Similarity Searching in Peer-to-Peer Databases

Indrajit Bhattacharya, Srinivas R. Kashyap, and Srinivasan Parthasarathy[1]

{indrajit, raaghav, sri}@cs.umd.edu
Department of Computer Science, University of Maryland, College Park, MD 20742.

Abstract

We consider the problem of handling similarity queries in peer-to-peer databases. We propose an indexing and searching mechanism which, given a query object, returns the set of objects in the database that are semantically related to the query. We propose an indexing scheme which clusters data such that semantically related objects are partitioned into a small set of clusters, allowing for a simple and efficient similarity search strategy. Our indexing scheme also decouples object and node locations. Our adaptive replication and randomized lookup schemes exploit this feature and ensure that the number of copies of an object is proportional to its popularity and all replicas are equally likely to serve a given query, thus achieving perfect load balancing. The techniques developed in this work are oblivious to the underlying DHT topology and can be implemented on a variety of structured overlays such as CAN,CHORD, Pastry, and Tapestry. We also present DHT-independent analytical guarantees for the performance of our algorithms in terms of search accuracy, cost, and load-balance; the experimental results from our simulations confirm the insights derived from these analytical models.

1 Introduction

Distributed Hash Table (DHT) based peer-to-peer systems such as CAN, CHORD, Pastry, and Tapestry [12, 18, 14, 20] support a basic abstraction: the lookup. Given a query for a specific *key*, the lookup efficiently locates the node which owns the key. Although all DHTs implement the basic lookup functionality efficiently, most real-life applications demand more. For instance, consider an Information Retrieval (IR) application where nodes publish a collection of text documents. Each document is characterized by a d-dimensional vector. The field of IR is replete with vector-space methods for such document representations (for e.g, see [2, 5]). A query consists of a vector and the user needs all documents in the database which match this vector or which are *semantically related* to it.

DHTs do not support information retrieval applications like the one above. The fundamental reason which renders DHTs ineffective in these situations is that data objects in a DHT are distributed uniformly at random across the network nodes. While this ensures that no node stores too many objects, it also scatters semantically related objects across the network. Thus, when a query is issued, the only way a DHT can return all objects relevant to it would be to flood the entire network, leading to unacceptable network loads.

Our focus in this work is to efficiently support *similarity queries for text information retrieval in DHT based overlay networks*. We introduce a new query model where users issue queries of the form (x, δ). Here x is a data object and δ is a distance measure. The search algorithm needs to return all data objects y in the network such that $f(x, y) \leq \delta$, where f is an application specific *distance function*. The schemes presented in this paper are geared towards the Cosine distance metric which is defined as follows: $f(x, y) = \cos^{-1} \frac{x \cdot y}{|x||y|}$, where $x \cdot y$ is the dot product between the vectors and $|\cdot|$ is the Euclidean (l_2) norm. The Cosine distance is a widely used distance function in text retrieval applications.

The key technical challenges which we attempt to address in this work are as follows:

1. Developing an efficient object placement and search mechanism such that given a query for an object, the search returns all objects in the DHT which are similar to the query object.

[1]Research supported in part by NSF Award CCR-0208005 and NSF ITR Award CNS-0426683.

2. Developing efficient mechanisms for adaptive replication of popular objects so that query loads are uniformly distributed across network nodes. This is particularly relevant for systems which support similarity searching: similar objects tend to be co-located with each other and if an object is popular, then other objects similar to it can also be expected to be popular.

3. Developing mechanisms which are oblivious to the underlying DHT technology so that the resultant system can be implemented over a variety of DHT topologies, making it possible to leverage other advantages specific to each DHT.

The techniques developed in this work address all the challenges identified above. In particular, we view the following as the main contributions of our work.

- We develop an indexing scheme which clusters data such that a group of closely related objects belong to a small set of clusters. This in turn paves the way for an efficient search mechanism for answering similarity queries.

- Our indexing scheme decouples object and node location in the DHT, allowing popular objects to be adaptively replicated in the DHT. We propose simple adaptive replication and randomized lookup algorithms to exploit this feature of our indexing algorithm. Our adaptive replication scheme ensures that the number of copies of a key in the DHT is proportional to its popularity and the randomized lookup scheme guarantees that a query is equally likely to be served by any of the replicas of that key in the DHT. Thus, the replication and randomized lookup algorithms together guarantee perfect load-balancing.

- We present precise analytical guarantees for the performance of our algorithms in terms of search accuracy, cost, and load-balancing. All the algorithmic and analytical results presented here are oblivious to the underlying DHT topology, thus making it possible for implementation over any DHT.

