# Range and $k$NN Searching in P2P

Manesh Subhash Ni Yuan Sun Chong
*National University of Singapore*
*School of Computing*
{*maneshsu, niyuan, sunchong*}*@comp.nus.edu.sg*

## 1 Introduction

The tremendous growth in the popularity of P2P networks has resulted in increasing demands in performance, scalability and functionality. P2P networks are mostly popular for locating and publishing information on the internet. This requires extensive and powerful querying techniques. The need to support not only exact queries, but range and similarity queries have brought about the need for similarity measures and metrics. This need has also extended itself to include nearest neighbor queries. For example, over a music file sharing network, one could try to locate a specific file, or try to locate files that match a range criteria or even try to find k files that are similar to a given file. In this report, we study some approaches that can handle range and kNN searching in peer-to-peer system.

Range queries can be divided into two class, i.e. single attribute range queries and multi-dimensional range queries. In our report we will first discuss P-tree algorithm, that supports single attribute range queries in P2P system. Then we will study the approaches which handle multi-dimensional range queries. These approaches can be classified into two types in terms of the routing network they adopt. The first type of approaches routs the queries in one dimension routing space, such as Chord [17] and Skip Graph [3]. ZNet [16], SCRAP [8], HSFC-based [14] are the approaches in this type. The second type of approaches routs the queries in multi-dimension routing space. MURK [8] is an approach in this type. In our report, we will study these approaches in detail and compare the advantages and disadvantages among them.

While most P2P networks like Gnutella [2], Freenet [1] support simple file name based matches, content based searches are also being researched upon. An example of this work is the semantic based content search in P2P networks [15]. This however introduces high dimensionality into the search scheme. With the limited bandwidth, complete content based searches are infeasible, thus summarized or keywords based searches are used. In this study we will also focus on the application of the GHT* distributed index to enable $k$NN searching for P2P networks as presented in the paper by Batko et al. [18]. We also introduce the approach pSearch [5], which is used for locating similar documents on the internet, in the kNN searching section, since the searching of similar documents can be considered as a kind of nearest neighbors.

In P2P networks, the search methodology also depends on the underlying architecture. One model uses the unstructured network like Gnutella [2], the searching models in these systems use the flooding technique. The flooding technique uses a finite number of hops known as 'time-to-live' (TTL) to restrict its search. The use of TTL however implies that only a fraction of the actual number of possible results are returned to the user. Worse, it might not be able to even locate files that do exist in the system. This scenario is typical if the file is a rare file and hence very few replicas exist in the P2P network. The flooding technique however is effective for popular files. Another alternative is the structured P2P network. In this model we are able achieve a system wide consensus on the location of a particular item. All lookups use keys that point a unique machine. Distributed hash tables (DHTs) have been developed for modeling these types of networks. Contrary to unstructured networks, these can locate all files in given system, including rare files. The effectiveness of flooding based techniques for popular

files and the ability of DHT based system to locate rare items have inspired the development of a hybrid model by Loo et al. [11] . We shall study this model.

The rest of the report is organized as follows. In section 2, we discuss the approaches for range query processing in P2P system. In section 3, we introduce some background for kNN searching in P2P system. Then in section 4, we discuss the approaches for kNN searching in P2P system. At the end of section 4, we also present an approach to locate rare items in P2P system as a complementary of the searching technique. The conclusion is given at section 5.

# 2 Range Query Processing in P2P

In this section, we discuss and compare different approaches to handle the range query in P2P systems. We classifies the discussion of approaches for range queries into two classes, i.e. approaches for one-dimensional range query and approaches for multi-dimensional range query.

## 2.1 Range queries over a single attribute

### 2.1.1 Range partitioning relations

The existing Distributed Hash Table (DHT) based systems such as Chord [17] makes use of *hash partitioning* to achieve load balance. However, the hash based partitioning is insufficient to support range queries. One way to handle range queries is to allow range partitioning in networks. One attribute is selected and range partitioned, and the tuples are allocated to the bucket in terms of the range they fall in. One well-known concern in range partitioning is *skew*. [7, 10] addresses the problem of load balance in range-partitioned peer-to-peer system.

### 2.1.2 P-tree index

The $B^+$-tree index is widely used for efficiently evaluating range queries in centralized database system. Crainiceanu et al [6] proposed an index called P-tree to support the range queries in peer-to-peer networks. P-tree is a set of semi-independent $B^+$-tree, which is maintained in each node. Before we present the semi-independent $B^+$-tree, we first introduce the fully independent $B^+$-tree. Suppose data items with index value 4, 8, 12, 20, 24, 25, 26, 35 are stored in peers $p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8$. Then each peer $p_i$ views the index values as being organized as a ring with $i$ as the smallest value, and builds a fully $B^+$-tree of all values in the network. The independent $B^+$-trees maintained in $p_1$ and $p_5$ are shown in **Fig** 1.

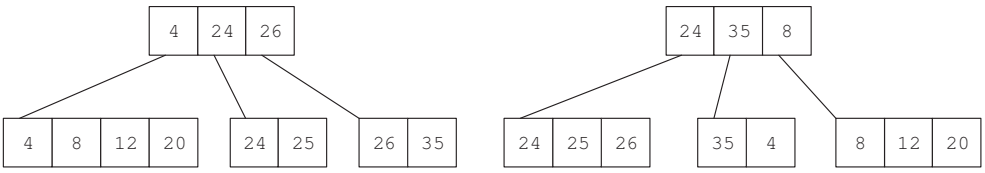


Figure 1: $B^+$-tree on peer $p_1$ and $p_5$

Since each peer maintains its independent $B^+$-tree, the consistency of each $B^+$-tree can be maintained in a fully distributed fashion. However, the fully independent $B^+$-tree also brings some problems. First, since each peer indexes all the values on the system, all peers have to be notified during each node insertion/deletion. Second, the space requirement at each peer is large.

The semi-$B^+$-tree, i.e P-tree is proposed to avoid the problem of fully independent $B^+$-tree. The key idea of semi-$B^+$-tree is that at each peer, only the *left-most root-to-leaf path* of the corresponding fully independent $B^+$-tree is stored. The pointers to other nodes which are not stored in the peer will be directed to the corresponding peers. Therefore, each peer relies on some other peers to complete

the fully $B^+$-tree. For example, $p_1$ only stores the root node containing the entry 4, 24, 26 and the left-most leaf node containing the entry 4, 8, 12, 20. The entry 24 in the root node will point to $p_5$ which stores the value 24 and the entry 26 will point to $p_7$ which stores the value 26. The set of semi-$B^+$-trees on each node forms the P-tree index. **Fig** 2 shows the P-tree structure of the eight peers aforementioned. Since the independent $B^+$-tree at each peer has two levels, we can see that for the P-tree index, each peer only stores two nodes.

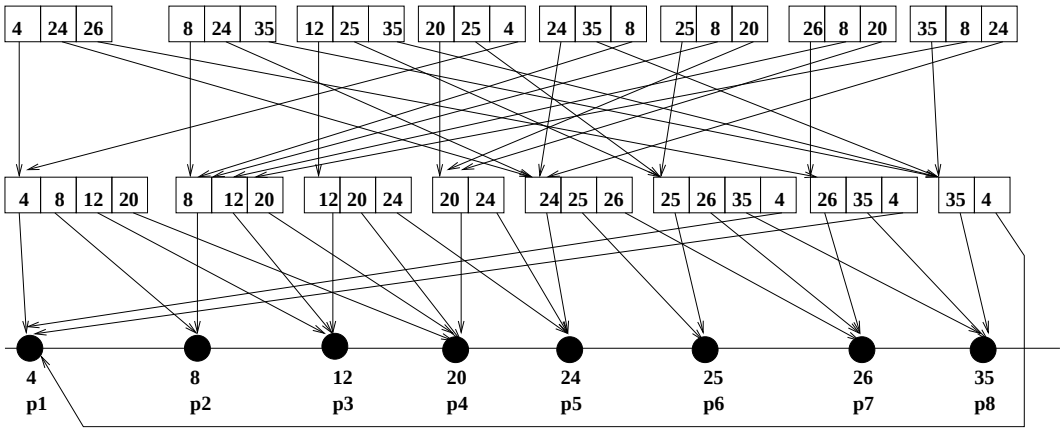


Figure 2: P-tree index structure for peers $p_1, p_2, \ldots, p_8$

Two requirements for the P-tree, which is not the concern for the $B^+$-tree in centralized environment, are *coverage* and *separation*. The *coverage* requires that all values are indexed by some node in the P-tree, i.e. no values are missed by the index. The *separation* requires that the overlap between adjacent peers is not excessive. In **Fig** 2, we can observe that some data values are indexed by more than one node, which is called the *overlap* between semi-$B^+$-trees. The excessive *overlap* between adjacent peers may make the height of the P-tree of order d no longer be guaranteed to be $O(\log_d N)$.

Since each peer only stores the left-most root-to-leaf path, the number of nodes stored in one peer equals the height of the $B^+$-tree, which is $O(\log_d N)$, where $N$ is the number of data items and $d$ is the order of the P-tree. Since each node has at most $2d$ entries (the same to $B^+$-tree), the total storage requirement per node is $O(d \cdot \log_d N)$ entries. The search cost for range query with P-tree is also similar with search cost for range query in $B^+$-tree, which is $O(m + \log_d N)$.

The algorithm to handle range query $q$ takes the lower bound $lb$ ,upper bound $up$, and the peer where the query is originated as input. The search algorithm for range query is similar with searching in $B^+$-tree. It first finds the peer which stores the first value which falls in the range, and then traverse the successors until the upper bound is exceeded. The searching of the first value is also similar with the point searching in $B^+$-tree, except that traversing from one level to the next level requires moving the searching from one peer to the other.

## 2.2 Range queries over multiple attributes

The approaches discussed in previous section support one dimensional range query, however in some applications, multi-dimensional range query will be needed. For example, the multimedia data always have multi-dimensions, then in a P2P system which shares multimedia data, a typical query would contain range predicates on multiple attributes. Therefore the supporting for multi-dimensional range queries is really necessary. In this section, we discuss the approaches for multi-dimensional range queries. The approaches can be divided into two classes, considering the types of network used to rout the queries. The first type of approaches routs the queries in one dimension routing space, such as Chord [17], Skip graph [3], therefore, the common task of these approaches is to mapping the multi-dimension data into one dimension space physical peers. The second type of approaches routs the

queries in multi-dimension routing space, such as CAN [12]. The most important problem for this type is efficiency routing.

There are three main requirements for the P2P systems that support multi-dimensional range queries:

- **Locality** : the data elements nearby in the data space should be stored in the same node or the close nodes.
- **Load balancing** : the amount of data stored by each node should roughly the same.
- **Routing efficiency** : the number of hops for evaluating a query should be small.

