Content-based Document Routing and Index Partitioning for Scalable Similarity-based Searches in a Large Corpus

Deepavali Bhagwat*
Storage Systems
Research Center
University of California,
Santa Cruz
1156 High Street
Santa Cruz, CA 95064, USA
dbhagwat@soe.ucsc.edu

Kave Eshghi
Hewlett Packard Labs
1501 Page Mill Rd., MS 1143
Palo Alto, CA 94304, USA
kave.eshghi@hp.com

Pankaj Mehra
Hewlett Packard Labs
1501 Page Mill Rd., MS 1143
Palo Alto, CA 94304, USA
pankaj.mehra@hp.com

ABSTRACT

We present a document routing and index partitioning scheme for scalable similarity-based search of documents in a large corpus. We consider the case when similarity-based search is performed by finding documents that have features in common with the query document. While it is possible to store all the features of all the documents in one index, this suffers from obvious scalability problems. Our approach is to partition the feature index into multiple smaller partitions that can be hosted on separate servers, enabling scalable and parallel search execution. When a document is ingested into the repository, a small number of partitions are chosen to store the features of the document. To perform similarity-based search, also, only a small number of partitions are queried. Our approach is stateless and incremental. The decision as to which partitions the features of the document should be routed to (for storing at ingestion time, and for similarity based search at query time) is solely based on the features of the document.

Our approach scales very well. We show that executing similarity-based searches over such a partitioned search space has minimal impact on the precision and recall of search results, even though every search consults less than 3% of the total number of partitions.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*

General Terms

Algorithms, Management, Performance

Keywords

similarity-based search, scalability, index partitioning, distributed indexing, document routing

*This research was done while the author was a Research Associate at Hewlett Packard Labs, Palo Alto

KDD'07, August 12-15, 2007, San Jose, California, USA.

1. INTRODUCTION

Finding textually similar files in large document repositories is a well researched problem, motivated by many practical applications. One motivation is the need to identify near duplicate documents within the repository, to eliminate redundant or outdated files and improve user experience [15]. Archival systems [26, 35] need to identify content overlap between files to save storage space by using techniques such as delta-compression [2, 11]. Other applications arise in information management: similarity based retrieval can be used to find all versions of a given document, e.g. for compliance, security, or plagiarism detection purposes. Notice that in this paper we are concerned with textual document similarity, where two documents are deemed to be similar if they share significant stretches of text. This is in contrast to natural language based approaches where the linguistic structure of the document is taken into account.

The operation that is the object of our study is *similarity based retrieval.* Here, a *query document* Q is presented to the system. The aim is to find all the documents $D_1, D_2, \ldots, D_n$ in the repository that are similar to Q, the most similar documents being presented first.

While finding textually similar documents can in principle be achieved by a pairwise comparison of the query document with each one of the documents in the repository using a program such as unix diff, this is clearly very inefficient. To solve this problem, the following framework is commonly used: from every document, a set of features are extracted, such that if two documents are similar, their sets of features overlap strongly, and if they are dissimilar, their sets of features do not overlap. Thus, the problem of document similarity is reduced to one of set similarity. Then, an inverted index [30, 31] is created mapping features to documents. Upon the presentation of the query document Q, its features are extracted, and used to query the inverted index. The result is a set of documents that share some

features with Q; these are then ranked with the document sharing most features coming first. Various authors have developed techniques for extracting features from a document: Manber [22], Broder *et al.* [5, 6], Kulkarni *et al.* [18] and Forman *et al.* [15]. All these approaches generate very specific features: when two documents share even a single feature, they share a relatively large stretch of text (tens of characters). As a result, when the inverted index is consulted, relatively few documents are returned. This is in contrast with techniques such as bag of words analysis, where most documents are expected to share at least a few words.

