A Landmark-based Index Architecture for General Similarity Search in Peer-to-Peer Networks [*]

Xiaoyu Yang and Yiming Hu
Dept. of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030 USA
{yangxu, yhu}@ececs.uc.edu

Abstract

The indexing of complex data and similarity search plays an important role in many application areas. Traditional centralized index structure can not scale with the rapid proliferation of data volume. In this paper, we propose a scalable index architecture built on top of distributed hash tables (DHT), to support similarity search in the general metric space. Based on efficient space mapping and query routing mechanisms, our architecture can provide a general platform to support arbitrary number of indexes on different data types. Significantly, it does not need to generate or maintain any search trees. Instead, the embedded trees in the underlying distributed hash tables are exploited to deliver queries. To deal with skewed data distribution, we also provide load-balancing mechanisms to ensure that no node in the system is unduly loaded. The performance of the proposed design is evaluated through simulations with a variety of metrics. The experimental results demonstrate that out approach can efficiently solve similarity query at a low cost.

1. Introduction

In recent years, there has been an increasing demand to share digital contents (e.g. music, video, images and text etc.) and support complex queries such as similarity search on these data. Similarity data retrieval has been applied to many fields such as data mining, information retrieval and computation biology. Similarity search has received extensive research attention in centralized database systems and many indexing schemes have been proposed in this context. However, these centralized solutions can not scale to massively distributed systems with a large amount of data. Distributing the index structures and query processing across multiple nodes becomes necessary.

The distributed hash table paradigms (Chord [20], Pastry [17], Tapestry [25] and CAN [16]) are appropriate for building large-scale distributed applications due to their scalability, fault-tolerance and self-organization. However, these DHTs are designed for *exact* key lookup. Complex Queries such as similarity search and range queries can not be efficiently supported since consistent hashing mechanisms destroy *data locality* (similar data objects in original data space are mapped to the same node or to nodes that are close together in the overlay network). The challenges of extending current DHTs to efficiently support similarity search on complex data include: (1) the design of an effective mapping mechanism to map data objects to nodes in the overlay network, while preserving the data locality; (2) the design of a light-weighted routing algorithm to efficiently deliver queries to the corresponding index nodes; (3) the design of load-balancing mechanisms to ensure uniform distribution of load among nodes.

In this paper, we propose a novel architecture, built on top of Chord, for supporting similarity search in peer-to-peer networks. Working as a scalable indexing platform, the proposed architecture can simultaneously support multiple indexes with various data types. Firstly, our design is based on generic metric space, any type of dataset with a corresponding "black box" distance function to compute the distance (dissimilarity) between members of the dataset can be indexed on the proposed indexing platform; Secondly, this architecture does not need to maintain multiple individual routing structures for each index scheme. Instead, it utilizes the trees embedded in the underlying DHT to resolve and deliver queries. So it can simultaneously support many index schemes without additional in-network data structures maintenance overhead. In addition, exploring DHT links can also eliminate the maintenance cost of routing table by piggybacking the maintenance messages onto the query de-

[*]This research was supported in part by the National Science Foundation under grant CCF-0541103.

livery messages. To deal with the load balancing problem, we provide both static and dynamic mechanisms to adjust the load among nodes, to ensure that no node in the system is unduly loaded.

The rest of this paper is structured as follows. In section 2, we present the necessary background of this work. Section 3 describes the key features of our design. In section 4, experiments and results are presented and discussed. Section 5 gives a short survey of related work. Finally, section 6 is the conclusion and future work.

2. Background

Our index architecture is based on the general metric space in [5]. To formalize the problem, we give standard definitions of the metric space and the near-neighbor search as below.

Definition 1 (Metric Space) *A metric space is composed of a data domain D and a distance function $d : D \times D \to \mathcal{R}$ which calculates the distance between any pair of objects in D. $\forall x, y, z \in D$, the distance function d satisfies the following properties:*

$d(x, y) \geq 0$	*(positivity)*
$d(x, y) = 0$ *iff* $x = y$	*(reflexivity)*
$d(x, y) = d(y, x)$.	*(symmetry)*
$d(x, y) + d(y, z) \geq d(x, z)$	*(triangle inequality)*

Definition 2 (Near Neighbor Search) *Given a metric space (D, d), a data set $X \subseteq D$, a query point $x \in D$, and a range $r \in \mathcal{R}$, the near neighbor search is to find the set of objects $Y \subseteq X$, such that $\forall y \in Y$, $d(x, y) \leq r$.*

For instance, the following six examples satisfy the model of searching near neighbors in the metric space: (1) searching similar DNA or protein sequences in a large genetics database; (2) searching similar vocal patterns in a pattern databse; (3) searching similar images in a large image library; (4) searching approximate time series in data mining; (5) searching related documents in information retrieval; (6) searching similar sentences in a large documents database. One can easily find many other applications where the model can be applied.

Among above examples, (2) and (4) find neighbors in a high-dimensional vector space under the L_1 or L_2 metric[1]. (1) and (6) find near neighbors in the metric space of strings under the *edit distance*[2]. (5) uses the *cosine* metric (the angle between the term vectors of the documents) to measure the dissimilarity between documents. (3) satisfies the model under some specific distance functions, e.g. Hausdorff metric [14].

[1] $L_k(x, y) = \sqrt[k]{\sum |x_i - y_i|^k}$, where L_1 and L_2 are called *Hamilton distance* and *Euclidean distance* respectively.

[2] The *edit distance* function of two strings, s_1 and s_2, is defined as the minimum number of point mutations (change, insert or delete a letter) required to change s_1 into s_2.

Our distributed index architecture intends to deal with arbitrary metric space. Given any type of data domain D and a "black box" distance function, which satisfies the properties in the definition of metric space, to compute the distance between data points in D, the data objects in D can be indexed on our indexing platform. Given a query point $q \in D$, the indexing platform can quickly answer the query by finding the set of data objects close to point q.