The key driver behind our techniques is the notion of *similarity preserving hash functions* (SPHs) [4]. SPHs provide a powerful and interesting property in the context of our work: given a set of points which are at a small distance from each other, with high probability, an SPH maps these points onto a "small" set of related indices. Such a mapping leads to a simple search strategy as follows: a node u which has a query (x, δ), computes the set of indices which are relevant to object x; u then queries all the nodes which own these indices. The queried nodes return the set of relevant objects back to u. The use of SPHs for developing provably good similarity search algorithms is one of the key innovations of this work.

The rest of the paper is organized as follows. We survey related work in Section 2. Section 3 formally describes our data and query model and Section 4 presents a detailed description of the major techniques developed in this work. Section 5 presents analytical performance evaluation of our schemes and Section 6 presents the results of our experimental studies.

2 Related Work

Several researchers have proposed mechanisms to extend the scope of DHTs beyond the traditional lookup. Vahdat *et al.* [13], Liu *et al.* [10], and Shi *et al.* [16] address efficient keyword searching in DHTs. The work of Gupta *et al.* [8] and Schmidt *et al.* [15] use SPHs to distribute high dimensional data vectors on top of a CHORD overlay. The former supports approximate range queries while the latter supports exact range queries. The work of Gopalakrishnan *et al.* [3] supports efficient set intersection operations using view trees. The `pSearch` system [19] comes closest to our work, since this is the only system prior to our work which supports similarity searching in any DHT.

2.1 The `pSearch` System

The `pSearch` system [19] supports similarity queries for real-valued data vectors for the Cosine distance metric (see Section 3). It is built on top of the CAN DHT [12] and uses Lexical Semantic Indexing (LSI) [5] for indexing text documents. Object coordinates derived from these indices are used for routing and object location. Although the basic goals of `pSearch` is the same as ours, the techniques presented here differ significantly from those developed in `pSearch`. We outline some of the key differences and the resulting trade-offs below.

1. `pSearch` uses projections of object coordinates derived from the LSI algorithm as object indices. This restricts object indices to only real vectors which makes it implementable only on the CAN DHT and *precludes* its implementation over other popular DHTs such as CHORD, Pastry and Tapestry. This is due to the fact that all DHTs invoke customized object-to-node mapping functions for mapping a given object onto a node in the DHT. While the range of this mapping function is a real vector for CAN, this is not the case in other DHTs (for CHORD it is a real number in the range $[0, 360)$; for Pastry or Tapestry it is a bit string). In contrast, our indexing scheme simply partitions the data into clusters and assigns the same index to each object in the cluster; we allow the underlying DHT mapping

functions to assign indices to nodes in the DHT. Thus, our schemes can be implemented over any underlying DHT topology.

2. The object placement algorithm in `pSearch` converts the CAN physical overlay into a semantic overlay such that nodes within a small physical neighborhood store data objects which are similar to each other. This has an implicit advantage since similar objects can be retrieved by flooding a small neighborhood of nodes within a region. This physical locality is not always possible with our object location technique, since we allow the underlying DHT mapping functions to assign object clusters to nodes. Hence, a single-hop neighborhood query in the *pSearch* system may correspond to a DHT lookup in our scheme resulting in increased network traffic. However, the tight coupling of object and node location in *pSearch* virtually makes it impossible for adaptive replication of highly popular objects in the system for effective query load balancing. Our indexing scheme provides for adaptive replication of popular clusters and avoids hotspots; this is one of most significant flexible features offered by our techniques vis-a-vis the `pSearch` system.

3. Our indexing scheme relies on the notion of Similarity Preserving Hash functions which hash a group of related objects onto a small set of indices. While the primary motivation for our work is supporting cosine similarity for use in text retrieval applications, the basic techniques developed here can be generalized to a large class of data and query models and similarity metrics which support SPHs. One such important category is image and multimedia retrieval which can be supported by the SPHs developed by Indyk *et al.* [9]. The use of SPHs also allows us to model the behavior of our system in terms of the search accuracy vs. search cost using precise analytical models which are independent of the underlying DHT. Thus, unlike the `pSearch` system, the use of SPHs make our techniques applicable to a variety of data models and similarity metrics, allows for precise analytical modeling, and is oblivious to the underlying DHT topology, thus staking a strong claim for widespread acceptance in peer-to-peer database applications.