A single feature index becomes a bottleneck as the size of the repository gets very large, and the index needs to simultaneously handle a large number of updates and queries. Such a situation is typical of document management systems for large enterprises, as well as archival systems dealing with large, continuous streams of documents. One solution is to partition the feature index into a number of subindices, placing each partition on a different server, so that the servers can be updated and queried in parallel. The partitioning scheme used must be such that only a small fraction of the partitions need to be accessed for each update and query. Here, the most obvious schemes, such as using a Distributed Hash Table (DHT) [28, 29, 33, 36] to store the $\langle feature, Document \rangle$ pairs fail. In this scheme, the partitioning of the index is based on the hash of the individual features. For example, given 2^k servers, each hosting a hash table, and the pair $\langle feature, Document \rangle$ that needs to be added to the index, the first k bits of the hash of the feature are used for identifying the server that this particular pair needs to be added to. The problem is that there is no locality of reference for the individual feature hashes: each one of the document features is routed independently, most likely to a different server. As a result, a large number of servers will need to be accessed for each document update and query.

In this paper we present an alternative index partitioning and document routing scheme; one that does not route each individual feature of every document independently, but rather routes all the features in a document together. The outline of our scheme is as follows:

- At *ingestion time*, i.e. when a document is being added to the repository, the document's features are extracted using a feature extraction algorithm. Based on these features, a fraction of the index partitions are chosen, and the document is *routed* to these partitions, i.e. the document's features are sent to these partitions and added to the index there. We call the algorithm by which the partitions are chosen the *document routing algorithm*.
- At *query time*, the same same feature extraction and document routing algorithms are used for choosing the partitions to query. The chosen partitions are queried with the document's features, and the results are merged.

This situation has been depicted in Figure 1. Notice that the choice of which servers to contact, both at ingestion and query time, is entirely based on the contents of the document; at no time do we have to have an interaction with the partitions to determine which one should be chosen. This sets us apart from approaches which apply a clustering algorithm [16, 25, 32] to all the documents in the repository; these approaches need to use knowledge of the existing clusters to route the new document. Our approach, by contrast, is incremental and stateless: only the contents of the document are used to decide which partitions it should be routed to. As a result, we have a very lightweight, client based routing capability.

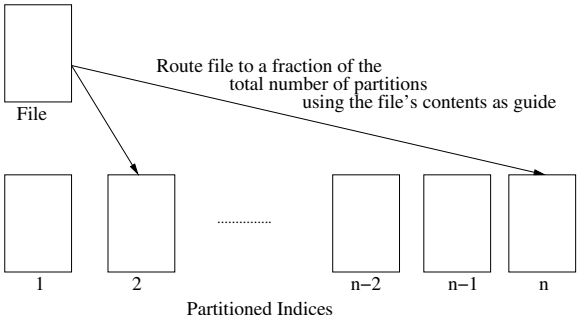

Figure 1: Our solution for document routing

2. BACKGROUND

As stated above, our work assumes the existence of a mechanism to extract features from documents, such that document similarity is reduced to set similarity. We use the Jaccard index as a measure of set similarity. Let H be an algorithm for extracting features from documents, where $H(f)$ stands for the set of features extracted by H from document f. Then, the *similarity measure* of two documents f_1 and f_2 according to Jaccard index is

$$\frac{|H(f_1) \cap H(f_2)|}{|H(f_1) \cup H(f_2)|}$$

There are a number of feature extraction algorithms in the literature that satisfy the requirements above. Shingling [4] is a technique developed by Broder for near duplicate detection in web pages. Manber [22], Brin *et al.* [3] and Forman *et al.* [15] have also developed feature extraction methods for similarity detection in large file repositories.

For the experiments reported in this paper, we used a modified version of the chunk based feature extractor described by Forman *et al.* [15]. This algorithm is described below.

2.1 Chunk Based Feature Extraction

Content-based chunking, as introduced in [24], is a way of breaking a file into a sequence of chunks so that chunk boundaries are determined by the local contents of the file. The Basic Sliding Window Algorithm is the prototypical content-based chunking algorithm. This algorithm is as follows: an integer, A, is chosen as the desired average chunk size. A fixed width sliding window is moved across the file, and at every position k, the fingerprint, F_k, of the contents of this window is computed. This fingerprint is calculated using a technique known as Rabin's fingerprinting by random polynomials [27]. The position k is deemed to be a chunk boundary if $F_k \mod A = 0$. We actually use the TTTD chunking algorithm [13], a variant of the basic algorithm that works better. See [15] for details.