3. System Design

In this section, we describe the design of our indexing platform on top of Chord. Techniques discussed in this paper are also applicable to other DHTs such as Pastry and Tapestry. In the rest of this part, we first introduce the construction of the landmark-based index space from the general metric space; then we give a description of the locality-preserving hashing mechanism to distribute the index entries onto nodes in the overlay network; afterwards, we explain the design of query routing mechanism; Finally, we present our load balancing mechanisms.

3.1. Landmark-based Index Space

Given a generic metric space (D, d), where D is the data domain and d is the corresponding distance function, the landmark-based index space is constructed as following:

(1) Select a set of data points in D as landmarks, let $L = \{l_1, l_2, \ldots, l_k\}$.
(2) Map each data object $x \in D$ to point $(d(x, l_1), d(x, l_2), \ldots, d(x, l_k))$ in the index space.

The index space construction can be viewed as a mapping from the original metric space to a k-dimensional vector space based on a group of pre-selected data points. Each data object in the original metric space is mapped to a point in the k-dimensional index space. Due to the triangle inequality, the above space mapping is contractive. And the data locality can be achieved since similar data objects are mapped to the close points in the index space.

Near neighbor search in the index space: Given a near neighbors query (q, r), where $q \in D$ is the query point and $r \in \mathcal{R}$ is the query range. Based on the triangle inequality, for any $x \in D$ and $l_i \in L$, we have

$$\begin{cases} d(q,x) + d(q,l_i) \geq d(x,l_i) \\ d(q,x) + d(x,l_i) \geq d(q,l_i) \end{cases}$$
$$\Rightarrow \begin{cases} d(q,x) \geq d(x,l_i) - d(q,l_i) \\ d(q,x) \geq d(q,l_i) - d(x,l_i) \end{cases}$$
$$\Rightarrow \quad d(q,x) \geq |d(x,l_i) - d(q,l_i)|$$

For any data object $s \in D$ that satisfies the query (q, r), we have $d(q, s) \leq r$. Hence $|d(s, l_i) - d(q, l_i)| \leq r$ holds. We only need to search the k-hypercube centered at $(d(q, l_1), d(q, l_2), \ldots, d(q, l_k))$ with edge size$2r$ in the k-dimensional index space to solve the query. Therefore the near neighbor querying in the original metric space is converted to the multi-dimensional range querying in the index space. Note that range queries in the index space will generate a superset result, which should be further refined to exclude the unsatisfied data objects.

Number of landmarks: The number of landmarks affects the tradeoff between querying quality and querying efficiency. If the amount of landmarks is too small, the index structure can not efficiently filter out the unrelated data objects to a query. The coarse results will increase the overhead of further refinement and waste network bandwidth when distributed processing is applied. Reversely, an excessively large number of landmarks will result in high dimensionality of the index space. Previous studies [5, 3, 23] have shown that complex queries in the high dimensional space have low efficiency.

Landmark selection: The landmark selection impacts the quality of querying results. A good landmark selection method should choose landmark points randomly in order to make them be close to the center of data clusters in the original data space; Another important issue is to keep these landmark points dispersive in the original data space. If the points bunch up, the distance calculation from them could be less informative, so they could not efficiently model the index space to filter out the data objects. In our simulations, we use two different schemes to select landmarks: the greedy method and the k-mean clustering method. We assume that a well-known node in the system is assigned the task of selecting landmark set at the system initiation time, and the landmark set is then used by every node in the system. The well-known node starts the landmark selection by randomly sampling a set of data objects S in the network, then uses the greedy method to pick up data objects from S to form the landmark set, or clusters the sampled dataset S and uses the cluster centroids as landmarks. The new joined nodes can simply obtain the landmark set from any nodes currently in the system. Algorithm 1 briefly describes the greedy method. Due to space limitation, we omit the description of k-mean clustering method here. Interested readers can refer to relevant documents for more details.

Algorithm 1 $GreedySelection$ ( )

```
1: $S \leftarrow$ randomly sample data objects in the network
2: $L \leftarrow \{\}$
3: Randomly pick up an object from $S$ and move it to $L$
4: while $size(L) <$ desired number of landmarks do
5:     Choose an object from $S$ which has the maximum distance to $L$ and move it
       to $L$. (The distance between an object s to a set L is defined as the minimum
       distance between s and all elements in L)
6: end while
7: return $L$
```

Boundary of index space: The boundary of the index space is required when partitioning and mapping the index space onto nodes in the overlay network (discussed in section 3.2). We provide two approaches to determine the boundary: (1) by the original metric space. Bounded metrics can be used directly, while unbounded metrics can be adjusted using the formula: $d' = \frac{d}{1+d}$. (2) by the landmark selection procedure. The minimum and maximum distance between the landmark set and the initially sampled set can be used as the boundary of the index space. The data objects whose distance to the landmarks goes beyond the boundary will be mapped to the boundary points in the index space.

3.2. Locality-preserving Hashing

For distributed index storage and query processing, the whole index space needs to be partitioned and mapped onto nodes in the network. To facilitate range queries, the data locality should be preserved. We propose a locality-preserving hashing mechanism to partition the multi-dimensional index space and to map the nearby data points in the index space to one node or nodes close together in the overlay network. The mechanism is based on the technique of k-d tree [2]. The whole index space is partitioned into 2^m equally sized hypercuboids (m is the number of bits in the key identifiers of Chord), each of which is identified by a *key* (a m bits integer).

Consider a k-dimensional index space $I[0..k-1]$, where each dimension is bounded by a pair $\langle L, H \rangle$. The k-dimensional cuboids are obtained by dividing each dimension alternately, for totally m times. The procedure satisfies the following two properties:

- After the i-th division, $1 \le i \le m$, $I[0..k-1]$ is partitioned into 2^i equal sized k-dimensional cuboids;
- The i-th division is performed on the j-th dimension, where $j = (i-1) \bmod k$.