2.2 Adaptive Replication

Object placement algorithm within DHTs typically place objects uniformly at random on one of the DHT nodes in an attempt to balance query load. While this is reasonable under assumptions of uniform query-rate for all objects, in practice, query behaviour tends to follow very skewed zipf-like distributions [17]. This behaviour is even more acute in systems which co-locate related objects to support similarity searching, since objects close to popular objects also tend to be popular. Most DHTs provide only for static replication where each object in the DHT is replicated a *fixed* number of times and hence do not deal with non-uniform query distributions.

The Lightweight Adaptive Replication (LAR) protocol of Gopalakrishnan *et al.* [7] addresses this problem by measuring the load on individual servers and using the load measurements to create appropriate number of copies of a key. They also modify the DHT lookup primitive by augmenting nodes in the DHT with information about the newly created copies. We note that the adaptive replication technique presented in this work is similar in spirit to this scheme, since our technique also relies on server load information for spawning and retracting copies of a key. However, our scheme differs from that of LAR in significant ways: a query node in our scheme is required to have a good estimate of the current number of copies of a key in the system (failing which the query may incur more than a single DHT lookup; however the query is still guaranteed to be successful even without a good estimate). However, unlike LAR, our scheme does not require nodes in the DHT to be augmented with "routing hints" to direct the lookups to the appropriate replicas. We also note that, like LAR, our scheme is also oblivious to the underlying DHT topology and can be implemented on top of any DHT. Finally, our techniques are also interoperable with LAR or any other adaptive replication protocol specific to any DHT.

3 Data and Query Model

Information Retrieval (IR) applications frequently model text documents as *term vectors*. A term vector is a vector of real numbers; coordinates in the vector correspond to *terms* and the value of each coordinate represents the relative frequency of the corresponding term within the document. In general, terms may correspond to keywords or a combination of keywords found within the documents. It is also usual to normalize the term vectors so that vectors are of unit length, in order to account for the variable sizes of the documents. Several techniques exist in the IR literature for representing documents as term vectors, most of which are variants the Vector Space Model (VSM) [2] and the Latent Semantic Indexing (LSI) schemes [5].

For the rest of this paper, we assume that all data objects are unit vectors in a d-dimensional Euclidean space. Equivalently, the data objects may also be viewed as points on the surface of the d-dimensional unit hyper-sphere. Two objects are considered similar, if they lie close to each other on the surface of the unit sphere. Formally, let x and y be two objects and let θ be the angle between them. The similarity between x and y is defined by the function f, where

$f(x, y) = \cos(\theta) = x \cdot y$, the so called dot-product or the inner-product of x and y. The distance between x and y is defined as the angle θ. The greater the value of f, the smaller the angle θ, and the more similar are the objects x and y. We note that the `pSearch` system also works with this data and similarity model.

We assume that user queries of the form $q = (x, \delta)$, where x is a d-dimensional unit vector and $0 \leq \delta \leq \frac{\pi}{2}$ is the distance measure. An object y *matches* query q if y is sufficiently close to x: i.e., $\cos^{-1} x \cdot y \leq \delta$. The search *accuracy* is defined as the fraction of matching objects in the system that are returned by the search. The search *cost* is defined as the number of lookups performed by the system during the search. The algorithms presented in this paper trade-off search accuracy with respect to search cost.

4 Design Details

Four basic techniques underlie the mechanisms developed in this paper. Following is a brief description of these techniques.

- The **Indexing Scheme** partitions the data-space into several clusters. Each data object is assigned an index and clustering is achieved implicitly by assigning all objects which have the same index to the same cluster. The indexing scheme guarantees that any set of objects which are sufficiently similar to each other are assigned either to the same cluster or to a small group of clusters. The indices are treated as keys by the DHT; each index is owned by some node and all objects with this index are stored by the node which owns the index.

- The **Search Algorithm** computes a set of indices S which are relevant to the given query $q = (x, \delta)$; it then performs a lookup for each index in S. These lookups terminate at a set of nodes, which return all objects owned by them that match the query q. In general, a higher search accuracy would require the algorithm to compute a larger set of indices S resulting in higher search costs.

- The **Adaptive Replication** algorithm ensures that the number of copies of each key in the network is proportional to its popularity. Specifically, the number of copies of each key in the DHT is proportional to the rate at which queries arrive for this key. The creation and retraction *thresholds*, which are global system-wide parameters determine how aggressively copies are created or retracted in the system.

- The **Randomized Lookup** algorithm guarantees that the lookup for a specific key terminates uniformly at random at one of the copies of this key. Thus, the lookup and the replication algorithms together guarantee that the load is balanced uniformly across all copies of all keys in the system.

In the following sections, we present the details involved in each of these techniques.