The rationale for using content-based chunking for similarity detection is that if two files share a stretch of content larger than the average chunk size, it is likely that they will

share at least one chunk. This is in contrast to using fixed size chunks, where inserting a single byte at the beginning would change every chunk due to boundary shifting.

We use the characteristic fingerprints of chunks (see below) as the features of the file. Again, the intuition is that if two files are similar, they share a large number of chunks, and thus their feature sets overlap strongly; if they are dissimilar, they will not share any chunks, and thus their feature sets will be disjoint.

Here is the feature extraction algorithm in more detail. This algorithm uses a hash function, h, which is an approximation of a min-wise independent permutation (see section 3.2 below). There are three steps in our feature extraction algorithm:

1. The given file is first parsed by a format specific parser. We handle a range of file formats, including PDF, HTML, Microsoft Word and text. The output of the parser is the text in the document.

2. The document text is divided into chunks using the TTTD chunking algorithm [13]. The average chunk size chosen for these experiments was 100 bytes.

3. For each chunk, a *characteristic fingerprint* is computed, as follows: let $\{s_1, s_2, \ldots s_n\}$ be the overlapping q-grams in the chunk, i.e. the set of all subsequences of length q in the chunk. Then the characteristic fingerprint of the chunk is the minimum element in the set $\{h(s_1), h(s_2), \ldots h(s_n)\}$, where h is the hash function described above. For the experiments in this paper we chose q to be 20.

To summarize, the features of the document are the characteristic fingerprints of the chunks of the document. This algorithm has been demonstrated to produce good features for document similarity. We will not discuss its properties further here, since it is not the subject of this paper.

2.2 The Structure of Feature Indices

In this section, we describe the structure of the feature indices, be they a monolithic feature index for all the documents in the repository, or one of the indices corresponding to a partition of the bigger index.

Figure 2 depicts one of the possible designs of a feature index. The index key is the feature itself. Each feature points to the list of files that it occurs in. This design is analogous to that of an inverted keyword index [30, 31] used commonly in Information Retrieval Systems. This index contains lookup information for every file that has been routed to it at ingestion time.

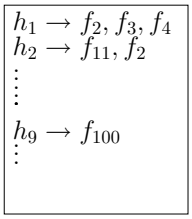

Figure 2: Feature Index

2.3 Building the Feature Indices

When a new file, f_n, needs to be added to the repository an entry for each feature in $H(f_n)$ must be added to the feature index. For every feature in $H(f_n)$, if an entry already exists in the feature index, then the detail for that entry is appended with f_n. If no entry is found then a new entry for that feature is inserted. If f_n is routed to multiple partitions this process is repeated at every one of the destination partitions.

2.4 Querying Feature Indices

When a feature index needs to be accessed to find files in the repository similar to a query file, f_q, then the index is queried using each feature in $H(f_q)$. The set of files similar to f_q is the set

$$\bigcup_{0 \leq i < |H(f_q)|} (I(h_i))$$

where $I(h_i)$ is the set of all the files that h_i points to in the feature index I. Each file in the result set is ranked based on its Jaccard similarity index with respect to f_q.

If multiple partitions need to be queried for f_q then the above querying process is carried out for every partition. The results obtained from each partition are collated such that the set of all the files similar to f_q is given by

$$\bigcup_{i \in R} \left(\bigcup_{0 \leq j < |H(f_q)|} (I_i(h_j)) \right)$$

where the set R is the set of all the partition numbers that were queried for f_q.

3. PARTITIONING THE FEATURE INDEX

As mentioned before, our main interest in this paper is to partition the index I into a number of sub-indices $I_1, I_2, \ldots, I_K$ while preserving the following properties:

- Each one of the partitions has the structure described in section 2.2, i.e. it is a reverse map from features to files.

- At ingestion time, the features of each file, f_n, are used to choose m partitions to which the file will be routed. We call m the *routing factor* and $m < K$. The chosen partitions receive *all* the features of the file, *i. e.* $H(f_n)$ is added to each one of the chosen partitions. This algorithm is the *document routing algorithm.*

- At query time, given the query document f_q, the same document routing algorithm is used to choose which partitions to query. The query process for each partition is as described in section 2.2. The chosen partitions are queried in parallel and independently; there is no background communication among them. The results of the queries from the chosen partitions are merged to form the answer to the query.