The key is defined as follows: on the i-th division, if a hypercuboid picks up the *higher half* of the splitting range, the i-th[3] bit of its key is 1, else 0. Algorithm 2 gives a detail description of the locality-preserving hash function. Given a point in the index space, the locality-preserving hash function identifies the hypercuboid that holds the point and returns the key.

Algorithm 2 $LocalityPreservingHash$ ($ipoint[0..k-1]$)

```
Require: $k$ : dimensionality of the index space
Require: $\langle L, H \rangle$ : boundary of each dimension
Require: $m$ : number of bits in the identifier
1: $key \leftarrow 0$
2: for $i \leftarrow 0$ to $k-1$ do
3:     $R[i] \leftarrow \langle L, H \rangle$
4: end for
5: for $i \leftarrow 1$ to $m$ do
6:     $j \leftarrow (i-1) \bmod d$
7:     $mid \leftarrow (R[j].l + S[j].h)/2$
8:     if $ipoint[j] > mid$ then
9:         $R[j].l \leftarrow mid$
10:        $key \leftarrow (key \ll 1) + 1$
11:    else
12:        $R[j].h \leftarrow mid$
13:        $key \leftarrow (key \ll 1)$
14:    end if
15: end for
16: return $key$
```

The Chord's key-mapping mechanism is utilized to map the hypercuboids onto the nodes in the network, i.e. each hypercuboid is mapped onto the node which is the *successor*

[3]The i-th bit is the one in the i-th position (from the left) of the m bits identifier (padded with zeros on the left if the length is less than m)

(node whose identifier is equal to or immediately after the key along the ring) of its key.

The above hashing and mapping mechanism can achieve data locality since nearby points in the multi-dimensional index space are hashed and mapped to the same node or the neighboring nodes in the overlay network. Since nodes are evenly distributed in the identifier space (Chord uses consistent hashing, e.g. SHA-1, to map nodes to the identifier space), the hypercuboids are evenly mapped to nodes in the overlay network. However, load may not evenly distributed among nodes due to the skew distribution of index entries in the index space. In section 3.4, we propose load balancing mechanisms to efficiently balance the load among nodes and make sure that no node is unduly loaded.

3.3. Range Query Resolving and Routing

As discussed in section 3.1, the near-neighbor queries in the metric space can be converted to range queries in the index space. Since index space is partitioned and distributed among nodes in the system, an efficient query routing algorithm is necessary to refine and deliver the range queries to the corresponding index nodes. A naive approach is to subdivide a range query into many subqueries, each of which is covered by only one of the 2^m hypercuboids, and to route each subquery to the corresponding index node. This method is obviously inefficient and will cause high overhead especially when the *query selectivity* (ratio of the query size to the domain size) is large. Accordingly, we propose a different approach to solve range queries by progressively splitting and refining a range query along the propagation path.

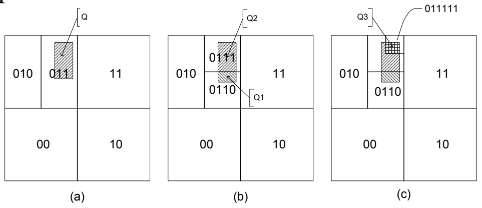

Figure 1. Range query refinement and the corresponding prefix

We define *prefix_key* and *prefix_length* to assist the query resolving and routing. The *prefix* is the code (bit string) of the smallest hypercuboid that can completely hold the query region when the data space is recursively partitioned. The *prefix_key* is a m-bit identifier by padding zeros to the right of *prefix*. The *prefix_length* is used to indicate the valid length of *prefix* in the *prefix_key*. As illustrated in figure 1(a), the rectangle 011 is the smallest one that can completely hold the query region (the shaded area) of Q when the data space is recursively partitioned for three times. Thus the prefix for Q is "011". The prefix_key of Q is "$0110\ldots0$", with m bits totally. When a range query is issued, the initial prefix_key is generated on the querying node. Then the query, as well as the prefix_key, is sent to the query routing module for refinement and delivery.

Upon receiving a range query Q, a node A (where A is any node on the propagation path, including the querying node) exploits the embedded tree (formed by the DHT links) to divide the query into multiple smaller sized subqueries and deliver these subqueries along the DHT links. The basic idea behind the query refinement and routing algorithms is that subqueries sent to the corresponding indexing nodes who share common ancestor nodes on the embedded tree are delivered as one larger sized query from the root node to their lowest common ancestor node. In other words, a query splits into multiple subqueries only when these subqueries need to take different ways to the destination on the distributed embedded tree.

Algorithm 3 $QueryRouting$ (Query q)

```
Require: $m$ : number of bits in the identifier
1: $p \leftarrow q.prelen$
2: if $q.prelen = m$ then
3:    $subquerylist$.push_back($q$)
4: else
5:    $subqueries \leftarrow$ QuerySplit($q, q.prelen + 1$)
6:    $N_1 \leftarrow$ nexthop($subqueries[0].prekey$);
7:    $N_2 \leftarrow$ nexthop($subqueries[1].prekey$);
8:    if $N_1 = N_2$ then
9:       $subquerylist$.push_back($q$)
10:   else
11:      $subquerylist$.insert($subqueries$)
12:   end if
13: end if
14: for each $sq$ in $subquerylist$ do
15:    $N \leftarrow$ nexthop($sq.prekey$)
16:    if $N = me$ then
17:       $Successor$.SurrogateRefine($sq$)
18:    else
19:       $N$.QueryRouting($sq$)
20:    end if
21: end for
```

