentID and the nodeID in Magnolia's model. Each nodeID n is also seen as a combination of k m' bit hash keys $n = n_1.n_2....n_k$. For a documentID whose key is $K = K_1.K_2...K_k$ the documentID K will be located at the closest successor node of K. A particular keyword w_i mapping to a key K_i can be *potentially* located in any node whose j^{th} key n_j is the closest successor to K_i for $j = 1, 2, .., \frac{m}{m'}$. This is because we want to return search results when the keyword is at any position in the title. Also we say *potentially*, because there might be more than node who will be the immediate successor of

K_i in the j^{th} key all with the same value of n_j. This will be true if the keyspace for n_j is smaller than total number of nodes in the system. This number is however low and tightly bounded (Section 6). Since more than one keyword can map to the same value of K_i, it is possible that a node who is the immediate successor if K_i in the j^{th} key does not actually contain the keyword w_i but corresponds to some other keyword. However, this can be determined as we also have each node store the document title as well along with its key. The key is used for location and efficient routing and the title itself is used for verification to make sure the keyword which is in the query is actually contained in the title. We call the set of nodes who have the same value for a m' bit key K in the j^{th} position of their nodeID , $siblings_{K,j}$ i.e. siblings in the j^{th} key for the value K. To summarize, a keyword w_i mapping to the key k_i could be potentially contained in one or more nodes in the set $siblings_{k_i,j}$ for $j = 1, 2, .., \frac{m}{m'}$. As we show later in analysis, this set can be made arbitrarily small with the right choice of m' and m. For example for $m' = 12$ and $N = 10^6$ nodes in the system, this set can be of size 256 with high probability.

Now we describe the routing infrastructure in the context of search for a single keyword w_i. Figure 3 illustrates how routing would proceed for searching for a single keyword. Each node maintains k=$\frac{m}{m'}$ finger tables called the keyword finger tables, corresponding to each of the k^{th} keyword in a title. If a particular keyword w_i maps to the m' bit key k_i, all nodes who are the immediate successor of k_i in any of its keys n_j, can potentially contain the keyword corresponding to k_i in the position j. We reach a node which is the successor of k_i in the j^{th} key by performing repeated recursive lookups in each of the j^{th} finger tables starting with a node n where the query is received initially. When routing to a node, we also forward the keyword w_i along with its key k_i. The lookups in different keyword finger tables are performed in parallel. The algorithm is described in Figure **??**. As described before, the approach here is that each node find the closest predecessor corresponding to key k_i in each of its j finger tables and sends the query request to that node. This repeats until we reach the immediate predecessor P_i of the successor node for k_i where the recursion loop ends and the query is then just sent to immediate successor node of P_i. This will take $O(log(m'))$ steps and the size of each keyword finger table is $log(m')$ entries.

Each node N also maintains a separate *sibling table* $siblingTable_{K,j}$ which stores the list of nodes belonging to the set $siblings_{K,j}$. We only require loose consistency here , i.e. it need not maintain perfect information about all the nodes. We later describe how this table is maintained taking into account transiency in the P2P network. Now when a immediate successor node in the j^{th} field for the keyword K with key k receives the query (one of the members of $siblings_{K,j}$) , it conveys this information to all its

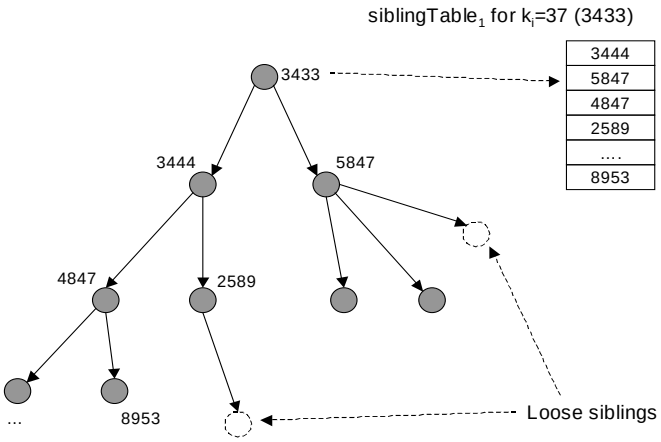

Figure 4. The Initiator Sibling which corresponds to key K_i informs all its siblings from $siblingTable_{K_i,j}$ via a multicast for the query for keyword w_i

$siblings \in siblingTable_{K,j}$. We call this node the Initiator Sibling. It constructs a binary multicast tree over this set and sends the query over this multicast tree.

Figure 4 shows the multicast phase of the keyword search. For the multicast, the Initiator Sibling constructs a sibling tree from $siblingTable_{K,j}$. It creates an ordered list $multicast\_list = \{sibling_i \in siblingTable_{K,j}\}$. Here $sibling_1$ is the Initiator Sibling itself. This ordered list represents the array form of a binary tree, where $sibling_i$ routes the query to $sibling_{2^i}$ and $sibling_{2^i+1}$. It sends this array to $sibling_1$ which then starts the multicast by sending the array to its children. Along with this array it also sends the keyword w_i along, the original query q and j i.e the keyword position corresponding to the finger list along which this query was forwarded and the address of the originating client node to which to send back replies. A $multicast\_message = \{multicast\_list, w_i, q, j, client\_address\}$.