4.1 Indexing

Each data object in the peer-to-peer database is assigned an index. We now propose a hash function h which takes a d-dimensional data object x as input and computes a k-bit string $h(x)$ as output. The string $h(x)$ is the index of object x. Let r be a d-dimensional unit vector. Corresponding to this vector, we define the binary function b_r as follows:

$$b_r(x) = \begin{cases} 1 & \text{if } r \cdot x \geq 0 \\ 0 & \text{if } r \cdot x < 0 \end{cases} \quad (1)$$

$b_r(x)$ defines the orientation of x w.r.t. r. This function was proposed by Charikar [4] for estimating cosine distances between points in high dimensional space. He also observed that if r is chosen uniformly at random from all d-dimensional unit vectors, then for any two vectors x and y, $\Pr[[]b_r(x) \neq b_r(y)] = \frac{\delta}{\pi}$, where $\delta = cos^{-1} \frac{x \cdot y}{|x||y|}$ is the angle between the two vectors in radians. Our hash function h is parametrized by a set of unit vectors $r_1, \ldots, r_k$, each of which is chosen uniformly and independently at random from the set of all d-dimensional unit vectors. The hash value $h(x)$ is simply the concatenation of the bits $b_{r_1}(x), \ldots, b_{r_k}(x)$. Objects with the same index belong to the same cluster. Object x is stored at the node which owns the DHT key $h(x)$.

The above hashing scheme essentially attempts to group nearby objects to indices with low hamming distance. However, there is still a reasonable chance that nearby objects can differ in some bit positions in their indices. In order to reduce the probability of this bad event from occurring, we construct t hash functions $h_1, \ldots, h_t$ as described above, which yields t sets of object indices. This ensures that there is a high probability of two related objects hashing onto indices with low hamming distance in at least one of these sets. We note that we can treat these sets of indices as a *static* replication of objects; the static replicas are also analogous to the "rolling indices" in the `pSearch` system. Further, we show both using theoretical analysis (see Section 5) and using simulations (see Section 6) that static replication boosts the search accuracy. However, it does not address the problem of load balancing, which we deal with in Sections 4.3 and 4.4. We emphasize again that these *static* replicas of objects are not the same as multiple *copies* of the DHT keys. For the remainder of the paper, we use the term replicas to denote the static replicas and the term copies to denote the multiple copies of a DHT key created by the adaptive replication algorithm.

4.2 The Search Algorithm

The search algorithm is parametrized by a radius r, which is a non-negative integer. A node u which generates a query (x, δ) first computes the index $h(x)$. It then computes the set S of all indices whose hamming distance from $h(x)$ is at most r (i.e., the set of indices which differ from $h(x)$ in at most r bit positions; note that S always includes $h(x)$). Let V be the set of nodes in the network which own the keys in S. Node u queries each of the nodes in V. Nodes in V return all data objects which match u's query.

How is the search radius r determined? The search radius r is affected by various parameters such as k, t, the query parameter δ, and the desired search accuracy. Fixing all other variables, an increase in the value of r would result in more objects which match the query being returned. Of course, the increased accuracy is also achieved at an increased search cost. We examine the effect of r on the search accuracy and cost in Section 5. We note that the search algorithm may be easily extended to the case where we have t static replicas of the objects in the system.

4.3 Adaptive Replication

We now present the details of our adaptive replication scheme which creates and retracts keys adaptively depending on load conditions. Keys in our system refer to indices within a specific static replication, although, in general, the replication scheme is oblivious to what the keys may refer to. Recall that in a DHT, each key y is mapped onto a random value $m(y)$ using a mapping function m; a lookup for $m(y)$ terminates at a specific node u which is said to own $m(y)$; this node stores a copy of the key y. Our replication scheme parametrizes the mapping function m with a positive integer i. Specifically, let $s = m(i, y)$. Then, the node which owns s is responsible for storing the i^{th} copy of the key y. We now describe how to a create a new copy and retract an existing copy of a key in the DHT.