- Even though we query only a small subset of the partitions (*i. e.* m is much smaller than K) there is minimal loss of recall compared to the case where there is one global index.

We first describe the document routing algorithm and then provide the justification for why it works.

3.1 The Document Routing Algorithm

The input to the algorithm is

- $H(f_n)$, the set of features of the document f_n
- an integer K, the number of partitions
- an integer m, the routing factor

We assume that $m < K$, and $|H(f_n)| \geq m$. The feature extraction algorithm, H, extracts features using a min-wise independent hash function as explained in section 2.1. The routing algorithm computes a set of integers $R = \{r_0, r_1, \ldots, r_{m-1}\}$ where $0 \leq i < m$. $r_0, r_1, \ldots, r_{m-1}$ are the partitions to which the document will be routed. The document routing algorithm is as follows:

1. Compute $bot_m(H(f_n))$ where bot_m is a function that picks the m smallest integers in a set. In other words, for a set of integers S where $|S| \geq m$, $bot_m(S) \subseteq S$, $|bot_m(S)| = m$, and $x \in bot_m(S) \wedge y \in S \Rightarrow x \leq y$.
2. For every hash h in $bot_m(H(f_n))$ compute (h mod K). R is the set of the resulting integers, $R = \{h \bmod K | h \in bot_m(H(f_n))\}$. The document is now routed to all the partitions indicated by R.

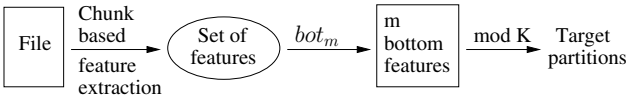

Figure 3: Document Routing Algorithm

The routing algorithm has been depicted in Figure 3.

3.2 Why It Works

The routing algorithm is based on a generalization of Broder's theorem [5]. Broder's theorem relies on the notion of a *min-wise independent family of permutations*. The following definition and theorem are from [5].

DEFINITION 1. *Let S_n be the set of all permutations of $[n]$. The family of permutations $F \subseteq S_n$ is min-wise independent if for any set $X \subseteq [n]$ and any $x \in X$, when p is chosen uniformly and at random from F we have*

$$Pr(min\{p(X)\} = p(x)) = \frac{1}{|X|}$$

In practice, truly min-wise independent permutation are expensive to implement. Practical systems use hash functions that approximate min-wise independent permutations.

THEOREM 1. *Consider two sets S_1 and S_2, with $H(S_1)$ and $H(S_2)$ being the corresponding sets of the hashes of the elements of $S1$ and $S2$ respectively, where H is chosen uniformly and at random from a min-wise independent family of permutations. Let $min(S)$ denote the smallest element of the set of integers S.*

$$P(min(H(S_1)) = min(H(S_2))) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

Broder's theorem states that the probability that the two sets S_1 and S_2 have the same minimum hash element is the same as their Jaccard similarity measure.

Now, consider two files f_i and f_j, and let $m = 1$, i.e. we route each file to only one partition. According to the theorem above, and the definition of similarity measure between two files, the probability that $H(f_1)$ and $H(f_2)$ have the same minimum element is the same as the similarity measure of the two files. In other words, if the two files are very similar, the minimum elements of $H(f_1)$ and $H(f_2)$ are the same with high probability. But if the minimum elements are the same, the two files will be routed to the same partition, since the partition number to which they are routed is the minimum element modulo K, the number of partitions.

While the probability of being routed to the same partition is high when the two files are very similar, the probability drops significantly when the degree of overlap between the two files goes down. For example, if half the features of the two files are the same, and the two files have the same number of features, the Jaccard similarity measure of the two files is 1/3, i.e. there is only one third chance that they would be routed to the same partition.

To overcome this problem, we route the files to more than one partition, i.e. we choose $m > 1$. The intuitive justification for using the bottom m features for routing is that if by chance a section of a file changes such that the minimum feature is no longer in the set of features, the second least feature will now become the minimum feature with good probability. Our experiments and the theorem below show that with m a modest number (less than 5), we have a very good chance that two files with a fair degree of similarity will be routed to at least one common partition.

