Supporting Multi-dimensional Range Queries in Peer-to-Peer Systems

Yanfeng Shu, Beng Chin Ooi, Kian-Lee Tan
School of Computing
National University of Singapore
{shuyanfe, ooibc, tankl}@comp.nus.edu.sg

Aoying Zhou
Department of Computer Science
Fudan University, China
ayzhou@fudan.edu.cn

Abstract

Today's peer-to-peer (P2P) systems are unable to cope well with range queries on multi-dimensional data. To extend existing P2P systems and thus support multi-dimensional range queries, one needs to consider such issues as space partitioning and mapping, efficient query processing, and load balancing. In this paper, we describe our scheme called **ZNet***, which addresses all these issues. Moreover, we conduct an extensive performance study which evaluates ZNet against several recent proposals, and our results show that ZNet possesses nearly all desirable properties, while others typically fail in one or another.*

1 Introduction

Today's peer-to-peer (P2P) systems are unable to cope well with range queries on multi-dimensional data. Early P2P systems, such as Gnutella, mainly depend on flooding techniques for searching, thus they offer no performance guarantee; data availability cannot be ensured unless all nodes in the network are visited. While more recent systems, such as CAN [9], can guarantee data availability and search efficiency, they are mainly designed for *exact* key lookup; range queries cannot be supported efficiently (as hashing destroys data locality).

To extend existing P2P systems and thus support range queries on multi-dimensional data efficiently and effectively, the following issues need to be addressed: (a) *space partitioning and mapping*. How is a data space partitioned and mapped onto nodes? Due to the dynamicity of P2P networks, space partitioning cannot be done statically. Also, when mapping spaces onto nodes, it is preferable that data locality is preserved, as such, data that are close in their native space can be mapped to the same node or nodes that are close in the overlay network. (b) *query processing*. What overlay is based on and what strategy is employed for query processing? An overlay determines how a query is routed and at what cost; by employing the right strategies for query processing, unnecessary node visits can be avoided. (c) *load balancing*. The load (e.g., storage for data indexing and/or computational for processing queries) across each node in the system should be approximately the same.

To address the above issues, in this paper, we describe our scheme called ZNet, which has the following main features: First, the whole data space is dynamically partitioned, with subspaces (zones) *at different granularity levels*. By partitioning the space into different granularities, ZNet can handle load balancing better. When mapping zones onto nodes, to preserve data locality, Space Filling Curves (SFCs) at different orders (corresponding to the space granularities) are used. Second, range queries are efficiently supported. In ZNet, Skip Graphs [3] are extended for query routing, with each node maintaining only $O(logN)$ states (N is the number of nodes in the network). Based on the skip graph overlay, we propose an efficient range query resolution strategy, which evaluates queries in a specific way to avoid unnecessary node visits. To further improve performance, optimizations are employed during query processing to refine the search space. Third, both static and dynamic load balancing are addressed. For new nodes that join the network, appropriate joining destinations are selected - more nodes will be clustered in more densely populated areas. Also, heavily loaded nodes can migrate some of their load to lightly loaded ones.

This paper builds on top of our preliminary workshop paper [11], by extending its functionality with a more flexible space partitioning and mapping strategy (each node now can manage a space not limited to a hypercube), and the support of dynamic load balancing. Also, this paper provides an extensive experimental study, which evaluates ZNet against several recent proposals, such as Squid [10], SCRAP and MURK [7], and SkipIndex [12]. To our knowledge, this is the first comprehensive evaluation of these proposals. Our results show that ZNet possesses nearly all desirable properties: it has both low routing cost and low maintenance cost, which is independent of data dimensionality and distribution, and only increases logarithmically with the network size; it has low range search cost - the number of nodes for

routing queries is much smaller, and no node is revisited during query processing; it achieves better load balancing - dynamic load balancing can be dealt with more easily with its space partitioning strategy. On the contrary, all other proposals mentioned above typically fail in one or two of these properties.

The rest of the paper is organized as follows: Section 2 discusses related work; Section 3 presents the design of ZNet, together with its range query processing; The experimental results are presented in Section 4; And finally, section 5 concludes the whole paper.