### 2.2.1 Routing in one dimensional space

This type of approaches arranges the physical peers into one dimension space. Therefore, the multi-dimensional data should be partitioned and allocated to one dimensional physical peers. There are two ways to achieve such mapping as from multi-dimensional data space to one dimensional routing space. The first way is to partition the data in the native data space, hence the data in multi-dimensional space is partitioned into subspace, which is called *zones*, then the *zones* are sorted according to some order into one dimensional space, finally the *zones* in the one dimensional space is mapped into one dimensional routing space. ZNet [16] is the approach in this way. The second way first maps multi-dimension data into one dimension using space filling curve, and then range partitions the data in one dimension, and finally the one dimensional data is allocated to peers. The example approaches in this way are SCRAP [8], HSFC-based [14]. In the following, we discuss these approaches in detail.

**ZNet** The ZNet partitions the data in the native data space in a way as in the generalized quad-tree. The space is halves in all dimensions for each partitioning, therefore, for a $d$ dimension space, $2^d$ subspaces will be produced for one partitioning, and the subspaces are called *zones* in ZNet. **Fig** 3 shows an example to partition a 2-dimension data space. As shown in **Fig** 3(2), the first partitioning partitions the data space in each dimension which generate $2^2$ subspaces. In **Fig** 3(3), the subspace with z-value 01 is further partitioned into four subspaces. Each *zone* assigned is considered at a certain level, the *zone* which is generated by the $i$th partition is called in level $i$.

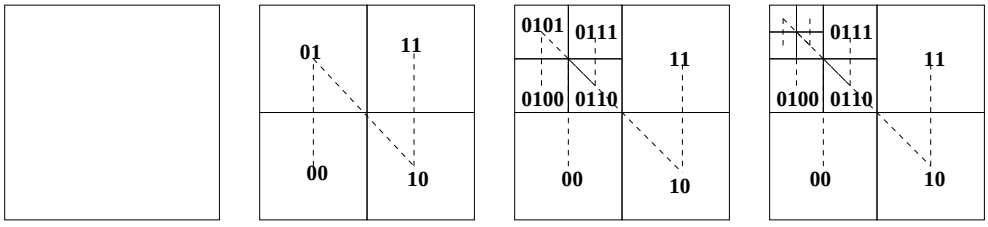


Figure 3: Data partition and z-ordering curve in ZNet

As aforementioned, ZNet makes use of the Skip graph which routs in 1-dimensional space, therefore multi-dimension data space should be mapped to 1-dimensional index space, such that it can be mapped to nodes in the network. ZNet utilizes the z-curve to arrange the *zones* in multi-dimension space into a sequence. A first order z-curve is used to fill zones in the same level. For a $d$ dimensional space, each zone at a certain level has a *z-value* of length $d$. The *z-value* of a zone $z_i$ in level $i$ is determined as follows. Suppose the centroid of the zone $z_{i-1}$ at level $i_1$ which covers zone $z_i$ is $(c_{i-1,0}, c_{i-1,1}, \ldots, c_{i-1,d-1})$ and the centroid of the zone $z_i$ is $(c_{i,0}, c_{i,1}, \ldots, c_{i,d-1})$, then the *z-value* for zone $z_i$ is $(b_0 b_1 b_2 \ldots b_{d-1})$, where $b_i = 0$ if $c_{i,k} < c_{i-1,k}$; otherwise $b_i = 1$. For example as shown in **Fig** 3, the data space is 2-dimension, thus the *z-value* is two-bit length. The *z-value* for the zone

in left-down corner of (2) is 00, since its values of centroid in both dimensions are smaller than the centroid of the zone in level 0.

As the definition of *z-value*, zones in different levels may have the same *z-value*. In order to sort the zones in a sequence, we need a unique identifier for each zone. *Z-address* is computed for each zone as the unique identifier. The *z-address* is computed in the following way: for each zone Z in level $l$, the *z-address* of this zone is like $z_1 z_2 \dots z_l$, where $z_i$ is the *z-value* of the zone in level $i$ which covers zone Z. The numbers in **Fig** 3 are the *z-address* for the corresponding zones.

The next step is to allocate the zones to physical peers. Suppose there are 8 peers in a P2P system, then the space is partitioned among 8 nodes in the following way : A(00), B(0100), C(010100), D(010101), E(010110), F(010111), G(0110, 0111), H(10, 11), which is shown in **Fig** 4.

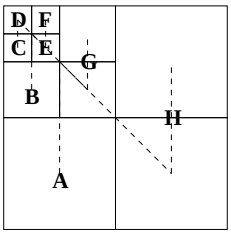


Figure 4: An example of allocating zones to physical peers

ZNet makes use of Skip graph [3] to rout queries. The Skip graph consists of multiple linked lists at multiple skip-levels. There are ⌈ logN ⌉ skip levels for an N-nodes network. The bottom skip-level is a double linked list consisting of all nodes in increasing order by key. Each node will be assigned a membership vector randomly, and the membership vector determines which lists a node belongs to. Every list at skip level $i$ has the identifier $L_w$, with the length of $w$ is $i$, and $w$ is the prefix of membership vectors of all nodes in the list. In other words, a node appears in list $L_w$ at skip-level $i$, if and only is $w$ is a prefix of its membership vector. The Skip graph formed by the 8 peers in our previous example is shown in **Fig** 5. Each node has two neighbors in each list, all neighbors of a node consist the routing table for the node. Given a searching key $k$, the node will first search its neighbors at the highest level. If there is a neighbor whose key is smaller than $k$, the query will be forwarded to that node; otherwise, the searching will go down one level.

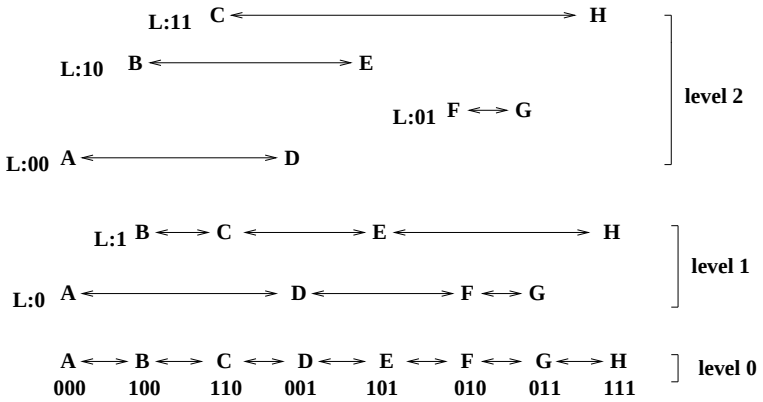


Figure 5: The Skip graph formed by 8 peers

To process the range query in ZNet, each node will be assigned a level in terms of the z-level of the zones which are allocated on it. For example in **Fig** 4 the node C is on level 3 since zone C is at z-level 3. Besides its own space, each node also has the knowledge about the spaces which cover its own space in each level. The spaces cover node C in each level is shown in **Fig** 6. For a $d$-dimensional space, a

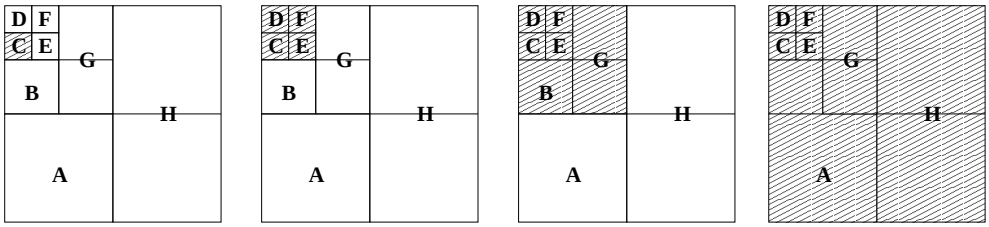


Figure 6: The levels for spaces

range query $q$ is in the format of $([l_0, u_0], [l_1, u_1], \ldots, [l_{d-1}, u_{d-1}])$. The first step is to determine the routing z-level $l$, whose space covers the range query. For example if the destination of range query is $C$ and $E$, then the space covering this range is on level 2, therefore the routing z-level is level 2. The next step is that the node computes the lowest and highest z-address of the range query $q$ according to z-level $l$. The lowest z-address is the point $(l_0, l_1, \ldots, l_{d-1})$, and the highest z-address is the point $(u_0, u_1, \ldots u_{d-1})$. Then the query is routed at level $l$ to find the nodes whose $l$ level space overlaps with query $q$. In this procedure, if a node $n$ whose $l$ level space is not overlap with the range, the neighbor of the node which is closer to the lowest bound is selected as the next node on which the query is evaluated; otherwise if node $n$ is at level $l$, it will notify the query initiator, if not, the query will go down one level to perform the further searching in the same way.

As mentioned above, another requirement for multi-dimensional range queries is load balancing. The load balancing in ZNet considers two aspects: static data distribution and dynamic data distribution. For the static data distribution, the load balancing is performed when a new node joins. The new joining node will choose appropriate nodes and splits their space to join the network. There are two strategies for candidates selection, the first is to select one candidate and the second is to select $m$ candidates and choose the one with the heaviest load. The second strategy will achieve better load balancing at the cost of higher joining cost. For the dynamic data distribution, each node estimates the average load $L$ by sampling loads on its neighbors. The node is considered to be lightly loaded if its load is smaller than $L$ by a selected threshold $\delta$ and heavily loaded if its load is larger than $L$ by $\delta$. Then each node periodically exchanges its load with its neighbors to achieve load balancing.

**HSFC-based** Although the HSFC-based approach also makes use of the one dimensional routing space Chord [17], it is different with ZNet. In ZNet the multi-dimensional data is partitioned in native data space, while in HSFC-based approach the multi-dimensional is first mapped into one dimension and than range partitioned in one dimension. In this section, we will study the HSFC-based approach in detail.

The Hilbert space filling curve is used to map multi-dimensional data into one dimension and preserves the locality during mapping. The Hilbert space filling curve is constructed recursively. Given a $d$-dimensional data space, suppose the keyword is base $n$, then in the first round, the $d$-dimensional space is partitioned into $n^d$ subspaces. The Hilbert space filling curve is a line that passes once through each subspace, and this is called the $1^{st}$ order space filling curve approximation. Then in the second round, each subspace which is generated in the first round is further partitioned into $n^d$ subspace, thus after the second round, there will be $n^{2d}$ subspaces. Then the Hilbert space filling curve is also a line that passed once through each subspace, which is called the $2^{nd}$ order space filling curve approximation. In the same way, we can generate the $k^{th}$ order space filling curve approximation which connects $n^{kd}$ subspaces. **Fig** 7 shows the first order and second order Hilbert space filling curve for a 2-dimensional data space and a base 2 keyword. A set of contiguous subspaces in $d$-dimensional space will be mapped to a collection of segments in the space filling curve. These segments are called clusters. The shaded parts in **Fig** 7 shows a cluster.