To formalize our intuition, we can generalize the Broder theorem as follows:

LEMMA 1. *Let S_1 and S_2 be two sets. Let $I = |S_1 \cap S_2|$ and $U = |S_1 \cup S_2|$. Let $B_1 = bot_m(H(S_1))$ and $B_2 = bot_m(H(S_2))$, where H is a min-wise independent hash function. Then*

$$P(B_1 \cap B_2 = \emptyset) \leq \frac{(U-I)(U-I-1)\ldots(U-I-m-1)}{U(U-1)\ldots(U-m-1)}$$

Let $s = I/U$, i.e. s is the Jaccard similarity measure between S_1 and S_2. A good approximation of the above, when m is small and U is large, is

$$P(B_1 \cap B_2 = \emptyset) \leq (1-s)^m + \epsilon$$

ϵ is a small error factor in the order of $1/U$.

When we translate this lemma to the case of documents, we get the following:

COROLLARY 1. *Let f_1 and f_2 be two documents with similarity measure s. When they are each routed to m partitions using the algorithm above, the probability that there will be at least one partition to which both of them are routed is at least $1 - (1-s)^m$*

Now, consider the case where the document f_1 has been ingested into the system, and we now wish to use f_2 as the query document to do similarity based retrieval. Let us say that the similarity measure of f_1 and f_2 is 1/3, and m, the routing factor, is 4. Since the same routing algorithm is used for ingestion and query processes, and for the query

to succeed it suffices that at least one partition be in common between the two files, the probability that we find the document f_1 when we query with f_2 is better than 80%. Contrast this with the case where $m = 1$, when the probability of finding f_1 is only 33%.

The following sections discuss the experimental setup and results.

4. EXPERIMENTAL SETUP

The experimental data set consisted of 179874 files. These were Hewlett Packard's internal support documents in HTML format. There were 1504984 unique features extracted from this set. A randomly selected subset, F_q, of 332 files was chosen from the original corpus to be used as query files. The rest of the files, the set F_d, was our document repository. Our goal was to find for every file $f_q \in F_q$ the files in F_d that were highly similar to f_q. The similarity measure between two files was calculated using their features as explained in section 2.

Since F_d was our document repository every file $f_d \in F_d$ was used to build the partitions using the document routing algorithm as explained in section 2.3. Every file in F_q was then used to query the partitions as explained in section 2.4 to find similar files to itself in F_d. The number of partitions, K, were varied from 1 through 128. The routing factor, m, used to route every file in F_d (for building the partitions) and in F_q (for querying the partitions) was also varied from 1 through 10.

The result set for every query in F_q using a single non-partitioned index was then used as a standard to judge the quality of results produced when the index was partitioned.

5. RESULTS

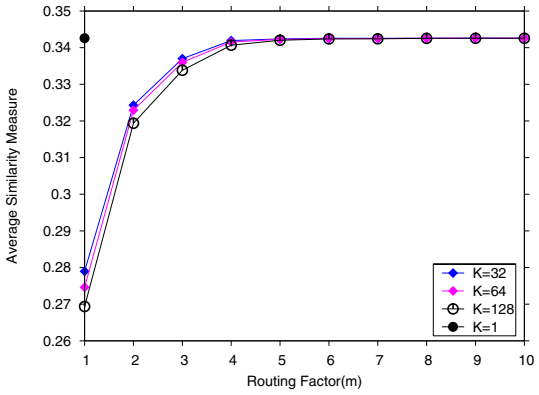

Figure 4: Effect of the routing factor on the average similarity measure

In the first set of experiments we have compared the quality of results obtained when using a single monolithic feature index ($K = 1$) to search for similar files with those obtained when we had multiple partitions ($K > 1$). First, a monolithic feature index was built using every file in F_d. Next, for every query file $f_q \in F_q$ the set of files similar to it were identified by querying the monolithic feature index. Each file, f_r, in the result set for every query f_q was then ranked based on its Jaccard similarity index with f_q. The file with the highest similarity measure was the file that was *most* similar to the query file and hence, the best result. The best result, thus calculated, was recorded for every query file. The average similarity measure, calculated as the average of the best results for all f_q, was then calculated for the entire query set F_q.