2 Related Work

Recently, there has been much work done on supporting range queries in P2P networks. Some mainly focuses on single-attribute range queries: Andrzejak *et. al.* [1] use the inverse Hilbert mapping to map one dimensional data space (single attribute domain) to CAN's d-dimensional Cartesian space; MAAN [6] uses a uniform locality preserving hashing to map attribute values to the Chord identifier space; Mercury [5] directly operates on an attribute space, with random sampling used to facilitate efficient query routing and load balancing. Though both MANN and Mercury can support multi-attribute range queries, they do so mainly through single-attribute query resolution. Load balancing issues of range-partitioned data are considered in [7, 2].

Other work like ZNet concerns more about multi-dimensional range queries: in both Squid [10] and SCRAP [7], a multi-dimensional data space is mapped to one dimensional space by using space-filling curves as in ZNet. However, they are different since Squid and SCRAP partition the space *statically*, while ZNet partitions the space *dynamically*. In Squid and SCRAP, the partitioning level needs to be decided beforehand. Also, as Squid is based on Chord, its performance may be affected by data skewness (routing efficiency in Chord is based on the assumption that the data and nodes are uniformly distributed). SCRAP does not have this problem, as it is based on skip graphs. However, it does not focus on designing efficient query processing scheme. Another approach proposed in [7] for multi-dimensional range queries is MURK, which uses k-d trees for space partitioning and mapping. SkipIndex [12] does nearly the same way. However, they have different routing overlays. Both MURK and SkipIndex cannot cope well with dynamic load balancing. In SWAM [4], a family of distributed access methods are proposed for similarity-search, which are derived from traditional database indexing models and small-world models. Different from the above work, each node in SWAM autonomously stores its own data, as such, the model proposed in SWAM is built on data, not on nodes, and a node which has more data may need to maintain more neighbors.

3 The Design of ZNet

In this section, we shall present the design of ZNet, and see how ZNet facilitates multi-dimensional range queries.

3.1 Space Partitioning and Mapping

Space partitioning occurs when one node needs to transfer some space (data) to another node: when a new node joins, it needs to find an existing node in the network, and gets some space from this node (for the first node in the network, it covers the whole data space); when a node leaves or fails, its space needs to be supervised by another node in the network; and when a node is overloaded, it needs to transfer some of its load to another node which is less loaded. When partitioning, the space is halved in all dimensions. The whole partitioning of a space can be modelled as a tree, which we refer to as the *partition tree*. For a d-dimensional space D, each partitioning will generate 2^d subspaces at a lower tree level, each of which can be further partitioned.

To facilitate space mapping, and also data indexing, we use the following space linearization process. For each subspace generated from one partitioning, we number it by a binary string of length d in the following way: For each dimension, we test whether the subspace lies in the lower half or in the upper half of the dimension. If the subspace lies in the lower half of dimension d_i, the i-th bit of its binary representation is set to 0, otherwise the i-th bit is set to 1. We call a subspace a *zone*. A zone Z in a space D can be uniquely identified by a zone code $z_1 z_2 ... z_l$, if it is at level l of the partition tree (obtained by partitioning D l times), where z_i is the binary representation of a subspace which is at level i in the partition tree and spatially contains Z.

In fact, this equals to filling subspaces (from each partitioning) with a first order z-curve [8] [1]. And a zone's code is just its Z-address. For a space which is unevenly partitioned, it is filled with z-curves at different orders (see Figure 1 for a 2-dimensional example), which results in zones in the space having Z-addresses (in binary representation) of different lengths. By comparing only the prefix part of longer Z-addresses with shorter Z-addresses, we can order zones (leafs in the partition tree) in a space. In this way, a multi-dimensional data space is mapped to 1-dimensional index space.

When mapping zones onto nodes, we need to make sure zones managed by a node are continuous (in the sense their Z-addresses are continuous), and at the same level in the partition tree. To achieve this, multiple partitionings may occur during space transferring, and one node always passes

[1] Note we chose z-curve simply because of its simplicity. If other Space Filling Curves (e.g., Hilbert-curve) are considered, the subspace numbering will be changed accordingly. ZNet, however, still applies.

