Adapting the Content Native Space for Load Balanced Indexing

Yanfeng Shu[1], Kian-Lee Tan[1], and Aoying Zhou[2]

[1] School of Computing
National University of Singapore, Singapore
{shuyanfe,tankl}@comp.nus.edu.sg
[2] Department of Computer Science
Fudan University, China
ayzhou@comp.nus.edu.sg

Abstract. Today, there is an increasing demand to share data with complex data types (e.g., multi-dimensional) over large numbers of data sources. One of the key challenges is sharing these data in a scalable and efficient way. This paper presents the design of ZNet, a P2P network for supporting multi-dimensional data. ZNet directly operates on the native data space, which is partitioned dynamically and mapped to nodes within the network. Particular attention is given to reduce load imbalance among nodes due to skewed data distribution. Results from an extensive simulation study show that ZNet is efficient in routing and processing range queries, and provides good load balancing.

1 Introduction

Today, there is an increasing demand to share complex data types (e.g., multi-dimensional) and to support complex queries (e.g., range queries). For example, in grid information services, computing resources are typically characterized by multiple attributes like the type of operating systems, CPU speed and memory size. It is not uncommon for such a system to search for resources that meet multiple attribute requirements, e.g, a resource with LINUX operating system and CPU speed of 1-2 GFlop/sec. As another example, in sensor networks, data or events are also characterized by a set of attributes. For a sensor network that monitors the weather, a typical query may be like this, to find regions whose temperature falls between [0,10] degrees, wind speed in [30,40] nautical miles, and so on.

One of the key challenges for these systems is to share these multi-dimensional data in a scalable and efficient way. Due to a large number of data sources, a centralized approach is always not desirable, sometimes, it may not even be feasible (e.g., in sensor networks). Though P2P technology, as an emerging paradigm for building large-scale distributed systems, could be used for sharing data in a scalable way, today's P2P systems are unable to cope well with complex queries (range queries) on multi-dimensional data. Early P2P systems, such as Gnutella[6], mainly depend on flooding techniques for searching, thus they offer no search guarantee; moreover, data availability could not be ensured unless all nodes in the network are visited. While more recent systems, such as

Chord[13] and CAN[9], can guarantee data availability and search efficiency, they are mainly designed for *exact* key lookup; range queries cannot be supported in most cases.

In this paper, as one of the initial attempts to address the above challenges, we present the design of ZNet. ZNet directly operates on the native data space, which is partitioned and then mapped to nodes within the network. ZNet focuses on addressing two important issues. The one is load balancing. We want to make sure that each node contains nearly the same amount of the data. Since data distribution in multi-dimensional data space may not be uniform, if the space is partitioned and assigned to nodes evenly, some nodes may contain more data than others. In ZNet, this issue is addressed by dynamically choosing subspaces which may be densely populated to be split further, so that space could be partitioned in a way that follows the data distribution.

The second issue is to facilitate efficient indexing and searching. Our basic idea is to partition the space in a quad-tree-like manner with some subspaces being recursively partitioned. To facilitate searching, all subspaces resulted from one partitioning are ordered by a first order Space Filling Curve (SFC). As such, the whole data space(multi-dimensional) is mapped to 1-dimensional index space by SFCs at different orders. Two data that are close in their native space are mapped to the same index or indices that are close in the 1-dimensional index space, which are then mapped to the same node or nodes that are close together in the overlay network. Any SFC could be used for the mapping. In our current implementation, Z-ordering is chosen mainly due to its simplicity. Skip graph [2] is extended as the overlay network topology (nonuniform node distribution in ZNet makes DHT-based P2P networks (such as Chord) unsuitable). From an extensive simulation study, it shows that ZNet is good in load balancing when the data distribution changes little, and efficient in supporting range searches.

The rest of the paper is organized as follows: Section 2 discusses the related work; Section 3 presents the system design in space partitioning, searching and load balancing; The experimental results are presented in Section 4; And finally, section 5concludes the whole paper.