When a sibling in this multicast tree receives the query request, it performs two tasks: 1) It received this query because its key n_j is the closest successor to key k_i corresponding to w_i. Therefore many of its documents could potentially contain this keyword in the j^{th} position of its title. It constructs a reply containing all document titles and their documentKey which have the j^{th} keyword equal to w_i and sends it to $sibling_1$. 2) It checks it siblingTable $siblingTable_{K,j}$ and finds all siblings which are not in the $multicast\_list$ by performing the set operation $\{multicast\_list\} - siblingTable_{K,j}$. It forwards the $multicast\_message$ to these siblings as well, as these siblings may not be a part of the original multicast tree because

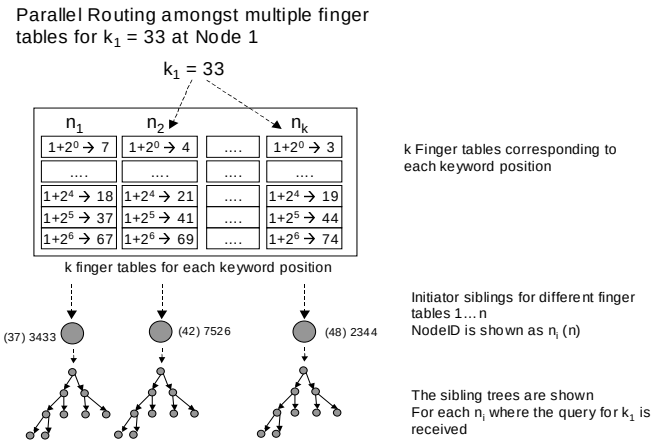

Figure 5. The parallel lookup process for a keyword for all the keyword positions in a title.

of loose consistency. It also adds these siblings to another list $loose\_siblings$ and passes the list to its children along with the multicast message, so that its children do not retransmit the multicast message to the same siblings again.

When $sibling_1$ receives the replies back from its child siblings, it aggregates this information together and sends it directly to the original client node from where the query was initiated. Using the multicast approach allows us to conveniently disseminate the query information amongst the siblings and aggregate the replies at the root to be sent back to the originating client. For clarity, Figure 5 shows how a single keyword query would proceed in parallel amongst different m' bit finger tables and proceed to the sibling nodes for their particular m' bit key corresponding to the keyword.

As we shall show later in the performance analysis section, the maximum number of hops in any of the parallel paths taken including multicast is $O(\log(n))$ with very high probability. The amount of traffic and number of nodes visited is also low and bounded. Moreover, this bounds can be adjusted by adjusting the values of m' and m.

5.1 Handling multiple keywords and boolean operations

In the previous discussion, we described lookup and routing in the context of searching for one keyword. Here we discuss searching for multiple keywords as well as handling boolean operations. For multiple keywords, each keyword query will be routed independently using the mechanism described previously. If the boolean operator is OR, it is equivalent to independent queries which each keyword searched separately. If there is a boolean AND operator between the keywords e.g. B̈uilding AND Mystery¿ each of the nodes which receive a search query for one of these keywords will not return any replies unless it also receives query requests for other keywords which are joined by the AND operator. Hence for AND queries , a node will wait for all keyword searches to arrive at it independently before searching its document titles for the AND query and returning any replies to $sibling_1$. This is because only a small fraction of the sibling nodes corresponding to the one of the keywords may actually also correspond to the keyword as well. Only those siblings need to search their list and reply which are routed requests for all the keywords connected by the AND operator. The nodes can know this because they also receive the complete query q as part of the multicast_message.

An exact phrase search is very similar to AND operator search. For example a query Q=Ḧarder to breathe ẅould be transformed into Q'="Harder AND to AND breathe" for the purpose of searching. When a node receives search requests corresponding to each of these keywords along with the original exact phrase search query, it can check all its matching entries for Q' to see if they contain the exact phrase specified in Q.

Note that there are $k = \frac{m}{m'}$ parallel route requests for one keyword. For y keywords in a query there will be k*y parallel route requests. This is necessary to allow keyword searches in any position of the title.