The next round of experiments was conducted with increasing number of partitions, $K > 1$. For every value of K, the routing factor, m for every file in F_d and F_q was varied from 1 through 10. Once again, the first step was to build the partitions using F_d for the appropriate values of K and m. Using the *same* values for K and m files in F_q were used to query the partitions. The results obtained from the respective partitions were collated and the best result was recorded for every $f_q \in F_q$. The average similarity measure, for every K and m combination, was then calculated for F_q.

Figure 4 shows the effect of increasing the number of partitions, K, and the routing factor, m, on the average similarity measure of F_q. In the figure, the data point corresponding to $K = 1$ and $m = 1$ corresponds to the average similarity measure for F_q with one monolithic index. This value is 0.342. We can see that for $K = 128$ and $m = 1$ this value is less than 0.27. This is because with increasing number of partitions while using only the minimum feature ($m = 1$) to route the query file it is possible to not find the best match, or the file with the highest similarity measure. When $m = 1$ even a single change that affects the minimum feature of the query file can prevent us from finding the best match as has been explained in section 3.2. The overall average similarity measure for F_q, thus, reduces. However, as we increase m, we improve our chances of finding the best result because we now route every file $f_d \in F_d$ and $f_q \in F_q$ to multiple partitions. We can see that even with $m = 3$ there is a significant improvement in the average similarity measure of F_q for all values of K. For $K = 128, m = 3$ this value is more than 0.33. This means that for a large percentage of query files we are being able to find the file in F_d that shares the highest content overlap with them. For $m > 4$ the average similarity measure for all values of K is 0.342 which means that for every query file we were able to find the best match.

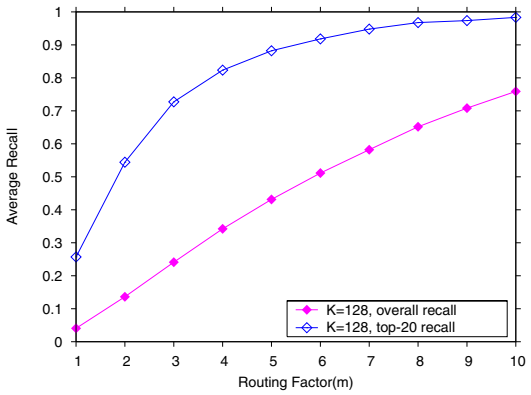

Figure 5: Effect of the number of partitions on the overall recall

Figure 5 depicts the average recall obtained for all the queries for increasing values of m and for $K = 128$. The recall for every query was calculated as the fraction of the size of the result set obtained when $K > 1$ with respect to the original size of results with $K = 1$. The ideal recall, thus, is 1.

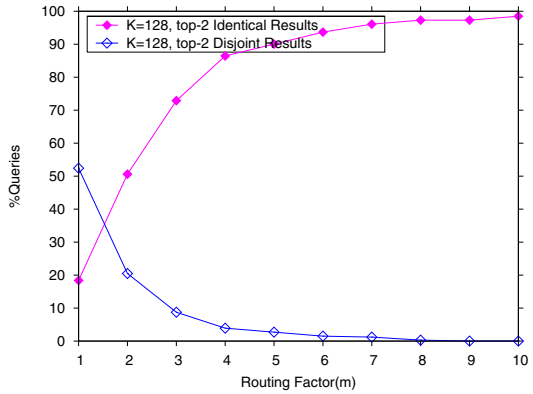

Figure 6: Identical and disjoint top-2 lists, 128 partitions

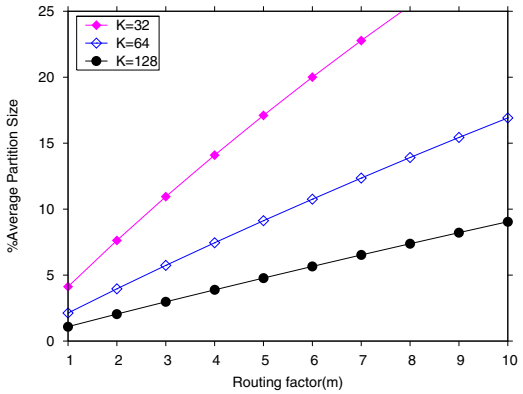

Figure 7: %Average partition sizes for increasing values of the routing factor

The graph in figure 5 depicts the average recall for $K=128$ for increasing values of m in two forms. The first form is the overall recall which takes into account the complete result set obtained for every query. The second form is the recall for only a subset of the result set — specifically the top-20 results in the result set. In this case we retained only the top-20 results for every query. We can see that for $m=3$ the overall recall is 24% whereas the recall for the corresponding top-20 results is 73%. This result shows us that though with $K=128$ and $m=3$ we were able to fetch only 24% of the total result set on an average, this subset contained most of the highest ranking results. This means that we were able to retain and produce the strong resemblances between documents.