The Hilbert space filling curve preserves the locality. Points which are close in the 1-dimensional space are mapped from the points that are close in $d$-dimensional space. However, the points that

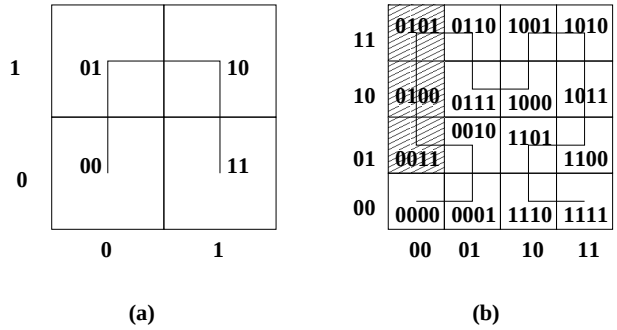


Figure 7: Hilbert space filling curve for d=2 and n=2 (a) $1^{st}$ order approximation (b) $2^{nd}$ order approximation

are close in $d$-dimensional space may not be mapped to the points that are close in 1-dimensional space. For example, as shown in **Fig** 7(b), two points 0000, 0011 are neighbors in 2-dimensional space, however, they are separated when mapped to 1-dimensional space. This is a drawback for such kind of algorithms that map data from multi-dimensional space into 1-dimensional space, since when the dimension is high, the data which are near in multi-dimension will be far apart in 1-dimension, and the locality property may not be guaranteed well. We will study this further in the performance study.

After mapping the multi-dimensional data into 1-dimension space, the next step is to map the 1-dimensional index space onto physical peers in the overlay network. This approach uses the Chord [17] as the overlay network. In Chord, each peer is assigned a unique identifier from 0 to $2^m - 1$ and peers are arranged as a circle. The Hilbert space filling curve based indexes are considered as the keys for data elements, and then the data element is allocated to the first node whose identifier is equal to or follows the key in the identifier space. **Fig** 8 shows a Chord overlay network with five nodes. We know that the data element with key 5, 6, 7 and 8 will be allocated to the node with identifier 8.

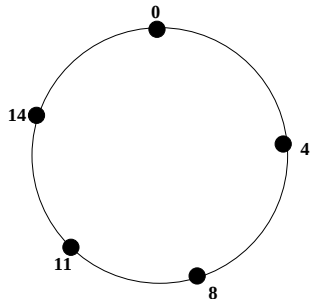


Figure 8: An example for Chord network with five nodes

There are two steps for query processing. First, the keyword query is translated to the corresponding clusters in the space filling curve based index space. Second, a message is sent to each cluster such that the query will be evaluated on appropriate nodes in Chord overlay network. For example, the query (00, [01,11]) will be first translated to the shaded cluster in **Fig** 7(b). Then a message is sent to this cluster and the query is to be evaluated on nodes with identifier 4, 8 in **Fig** 8, since the cluster is stored on these two nodes.

The above query processing is a little straightforward, since we need to send a message to each cluster, however, the number of messages sent can be reduced. We explain this more clearly using example in **Fig** 9. Suppose there is a 2-dimensional data space and the range query issued is (010, *). The range query will be translated into five clusters in the index space, i.e. (000100), (000111, 001000), (001011), (011000, 011001), (011101, 011110). The first cluster will be allocated to the node

with identifier 000100, the second cluster will be allocated to the node with identifier 001001, and the last three clusters will be allocated to the node with identifier 011110. For the straightforward query processing, a message is sent to each cluster, the total number of messages needed is 5, with the last three messages sent to the same node. However, for each node, only one message is enough. The query optimization is to reduce the number of messages.

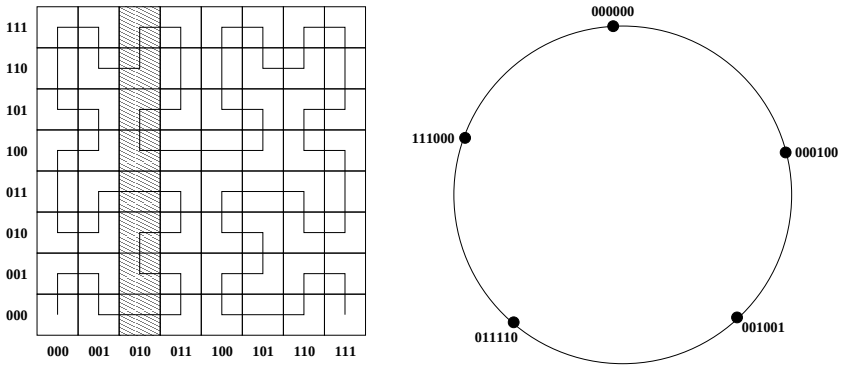


Figure 9: An example to show the non-optimal in message sending

The refinement of queries on the data space is recursive and can be modelled as a tree structure. Suppose the query (010, *) is issued on a 2-dimensional data space. **Fig** 7 and the left figure in **Fig** 9 shows the refinement of the query on the data space. The corresponding tree structure is shown in **Fig** 10. Each node in the tree corresponds to one cluster in the data space. Query (010, *) will be translated to one cluster in the first order Hilbert space filling curve, which corresponds to the root node of the tree, i.e. (00, 01); one cluster will be refined into two clusters in the second order Hilbert space filling curve, which corresponds to the two children of the root; finally, the five clusters in the third order Hilbert space filling curve correspond to the five leaf nodes in the tree. The query optimization is to prune nodes from the tree during the routing.

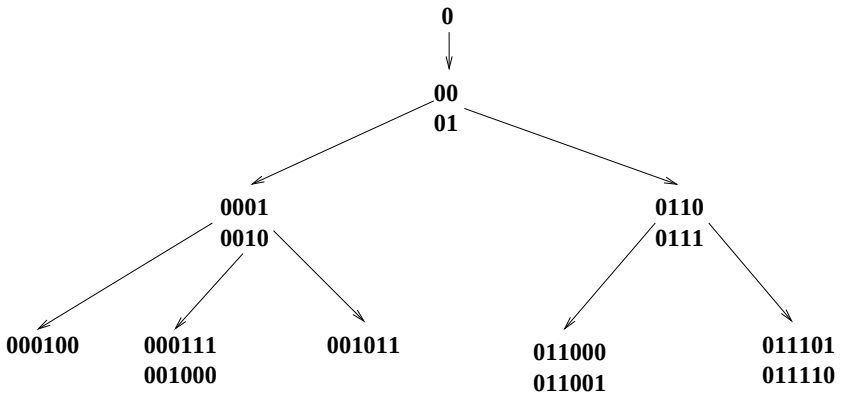


Figure 10: The tree structure for the query refinement

Suppose the query is issued on the node with identifier 111000 in **Fig** 9. Since the prefix of the cluster which is the root of the tree is 0, then the query is routed to the node with identifier 000000. For the cluster which is the right child of the root node, its prefix is 01, then the query is routed to the node with identifer 011110. Since the identifer of the node is larger than the identifiers of all the query nodes, then there is no need to further refine the tree. The children nodes of the right child of the root node can be pruned from the tree.

This approach also achieves load balancing in two ways. First, the load balancing is performed at node join. The incoming node will select several nodes and send joining messages to them. Then the incoming node will select the most loaded node as the joining point. Second, the load balancing is performed at runtime. A load balancing algorithm will be run periodically to perform the load balancing between neighboring nodes.

**SCRAP** The SCRAP [8] is similar with the HSFC-based [14] approach, in which the multi-dimensional data is mapped into 1-dimension with some space filling curve, and then range partitioned the 1-dimension data to allocate them onto physical peers and finally makes use of the 1-dimension routing space to process queries. The mapping mechanism in SCARP can use z-ordering and Hilbert space filling curve. The Hilbert space filling curve is the same with what we have discussed in HSFC-based approach. The z-ordering is just to interleave the bits between every dimension. For example, the two dimension point (0110, 1110) will be mapped to 01111100 in 1-dimension.

After the multi-dimensional data have been mapped into 1-dimension, the 1-dimension data is range partitioned and allocated to the physical peers in the network. Each physical peer will take charge of data with contiguous 1-dimensional values.

The different part of SCARP with the HSFC-based approach is that it uses Skip graph [3] as the routing network rather than Chord. The Skip graph is discussed in the ZNet part, which is multiple linked lists for efficient routing.

In SCRAP, there are two ways to achieve load balancing. First, the partition boundary between two nodes which manage the adjacent ranges can be adjusted such that the load can be transferred from one node to the other. Second, a new node with the empty range can be added to split the range on the node with heavy load such that the load on the node can be distributed.

**Performance study and discussion** In this section, we will study the performance of the above approaches in terms of the requirements for multi-dimensional range query processing, and we will also compare these approaches.[1]

The first aspect we want to compare is the locality. The performance for the ZNet and SCARP approaches is shown in **Fig** 11. The HSFC-based approach does not provide the performance for the locality, however since it is similar with the SCARP algorithm, we can expect that they achieve the similar for the locality requirement. The y-axis in the chart is the average number of nodes on which the answers are stored. The numbers on the y-axis do not matter since they are from different datasets, what we care is just the trend of the curve. We can see that both ZNet and SCRAP do not scale well with the dimensionality for the locality requirement. The average number of nodes which are needed to search increases fast with the dimensionality. It is due to the mapping from multi-dimensional space to 1-dimensional space. When the dimension is high, the points which are close in high-dimension may be far away when mapped to 1-dimensional space. This type of approach can not avoid this, since they make use of 1-dimensional routing space, therefore the multi-dimensional data have to be mapped into 1-dimension such that the 1-dimensional routing space can be used.

In **Fig** 11, we also notice that the ZNet achieves better than the SCARP on the dimensionality. In ZNet, the curve increases fast when the dimension is larger than 12, while in SCARP, the curve increases fast when the dimension is only larger than 3. This may because that the ZNet partitions data in the native space which is better for locality compared with the range partition in 1-dimensional data in SCARP.

The second aspect we want to compare is the load balancing. The first three figures in **Fig** 12 show the load balancing in HSFC-based approach, and the forth figure shows the load balancing in ZNet. For the HSFC-based case, the first figure is the original distribution of keys in the index space; the second figure is the key distribution using only load balancing at node join; the third figure is the key distribution using load balancing at both node join and runtime. We can see that only load balancing at node join can achieve the load balancing at some degree, and the runtime load balancing

[1] Figures illustrating performances have been borrowed from the original papers.