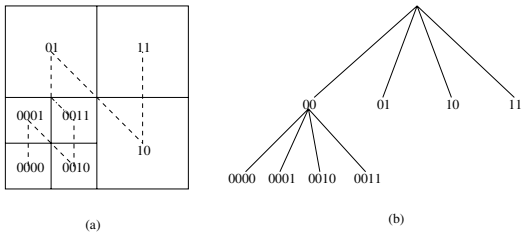


Figure 1. Space partitioning and mapping: (a) Z-curves at different orders; (b) corresponding partition tree

part or all of its continuous zones to another node to preserve data locality.

3.2 Overlay Network

Skip Graphs. In ZNet, the overlay network is based on skip graphs [3], which generalize skip lists for distributed environments. Each node in skip graphs is a member of multiple doubly-linked lists at several levels. The bottom-level list consists of all nodes ordered by their keys. At upper levels, the list in which a node belongs to is controlled by the node's membership vector, which is generated randomly. Specifically, a node is in the list L_w at level i, if and only if w is a prefix of its member vector of length i. Each node stores the addresses and keys of its left and right neighbors at each level. When searching, a node first checks its neighbors at the highest level. If there is a neighbor whose key is not past the search key, the search request is passed to the neighbor; otherwise, neighbors at a lower level are checked.

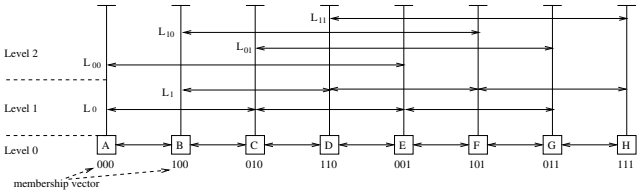


Figure 2. An example of skip graphs

An example of skip graphs is shown in Figure 2. There are eight nodes $A - H$ in the overlay, whose membership vectors are shown as in the figure (For clarity, each node has a distinct membership vector). Membership vectors decide nodes' neighbors at each level (except level 0). For node C, it has neighbors, B and D, at level 0; A and E at level 1; G at level 2. When C receives a query whose destination is F, it checks G first. Since G's key is larger than F's, C checks E next, and passes the query to E, and so on.

Query Routing. In skip graphs, each node needs to maintain $O(logN)$ states, and a search needs to take expected $O(logN)$ time. Since skip graphs were designed with one key per node, N is in fact the number of keys. This is not acceptable, as the number of keys is usually much larger than the number of nodes in a system. ZNet extends skip graphs by assigning continuous zones to nodes while ensuring system-wide load balancing, thus it can leverage the useful routing properties of skip graph even when there is a data skew. In ZNet, each node maintains only $O(logN)$ neighbors, where N is the number of nodes. Routing in ZNet is a little more complicated, as each node has only incomplete knowledge about space partitioning. However, as we shall see in the experiments (Section 4), this incomplete knowledge does not affect the search performance significantly, the search cost is still bounded by $O(logN)$. When given a search point, a node may transform the point to a Z-address which is only a prefix of the key's full Z-address. Nevertheless, with each routing step, the query is routed to a node which has more knowledge about the destination's partitioning status. Before describing the routing procedure at detail, we need to introduce the following three terms:

- **Z Level**: For a zone Z, its Z Level is its level in the partition tree. As a node always covers continuous zones at the same partition level, a node's Z Level is its zones' Z Level.
- **Z Range**: For a node A, its Z Range is the interval of Z-addresses of zones A covers, defined as [zMin, zMax]. Given a Z-address z, we use the following notations to denote its relationship with A: if $zMin \leq z \leq zMax$, z is covered by A, noted as $z \in A$; else if $z < zMin$, then $z < A$; else, $z > A$.
- **Routing Table**: Each node maintains its neighbor information in a table RT with m entries (m is at most $\lceil logN \rceil$). The ith entry in the table contains the identities and Z Ranges of its neighbors (left neighbor lN and right neighbor rN) at level i of the skip graph.