Algorithms 3, 4 and 5 outline the procedure of range query refining and routing. The notation $N.foo()$ in the pseudocode stands for the function $foo()$ being invoked at and executed on node N. The procedure for query refinement and delivery is a distributed recursive process that starts from the querying node which initializes the query and then locally invokes $ETBRouting()$ for query routing. Once receiving a query Q, $QueryRouting()$ first invokes $QuerySplit()$ to divide Q into small sized subqueries based on the routing table. The query splitting procedure ensures that no subqueries share common $next\_hop$ [4] nodes when delivered along DHT links. Figure 1(b) gives an example of query splitting on a 2-dimensional data space. Query Q is divided into two subqueries by horizontally partitioning rectangle 011 in half, and the prefix for the subqueries are "0110" and "0111" respectively. After the query is divided into subqueries, the next step is to deliver these subqueries along the DHT links by remotely invoking the

[4]The *next_hop* node is the one from the routing table whose identifier is immediately before the *prefix_key* of the query on the ring. In the implementation of Chord, the routing table is composed of a finger table, a successor list and the current node itself.

$QueryRouting()$ function on the next_hop nodes in parallel. If the *next_hop* is the current node (current node is the *predecessor* of the prefix_key), the subquery is sent to the surrogate node (the *successor* of the current node) for refinement by remotely invoking procedure $SurrogateRefine()$ on the successor node.

Algorithm 4 $QuerySplit$ (Query q, Pos p)

```
Require: $k$ : dimensionality
Require: $\langle L, H \rangle$ : boundary of each dimension
Require: $m$ : number of bits in the identifier
 1: $j \leftarrow (p-1) \bmod k$
 2: $R \leftarrow \langle L, H \rangle$
 3: $i \leftarrow (p \bmod d = 0)\ ?\ k : (p \bmod k)$
 4: while $i < p$ do
 5:   if getbit($q.prekey$, $i$) = 1 then
 6:     $R.l \leftarrow (R.h + R.l)/2$
 7:   else
 8:     $R.h \leftarrow (R.h + R.l)/2$
 9:   end if
10:   $i \leftarrow i + d$
11: end while
12: $mid \leftarrow (R.h + R.l)/2$
13: if $q.range[j].l > mid$ then
14:   setbit($q.prekey$, $p$)
15:   $q.prelen \leftarrow p$
16:   $subquerylist$.push_back($q$)
17: else if $q.range[j].h < mid$ then
18:   $q.prelen \leftarrow p$
19:   $subquerylist.push\_back(q)$
20: else
21:   $nq_1 \leftarrow nq_2 \leftarrow q$
22:   $nq_1.range[j].l \leftarrow nq_2.range[j].h \leftarrow mid$
23:   setbit($nq_1.prekey$, $p$)
24:   $nq_1.prelen \leftarrow nq_2.prelen \leftarrow p$
25:   $subquerylist$.push_back($nq_1, nq_2$)
26: end if
27: return $subquerylist$
```

The surrogate node C refines a query q based on the overlapping relation between the data space that C covers and the range of q. If the range of q completely falls into the range of C, C will fully accept query q; If there is no overlapping between them, C will forward q after generating a refined prefix_key; If there is overlapping between them, C will divide q into multiple *subqueries* and forward the subqueries which are not covered by the current node. The surrogate refinement is a recursive process which progressively prunes the query range to fit the data space region covered by the surrogate node. Figure 1(c) gives an example of query refinement. Subquery Q_3 is cutted out from Q_2, and the remainder of Q_2 is completely covered by the surrogate node C. Node C then replies to the remainder of Q_2 with index entries stored locally and sends out subquery Q_3 using the procedure outlined above.

The range query resolving and routing algorithm is essentially a recursive process where queries are progressively refined and delivered on the embedded trees formed by the DHT links. Thus the overall number of messages needed to resolve and route a query can be significantly reduced. Through exploring DHT links, the overhead of maintaining additional in-network structures specific to different index schemes can be eliminated, therefore the proposed index architecture can inherently support many index schemes. In addition, the maintenance messages for the DHT links can be piggybacked onto the query delivery messages, so as to reduce the maintenance cost.

Algorithm 5 $SurrogateRefine$ (Query q)

```
Require: $m$ : number of bits in the identifier
Require: $me.id$ : the identifier of current node
 1: if prefix($q.prekey$, $q.prelen$) $\neq$ prefix($me.id$, $q.prelen$) then
 2:   surrogate node fully covers q
 3:   solve $q$ locally and return results to querier
 4: else
 5:   $j \leftarrow$ first 0 bit position in me.id from q.prelen+1 to $m$
 6:   if $j$ not exists then
 7:     surrogate node fully covers q
 8:     solve $q$ locally and return results to querier
 9:   else
10:     $q.prekey \leftarrow$ prefix($me.id, j-1$)
11:     $q.prelen \leftarrow j - 1$
12:     $subquerylist \leftarrow$ QuerySplit($q, j$)
13:     for each $sq$ in $subquerielist$ do
14:       if prefix(sq.prekey,sq.prelen)=prefix(me.id,sq.prelen) then
15:         SurgateRefine($sq$)
16:       else
17:         QueryRouting($sq$)
18:       end if
19:     end for
20:   end if
21: end if
```

3.4. Load Balancing

An important issue in the distributed system is load balancing. In this section, we propose static and dynamic load balancing mechanisms to adjust load among nodes and ensure that no node in the system is unduly loaded. In this paper, we measure the load on a node by the amount of index entries stored on it (the overhead of index storage and computation cost of query evaluation); however, other information, such as the number of messages, can be easily incorporated into the load value.