Consider a key y which currently has l copies. The main invariant maintained by the replication algorithm is that the copies are contiguous, ranging from 1 to l: i.e., the l copies are placed at nodes which own values $m(1, y), m(2, y), \ldots, m(l, y)$ respectively. The copies can be visualized as nodes of a complete binary tree, with copy i being the parent of copies $2i$ and $2i + 1$. We note that the binary tree abstraction is completely implicit and there are no pointers associated with children or parents of copies in reality. All nodes in the system maintain two thresholds r_{high} and r_{low} and a periodic local timer. A node which owns a copy of y performs the following check at the end of each period: let the number of queries it received for key y in the previous time period be q; if $q \geq r_{high}$, it creates copies $l + 1$ and $l + 2$ of y. If $q < r_{low}$, it retracts copies l and $l - 1$ of y. Creation and retraction are both achieved by sending a message to the parent of the nodes which own the corresponding copies; if the parent has not already performed the creation (retraction) it performs this action after receiving a creation (retraction) message. These messages are routed using the standard lookup primitives of the DHT. We observe that the creator or retractor of a copy need not know the node which owns the last copy l of y, or its parent, but just the value of l.

How does a creator (or a retractor) of a copy know the value of l? We note that a simple solution is to notify all nodes which own a copy of y, whenever a copy is created or retracted in the system. Yet another solution is to perform a simple binary search in the range $1, \ldots, l_{\max}$, where $l_{\max}$ is the maximum number of replicas allowed for any key within the DHT. We note that the latter discovers the value of l after $O(\log(l_{\max}))$ lookups with high probability.

4.4 Randomized Lookup

Let node u generate a query for key y and let there be l copies of y currently in the system. If node u knows the exact value of l, it chooses a random number i in the range $1, \ldots, l$ and performs a lookup for $m(i, y)$. This ensures that all copies of the key are equally likely to serve this query. However, in general, nodes can not be expected to have exact information about the number of replicas for a specific key in the DHT. We now show how our randomized lookup solves this problem in two scenarios. In the first scenario, node u does not have any information about l. In this case, it performs a randomized binary search in the range $1, \ldots, l_{\max}$ to obtain a copy of y. Specifically, u selects a random number l_1 uniformly in the range $1, \ldots, l_{\max}$ and performs a lookup for $m(l_1, y)$. If this lookup returns a copy of y, the lookup terminates. Else, the node repeats the randomized binary search in the range $1, \ldots, l_1 - 1$. The randomized lookup terminates whenever any of these DHT lookups returns a copy of y. Observe that the randomized lookup is guaranteed to terminate for any initial estimate of l since the successive DHT lookups are in strictly decreasing ranges and a lookup for $m(1, y)$ is guaranteed to terminate successfully.

Fixing the initial estimate of l at $l_{\max}$ results in at most $O(\log(l_{\max}))$ DHT lookups with high probability. However, this increased lookup latency may be unacceptable for many applications. One way to avoid this problem is for each node to estimate the value of l using counting bloom filters [6]. Counting bloom filters are compact data structures for checking set membership in distributed environments. In our setting, the entries in the counting bloom filter are of the form (i, y). If such an entry exist, it indicates that the i^{th} copy of key y exists in the system. These bloom filters are updated periodically to reflect any changes

in the number of copies of any key. We note that an exact estimate for the number of copies is not required for the correctness of our lookup algorithm. Hence, one possible optimization in the bloom filter design is to just store entries of the form (i, y) where i is a power of two, instead of all values of i in the range $1, \ldots, l_{\max}$. This results in an estimate of l which is at most within a factor of two from the correct value, thus only slightly increasing the randomized lookup latency while decreasing the size of the bloom filter substantially (from $O(l_{\max})$ to $O(\log(l_{\max}))$. The work of Mitzenmacher [11] discusses several techniques for updating counting bloom filters in distributed settings with low message overhead.

5 Analysis

Consider a query $q = (x, \delta)$. Let S be the set of all objects in the database which matches this query. Let S' be the set of objects returned by the search algorithm. Recall that t is the number of static replications, k is the the number of bits in the index and r is the search radius. We define the accuracy of the search to be $|S'|/|S|$, i.e., the fraction of objects in the database which match the query and which are returned by the search. $\mathbf{E}[|S'|/|S|]$ denotes the expected accuracy. Due to lack of space, we present the following theorems without proof and defer the detailed proofs to the journal version of this paper.

Theorem 1

$$\mathbf{E}[|S'|/|S|] \geq 1 - \left(1 - \sum_{i=0}^{r} \binom{k}{i} \left(\frac{\delta}{\pi}\right)^{i} \left(1 - \frac{\delta}{\pi}\right)^{k-i}\right)^{t} \quad (2)$$

Theorem 2 *Let the number of keys being looked up be C. Then,*

$$C = t \sum_{i=0}^{r} \binom{k}{i} \quad (3)$$

We note that in general, the search cost could potentially be greater than C. This is because, each key lookup could result in more than a single DHT node lookup in the randomized lookup algorithm. However, we show in Section 6 that this is not the case: each key lookup on an average incurs only slightly more than one node lookup even with a bloom filter of small size.