5.2 Changes required for supporting multiple search attributes

In this section we describe how multiple search attributes can be supported using Magnolia. We described search for more than keyword in a single attribute in the previous sections. However many documents like MP3 files or research papers can contain multiple attributes like Title, Album, Artist, Genre etc. When a document is initially stored in the DHT system, a separate document ID is computed corresponding to each attribute using a different set of hash functions for each attribute (remember that for each attribute we use $\frac{m}{m'}$ different hash functions for Modular Hashing to yield a m bit key). For example an MP3 file may have $documentID_{title}$, $documentID_{artist}$, $documentID_{album}$, $documentID_{genre}$ corresponding to each attribute. The document is stored in only one position as defined by Key K_s, but pointers to it are stored at successor nodes corresponding to each of its documentIDs for different attributes. For supporting routing for each attribute, each node would have to maintain a separate set of k finger tables, enabling keyword search for each different attribute. In this case search based on different attributes essentially is independent of each other.

When a user searches for a document with keywords

in more than attribute, these amount to a AND operator amongst different attributes. For example a user may search for Q=älbum: horse AND title: headlight. The mechanism for achieving search for this is similar to supporting search for AND amongst multiple keywords in one attribute. The search query for each attribute is routed separately and in parallel. When a sibling node receives a search query for a particular attribute, it waits for other attribute queries to arrive as well. Various queries for different attributes must intersect at a node for that node to search its document list and send a reply back. A node can decide to wait because it is also sent the full query Q and it can know if it is required to wait for all search queries corresponding to each query partition separated by AND to arrive before taking any further decision.

6 Performance Analysis

In this section we analyze some important properties of Magnolia which characterize its performance, routing state and scalability.

Performance Results:
$Theorem1$: For a query consisting of a single keyword, results are returned in $O(\log(N))$ hops from the originating node, where N is the total number of nodes in the system.
$Proof$: For every keyword query received at an arbitrary node n, keyword w_i may be stored on any of the keywordGroups corresponding to $h_1(w_i), h_2(w_i), ...h_k(w_i)$. There the query for that keyword is routed for each of these groups using the group finger table in parallel. For each of the k independent parallel routes, each route ends up in one of the random siblings belonging to that group. For any of these routes, the number of hops taken in the group finger table will be m' since the number of entries in the group finger table is m'. This particular result is proved in Theorem 4 in [4].

After the keyword query reaches one of the sibling nodes, it needs to be distributed to the rest of the siblings. Using the multicast approach described in Section 5, the height of the tree can be $\log(\frac{N}{2^{m'}}) + 1$. This is the expected value because, number of nodes in a sibling group is $2^{m'}$. The additional 1 factor is due to delivering queries to loose siblings which can result in one extra hop from the total depth of the tree. Each of these sibling nodes then searches its hash table for presence of keywords in its document pointers stored and returns the replies directly back to the client.

The total number of hops is therefore $m' + \log(\frac{N}{2^{m'}}) + 1 = \log N + 1$. Since the keyword queries are routes in parallel to the different k possible groups for a keyword, this is the number of hops for each parallel route.
Scalability Results:
$Theorem2$: For a single query , number of total nodes visited is $O(k(m' + \frac{N}{2^{m'}}))$
$Proof$: The total number of nodes visited while reaching each of the random siblings is m' for each parallel query route for a keyword. The expected number of siblings for each of the k possible groups for a keyword is $\frac{N}{2^{m'}}$. Thus the expected number of nodes visited for a single keyword query is $k(m' + \frac{N}{2^{m'}})$.

To give a quantitative idea, for 1 million nodes, with $m' = 16$ and $k = 10$, the total number of nodes visited is 320.

$Theorem3$: Amount of traffic generated for a single keyword search is $O(k(m' + \frac{N}{2^{m'}}) + r)$ units where r is the number of document pointers matching the query keywords.
$Proof$: The amount of traffic generated is proportional to the number of nodes visited for each query (See Theorem 2). In addition, a unit of traffic is generated for each of the r replies. Note that r depends on the occurrence frequency distribution of the keywords in the documents. This can be limited by specifying a search bounded on r where a user also specifies the number of results requested. Each traffic unit consists of $\{Query, h_i(keyword), client\_address\}$ for the query and replies consist of $\{document\_title, document_pointer\}$.

References

[1] BAUER, D., HURLEY, P., PLETKA, R., AND WALDVOGEL, M. Bringing efficient advanced queries to distributed hash tables. In *Proceedings of IEEE LCN* (Nov. 2004).

[2] GARCES-ERICE, L., FELBER, P., BIERSACK, E. W., URVOY-KELLER, G., AND ROSS, K. W. Data indexing in peer-to-peer DHT networks. In *24th International Conference on Distributed Computing Systems (24th ICDCS'2004)* (Tokyo, Japan, Mar. 2004), IEEE Computer Society, pp. 200–208.

[3] MAYMOUNKOV, P., AND MAZIèRES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (2002), Springer-Verlag, pp. 53–65.

[4] MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001* (San Diego, CA, September 2001).

[5] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Inter-*

national Conference on Distributed Systems Platforms (Middleware 2001) (Heidelberg, Germany, November 2001).

[6] TANG, C., MAHALINGAM, M., AND XU, Z. psearch: Information retrieval in structured overlays, Oct. 20 2002.

[7] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004), 41–53.