The routing procedure is shown in Figure 3. When a node A receives a search request for a point p, it first calls getZAddress(), transforming p to a Z-address z ($z_1 z_2 ... z_l$) based on its local knowledge about the partition tree. getZAddress(p) is computed in the following way: first, a zone at Z Level 1 (with Z-address z_1) in the partition tree, which covers p, is decided by comparing each dimension of p with 0.5 (if $p_i \geq 0.5$, $z_{1,i} = 1$; else $z_{1,i} = 0$); next, a zone at Z Level 2 (with Z-address $z_1 z_2$) is decided, z_2 is obtained by similarly comparing each dimension of p with the centroid of z_1, and so on, until node A's Z Level l is reached. After getting p's Z-adddress z, A compares z with its Z Range, if $z \in A$, the search destination is itself; otherwise, it calls findCloserNode(), which follows the routing

```
A.Routing(p)
1.  z = getZAddress(p);
2.  if z ∈ A
3.      return A;
4.  else B = findCloserNode(z);
5.      return B.Routing(p);

findCloserNode(z)
1.  for i = m downto 0
2.      if z < RT[i].lN || z ∈ RT[i].lN
3.          return RT[i].lN;
4.      if z > RT[i].rN || z ∈ RT[i].rN
5.          return RT[i].rN;
```

Figure 3. The routing algorithm

process of skip graphs by examining the neighbors from its Routing Table, and then passes the search request to one of its neighbors without overshooting the search point.

Following the example in Figure 2, assume the space is partitioned among $A - H$ like this: A(00,01) (which means A contains Z-addresses 00 and 01), B(1000, 1001), C(101000), D(101001), E(101010), F(101011), G(1011), J(11). Suppose A receives a point query, whose destination is node D. Since A's zones are at z-level 1, it can only transform the point to Z-address (10) according to Z-address transformation process (A has no idea about the complete space partitioning status). For Z-address (10), all of A's neighbors, B, C, and E, are qualified. We may choose one based on consideration of load or network proximity. Suppose E is chosen, the query will be forwarded to E. When the query arrives at E, another Z-address transformation will be done again, and at this time, full Z-address (101001) of the search point is obtained (since zones covered by both E and D are at the same Z Level). By choosing D from E's neighbors as the forwarding node, the query is finally resolved.

Node Join and Leave. When a new node A wants to join the network, it needs to find an existing node B in the network and split B's space (we defer the discussion on how B is determined to Section 3.4). Since B always passes A the lower or upper part of its Z Range (randomly decided), A can establish its neighbors at level 0 in skip graphs. For A's neighbors at other levels, they are determined by the underlying skip graphs. After all of its neighbors are decided, A joins the network. As shown in [3], the join operation takes expected $O(logN)$ time and $O(logN)$ messages.

To deal with node failure or departure, each node in ZNet maintains a redundant neighbor list of length $2r$, which includes the closest r left and r right nodes along the bottom level list in skip graphs. If a node notices its level 0 left or right neighbor fails (detected by periodically sending 'heartbeats'), it replaces the neighbor with the first live left or right neighbor in its redundant neighbor list (data loss can be avoided by storing replicates in these neighbors). A background stabilization process runs periodically at each node to fix neighbors at upper levels in skip graphs. To improve system performance, for a node which voluntarily leaves the network, it may pass its data to one of its level 0 neighbors, and notify all of its neighbors before leaving.

3.3 Distributed Query Processing

ZNet supports both point queries and range queries. Point queries, as a special case of range queries, can be resolved by simply routing queries to nodes which cover the points. Thus, we mainly focus on resolving range queries. One method to resolve a range query is to convert the query range (QR) to a set of Z Ranges which are covered by QR, and then route a request to each node which contains $zMin$ of each Z Range. This method can be inefficient if the number of Z Ranges is very high. Also, it precludes the possibility for possible query optimizations, as a node may contain several Z Ranges which are included in the set. Further, it requires that the whole data space be statically partitioned. Thus, in ZNet, we use a different way to resolve a range query: Initially, a range query is converted to a set of zones, which is a superset of zones covered by the query range. As the query is routed, this superset is refined. To avoid unnecessary visits, the query is routed along two opposite directions - in one direction, the query is routed to nodes which contain zones of larger Z-addresses than zones' of the current node; in the other direction, the query is routed to nodes which contain zones of smaller Z-addresses than zones' of the current node.

The pseudo-code is shown in Figure 4. Given a query QR, node A first computes a superset of zones covered by the query range, $[zL, zH]$. Then, A checks whether its Z Range overlaps with $[zL, zH]$: if not, it forwards the query to a node which covers zL or zH (randomly decided); otherwise, it checks whether the space it covers overlaps with QR by isOverlap(QR) and returns a message to the query initiator if they overlap. Meanwhile, it refines the search range by computing the next bigger Z-Address zL' and smaller Z-address zH' (as compared to its own Z Range) which overlap with QR. By forwarding requests to the node which contains zL' and the node which contains zH' respectively, the search is recursively resolved.