Theorem 3 *Let the number of copies of a key y at time t be l. Then a lookup for key y at time t will terminate at one of these l copies uniformly at random.*

6 Experiments

Our experiments assume an underlying CHORD network that provides lookup, insert and delete primitives. The number of nodes in the network is fixed throughout all our experiments.

6.1 Similarity Search

Data objects in our simulations are sampled uniformly at random from the surface of the d-dimensional unit hypersphere. This is achieved by sampling each coordinate of the vector independently from a standard normal distribution. A k-bit index of a data object is created using k random unit vectors. We use static replication with t hash functions. We use the publicly available CHORD simulator [1] for evaluating the accuracy results. Initially all nodes and keys are inserted into the CHORD simulator. Each query object is a randomly sampled d-dimensional unti vector. We observe the effect of the number of replicas t, the search radius r, the size of the index k, the dimensionality of the data d, the number of nodes n, the number of data objects N, and the query parameter δ on the search accuracy and storage load. The default values of these parameters are in the table below. Results are averaged over 100 trials.

N	d	k	t	r	δ	n
50,000	15	10	1	1	0.75	2^k=1024

Table 1. Default values for network parameters used in the similarity search experiments

To evaluate storage load, we sort the nodes in decreasing order of number of objects they store and group them into 20 buckets. For each of the buckets, we plot the percentage of the total number of objects stored in the nodes of the bucket. The baseline for comparison is the uniform distribution where each bucket stores 5 percent of the objects. Figure 1 (a)-(g) plot the effect of the various system parameters on accuracy. We plot both the experimentally observed values as well as the analytically predicted ones. The accuracy increases as a function of the number of replicas t and the search radius r. It does does not vary much as a function of the data dimension d or the number of nodes n or the number of data objects N in the system. However, the accuracy decreases with the size of the index k as well as the query parameter δ. Our analysis predicts the experimental trends accurately in all the trials. This suggests that the accuracy guarantees provided by our analysis do not only hold in expectation, but also with high probability. Also note that the experimentally observed values are always higher than the analytically predicted ones. This is

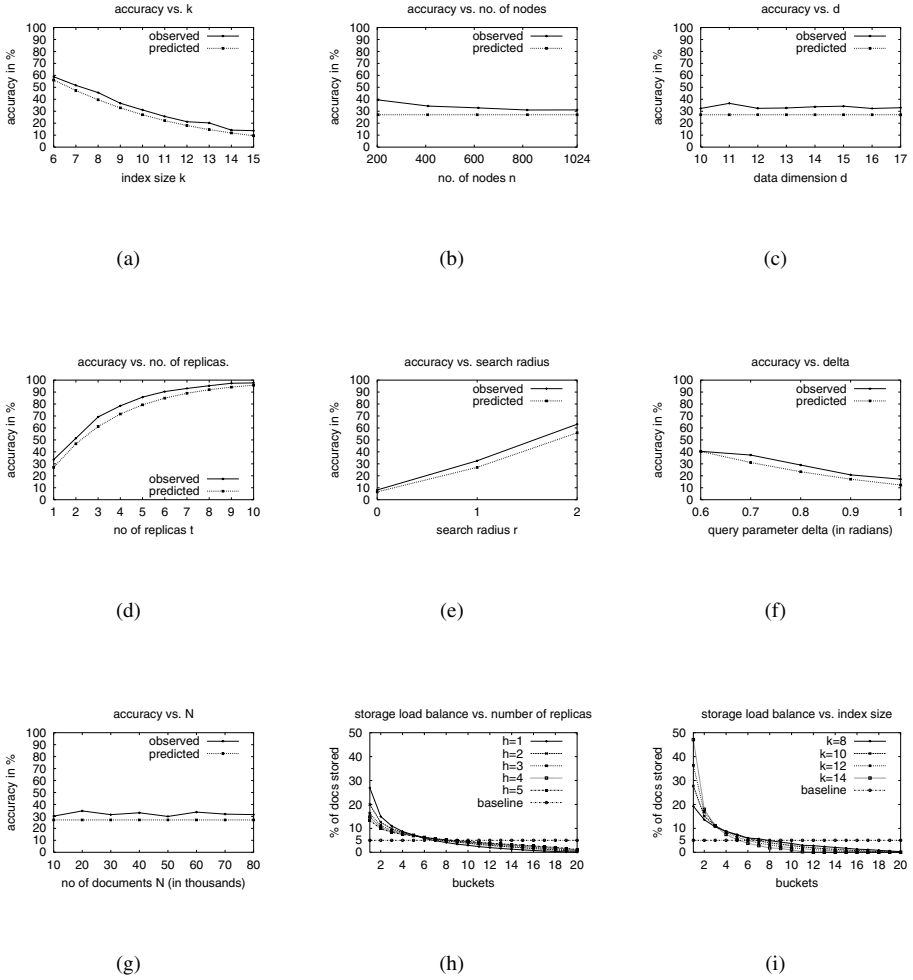

Figure 1. Experimental Results for Accuracy and Storage Load Balance

explained by the fact that our analysis always yields a lower bound on the expected accuracy rather than the exact value.