We may have lost some of the weak similarity relationships but this loss is acceptable given the gain in scalability. Moreover, for many applications, only the documents with the strongest similarity to the query are of interest, and the low-similarity hits may not be of interest or get filtered out. For example, in the case of a standard search engine users are interested in only the the top few results of their query or the first page of results returned by the search engine. In such a situation the recall achieved by our routing algorithm with respect to the top-20 results is sufficient. However, if an application requires that *every* similar document to a query be found, then one can easily adopt a policy of querying each and every partition instead of just m out of K. Such a scheme will preserve the original recall of every query as was the case when there existed just one index($K=1$).

Figure 6 shows us exactly how many of the top-2 results with $K=128$ were identical or disjoint when compared to those with $K=1$. This data was obtained using techniques developed by Fagin [14]. Identical results were those in which the contents of results were preserved with $K=128$. Disjoint results were those in which none of the contents of the original top-2 results with $K=1$ were preserved when $K=128$. The rest of the top-2 results for $K=128$ contained at least one of the original top-2 results. We can see that as m increases, even with $m=3$, more than 70% of the top-2 lists were identical and less than 10% were disjoint. This means that overall more than 90% of the queries returned at least one of their top results.

Figure 7 depicts the average partition sizes as compared with the size of the monolithic index for increasing values of m. The average partition sizes have been shown as a percentage of the size of the monolithic index. The partition size is the number of keys in the partition — the number of features indexed at every partition. This size does not take into account the list of files that every feature occurs in. Even if we had accounted for it we would observe the same trend as has been shown in the figure. We can see that with $K=128$ and $m=3$ the average partition size is less than 3% of the size of the monolithic index.

We have already seen from our previous results that with $K=128$ and $m=3$ we obtain very good average similarity for our queries (Figure 4) more than 70% of the queries returned their top-2 results identical to when $K=1$ and more than 90% of the top-2 results contained at least one of the original top-2 results. We can clearly see that our routing algorithm has performed very well while reducing the individual partitions to more manageable sizes, enabling the parallel execution of similarity based searches while at the same time has not compromised the quality of our results.

6. RELATED WORK

Routing keyword queries to promising sources of information has been an active area of research in the field of distributed information retrieval and peer-to-peer networks [7, 8, 10, 21]. Cooper [8] and Lu *et al.* [21] use past information about query results to guide queries to promising sources in peer-to-peer and federated information systems. Distributed Hashing has also been studied widely. Litwin *et al.* [20] proposed Scalable Distributed Data Structures based on linear hash tables for parallel and distributed computing. Distributed Hash Tables(DHT) have also been widely used in the area of peer-to-peer systems to distribute and locate content without having to flood every node in the system with content and queries. Content Addressable Network (CAN) [28], Chord [33], Pastry [29] and Tapestry [36] are some of the DHT implementations used in a distributed environment. Manku [23] has categorized the DHT routing methods into deterministic and randomized. Oceanstore [17] is an infrastructure that provides access to data stored across a large-scale globally distributed system and uses the Tapestry DHT protocol to route queries and place objects close to their access points with the objective of minimizing latency, preserving reliability and maximizing the network bandwidth

utilization. PAST [12] is an internet scale global storage utility that uses Pastry's routing scheme. PAST routes a file to be stored to k nodes within the network such that those node identifiers are numerically closest to the file identifier. Pastiche [9] is a peer-to-peer data backup facility that aims to reduce the storage overhead by identifying nodes that share common data at a sub-file granularity. Pastiche aims to conserve storage space by identifying overlapping content using techniques introduced in the Low-Bandwidth Network File System [24]. In order to route data to appropriate nodes, Pastiche needs to access and maintain an abstract of the file system's contents.