During query processing, we may need to get a zone's space, which can be computed recursively as follows: For a zone Z of Z-address $z_1z_2...z_l$, we first compute the space of a zone of Z-address z_1, which in turn can be used to determine the space of a zone of Z-address z_1z_2. The space of a zone of Z-address $z_1z_2...z_i$ is computed by first deciding its

A.**RangeSearch**(ll, hr, QR)
// ll and hr are initially the low left and
// high right point of query range QR;
1. zL = getZAddress(ll)
2. zH = getZAddress(hr)
3. **if** A's Z Range $[zMin, zMax]$ doesn't overlap $[zL, zH]$
4. B = FindCloserNode(zL or zH)
5. B.RangeSearch(ll, hr, QR)
6. **else if** isOverlap(QR)
7. **return** B
8. **if** $zMax \in [zL, zH]$
9. zL' = getLowestOverlapZ(QR)
10. B = FindCloserNode(zL')
11. reset ll to ll' (the low left point of zL''s space)
12. B.RangeSearch(ll', hr, QR)
13. **if** $zMin \in [zL, zH]$
14. zH' = getHighestOverlapZ(QR)
15. B = FindCloserNode(zH')
16. reset hr to hr'(the high right point of zH''s space)
17. B.RangeSearch(ll, hr', QR)

Figure 4. The range search algorithm

centroid $C_{i,1}...C_{i,d}$ based on its radius r_i and z_i. Suppose z_i is represented by $z_{i,1}...z_{i,d}$, if $z_{i,j} = 1$, $C_{i,j} = C_{i-1,j} + r_i$; else $C_{i,j} = C_{i-1,j} - r_i$, where $C_{0,j} = 0.5$, $r_j = 0.5/2^j$, $j = 1..d$. From the computation of a zone's space, we can get an important property of zones: for two zones Z_1, and Z_2, if Z_1's Z-address is a prefix of Z_2's Z-address, then Z_2 is covered by Z_1. We use this property to efficiently test isOverlap() and compute getLowestOverlapZ() and getHighestOverlapZ().

3.4 Load Balancing

In ZNet, the load on a node is measured by the amount of data the node indexes, and load balancing is addressed from the following two aspects: first, for new nodes, appropriate joining destinations are chosen. When a node joins the network, it randomly chooses several of its data points as possible joining destinations, and the one whose corresponding destination node has the heaviest load is chosen as the joining destination, which will pass some space (zones) to the new node, meanwhile ensuring load on each is nearly the same.

Second, at run-time, heavily loaded nodes can migrate some of their load to lightly loaded ones. A node is said to be heavily loaded if its load is larger than $\overline{L} + \delta\overline{L}$ and lightly loaded if its load is less than $\overline{L} - \delta\overline{L}$, where $\overline{L}$ is average load, and δ is a variable which represents the trade-off between the amount of load moved and the quality of balance achieved. By sampling loads on its neighbors, each node can estimate the average load $\overline{L}$. To balance the load at run-time, each node periodically exchanges its load information with its neighbors (We differentiate two kinds of neighbors of a node: neighbors whose Z-addresses are continuous with the node's Z-addresses are the node's *close neighbors*; all other neighbors are the node's *far neighbors*). For a lightly loaded node, it will average its load with its more loaded close neighbor, as such, close neighbors of a lightly loaded node will be lightly loaded with a high probability. For a heavily loaded node, it will first try to transfer part of its load to its close neighbors. In case it is not successful (e.g., all of its close neighbors are heavily loaded), it will send requests to one of its far neighbors to find a lightly loaded node in the network, which will gracefully leave the network and rejoin at the location of the heavily loaded node. Heuristics are adopted to find a lightly loaded node: when a heavily loaded node sends requests to one of its far neighbors, the far neighbor which is less loaded is always chosen, in the hope that the far neighbor is itself lightly loaded or it is close to lightly loaded nodes (as lightly loaded nodes always average their load with their close neighbors). With the randomness of skip graphs, a heavily loaded node can find a lightly loaded node in the network with a high probability.