Figure 1 (h)-(i) plot the effect of the number of replicas and the size of the index on the storage load across nodes. We observe that increasing the size of the index k adversely affects the storage load balance while increasing the number of replicas t aids load balance. Varying other parameters does not seem to change the storage distribution across the nodes.

6.2 Adaptive Query Load Balancing

In our adaptive query load balancing experiments, we have 100,000 data objects distributed over a network with 5000 nodes. The data and keys are generated as mentioned in the previous section. We generate 100,000 queries for these objects according to a Zipf distribution. The skew in the distribution causes a small number of keys in the network to become load hot-spots while most other keys receive very few queries. The queries arrive according to an exponential distribution with an expected inter-arrival time of 1 time unit . Local timer events occur every 1000 time-units.

All nodes maintain a query log (since the last local timer event) for each copy of a key assigned to it. At a local timer event, a node calculates the query rates for each of the keys assigned to it and then decides for each key whether to create a copy or retract an existing copy. This decision in determined by global creation and retraction thresholds. We study the effect of load balance with respec to the thresholds. In all our experiments, retraction is turned off ($r_{low} = 0$). This also makes analyzing the results easier. We set $l_{\max}$, the maximum number of copies of a key to 250.

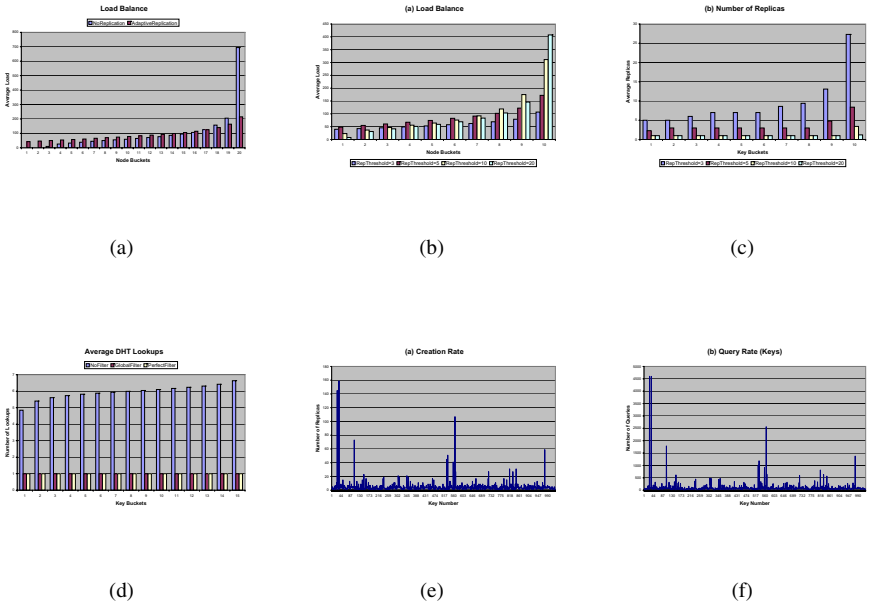

Figure 2. Experimental Results for Adaptive Replication

When querying a key y, a node needs to estimate the number of active copies of y. The query converges to an active copy using randomized binary search over the range $[1, est(y)]$, where $est(y)$ is the estimated number of current copies of y. This range determines the number of DHT lookups before the query terminates. We compare two different schemes for estimating the number of copies. First, we use a global counting bloom filter that has 4-bit counters, 2 hash functions and whose size is $3 \times 2^k \times l_{\max}$, where k is the size of each key (10 in our case). This bloom filter experimentally generates a false positive rate of 0.237. We compare this with the pessimistic estimate $est(y) = l_{\max}$. We also compare with the ideal (hypothetical) scenario where every node has perfect knowledge about the number of copies for each key.