2 Related Work

Existing P2P systems can be generally classified as unstructured or structured. For unstructured systems (such as Gnutella [6]), there has no guarantee on data availability and search performance. Therefore, research on range query support in P2P is mainly focused on structured systems.

There are two kinds of structured P2P systems: DHT-based and skip-list based. DHT-based systems, like Chord [13], CAN [9], Pastry [10], and Tapestry [15], use a distributed hash table to distribute data uniformly over all nodes in the system. Though DHT systems can guarantee data availability and search efficiency on exact key lookup, they cannot support range searches efficiently, as hashing destroys data locality. Skip graph [2] and SkipNet [7] are two skip-list based systems, which can support range queries. However, they did not address how data are assigned to nodes. As such, there is no guarantee about data locality and load balancing in the whole system.

In [3], Chord and skip graph are combined into one system to support range searches. Chord is used to assign data to nodes, while skip graph is used to do range searches.

Though load balancing can be ensured in [3], searching is not efficient, which is at a cost of $O(\log m)$, where m is the number of data.

Most work supports range queries by drawing its inspiration from multi-dimensional indexing in the database research [5]. Specifically, locality-preserving mapping is used to map data that are close in their data space to nodes that are close in the overlay network. For example, in [1], the inverse Hilbert mapping was used to map one dimensional data space (single attribute domain) to CAN's d-dimensional Cartesian space; and in [12], the Hilbert mapping was used to map a multi-dimensional data space to Chord's one dimensional space. Though [12] can support multi-dimensional range queries, its performance is poor when the data is highly skewed as the node distribution (which follows data distribution) is not uniform any more. DIM [8] supports multi-dimensional range queries in sensor networks by using k-d trees to map multi-dimensional space to a 2-d geographic space. Load balancing, unfortunately, is not addressed in DIM.

Different from above work, MAAN [4] uses a uniform locality preserving hashing to map attribute values to the Chord identifier space, which is devised with the assumption that the data distribution could be known beforehand. Multi-attribute range queries were supported based on single-attribute resolution. In our work, we do not assume any a prior knowledge on the data distribution, and load balancing is achieved fully based on heuristics that partition dense subspaces.

Still, there are some other orthogonal work. pSearch[14], proposed for document retrieval in P2P networks by extending CAN, bears some similarity to our work in load balancing. However, its main focus is to retrieve some relevant documents, and not to support range searches. [11] proposed a framework based on CAN for caching range queries. By caching the answers of range queries over the network, future range queries can be efficiently evaluated.

3 ZNet

The whole system consists of a large number of nodes, each publishing its data objects (multi-dimensional) and sending queries for other data objects over the network. Range query is the kind of query ZNet is mainly interested in.

To support range queries efficiently, data that are close in their native space needs to be mapped to nodes that are close in the network. In ZNet, a kind of locality preserving mapping is used, and multi-dimensional data space is mapped to 1-dimensional index space by z-curves at different orders. And also, by extending skip graph as the overlay network, queries can be routed efficiently in ZNet, with each node maintaining $O(logN)$ neighbors (N is the number of nodes). Besides query processing, ZNet also addresses the load balancing issue. Two strategies are employed in ZNet to reduce load imbalance. All this will be described next in detail.

3.1 Space Partitioning and Mapping

In ZNet, data space is partitioned in a way as in the generalized quad-tree, that is, each partitioning halves the space in all dimensions. As such, for d dimensions, 2^d subspaces are generated from one partitioning. We call each of such subspaces a *zone*.

Partitioning always occurs when a new node joins and the joining destination (an existing node in the network) has only one zone; if the joining destination has more than one zone, it just passes part of its zones to the new node. Zones from one partitioning are at the same level, which is one level lower than the level of the zone where the partitioning occurs. For the first node (and also the node in the network), it covers the whole data space (at level 0).