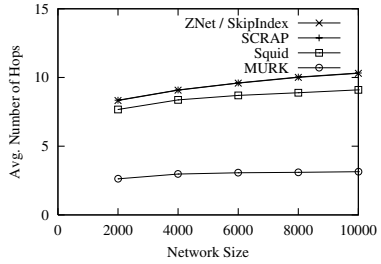
4 Experimental Results

In this section, we evaluate ZNet in routing cost, maintenance cost, range search cost, and load balancing, by comparing it with several existing systems - Squid, SCRAP and MURK, and SkipIndex. The evaluation is conducted via simulations with up to 10,000 nodes. Two kinds of synthetic datasets are generated for the experiments, each of increasing dimensionality (from 2 to 20): one is skewed datasets based on normal distribution; the other is uniform datasets. By default, we use the synthetic data set with skewed 8-dimension 300,000 data points, and 6,000 nodes in the network.

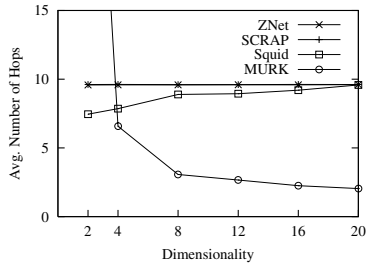
4.1 Routing Cost

Routing cost mainly depends on the underlying overlay networks. Squid is based on Chord; MURK is based on CAN [2]; SCRAP, ZNet, and SkipIndex are all based on Skip Graphs: SCRAP can be regarded as a static version of ZNet in query routing; ZNet and SkipIndex are similar in their query routing, thus we mention ZNet only in the following. One concern in this part is whether the *add-on dynamicity* of ZNet affects its routing performance.

[2]Though three implementations of MURK (MURK-CAN/ MURK-Ran/ MURK-SF) are described in [7], in this section, we only consider MURK-CAN, as it is the most basic implementation of MURK.

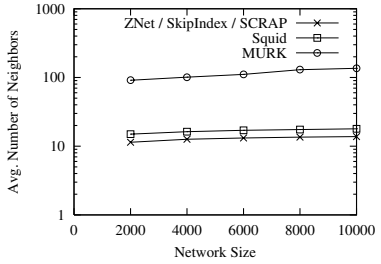


(a) Dimensionality = 8

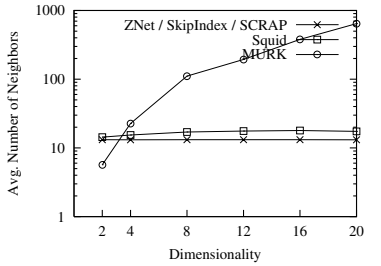


(b) Network size = 6000

Figure 5. Routing cost measurement



(a) Dimensionality = 8



(b) Network size=6000

Figure 6. Maintenance cost measurement

For comparison, we assume the maximum partition level of ZNet is globally known in SCRAP and Squid. First, we use skewed datasets to test the routing performance, which is measured by the number of hops between two randomly selected nodes, averaged over 10 times the network size. Figure 5 shows the results, in which (a) depicts the effect of network size, while (b) depicts the effect of dimensionality. As shown from the figure, the routing performance of ZNet is comparable to SCRAP's; both are independent of dimensionality and increase logarithmically with the network size, which means the add-on dynamicity of ZNet does not affect its routing performance. Also, in the experiment, we find that the skewness of data affects Squid's routing performance. As shown in Figure 5(b), the routing cost of Squid increases with the dimensionality. This may be because, with higher dimensionality, data distribution becomes more skewed, which leads to nonuniform node distribution, and thus deteriorates the routing performance (Chord's performance relies on uniform node distribution in the overlay). To confirm this, we use uniform datasets next and find that its routing cost does not change with the dimensionality any more (we do not show this result in the paper, due to space constraints).

Figure 5 may give us the illusion that MURK performs best among all systems. In fact, it performs badly when the dimensionality is low, as indicated in (b) (The data dimensionality in (a) is 8). Also, the much lower routing cost for MURK is derived from maintaining a large number of neighbors when the dimensionality is high, which incurred high maintenance cost as we shall discuss in the following subsection.