Space mapping rotation: Recall that our index architecture can simultaneously support multiple index schemes. For each index scheme, the locality-preserving hashing mechanism partitions and maps the multi-dimensional index space to a 1-d key space ranged $[0..2^m - 1]$. If several index schemes have similar distribution of the *hotspots* in the index space, the hot region for each index will be mapped to the common ranges in the 1-d key space, therefore nodes with identifiers located in these ranges will be overloaded. For example, in the high dimensional vector metric space, the volume of hyperball centered at the landmark point increases quickly when the radius becomes large, given formulas $V_n = \frac{\pi^{n/2} R^n}{\Gamma(1+\frac{n}{2})}$ and $\frac{dV_n}{dR} = \frac{n\pi^{n/2} R^{n-1}}{\Gamma(1+\frac{n}{2})}$. Thus the index entries will be densely distributed to the area close to the upper boundary of the index space. Therefore, a few of such index schemes will overload the nodes located in the higher range of the identifier space.

If each index scheme is given a random rotation offset ϕ when mapped to the 1-d key space, index i will be mapped to $[\phi_i .. \phi_i + 2^m - 1]$ (arithmetic is modulo 2^m), and the hot regions of these index schemes will be mapped to different ranges on the Chord ring. The randomness of ϕ for each index scheme can be achieved by hashing (*random hashing function*) the name of the corresponding index. The locality-preserving hashing and query delivery algorithms presented in the previous sections can be easily modified to reflect the space rotation.

Dynamic load migration: At runtime, heavily loaded nodes can dynamically migrate some of their load to lightly loaded ones in two ways: first, when a new node joins the system, the join request is forwarded toward a heavily loaded node, which will divide its key range and assign one half to the new node. The indices with keys located in the corresponding range are transferred to the new node. The split point should be chosen carefully to ensure load on each node is almost even. Second, a node, called N, periodically samples the load on its neighbors (and neighbors' neighbors if the probing level P_l is greater than 1). Node N is said to be heavily loaded if its load oversteps the average load on the neighbors by a threshold factor δ_N, that is $L_N > \overline{L} \times (1 + \delta_N)$. The value of the threshold factor δ for each node is based on the node's capacity. The average value of δ and P_l control the tradeoff between the overhead and quality of the load balancing. A heavily load node can find a lightly loaded node and ask it to leave and then rejoin the system with a given node identifier (the split point of the key range to divide the load in halves).

To facilitate load probing, each node in the system keeps its neighbors' load information in the routing table. And the load information refresh message can be piggybacked onto the routing table maintenance message, which can be further piggybacked onto the query delivery messages as discussed in section 3.3 to reduce the maintenance cost.

It should be noted that this load balancing approach may cause nodes not uniformly distributed in the identifiers space, which will more or less impair the performance of the query routing algorithm. This is because uniformly distributed nodeIDs will increase the depth of the search tree, thus the query propagation path will also increase and the concurrent degree of subqueries will be low. However, the tradeoff between the quality of load balancing and query routing performance can be controlled by the threshold factor δ and the probing level parameter P_l.

4. Experimental Evaluation

In this section, we evaluate performance of the proposed design through simulations. We start our discussion by describing the experimental setup and metrics used for evaluation. Afterwards, the experimental results are presented and discussed.

4.1. Experimental Setup

We implement our index architecture on top of **p2psim**[8], a discrete event-driven, packet level simulator for many DHT protocols. We use Chord-PNS (*Chord with proximity neighbor selection [9] allows each node to choose physical closest nodes from the valid candidates as routing entries, thus to reduce the lookup latency.*) protocol with its default parameters (*base=2, successors=16* etc.). The number of bits in the key/node identifiers in the simulator is 64. The network model used in our simulation is derived from the King dateset, which includes the pairwise latencies of 1740 DNS servers in the Internet measured by King method [12]. The average round-trip time of the simulated network is 180 milliseconds. We use both synthetic datasets and real world datasets for experiments.

The simulations are initialized with one node in the system, which randomly samples data objects and performs the landmark selection procedure. Then other nodes join the system at a randomly chosen time. After system stabilization, we schedule 2000 queries issued on the randomly chosen nodes. The interarrival time of queries is exponentially distributed with average value of $150s$.

A set of cost metrics are used to evaluate the performance of our index architecture: (1) *hops*: the maximum path length required to deliver a query to all of the corresponding index nodes; (2) *response time*: the elapsed time between injecting a query into the system and receiving the first query result; (3) *maximum latency*: the elapsed time between injecting a query into the system and receiving responses from all of the corresponding index nodes; (4) *bandwidth cost*: the total bandwidth consumption for delivering a query to the corresponding index nodes (query delivery bandwidth) and delivering the query results from the index nodes to the querying node (results delivery bandwidth); (5) *recall*: a metric used to quantify the quality of query result. For each query, the k-nearest data objects obtained by searching the whole dataset, set X, are considered as the theoretical results. Then we use our system to retrieve k-nearest data objects, set set Y, the recall rate for a query is calculated as $Recall = \frac{|X \cap Y|}{|X|}$. We set $k = 10$, which is a reasonable value since most users only have interest in the top 10 results [21]. Each queried index node returns the 10-nearest local results and the querying node merges these results to calculate the recall rate.

The size of each query message is modeled in bytes as: $20 + 4 + n \times (2 \times 2 \times k + 8 + 1)$, where 20 bytes are for packet header, 4 bytes are for IP address of source node, n is number of subqueries, k is the number of landmarks, 8 bytes are for prefix key, and 1 byte is for prefix_length. The size of the result message is modeled as 20 bytes for packet header and 6 bytes for each index entry in the result.