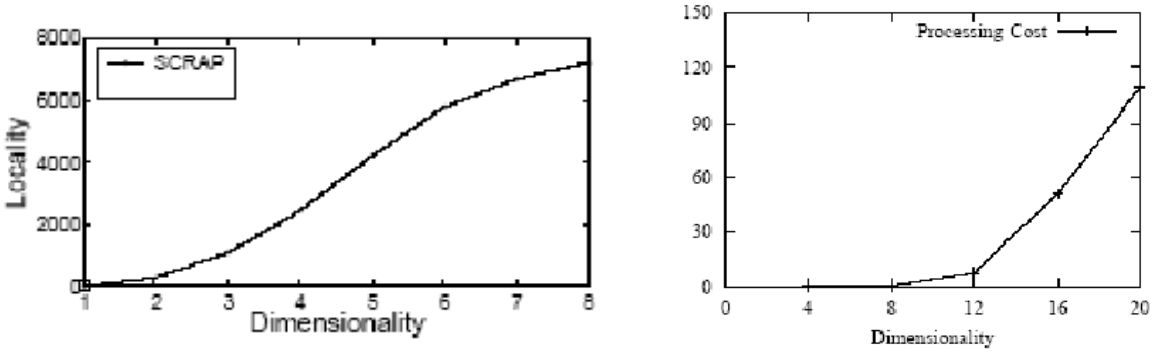


Figure 11: The comparison for the locality

can improve the performance. The case in ZNet appears the same property. The two load balancing mechanisms together can achieve good performance.

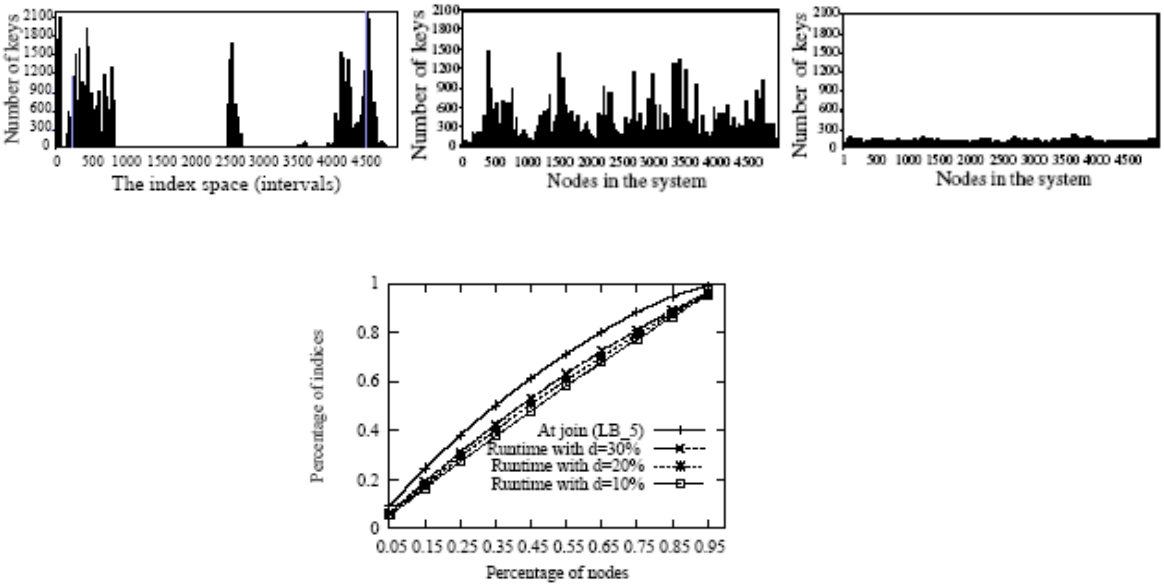


Figure 12: The comparison for the load balancing

For the routing efficiency, since all above approaches makes use of the 1-dimensional routing space such as Chord, and Skip graph, the number of messages which are needed to be sent for routing is the same, which is O(logn) and it does not increase with the dimensionality of data.

As a conclusion for the approaches that use one dimensional routing space, the locality is satisfied when the dimension is not high. However, it does not scale well with the dimensionality, since the high dimension will make the points which are close in high dimension far away when mapped to one-dimension. This drawback can be improved by the approached discussed in the next session. The load balancing can be achieved by the load balancing at node join and runtime together. The routing efficiency of this type of approaches is good.

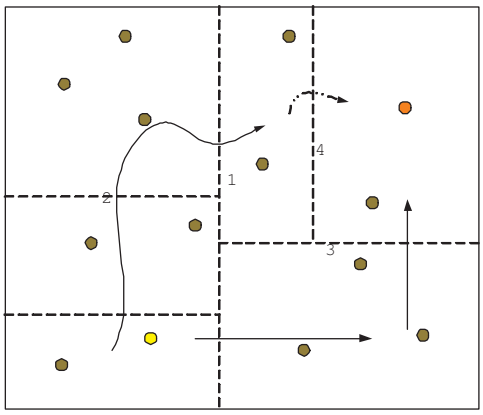


Figure 13: Partition and routing in MURK

### 2.2.2 Routing in multi-dimensional space

The basic idea for the SCRAP is to map the multi-dimensional data into one dimension according to the space filling curve first and then query process is implemented based on the one dimensional data such as the range partition. While the MURK, short for Multi-dimensional Rectangulation with Kd-tree, is based on mapping the multi-dimensional data space into rectangles and then each node manages a certain rectangle, and the queries are routed in multi-dimensional routing space. Essentially, efficient range query requires the data keeping locality while routing cost from the node issuing the query to the node storing the target data is small enough. The basic concept for both SCRAP and MURK is to partition the data space and balance the work load for the nodes in the network while guaranteeing the data in each node having certain locality. The difference lies in the ways of the partitioning. Other than partitioning, an important component for the range query is the routing in the new data space. The efficiency of the routing has great impact on that of the range query. In all, the partitioning tries to equally distribute the work load into the nodes and routing in the new network space make the query more efficient. In the following, the partitioning and routing for the MURK will be separately talked about and finally a performance comparison will be given.

#### MURK

**Partitioning based on KD-tree** KD-tree is a multi-dimensional search tree for points in K dimensional space. The basic property is that levels of the tree are split along successive dimensions of the points. It is like a tree combining the binary search trees in all the dimensions and for each level it functions like a binary search tree. So the dimensions of the data space would be split cyclically until no nodes left. Finally, the partition would make sure that each leaf node of the KD-tree manages one rectangle in the data space. Based on the conception of the KD-tree, the procedure of the multi-dimension space partitioning could be illustrated as follows.

**Fig** 13 describes the partitioning procedures for the MURK and the basic routing algorithm in the two dimensional space. Points in the rectangles presents the data elements and dash lines are for the partitioning edges. Partitioning(construction): The data elements in the whole space are initially managed by one node. When a new node applies to join the network, then the initial node would split the data equivalently along the first dimension and transfers half of the load to the new node, just as dash line 1 separates the date elements. For easy expression, these two nodes could call each other buddy. When another new nodes joins the data space managed by one node, then this space owners would split the data again as before but along the second dimension and assigns the data parts to the nodes. The procedure would go on and on whenever there is an incoming node. Accordingly, the

reverse steps would be adopted when a node leaves the network. The work load of the leaving node would be taken over by the existing node. The ideal case is to find its buddy from whom its work load coming from to recover to the state before the splitting. While sometimes, the buddy of the leaving node may have been split recursively. In this case, the leaving node would find one neighbor with the least work load to take over its part.

**Comparison** The basic idea for the partitioning in the MURK is very similar as that of CAN. Both the CAN and MURK try to map the data into multidimensional data space as presented above, while the partitioning of CAN is based on the assumption that data are uniformly distributed in the multidimensional space, so the work load distribution just equally partitions the data space rather than that in the MURK. All the rectangles of CAN in one partitioning level are of the same size, but different in the MURK. In this sense, CAN could be considered as a special case in MURK with evenly distributed data in the space. This difference affects the routing strategies for the query. While at a philosophical level, a distinct difference for the number of dimensionality in the CAN is decided by the routing consideration,not by the dimensionality of the data. Because in CAN the data is uniformly hashed in to a virtual d-dimensional Cartesian coordinate data space without considering the property of the data. The reason of building this hash is for convenience of later routing. MURK makes the partitioning according to the dimensionality of the data with more concern on the load balancing for each node. An obvious property for the MURK is that, different rectangles would have different number of neighbors resulting from different sizes of the rectangles, which makes the routing more complex.

**Routing** The basic conception for Routing is to transfer the query from the node issuing it to the nodes having related data. Partitioning in MURK intents to make related data of a range query distributed in fewer nodes, while the routing tries to locate the related data more efficiently. The intuitive way of routing is very simple. Since nodes are interconnected in an overlay network, each node has links with its neighbors.The distance for the routing between two nodes is the Minimum Manhattan distance. Based on this binary relationship for each link of the node, named grid link, in the space, it would be natural to use the greedy algorithm find one near way from the source node to the target according to the heuristic of reducing the Manhattan distance in each dimension.The algorithm could be illustrated by the **Fig** 13 as following: The arrow lines in the rectangles are the routing paths. The node managing the yellow node is the source issuing the query and target element is the red element shown in the graph. According to the basic grid link and greedy algorithm, each time the node selects a way in which the Manhattan distance could be reduced to the Maximum degree. The arrow lines provide one path solution. To achieve this algorithm, each node must keep the boundary information of its neighbors. This grid link based routing is not efficient enough. In some sense, it is like the sequential search which would cost linear time as O(n).

Similarly as using binary search to improve the performance of sequential search, which has made the complexity reduced to O($log^n$), some skip pointers may be helpful to speed up the routing. The basic idea is build on the greedy algorithm talked as before, the only difference is the definition of a neighbor not only in term of a grid link, but also in term of a node linked by the skip pointer. How to select a skip pointer would be the key issue. Two kinds of strategies for establishing the skip pointers have been proposed in [one torus] as following:

- Random: The skip pointer maintained by the each node is randomly selected from the node space.
- Space-filling skip graph: Each node maintains the skip pointers at exotically distance from it.

The idea for randomly selecting a node is intuitive, which is to keep more routing choices. Some choices may break the limit for sequential search and jump to a much more close space to the destination. To some certain degree, this is a little like the idea to keep multi-reality in one node in the CAN. For the Space-filling curve based skip pointer, there is some guarantee that node pointed must be at

exponential increasing distance away. With the help of this skip pointer, the routing would be much faster in exponential manner. The basic concept to achieve the exponential distributed skip pointers is as follows: Create a linear ordering of nodes and the distance between the nodes approximates the grid distance between the nodes in the native space. One way to achieve the linear ordering is as following: Use the coordinate of the centroid of the partition as the ID of the node and then map the nodes into one dimension via the space-filling curve such as Z-curve. Next, skip graph is built on the linear ordering of the nodes. This method is based on the exponential distribution property of the skip graph. Without considering the different ways for constructing skip pointers, one example in **Fig** 13 is to show the efficiency of the skip pointer in the routing: The curve lines show one possible routing way. With the help of the skip pointer, the query may be routed to a place much nearer to the target than the basic grid link in fewer steps.