4.2 Maintenance Cost

When a node joins or leaves, the overlay network needs to be reconfigured, whose cost is mainly decided by the number of neighbors the node maintains. Thus, we discuss maintenance cost in terms of the number of neighbors maintained by each node. Figure 6 shows the average number of neighbors maintained by each node in each system (ZNet, SkipIndex, and SCRAP have same maintenance cost, thus in this part we mention ZNet only). As shown from the figure, the average number of neighbors at each node in ZNet increases logarithmically with the network size, which is independent of dimensionality. In Squid, the average number of neighbors slightly increases with both network size and dimensionality, which is a little higher than ZNet. This may partly explain relatively higher routing cost of ZNet in Figure 5. We also test ZNet and Squid on uniform datasets, and find that, ZNet is not affected by the data skewness at all,

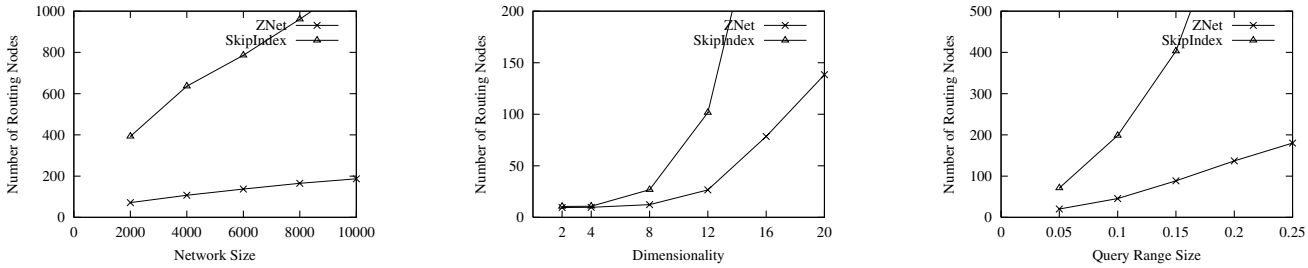


Figure 7. Range search cost measurement

while Squid does. When the data distribution is uniform, the number of neighbors at each node in Squid is nearly the same. For MURK, we notice that, when the dimensionality is low, each node maintains only a few neighbors. However, when the dimensionality is high, each node needs to maintain a large number of neighbors.

4.3 Range Search Cost

In this part, we will not compare all systems, as it may not be meaningful to compare two systems with different routing overlays in their range search cost (routing cost directly affects range search cost). In the following, we only compare ZNet and SkipIndex[3], which are both built on skip graphs, but have different range search strategies.

For the comparison between ZNet and SkipIndex, we measure *the number of routing nodes*. Routing nodes are such nodes that are visited as part of the routing process, and they themselves do not contribute answers to the query results. Four factors are involved in range searches: network size, the dimensionality, the query range size and the data distribution. As results for uniform datasets present similar trends to skewed datasets, we omit them here. Thus, only the other three factors are considered. Queries for experiments are generated according to the data distribution in the space, which can be initiated from any node in the network. For each measurement, results are averaged over 200 randomly generated range queries with fixed range sizes, each is initiated from 100 random nodes in the network.

Figure 7(a) - (c) show the effect of network size, dimensionality, and query range size on range search cost respectively. In (a), query range size at each dimension is fixed to be 0.2, and data sets are the same for all network sizes. As shown in the figure, for both systems, the number of routing nodes increases with the network size. In (b), the query selectivity is fixed for all dimensionalities. Since query range size for the same selectivity increases rapidly with higher dimensionality, the number of routing nodes increases quickly. In (c), query range size at each dimension is varied from 0.05 to 0.25. As query range size increases, more nodes' spaces overlap with the query range, thus incurring larger number of routing nodes. In all these measurements, the number of routing nodes in ZNet is much smaller than in SkipIndex. This is mainly because a node in SkipIndex may be revisited many times during query processing for routing a query to remote regions for further processing. ZNet avoids this problem with its two-way query resolution strategy.