4.2. Experimental Results with Synthetic Datasets

We generate multi-dimensional datasets. Each dataset contains 10^5 data objects which are clustered in the data space. Data in each data cluster are modeled as normal distribution. Thus less number of clusters and less deviation

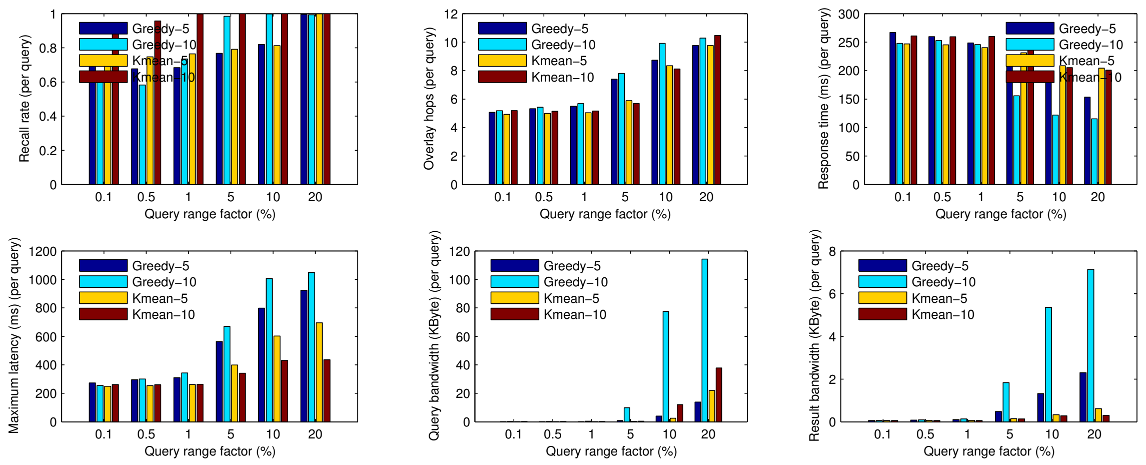

Figure 2. Performance with respect to different landmark selection schemes (without load balancing)

in each cluster will generate more skewed dataset. The corresponding query sets are generated with the same method. The values of the parameters for generating the datasets are list in table 1. Distance between two data points is measured using the euclidean metric. Given two data points x and y in a d-dimensional data space, the distance between x and y is computed as follows: $d(x, y) = \sqrt[2]{\sum_{i=1}^{n}(x_i - y_i)^2}$.

Table 1. Parameters for Datasets Generation

Dimension	100
Range of each dimension	[0 .. 100]
Number of clusters	10
Deviation of each cluster	20

The landmarks are chosen with greedy method or k-mean clustering method by randomly sampling 2000 data objects from the dataset. We define *query range factor* as the size of query range divided by the maximum theoretical distance between two data points in the original data space. The theoretical maximum distance is $\sqrt[2]{\sum_{i=1}^{100}(100 - 0)^2} = 1000$. The boundary of index space is determined by the original metric space, each dimension bounded by $[0..1000]$. We evaluate the performance by ranging the query range factor from 0.1% to 20%.

Figure 2 illustrates the performance without applying load balancing mechanism. Almost all landmark selection schemes can achieve high recall rate with low cost. *K-mean-10* and *Greedy-10* can achieve 100% recall rate when the query range factor is about 5%. The 10-landmark schemes outperform the 5-landmark schemes because the data in the dataset are distributed in 10 clusters, thus the indices will be correspondingly clustered in the index space and index entries will be distributed onto a small amount of nodes during partitioning and mapping. Therefore the recall rate is high, with a low query routing cost. The k-mean clustering schemes outperform greedy schemes in that the k-mean method uses the centroids of the data clusters as landmarks, so it can efficiently model the index space and filter the data objects.

Next we evaluate the performance when the load balancing mechanism (dynamic load migration) applies. Assume that all nodes in the system have same capacity and each node uses a threshold factor $\delta = 0$ and a large probing level parameter $P_l = 4$. These values are set to evaluate the maximum effect of load balancing on the performance of query routing. As illustrated in figure 2 and 3, for all landmark selection schemes, the recall rate decreases and the cost of query routing increases when load balancing is applied. However, a high recall rate, although affected by load balancing, can still be achieved with a reasonable routing cost. The 5-landmark schemes outperform the 10-landmark schemes. This is because the 5-landmark schemes can distribute the index entries more evenly onto nodes, which are less impacted by the load balancing mechanism. Figure 4 shows the load distribution on nodes (The nodes are sorted in the decreasing order of the load). We can see that the load balancing mechanism can achieve an even load distribution among nodes. And the maximumly loaded node only has 97 index entries for all of the landmark selection schemes.

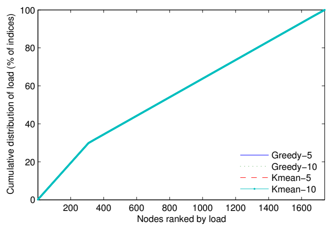

Figure 4. Load distribution on nodes

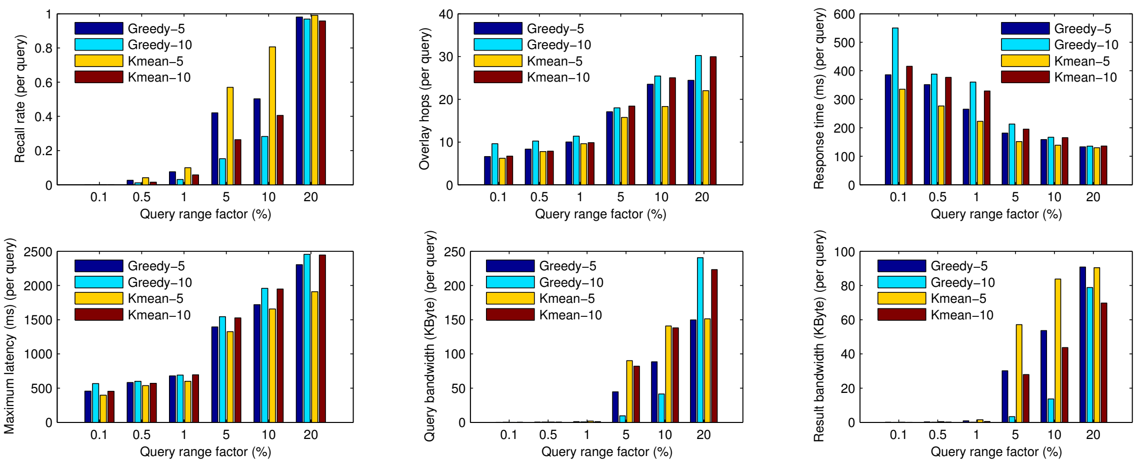

Figure 3. Performance with respect to different landmark selection schemes (with load balancing)