**Performance analysis and comparison** Two metrics are used to evaluate the performance of the SCRAP and MURK approaches: locality and routing cost. Locality is measured by the average number of nodes that a given query has to access to get the exact answer. The routing cost is measured based on the average number of messages exchanged between nodes to route the query. Good performance in these two metrics could result in better efficiency in the range query. To a certain degree, locality shows the nodes needing access and routing cost shows the cost for reaching the node.

Lots of experiments have been implemented to compare the performance of the several strategies according to the variation of the parameters such as number of dimensions, node number and query selectivity. It would be presented in terms of the following two metrics.

For the locality, it becomes very poor with the increasing dimensionality in the SCRAP, for in which the query has to visit more nodes to get all the relevant nodes. In MURK, the query locality keeps nearly stable with the dimensionality increasing. When varying the number of node in the network to check the scalability of the approach, the MURK present much better, even ideal performance compared to the SCRAP, in most cases, the MURK just needs visit one node. When considering the query selectivity, the experiment shows that the query locality in the MURK increasing linear with the increasing selectivity and even for clustered data,showing that, the relevant data is well balanced among the nodes. Correspondingly, the SCRAP performs much worse for some nodes nearby in the data space are not that near when mapped into 1-d space. In all, MURK achieves better performance in query locality compared to the SCRAP, and even better when varying the parameters of dimensionality, network size and query selectivity.

For the routing cost, the performance is still compared according to the parameters used for the locality. The locality of the SCRAP is independent of the dimensionality of the data, for it would just need to rout in the 1-d space. While for MURK, after a certain number of dimensions, all the MURK-sf(with space-filing skip pointer) MURK-ran (with random pointer) and MURK-can( plain murk) would outperform the SCRAP. When comparing based on the selectivity, MURK-can performs poorly, while MURK-sf performs consistently well. When the network size is small enough, MURK-sf performs not as good as MURK-ran, in which case, the skip graph are too close to each other that the efficiency could not be shown. Accordingly, the performance of MURK-sf is much better when the network reaches a certain size. From the above experimental result, we can conclude that: Routing based on MURK-can is expensive and not scalable. Skip pointer could greatly speed up the routing especially for MURK-sf in large network. SCRAP is efficient for the routing but not good at locality.

**Discussion** Though MURK achieves good performance, some issues still need further study. One is the non-uniformity number of neighbors. Since the MURK partitions the space into parts with equivalent load, it would naturally result in the cases that some nodes have more neighbors, bringing heavy routing load. One tentative process for the partitioning is to add some replicates and make the partitioning more balanced in terms of dimensionality space. Another way is to add some more skip pointers to function as neighbors. The other important issue for the MURK is that how to keep

load balance in case of dynamic data distribution. What we have considered before is related to the dynamic network such as node joining and leaving, while considering the load in one node keeps stable, how to process the case of the dynamic work load in one node. The basic solution may be to borrow some data from the heaviest neighbor periodically when the unbalance reaches a certain degree. If the data increase in one node, then this node could cut down some work and move it to the node having light work load. This should be done periodically to reduce the network load. When data is transmitted between nodes, the according hash table should be updated to keep consistency.

# 3 Background on *k*NN searching

Several approaches to searching in P2P networks have been developed, These approaches have been well summarized in [15]. We shall include them here for completeness. Primarily for the unstructured networks the flooding based techniques [2] have been developed. If the system uses centralized servers then all searches are forwarded to it before actually retrieving the result data and for structured networks the usage of distributed hash tables (DHT) like in Chord [17] has been applied. Pre-dominantly these searches have been developed to locate exact matches such as music, video file names. But with the development of P2P systems supporting various other types of content such has image databases, text documents, web repositories, biological data searching requirements have grown complex. The searching requirement for these kinds of data not only include exact matches, but also include near matches. For this there exists the requirement of defining a notion of similarity. To define similarity one needs to develop metrics. With the help of metrics, one can define exact, ranges and k most similar searches. The challenge in developing such a metric system in a P2P environment lies in the scalability aspect. As large number of comparison operations are going to be computed across several thousand nodes. The metrics must lead itself to be easy to compute and to locate objects. The work by Batko et al [18] is one of the foremost in developing a distributed index using metrics that is suitable for $k$NN searching in P2P networks. It is an extension of their work in scalable similarity search in metric spaces [4].

Another important aspect in searching over P2P networks is obtaining excellent query response times and quick preliminary results. Loo [11] has found that the Gnutella based system perform very well with popular files, but had high latency for rare files. There have been studies to show that general querying patterns have a Zipfian distribution [11]. The Zipfian distribution in this case has a head of popular items and a long tail of rare items. This brought about the inspiration behind the PIERSearch based hybrid model. PierSearch has been developed over PIER [9] , a DHT based internet scale relational query engine developed by the author's research group. The authors performed real time studies on the performance of the Gnutella network and identified key performance parameters. One of the challenges faced in this problem is that of identifying rare items to be indexed into the DHT of the Hybrid model. We shall detail these in the section on $k$NN searching.

The first approach we want to discuss is called pSearch, which retrieve the document which the users are interested in from the internet. It can be considered as a kind of kNN searching, since it selects the nearest neighbors of the document that users are interested in.

# 4 *k*NN Searching in P2P

The need for searching the $k$ nearest neighbors of a given object can be explained as follows. In most cases where the nature of the underlying database is not known, one cannot estimate that a certain number of similar objects can be found between an upper and lower bound of a metric. Thus repeated searches using varying ranges might be required. Alternatively, using $k$NN searching, heuristics are applied such that $k$ similar objects are found using minimal number of iterations of searches. Additional applications of $k$NN searching include approximate pattern matching and semantic matching. Let us start by discussing a pSearch that is used for $k$NN documents in the internet.

## 4.1 pSearch

### 4.1.1 Motivation

With rapid development, Internet has become the most important platform for data exchange. More and more documents have been stored over the network. How to efficiently search the documents you are semantically interested in becomes a big issue. A bad searching engine may return a large amount of documents including some that are not so close to what you are looking for, bringing you more burden to filter, and even sometimes some very important documents may be missed. [5] proposes a IR system over the p2p network to address this challenging problem.

In fact, lots of research work has been studied in the searching techniques over the p2p network, however, most of the searching systems are either unscalable or could not provide deterministic guarantee. For the unstructured p2p network, following techniques are popular:

- Centralized index: Facing the fatal problem of scalability, single node failure and performance bottleneck.
- Flooding: Naturally bringing lots of overhead.
- Heuristic based flooding: The price of reducing searching space is probably missing some important documents.

For the structure network, such as CAN [12], Chord [17] or Pastry [13] and so on, are mainly built on the distributed hash table(DHT), which has very good scalability and fault-tolerance. But coming from the nature of hash, DHT is more suitable for keyword match, not efficient for range query and especially content-based semantic search. While in traditional IR system, content and semantic based search has been well studied and some mature techniques have been used such as VSM and LSI. Fully exploiting the mature IR techniques over the p2p network to make the searching system have efficiency of DHT system as well as accuracy of IR system is the basic design target of pSearch.

In IR system, content based search has been one of the major issues, thus lots of algorithms have been proposed. Till now mainly two kinds of techniques are popularly used: VSM and LSI. VSM, short for vector space model, represents the documents and queries as term vectors. The weight of a term is measured by the statistical term frequency*inverse document frequency(TF*IDF). In the IR system, the rank of the result document is evaluated according to the similarity of term vectors X of the document to the term vector Y of the query, and the similarity could be expressed by the cosine value of two vectors cos(X,Y). To compensate the different document length, usually VSM normalizes vectors to unit Euclidean norm before calculating the cosine value. The techniques based on the VSM face a important problem that is synonymous and noise. In many cases,different terms may have similar meaning. However, VSM would differentiate the terms which are semantically the same. LSI,short for Laten Semantic Index, could overcome these problems based on the statistically derived conceptual indices instead of term frequency itself. The basic concept for LSI is as follows: LSI uses the singular value decomposition (SVD) to transform a high dimensional term vector got in the VSM into a lower-dimensional semantic vector based on projecting the former into a semantic subspace.

### 4.1.2 pSearch system

In the pSearch system, the most important component is the pSearch engine which is organized by CAN and all the nodes in the pSearch engine have homogeneous functions.The nodes in the pSearch engine would be in charge of the operation such as document publishing and query routing. Not all the nodes over the network would be in the pSearch engine and only the subset of nodes which are stable and have good network connectivity would be included. And the pSearch boundary could be adjusted according to the work load. The work procedure of the pSearch based on IR techniques could be illustrated as follows in **Fig** 14. The nodes in the rectangles forms pSearch engine.

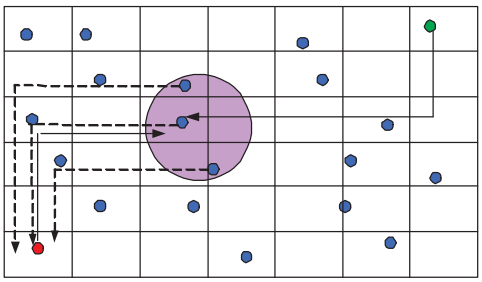

Figure 14: pSearch

- Document publishing. After receiving a document form a client, such as the the green element in the graph, the node in the engine would calculate the vector representing the document either based on VSM or LSI. Then use the vector as the key to store the index in the CAN. In the example, the store place would be node managing the yellow element.
- Issuing query. After receiving a query from a client, the node managing the red element in the engine would generate the vector for the query. Based on the hash function, the destination of the query is decided and would then route to the target. In the example, the target node is the one managing the yellow element.
- Routing query. The query would be routed to the destination node and then it would be transmitted to nodes in a certain area within a bounded radius, Such as the light blue area in the graph.
- Local searching. The nodes receiving the query would then do the local search and return the result to the client. In the example, all the elements falling into the light blue area would be returned to the node issuing the query.

How to generate the vector would affect the accuracy of the pSearch system without changing the basic operations. The algorithm based on VSM, named pVSM, would generate the term vectors. In the operation, we just select some term vectors of most weight to make the hash and would routed to several distinct pointers in the CAN space. So the result of the query would also be returned from several destines. After receiving the several partial results, the engine node would then make the ranking for them based on the similarity and discard the less related results. Since, according to the character of the VSM, pVSM still have the problem for the synonymous and noise. So in the later part, I would mainly talk about the pSearch based on the LSI, name pLSI, in which each document or query would just be projected to one certain space managed by a node. But according to the document publishing system, semantically close documents would also be hashed into close space in the CAN. So the node receiving the query would then make a local search in a certain bounded area, all the semantically closely related documents would be returned as results.

pSearch as an IR system built over the p2p network has several advantages:

- Obviously, pSearch has inherited the nice properties of p2p system such as scalability and self-organizing.
- Without exhaustive search over the whole space, but just locally within a certain bounded area, the query could achieve ideal result.
- The communication overhead is also limited to the query and reference to the top document which are independent of the corpus size.
- Another merit for pSearch is that approximation of global statistics is good enough for generating the semantic vector.