7. DISCUSSION AND FUTURE WORK

The document routing algorithm is an effective method for scalable and parallel similarity-based searches. The documents are routed based solely on their contents to only a small fraction of the total partitions while still being able to preserve the precision of the results. We conclude that this algorithm is a good scalable solution.

Similarity-based searches in large scale repositories is only one of the applications for our document routing algorithm. Besides archival systems our document routing algorithm can be used to distribute and locate content in peer-to-peer cooperative storage and backup systems [9, 12, 19] and distributed storage systems [1, 17]. Such systems can save storage space by routing documents to nodes that are expected to store similar content. The recipe for a document [34] consisting of the feature hashes can be used to locate it without having to consult a large number of indices.

The future work in this direction would consist of evaluating schemes that allow the dynamic growth in the number of partitions. We will investigate methods to divide those partitions that become overloaded and the effects of such a scheme on the quality of our results. We will also investigate the efficacy of our partitioning scheme for large scale archival systems that need to identify similar files within their repositories with the intention of conserving storage space. What we gain by partitioning the feature indices, used primarily for the de-duplication of archival data, and using our partitioning method may cost us some storage space as we miss identifying the files with high similarity. Future work will consist of quantifying our losses in the form of storage space and finding out if our gain, in the form of better bandwidth utilization and throughput, outweighs this loss.

8. ACKNOWLEDGMENTS

The authors would like to thank Vinay Deolalikar of Hewlett Packard Labs, Palo Alto for contributing the lemma in section 3.2. The authors would also like to thank Jaap Suermondt and Mark Lillibridge of Hewlett Packard Labs, Palo Alto for their comments.

9. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 36(SI):1–14, 2002.

[2] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, 49(3):318–367, May 2002.

[3] S. Brin, J. Davis, and H. García-Molina. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, New York, NY, USA, 1995. ACM Press.

[4] A. Z. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, pages 21–29, Washington, DC, USA, 1997. IEEE Computer Society.

[5] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[7] J. J. Chua and P. E. Tischer. Strategies for cooperative search in distributed databases. In *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, page 325, Washington, DC, USA, 2003. IEEE Computer Society.

[8] B. F. Cooper. Guiding queries to information sources with infobeacons. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 59–78, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[9] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, New York, NY, USA, 2002. ACM Press.

[10] P. B. Danzig, J. Ahn, J. Noll, and K. Obraczka. Distributed indexing: A scalable mechanism for distributed information retrieval. In *SIGIR '91: Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 220–229, New York, NY, USA, 1991. ACM Press.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, June 2003.

[12] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.

[13] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report HPL-2005-30(R.1), Hewlett Packard Laboraties, Palo Alto, 2005.

[14] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *SODA '03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–36, Philadelphia, PA,

USA, 2003. Society for Industrial and Applied Mathematics.

[15] G. Forman, K. Eshghi, and S. Chiocchetti. Finding similar files in large document repositories. In *KDD '05: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 394–400, New York, NY, USA, 2005. ACM Press.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, Sept. 1999.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, Cambridge, MA, Nov. 2000. ACM.

[18] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, Massachusetts, June 2004.

[19] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 29–41, June 2003.

[20] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[21] J. Lu and J. Callan. User modeling for full-text federated search in peer-to-peer networks. In *SIGIR '06: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 332–339, New York, NY, USA, 2006. ACM Press.

[22] U. Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 1–10, San Fransisco, CA, USA, Jan. 1994.

[23] G. S. Manku. Routing networks for distributed hash tables. In *Proceedings of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 133–142, New York, NY, USA, 2003. ACM Press.

[24] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, New York, NY, USA, 2001. ACM Press.

[25] Z. Ouyang, N. D. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society.

[26] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[27] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[30] G. Salton. *Automatic Text Processing*. Addison-Wesley, 1989.

[31] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[32] H. Schütze and C. Silverstein. Projections for efficient document clustering. In *SIGIR '97: Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 74–81, New York, NY, USA, 1997. ACM Press.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[34] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, June 2003.

[35] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 804–815, Tokyo, Japan, Apr. 2005. IEEE.

[36] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.