4.3. Experimental Results with Real World Datasets

We evaluate our design with *TREC-1,2-AP* datasets, which contains AP Newswire documents in TREC CDs 1 and 2. We extracted the documents with the *head* and *text* fields and excluded $7,576$ documents that do not have a valid *head* field. This results in $157,021$ documents in the final dataset. The queries are obtained from TREC-3 ad hoc topics (151-200). We use 2000 queries in the simulation by repeating these 50 topics on randomly selected nodes. Landmarks are chosen with greedy or k-mean clustering method by randomly sampling 3000 documents from the dataset.

We use VSM [4] to represent documents and queries as term vectors. Each component of the vector represents the importance of the corresponding term in the document or query. We use TF/IDF scheme to calculate the weight of components, where TF is the frequency of terms in a document and IDF is the inverse of the number documents in which the term appears. We also use a list of 571 stop words from SMART [6] to remove the stop words from the document vectors. Thus each document vector has 155 terms on average and the whole document set has 233640 distinct terms. Therefore the whole document set can be represented as a 233640×157021 sparse matrix. The distribution of the document vector size is list in table 2.

Table 2. The Distribution of Doc Vector Sizes

minimum	5th	50th	95th	maximum	mean
1	50	146	293	676	155.4

The distance between two vectors is measured as the angle between them (based on the well known *cosine similarity* measure). Given vectors X and Y, the distance between them can be calculated as:

$$d(X,Y) = arccos\left(\frac{X \cdot Y}{|X||Y|}\right)$$

We study two landmark selection schemes in this experiment: k-mean-10 and Greedy-10. The boundary of the index space is determined by the landmark selection procedure. As illustrated in figure 5, when the query range factor is less than 1%, the greedy method achieves higher recall rate with lower query routing cost. When the query range factor increases from 1% to 20%, the k-mean clustering method achieves high recall rate with lower query routing cost. As we have discussed before, the whole document set has been represented as a high dimensional sparse matrix, and each document vector has a very small number of terms (the maximum size of document vector is 676, the total number of distinct terms in the whole dataset is 233640), thus given a document vector v, there are a large amount of vectors which have maximum distance $(\pi/2)$ to v. Since the greedy method choose landmarks directly from the document set, the landmark vectors will also have less number of terms. Therefore a large number of unrelated documents will be mapped to the same point close to the upper boundary of the index space. In other words, landmarks chosen by greedy method can not effectively model the index space to filter the documents. On the contrary, k-mean clustering method efficiently groups the sampled documents and uses the centroids of the clusters as landmark vectors, so each landmark vector has more terms and can be used to measure the documents effectively.

The greedy method can not efficiently filter the documents. It maps a large amount of unrelated documents to the same point in the index space. As a result, the locality-preserving hash function will hash these documents to a single key. The load balancing mechanism can not divide the

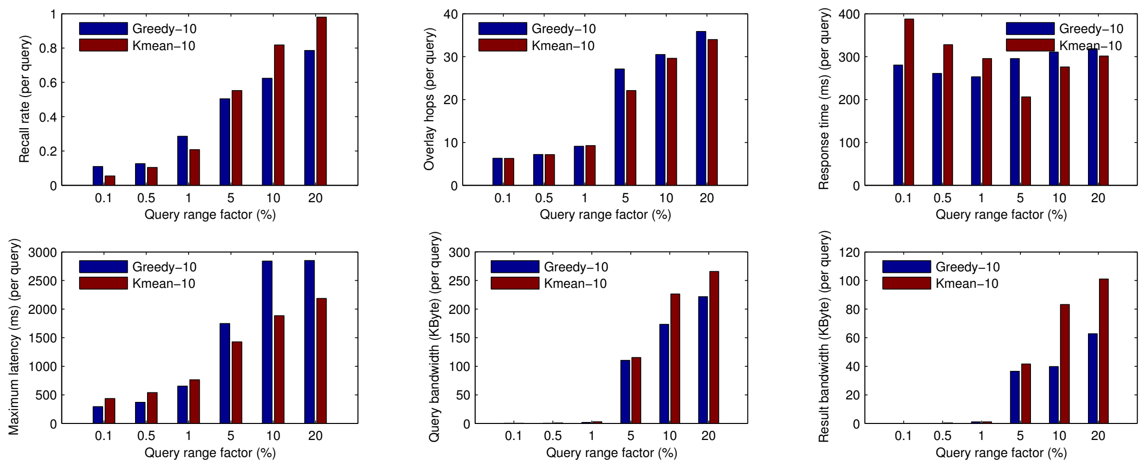

Figure 5. Performance with TREC dataset (with load balancing)

index entries associated with a single key. Therefore, the indices are still stored on a small amount of nodes even with load balancing, as shown in figure 6.

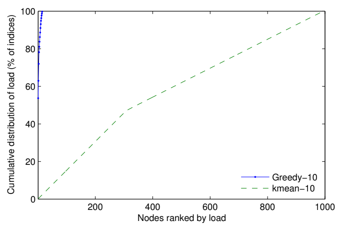

Figure 6. Load distribution (TREC dataset)

Since the queries have 3.5 unique terms averagely, the greedy method will map them to the points close to the upper boundary of the index space with high probability. Therefore, the actual query range is less than the one depicted in figure 5. For example, given range r, the query range for a query q is $[I_q - r, \text{upper\_boundary}]$, instead of $[I_q - r, I_q + r]$, where I_q is correspond index point of q. This is the reason why the query routing cost of the greedy method is less than that of *k*-mean clustering method when the query range is relatively small. Since the greedy method only distributes indices onto a small amount of nodes, the load balancing mechanism may cause a skew distribution of nodes in the identifier space, which will increase the query routing cost, as discussed in section 3.4.