Though pSearch shows great success in terms of the advantages, there are still problems left for further study:

- Dimensionality mismatching between CAN and LSI.
- Uneven distribution of indices.
- Large searching region.

These problems would be detailed discussed in the following sections.

**Dimensionality mismatching between CAN and LSI** The nodes over the network are managed by the CAN, while there are usually not enough nodes in the can to partition all the dimensions in the LSI, resulting in that some dimensionality can not be partitioned. While the documents with similar semantic content along these dimensions would be spread over all the nodes, so the search would be expensive and is not what the pSearch would like to have. The actual partitioned dimensionality of the CAN is called effective dimensionality. Though it is impossible to partition all the dimensionalities with the limited nodes, some properties of the documents need noticing. The similarity of a query and document is usually decided by a small fraction of the elements in the semantic vectors. And only a small number of the semantic dimensions related to a certain amount documents. More importantly, low-dimension elements in the semantic vectors contribute more to the similarity. Based on these important observations, rolling-index is proposed to overcome this problem. The basic concept is as follows: For a semantic vector, V=($v_0$,$v_1$,...,$v_l$), it would be rotated by m dimensions each time to generate a series of new rotated semantic vectors, called support sub-vector. The ith vector is $V^i$= ($v_{i*m}$,...,$v_0$,$v_1$,...,$v_{i*(m-1)}$). The formula m=2*3*ln(n) is used to approximate the effective dimensionality of the CAN. Each time a different $V^i$ is used to route the query and search and all the matching results would be returned. since the similarity measure has not been changed by the rotated space according to the evaluation of the similarity, the close document in the CAN is still close semantically, which is the base for correctness of the rolling-index.

The rolling-index is based on some important observations, and multiple rotated support vector could usually cluster the closely related documents with storage cost as p times as the base storage space. The observation is not always the truth, sometimes some high-dimensional elements may have heavy weight, which makes the rolling-index ineffective. Then selective rotation is proposed which would store the indices in the stored rotated space whose corresponding support vector cover the important elements which are not covered by the first p spaces. The selective rotation is theoretically effective but evaluation has not been implemented in the paper.

**Uneven indices distribution** In the CAN, when a new node is going to join the network, it would randomly select an existing node and split the work load of the existing node. This would achieve good performance when the load is uniformly distributed over the whole space. But for the pSearch system, in certain semantic space, there would be more documents resulting in heavy load. How to evenly distributed the load mainly indices becomes an great issue. Content-aware node bootstrapping is proposed to make the nodes distributed according to that of the indices. The basic conception is as follows: When a new node comes, it randomly selects one document that is going to be published, and this semantic vector is randomly rotated to a space i, then the rotated vector would decide the space where the new node should join. Though the load for the nodes are completely balanced due to the randomness, when reaching a large corpus, the load is expected to become more balanced.

**Reducing search space** Based on the techniques discussed before, the query would be limited to search a fraction of local nodes to find the most relevant document. While because of the curse of dimensionality, the distance between a query and its nearest neighbor becomes large with the increasing dimensionality of the space. In this case, a naive nearest neighbor search would not work efficiently for it has to get access to lots of nodes. Based on the observation that relevant document is likely to be

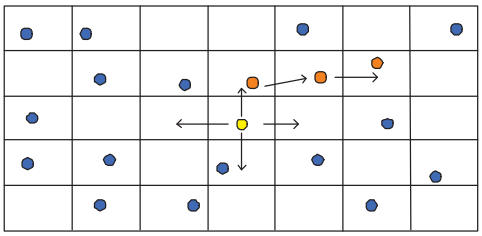

Figure 15: Content-directed Search

surrounded by other relevant documents, content-directed search is proposed. The basic idea comes from the observation and could be illustrated in the following example as **Fig** 15. When the node managing the yellow node issues a query for the red elements and search the near neighbors randomly. Then the query finds one node with the red element, and later query would starts from the node with the target element until the related elements found.

### 4.1.3 Performance

Lots of experiments are implemented based on the pLSI to evaluate the performance of the pSearch system. Two main metrics are used for the evaluation: visited nodes and accuracy. The number of visited nodes is the indication of the consumption of the system resources. Compared to the exhaustive search techniques, such as flooding, which results in very expensive cost, pSearch just needs to get access to a limited number of nodes. As far as we know, pSearch is a decentralized IR system built over the p2p network, so we would like to used the relative accuracy compared to the centralized IR system to show the performance of pSearch. The accuracy could be shown in the following form: Accuracy= $|A \cap B|/|A| \times 100\%$. A is the returned result for a certain query over the pSearch system implemented by the pLSI, while B is the result retrieved by LSI for the same query in the centralized environment, which is used as the baseline for the evaluation.

When the system size increases exponentially, the number of visited nodes grows slightly, for which is decided by the logarithmically increased number of neighbors. The experiment also shows that the accuracy could easily become high without visiting too many nodes. When considering both the size of the system and the corpus, it is found that the search cost increases moderately as they grows. For the number of returned documents, though the visited nodes increase, the average number of visited nodes needed to return a one relevant document is decreasing drastically.

To reduce the search space, the paper introduces the heuristic based content-directed search. It has been experimentally shown that the search with a warmup period to gather the past query information would not only increase the accuracy, but also reduce the number of visited nodes. When the experiment used replicates to improve the accuracy and efficiency, it finds that a small number of rotated spaces and a tight quit bound could achieve good accuracy. Based on the analysis of the result source distribution, it proves that a large amount of documents are clustered by the low-dimensional elements and also shows that the document found curve fits well with the nodes visited node curse, which proves the efficiency of the query heuristic in guiding the search. The later experiments present that the pSearch system is not very sensitive to the parameters such as query length and so on.

### 4.1.4 Discussion

pSearch system have achieved appealing performance, but there are still some research work left for further study to improve the performance.

- Sometimes, a node stores lots of homogeneous documents and there is no need to index for each of them. Maybe an index at coarse granularity is more efficient and hierarchical k-means could be used to cluster the documents at one node.

- Another more intelligent way could be used to assign weight for document. Though the documents the pSearch returned are all closely related to the query, some of which are more popular for client to use. So we can add additional weight for the document of more popularity, thus making the ranking in pSearch more intelligent.
- In the pSearch system, we use replication to improve the accuracy and efficiency, and if we could consider the system capability of the node when we make the replication, the pSearch system may perform much better.

#### 4.1.5 Conclusion

pSearch is an IR system build over the p2p network. It combine both the techniques in IR and p2p areas. Based on the comparison of the performance of VSM and LSI, it is mainly implement on the LSI and in fact LSI is improved and derived from VSM. The nodes in the system are managed by the CAN. The basic idea if from CAN, but there are some improvements which are illustrated as follows based on comparison:

- In the original CAN, the node are projected into a multi-dimensional space which is predetermined. While in pSearch, the dimensional of the space is decide by the semantic vector generated form pLSI. The number of the nodes in the network decides the number of dimensions partitioned. Rolling index is proposed to overcome the dimensionality mismatch problem between CAN and semantic vector.
- For publishing, the index of a document is randomly stored in one node of the CAN according to a uniform hash function, which makes the work load uniformly distributed over the whole space. While in pSearch, the node storing the index of a document that is going to be published is decided by the semantic of the document, which is the base of the pSearch system to make semantic clustering search.
- When a node joins the network, CAN just randomly distributes the node into the space, while in the pSearh, the new join point is affected by the indices load of the space. More nodes would be distributed into the space having more indices.
- The search in the original CAN is based on the greedy algorithm to return one result, while in the pSearch, several closely related documents would be returned. In order to reduce the search space, heuristic based content-directed search is implemented.

### 4.2 Scalable nearest neighbor search in P2P systems [11]

After pSearch let us look at a paper that extends a similarity metric approach developed earlier [9] for range and similarity searching to support $k$NN searching. The essential idea is to use a distributed index structure known as GHT$^*$. To begin illustrating this approach we first introduce 'search metrics' and the notion of metric space.

#### 4.2.1 Search metrics

Search metrics are required to locate objects that match certain criteria. A metric provides a quantitative measure of similarity or even exact matches. In addition, a metric also enables comparisons to be made. Metrics have the properties such as strict positiveness, reflexivity, symmetry and satisfy the triangular inequality. The usefulness of metric search is the ability to answer varied types of queries including exact and range queries over sortable data, it can also support semantic similarity searches using vector spaces and a suitable distance function. There are several types of metrics that can used based on the domain. For example one can use *edit distance* to compare sequences or tree patterns or one can use the *Jacard co-efficient* for similarity of sets [18]. The searching metrics used in a P2P environment need to exploit the parallelism supported.

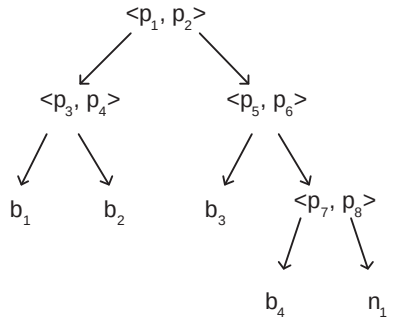


Figure 16: Example of an address search tree

Formally, a metric space is a pair ($D$, d) where $D$ is the set of objects and 'd' is the distance function that is used to compute distances between any pair of objects in $D$. The general idea of a distance is that, if it is small then they are closer (or similar). If $F \subseteq D$ is the data set, the range queries on F over query object 'q' retrieves all objects that are at a distance(radius) of $\rho$

$$\{x \in F \mid d(q, x) \leq \rho\} \tag{1}$$

The $k$ nearest neighbor queries returns the 'k' objects which is a subset of $F$ having the shortest distance to q.

$$|K| = k \wedge \forall x \in K, y \in F - K : d(q, x) \leq d(q, y) \tag{2}$$

#### 4.2.2 KNN searching using GHT*

GHT* is designed to be a distributed index structure, it assumes the following. The systems supports a message passing interface. Each node(peer) in the network has a unique identifier (NNID). Each peer stores its objects (data) in a set of buckets, each of a fixed capacity(total number of objects it can hold) and within a peer each bucket has a unique bucket identifier (BID). Also an object can be stored only on one bucket. Central to the GHT* is the address search tree (AST). The AST is a binary search tree, it is used to locate the buckets where the objects are to be stored or retrieved. Its inner(Non-leaf) nodes essentially contain routing information. Each inner node stores a pair of pivots that are representative of two metric objects form the dataset. The idea being all objects that lie in that metric range will be found in its subtree. Given an object O, if its metric is closer to the left pivot (Say P1) then that is traversed, else the right pivot(Say P2) branch is taken.

$$d(P_1(i), O) < d(P_2(i), O) \Rightarrow leftbranch \tag{3}$$

The leaf nodes in the AST point to buckets (BID) thats local to the peer or to NNIDs (Another peer). Thus, after navigating the AST, the data is either found on the local system if the leaf is a BID, or on the peer identified by the NNID. An example of an AST is shown in **Fig** 16. Inserting an object to the AST involves searching for the bucket at which it needs to be stored. The AST is traversed and if the leaf node reached is a BID, it is stored in that bucket, else if it is a NNID node, the corresponding peer is requested to find the right BID to store this object. Like all metric computations, this traversals are computationally expensive. Hence an encoding of the path traversed to reach each object can avoid repeated computations being made. This encoding is called a BPATH. The BPATH is a string of 1's and 0's, where a 0 implies that the left path was taken and a 1 implies that the right path was taken.