4.4 Load Balancing

We measure load balancing mainly by the amount of data indexed by each node in the network. Initially, we randomly assign data in a dataset to all nodes. For a node joining the network, it randomly chooses several of its data points as joining destinations, and the one with the heaviest load is chosen as the final destination. As shown in Figure 8, where nodes are sorted in decreasing order according to the number of data indexed by them, the load distribution becomes more balanced as the number of tries increases. However, when the number of tries is larger than 5, only marginal improvement can be observed. The number of appropriate tries is mainly decided by the system setup; also, we need to consider the tradeoff of this strategy - the join cost/overhead increases with more tries.

Two systems are compared with ZNet in load balancing[4]: MURK, and SkipIndex. CAN is also compared as a representative of a "worst case" scheme. For comparison, we use 5 tries for all systems. In CAN, a random point in the space is chosen as the joining destination and the destination always splits its space in half and assigns one half to the new node. This may result in bad load distribution when the data is skewed. As shown in Figure 9, 65% of data are indexed by only 5% of nodes, the load distribu-

[3]SCRAP is not considered here, since [7] only provides a very naive query processing strategy for SCRAP, which first converts a multi-dimensional range query to a set of 1-d range queries, and then routes each of these 1-d range queries to appropriate nodes.

[4]We do not compare ZNet with SCRAP and Squid. In theory, they can achieve the same load balancing as ZNet.

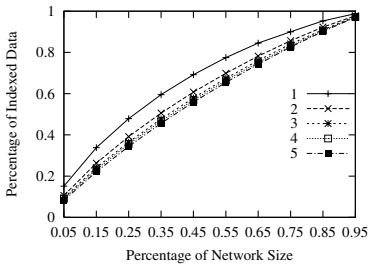


Figure 8. Different Tries At Node Join

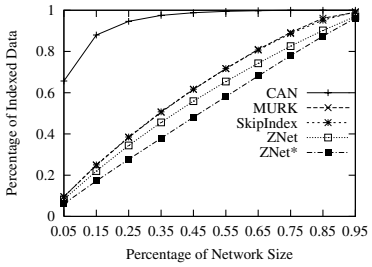


Figure 9. Comparison of Systems

tion among nodes is severely unbalanced. MURK improves on CAN by always splitting the space by load. SkipIndex does similarly. The difference is, in MURK, the dimensions are used cyclically in splitting, while in SkipIndex, the dimension with the maximum span is chosen as the splitting dimension. However, from the figure, we do not observe much difference between them. Though ZNet also splits the space by load, it does not require the space managed by each node to be a hyper-rectangle as in MURK and SkipIndex.

ZNet can handle run-time load balancing well, by passing some zones to its close neighbors, or selecting a lightly loaded node to leave and rejoin the network. However, This may not be easily done in MURK and SkipIndex. As each node in MURK and SkipIndex keeps a hyper-rectangle, when a node wants to transfer its load, it needs to find a neighbor which abuts along one dimension but shares all other dimensions with itself. Sometimes, such a neighbor may not exist. Figure 9 also shows the effectiveness of dynamic load balancing in ZNet (ZNet*): each node in the system nearly has the same load. To evaluate the efficiency of our dynamic load balancing algorithm, we did some experiments and get the following results: for a network of 2000 nodes, average load is 150, the system only needs 18 rounds to arrive at load balancing for δ=10%. Fewer rounds are needed for larger δ.

5 Conclusion

In this paper, we have addressed the problem of processing range queries on multi-dimensional data in P2P environments. We have proposed a load-balanced P2P system, ZNet, which efficiently supports range searches on multi-dimensional data space. We conducted an extensive performance study comparing ZNet against several existing proposals with similar features. Our results showed that ZNet can achieve good load-balancing, efficient query processing at low maintenance overhead, while existing schemes typically fail in one or two of these aspects.

References

[1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proceedings fo the Second IEEE International Conference on Peer-to-Peer Computing*, 2002.

[2] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. In *Twenty-Third ACM Symposium on Principles of Distributed Computing*, 2004.

[3] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[4] F. Banaei-Kashani and C. Shahabir. Swam: A family of access methods for similarity-search in peer-to-peer data networks. In *Proceedings of the Thirteenth ACM conference on Information and knowledge management*, 2004.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM*, 2004.

[6] M. Cai, M. Frank, J. Chen, and P.Szekely. Maan: A multi-attribute addressable network for grid information services. In *4th International Workshop on Grid Computing*, 2003.

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

[8] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, 1984.

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

[10] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distriubted systems. In *12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

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

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