Similarly, since the greedy method maps documents and queries to a less number of nodes, it can achieve a high recall rate when the query range factor is relatively small. However, the overhead of query processing on the corresponding index nodes is significantly high. When the query range factor is large, the greedy method can not effectively retrieve other related documents due to its poor performance in filtering documents.

5. Related Work

Recently there has evolved considerable work for supporting complex queries in structured P2P networks. Gupta, Agrawal *et.al* [13] attempt to hash ranges instead of keywords to nodes. They use locality sensitive hashing to ensure that similar ranges are mapped to the same node with high probability. MAAN [7] uses locality preserving hashing to support multi-dimensional range queries on Chord. SCRAP [11] uses Hilbert Space Filling Curve [18] to map multi-dimensional data space to a one-dimension key space. Their query routing is based on Skip Graph [1]. MURK [11] and SkipIndex [24] use k-d trees to partition and map the multi-dimensional space to nodes, for supporting multi-dimensional range queries. Our index architecture differs from above solutions in that it can efficiently support multiple multi-dimensional indices with different dimensionality, without maintaining multiple individual routing structures.

There has been other work aiming to extend the DHTs to support similarity search. pSearch [22] is built on top of CAN and leverages LSI for indexing the documents. As mentioned in [22], pSearch is less efficient as the size of the corpus increases. Sahin *et.al* [19] use reference set to map documents onto the nodes in the Chord overlay. Since data locality is not preserved, their system suffers from low recall rate and high query overhead. In order to achieve high recall rate, they use multiple reference sets, which will increase the cost of storing indices and maintaining the reference sets. MCAN [10] is built on top of CAN for supporting similarity queries in the metric space. The main drawback of MCAN is that the same querying message may be send to one node for several times.

6. Conclusion and Future Work

In this paper, we have proposed and evaluated the design of a distributed index architecture, which is built on top of DHT for efficiently resolving similarity queries in peer-to-peer networks. By choosing a set of data points as landmarks, we map a general metric space to a multi-dimensional index space and convert the near-neighbor queries in the original metric space to range queries in the index space. We have proposed a locality-preserving hashing mechanism to partition and map the index space onto the nodes in the overlay network; and an efficient query routing algorithm to progressively refine and deliver the range queries to the corresponding index nodes. There are two distinct features in our design: (1) any type of datasets with a corresponding distance function can be indexed on our indexing platform; (2) multiple index schemes can be simultaneously supported without maintaining multiple routing structures. We have also developed light-weighted load-balancing mechanisms to adjust the load among nodes to make sure that no node in the system is unduly loaded.

This paper constitutes an initial step to build an efficient and distributed platform for supporting similarity queries in general metric space. There is plenty of future work to do. One is to enable the execution of other metric spaces. Detailed evaluations will be performed on the space mapping, query routing and load balancing mechanisms, based on which more optimizations may be proposed. Another future work is to support *automatic query expansion* [15], which is already an effective technique to improve recall and precision in centralized information retrieval systems. The third direction is to support dynamic datasets. New landmark sets can be periodically generated and evaluated. If the new landmark set outperforms the current one according to some threshold, the new landmarks will be disseminated to the nodes in the system. Indices will be recalculated and migrated to new nodes accordingly.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Jan 2003.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] S. Berchtold, C. Böhm, D. A. Keim, and H. P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 78–86. ACM Press, 1997.

[4] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 1999.

[5] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 574–584, Zurich, Switzerland, Sep 1995.

[6] C. Buckley. Implementation of the smart information retrieval system. Technical Report TR85-686, Department of Computer Science, Cornell University, May 1985.

[7] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. Maan: A multi-attribute addressable network for grid information services. In *Fourth International Workshop on Grid Computing*, pages 184–191, Phoenix, AZ, Nov. 2003.

[8] Computer Science and Artificial Intelligence Lab, MIT. p2psim: a simulator for peer-to-peer protocols. http://pdos.csail.mit.edu/p2psim.

[9] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceeding of the First Symposium on Networked Systems Design and Implementation (NSDI)*, pages 85–98, San Francisco, CA, Mar. 2004.

[10] F. Falchi, C. Gennaro, and P. Zezula. A content-addressable network for similarity search in metric spaces. In *3rd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, pages 126–137, Trondheim, Norway, 2005.

[11] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in p2p systems. In *Proceedings of the Seventh International Workshop on the Web and Databases*, pages 19–24, Maison De la Chimie, Paris, France, Jun. 2004.

[12] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[13] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2003.

[14] D. Huttenlocher, G. Klanderman, and W. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[15] M. Mitra, A. Singhal, and C. Buckley. Improving automatic query expansion. In *Proceedings of ACM SIGIR*, pages 206–214, Melbourne, Australia, 1998.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172, San Diego, CA, Aug. 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[18] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[19] O. D. Sahin, F. Emekçi, D. Agrawal, and A. E. Abbadi. Content-based similarity search over peer-to-peer systems. In *2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, pages 61–78, Toronto,Canada, 2004.

[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug. 2001.

[21] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOMM*, pages 175–186, Karlsruhe, Germany, Aug. 2003.

[22] C. Tang, Z. Xu, and M. Mahalingam. psearch: information retrieval in structured overlays. *Computer Communication Review*, 33(1):89–94, 2003.

[23] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 194–205, New York, USA, Aug 1998.

[24] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, 2004.

[25] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U.C. Berkeley, Apr. 2001.