Range searching using the AST is similar to the insert. It too starts by traversing the AST. Given the radius $\rho$ a traversing operator $\psi$ is used to return all the BPATHs that need to be traversed to retrieve all the objects that satisfy the search range (radius). The formal algorithms insert and range searching can be found in [11]. An example of the GHT* is shown in **Fig** 17

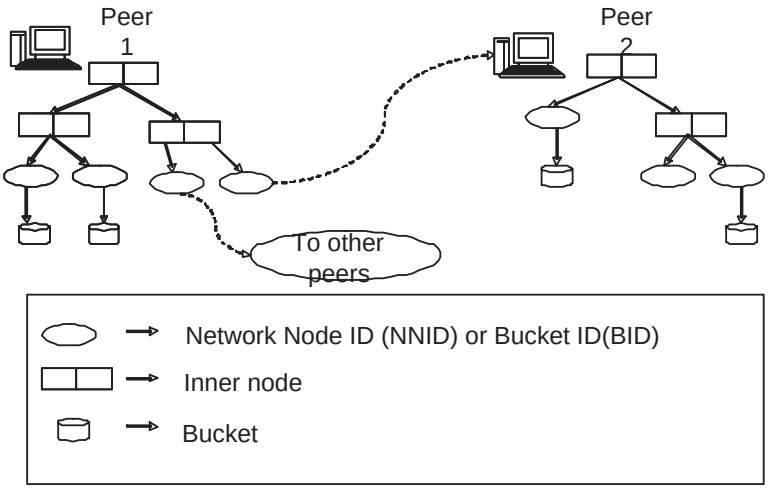


Figure 17: Example of the GHT* network

$k$NN searching: The problem in this context is to identify k objects similar to a given object O using the distributed environment. As the efficiency of this process relies on exploiting parallelism across the network, one heuristic is to start searching at a certain region and based on the initial findings modify the search radius and locate other candidate regions. Thus after locating the first region, all the computations hence forth can be executed in parallel and independent of each other.

In the GHT* approach, a initial search is made for the bucket with a high probability of occurrence for the nearest neighbors. The bucket that is likely to be selected for the insert of this node is a good start. If 'k' objects are found the distance to the $k^{th}$ object, it provides a highly probabilistic range to find 'k' objects on other peers. Thus, a range query is issued with a radius equal to the distance to the $k^{th}$ object. The results obtained are collected and sorted. We can obtain the first 'k' objects with the shortest distances to match the $k$NNs.

There exists the possibility that less than 'k' objects are found in the initial bucket searched. In this scenario an approximate radius needs to be selected. There are two options to select a suitable radius, one optimistic and the other pessimistic. In the optimistic approach, the idea is to minimize the number of buckets accessed and the number of distance computations. The trade-off is in the risk of having to execute another iteration if few than 'k' objects are found. In this approach, even though the first bucket accessed has few than 'k' objects, the distance to the furthest object in that bucket is used as the radius. It is optimistic in the sense, it hopes that other peers might locate enough objects within this radius. If 'n' is the number of objects returned. if $n \geq k$, we have found the result, else the next iteration uses $\rho + \rho(k-n)/k$ as the radius.

Alternatively in the pessimistic approach, it effort is to minimize the likelihood of another iteration at the cost of larger number of computations and bucket accesses. To estimated radius is the distance between pivot values of the bucket initially accessed. If fewer than 'k' objects are returned the pivot values in the next higher inner node is used. The algorithm for $k$NN searching can be found in section 4.2 of [4].

### 4.2.3 Performance

Let us now discuss the experimental results[2] obtained on two real-life datasets. One a vector of 45 dimensions using the Euclidean distance metric (VEC dataset), the second a sentence based dataset using edit distance as the metric(TXT dataset). Experiments were carried out on a hundred networked

[2]The experimental results have been taken from the original paper without the author's [18] permission

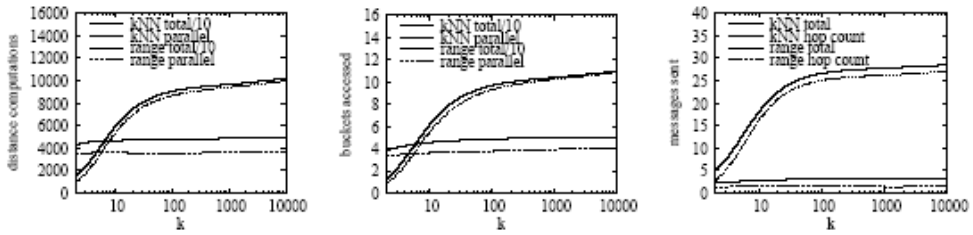


Figure 18: The variation of costs with increasing k on the VEC dataset

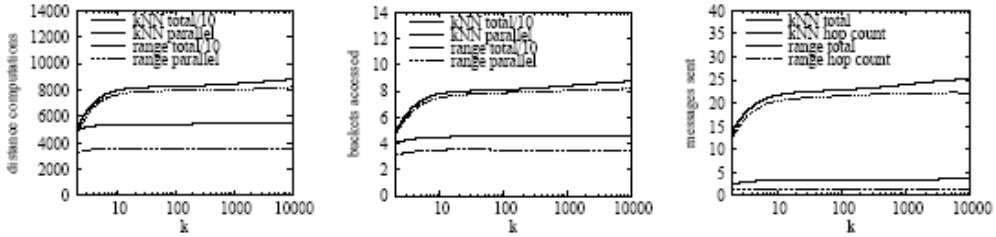


Figure 19: The variation of costs with increasing k on the TXT dataset

computers. The performance parameters measures include the number of distance computations necessary to evaluate $k$NN queries. The parallel cost is the maximal computations made on a peer or a set of peers accessed serially. As the search for the $k$NN involves locating the first bucket and examining its objects, before executing a range query. This is going to add to the computational complexity. Also measured in this study are the number of buckets accessed per query and number of buckets per peer in addition to the number of messages exchanged. The following observations have been made from the figures 18 and 19.

- The parallel costs of the $k$NN queries remain largely stable over increasing values of 'k'.
- The number of bucket accesses and distance computations increase very quickly with increasing 'k'.
- The $k$NN search is slower than the range query of radius equal to distance of the k$^{th}$ NN, but the overhead incurred is not depended on 'k'.

The scalability aspect of a distributed index is very important when it comes to real datasets. This approach using a GHT$^*$ exhibits a nearly constant costs even with inceasing size of the dataset. The experimental observations can be seen in figures 20 and 21. The graph on the left show that the parallel computations remains largely constant. The middle graph represents the number of messages exchanged. This increase is expected as more peers are used to store data. Lastly the graph on the right shows a gradual increase in the hop count.

### 4.2.4 Observations

The authors have described this research effort as the first in distributed index structures supporting the execution of the nearest neighbor queries on metric datasets. While, this approach seems to be scalable, the experiments lack comparisions with other $k$NN approaches. Another drawback, is that this approach hasn't discussed the possibility of high dimensionality datasets. The distance metrics and computations could be very complex in the high dimensional case.

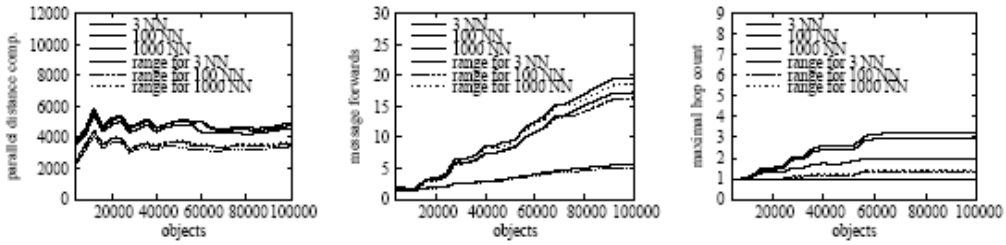


Figure 20: Scalability of GHT* with increasing VEC dataset

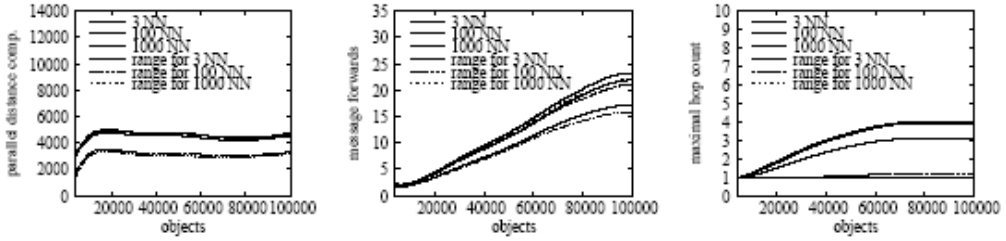


Figure 21: Scalability of GHT* with increasing TXT dataset

## 4.3 Enhancing P2P file-Sharing with an Internet-Scale Query Processor

In this section we will outline the research study by Loo et al. [11]. This paper does belong to the general category of searching methodologies, It however does not directly extend itself to $k$NN searching. However, the ideas for $k$NN searching can be applied to this kind of a querying system. As mentioned in the introduction, this paper focuses on a hybrid model that uses both the properties of structured networks and unstructured networks. Initially, we will study the Gnutella system.

**The Gnutella approach:** Gnutella [2] connects peer machines into an ad-hoc, unstructured network. Query processing uses the flooding approach: A node issues a query against its P2P neighbors. The neighbors forward the request recursively for a finite number of times (using TTL). All matches are returned directly to the query initiator. Flooding based schemes have the limitation that not all possible results are returned. It has also been observed that certain files although existing in the network cannot be found as they like outside the search horizon. These characteristics have been studied in detail and is explained in the following section. In the Gnutella network each node that joins the network shares its files. It is not required to publish its files like in the case of structured networks.

Gnutella has made a few optimizations to the pure flooding based system. It has introduced the idea of using ultrapeers and dynamic querying for optimizing its performance. When a Gnutella client joins the network, it joins as a leaf node of the network. It doesnot answer or forward any query requests, but it can issue them. After it joins the network, it publishes to a few selective nodes (its ultrapeers) its entire file list. Ultrapeers perform the query processing on its behalf. A nodes monitor itself to determine if they can be converted to ultrapeers based on factors such a uptime, bandwidth etc. Dynamic querying is used when very few results are returned, this essentially involves using a larger TTL. But as we shall see later, this is not an optimization with high benefits. This can be partly attributed to the large increase in number of messages to be sent even for an increase of one hop.