In Figure 3 and 4(a) respectively, we compare load balance across nodes with and without adaptive balancing and with different creation thresholds. Both figures show the top 20 percent of the nodes that have the highest loads. Clearly, the load on the hot-spot nodes is alleviated by spreading the load over other nodes. It can be observed that nodes that had low load with no adaptive replication have progressively higher loads with more aggressive copy creation. However, the load balancing does not come for free. We can see from Figure 4(b) that the lower the threshold, the more the number of copies created, which explains the reduced load on hot-spots. It can also be seen from the plot that the number of copies created is not uniform over the keys. Some keys have significantly more copies than others for all thresholds. Ideally, we would like the popular keys to have more copies. This is confirmed by Figure 6. Figure 6(b) shows the query rate over the 1024 keys in the network. Note the significant spikes which denote the popular keys. Figure 6(a) shows the number of copies created for the same keys when using the most aggressive threshold of 3. We can see that the replica creation correlates with query rates. Specifically, the following table presents the correlation coefficient of the distribution of requests over keys and the distribution of number of copies created for the keys for different creation thresholds. We can see that in all cases the distributions are very strongly correlated, and the more aggressive the creation threshold, the stronger the correlation.

thresh	3	5	10	20	50
CorCoeff	0.975	0.971	0.939	0.899	0.724

Table 2. Correlation Coefficients between query rate on keys and number of replicas created for key when using different creation thresholds. 1.00 indicates perfect correlation

In Figure 5, we compare the average number of DHT lookups for a query on a key when estimating the current number of copies for the key using a bloom filter with the pessimistic estimate of $l_{\max}$ and the hypothetical scenario

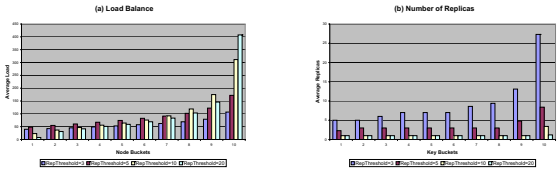

Figure 4. Effect of Varying Replication Threshold

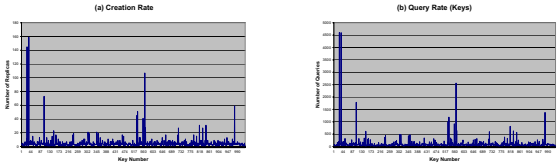

Figure 6. Replication Response to Query Rate

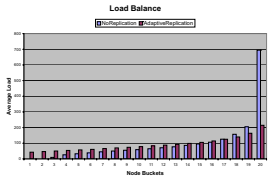

Figure 3. Effect of Adaptive Replication on Load Balance.

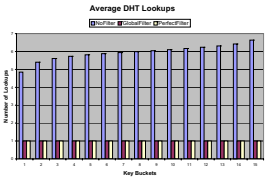

Figure 5. Average Number of DHT Lookups with Global Bloom Filter, Perfect Knowledge and Randomized Binary Search.

when the correct number of copies in known. We observe that the use of the bloom filter with a small false positive rate reduces the number of lookups to 1 per query, which is the case for perfect knowledge. However, without the bloom filter, the average number of lookups is about 5.

References

[1] http://www.pdos.lcs.mit.edu/chord/.

[2] Michael W. Berry, Zlatko Drmac, and Elizabeth R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[3] Bobby Bhattacharjee, Sudarshan Chawathe, Vijay Gopalakrishnan, Peter Keleher, and Bujor Silaghi. Efficient peer-to-peer searches using result-caching. In *The Second International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[4] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM Press, 2002.

[5] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[6] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[7] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer sys-

tems. In *The 24th International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[8] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[9] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625. ACM Press, 1997.

[10] Lintao Liu, Kyung Dong Ryu, and Kang-Won Lee. Keyword fusion to support efficient keyword-based search in peer-to-peer file sharing. In *Cluster Computing and the Grid (CCGrid)*, 2004.

[11] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 144–150. ACM Press, 2001.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[13] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Middleware*, 2003.

[14] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed 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.

[15] Christina Schmidt and Manish Parashar. Flexible information discovery in decentralized distributed systems. In *IEEE International Symposium on High-Performance Distributed Computing (HPDC-12)*, 2003.

[16] Shuming Shi, Guangwen Yang, Dingxing Wang, Jin Yu, Shaogang Qu, and Ming Chen (Tsinghua University). Making peer-to-peer keyword searching feasible using multi-level partitioning. In *The Third International International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[17] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. *Featured on O'Reilly's www.openp2p.com website*.

[18] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[19] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186. ACM Press, 2003.

[20] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. To appear in IEEE Journal on Selected Areas in Communications.