### 4.3.1 Performance evaluation of Gnutella

The Gnutella network was studied in terms of its size and topology using a crawling method in parallel over various Gnutellla nodes. The crawling help establish that most ultrapeers support from anywhere

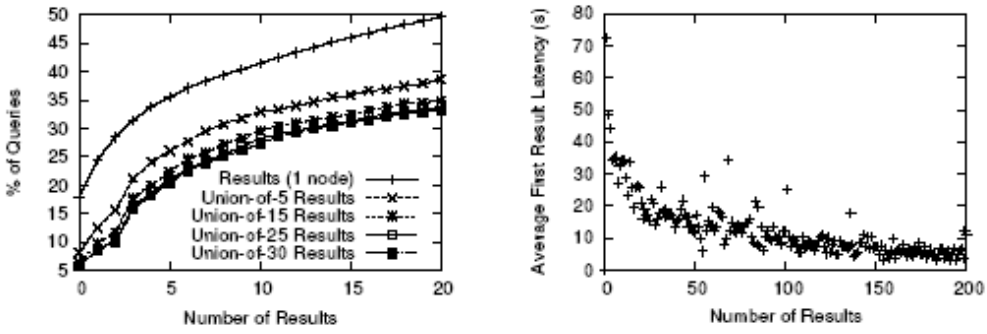


Figure 22: Performance analysis of the Gnutella network

between 30 and 75 leaf nodes. To measure the quality of the searching techniques metrics called the query recall and the query distinct recall are used. The query recall(QR) represents the total fraction of results returned against that which is available in the system. Query distinct recall(QDR) is the percentage of distinct results return from that which is available in the entire system. Estimating these correctly requires complete information (snapshot) about the entire system. As an approximation, simultaneous queries are executed against all the ultrapeers and the results obtained from them are merged,it was found that after a certain threshold of ultrapeers there was not significant increase in the size of the result set. Refer to the graph on the left of **Fig** 22.

It has been shown that there is an expected but strong correlation between the number of replicas and the number of results in the query. This has induced an inference that queries with small result sets are generally with few replicas and hence rare. While the opposite cannot always be said about large results being popular alone. However, Gnutella has been found to be effective for highly replicated files. Experiments have also shown that Gnutella is very inefficient for searching rare items. Refer to the graph on the right of **Fig** 22.

An important observation is the the graph in the left half of **Fig** 22 shows that for union of 30 only 6% of the queries returned no results, while union of 5 has upto 18% of the queries returning no result. Thus, a scheme that can unitize the available data in the system can help reduce this percentage queries with no results.

### 4.3.2 PierSearch

PierSearch is an distributed hash table (DHT) based querying system developed over the PIER system [9]. PIER is a DHT based internet scale relational query engine. PierSearch is essentially used to build a partial index that uses a DHT. The PierSearch system modeled on one node is shown in **Fig** 23. PierSearch supports keyword based searching. Items with filenames that contain all the terms of the search terms will match the query. The publisher component maintains an inverted file. The inverted file is used to quickly access all filename that contain a certain term. In the case of PierSearch, this is indexed using the DHT. The publisher generates the two tuples for each item.

- Item(fileId, filename, filesize, ipAddress, port)
- Inverted(keyword, fileID)

The item table stores an entry for each item that is to be shared. Essentially it requires a unique identifier. This identifier is generated using a hash function and is used as the publishing key for the DHT.

The searching process involves the local PIER engine generating a query plan, and posting it to all the sites that host a keyword of the query. This is done using the DHT. Also PIER performs a

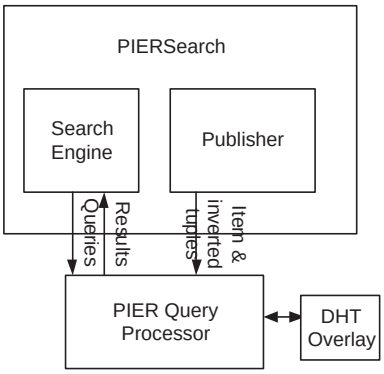


Figure 23: Example of PIERSearch on a single node

distributed join on the matching tuples and the node hosting the last keyword routes the results to the query originator.

### 4.3.3 Introduction to the hybrid search model

The key idea in this approach is the merger of two different methodologies. We have seen that Gnutella performs well to locate popular networks but fail to achieve the same for rare files. On the other hand a DHT based scheme like PIERSearch provides perfect recall, a complete implementation of PIERSearch alone would cause excessive loads on the network. Most of this can be owed to large list postings that need to be need to be sent to other peers. However, if the file is rare, one can be almost certain there are very few files that match it in the network, hence a lot more easier to compute. This has brought about the design of the hybrid model, where a Gnutella based search is used to find popular and highly replicated items and PIERSearch that builds a partial index for locating rare items. In this model, each ultrapeer identifies the rare items and publishes them into the DHT. The search is first carried out using the Gnutella, if sufficient results are not found, then the query is reissued using the DHTs.

Thus, the key to this model depends on the classification model used to identify if an item is rare or popular. We shall discuss some of the techniques below.

**Query result size:** Rare files typically have small result sets, hence if a query returns a small result it could be published into the DHT. The drawback of this approach lies in the fact that one cannot identify if an item is rare until it is queried for the first time.

**Term frequency:** In this scheme, each peer gathers statistics on the terms in the filenames used for searching. If a search contains terms that has a an occurrence frequency smaller than a certain threshold it could be classified as rare.

**Term pair frequency:**Sometimes, individual terms of a search may be popular, but when used in conjunction represents a rare file. So, in a multiterm search string of atleast one term-pair is below a a fixed threshold it is classified as rare.

**Sampling:** This technique samples the neighboring nodes and computes a lower bound estimate for each item. Items with a lower bound lesser than a threshold can be set as rare. This method is an alternative to gathering counts over a period of time.

### 4.3.4 Evaluation of the hybrid model

The goal of this hybrid system is to improve query recall for both popular and rare objects[3]. The probability that an object found in the hybrid system needs to be maximized, while keeping the publishing costs to a minimum. In the perfect system, where all peers have knowledge about the

[3]The experimental results have been taken from the original paper without the author's [11] permission

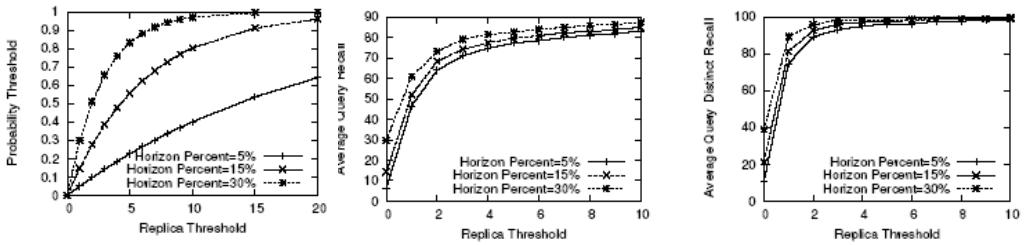


Figure 24: Query recall vs replica threshold

counts of replicas for each item, each peer needs to publish only those items below the threshold. As seen from the left graph of **Fig** 24.

From the graph in the center of **Fig** 24 it is seen that, when the replica threshold is zero, the query recall is equal to percentage found in the nodes in the horizon. As replica threshold increases, its seen that the query recall increases fast. Also, it can be observed that there is a smaller increase in query recall after query threshold reaches a significant value.

Also, from the experiments it has been observed that the sampling method to determine rare items results in the highest query recall. Additionally it has been shown that the term frequency method does well when there are low publishing overheads and the term pair frequency is better for large overheads.

### 4.3.5 Observations

This paper has tried to improve the query recall for searches for rare items. The studies on the Gnutella network have identified this limitation and brought forward the potential for improvement. The hybrid model using PIERSearch has approached this problem. It was clearly shown that, building a partial index over least replicated items can improve query recall significantly. There is potential for improvements on the determination of rare items. As was observed from the experiments there are significant publishing overheads and no one method could be singled out as the best approach. One possible alternative solution could lie reversing the sequence of the search. Given the search string, using its terms one could easily determine if the item is rare. If it is it can directly be looked up in DHT. Otherwise, a query is executed on the Gnutella network, and based on the result one can update the DHT if required.

# 5 Conclusion

In the first part of the report, we study several approaches which can support range query searching in P2P system. We first discuss the P-tree index which only handles the one-dimensional range query. Then we discuss the approaches for multi-dimensional range query. We divide these approaches into two types according the routing network they use. For the approaches which make use of one dimensional routing space, the load balancing and routing efficiency are satisfied, the locality is preserved when the dimension is not high. The locality does not scale well with the dimensionality. For the approaches which use multi-dimensional routing space, the locality is better preserved compared with the first type of approaches. However, the routing efficiency is not good when the dimension is 2.

Later sections of this study further explored the area of effective nearest neighbor searching P2P networks, we have studied for $k$NN searching pSearch and metric based GHT$^*$ approaches. We have also studied PIERSearch a hybrid model that uses principles from both structured and unstructured networks.

# References


[1] Freenet. http://freenet.sourceforge.com/.

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

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of SODA*, 2003.

[4] M. Batko, C. Gennaro, P. Savino, and P. Zezula. Scalable similarity search in metric spaces. In *Digital Library Architectures: Peer-to-Peer, Grid, and Service- Orientation, Pre-proceedings of the Sixth ThematicWorkshop of the EU Network of Excellence DELOS, S. Margherita di Pula, Cagliari, Italy*, 2004.

[5] Z. X. C. Tang and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of SIGCOMM*, 2003.

[6] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *WebDB*, 2004.

[7] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of 30th VLDB Conference*, 2004.

[8] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In *WebDB*, 2004.

[9] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, 2003.

[10] D. R. Karger and M. Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. In *Proceedings of IPTPS*, 2004.

[11] B. Loo, J. Hellerstern, R. Huebsch, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *Proceedings of VLDB Conference*, 2004.

[12] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *Proceedings of SIGCOMM*, San Diego, California, USA, August 2001.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer system. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms*, 2001.

[14] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

[15] H. T. Shen, Y. F. Shu, , and B. Yu. Efficient semantic-based content search in P2P network. In *IEEE Transactions on Knowledge and Data Engineering*, 2004.

[16] Y. Shu, K.-L. Tan, and A. Zhou. Adapting the content native space for load balanced indexing. In *Database, Information Systems and Peer-to-Peer Computing*, 2004.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceesings of SIGCOMM*, San Diego, California, USA, August 2001.

[18] P. Zezula, M. Batko, and C. Gennaro. A scalable nearest neighbor search in P2P systems. In *Database, Information Systems and Peer-to-Peer Computing*, 2004.