By filling zones (from one partitioning) with a first order z-curve, each zone which is at a certain level (call *z-level*) corresponds to a *z-value* in $0..2^d-1$ (for d-dimensional space), which can be computed in the following way: suppose the centroid before partitioning (at z-level i) is $(C_{i,0}, C_{i,1}, ..., C_{i,d-1})$ (for z-level 0, the centroid is (0.5, 0.5, ..., 0.5)), the centroid of a new zone (at z-level $i+1$) generated from partitioning is $(C_{i+1,0}, C_{i+1,1}, ..., C_{i+1,d-1})$, then the new zone's z-value at z-level $i+1$ is $(b_0 b_1 .. b_{d-1})_2$, where b_k is 0, if $C_{i+1,k}$ is less than $C_{i,k}$; otherwise it is 1 ($k = 0..d-1$).

A zone in the space can be uniquely identified by its *z-address*. For a zone Z at lth z-level, its z-address will be like $z_1 z_2 ... z_l$, where z_i is the z-value's binary representation of a zone which is at i z-level and contains Z. Z's z-address can be recursively constructed: first, z_1 is decided in the same way as the above z-value computation by comparing Z's centroid with the centroid (0.5, 0.5, ..., 0.5) at z-level 0; then z_2 is decided by comparing Z's centroid with the centroid of the zone of z-value z_1, and so on, until the level is l.

The z-address of a point in the space is the same as the z-address of a zone which covers the point and is at the lowest z-level. Since the space is unevenly partitioned, z-addresses of two points may be of different lengths. When comparing two z-addresses of different lengths, only the prefix part of the longer one is compared to the shorter one.

Figure 1 illustrates the space partitioning process in a 2-dimensional data space. In the figure, (a) is the initial state of the data space (z-level 0). After the first partitioning (b), four zones are generated with z-values from 0 to 3 (corresponding z-addresses are from 00 to 11). The new zones are at z-level 1. In (c), zone 00 (the zone's z-address is 00) is further partitioned, forming the second level (z-level 2). Suppose zone 0010 is partitioned again, the third level (z-level 3) will be formed.

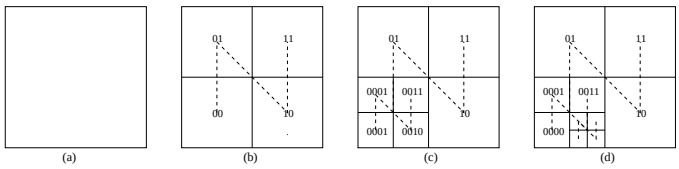

Fig. 1. Z-curves at different levels (0-3)

In sum, by z-curves at different levels (lower level z-curves correspond to higher order of z-curves), multi-dimensional data space is mapped to 1-dimensional index space. Meanwhile, this 1-dimensional space is mapped to nodes in the network. In ZNet, each node always contains continuous zones (zones are continuous in the sense that their z-addresses are continuous)

3.2 Query Routing and Resolving

Since routing in ZNet is based on skip graph[2], in this subsection, we will give a brief description of skip graph first, then we describe query routing and resolving in ZNet in detail.

Skip Graph Skip graph generalizes skip list for distributed environments. Each node in a skip graph is a member of multiple linked lists at ⌈logN⌉ skip-levels, where N is the number of nodes. The bottom skip-level is a doubly-linked list consisting of all nodes in increasing order by key. Which lists a node belongs to is controlled by its membership vector, which is generated randomly. Specifically, a node is in the list L_w at skip-level i, if and only if w is a prefix of its member vector of length i.

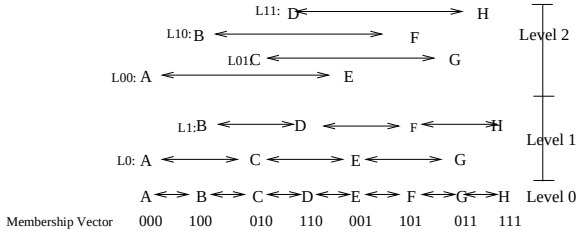

Fig. 2. A skip graph with 3 skip-levels

Figure 2 gives a example of a skip graph with 3 skip-levels. In the figure, there are 8 nodes (from A to H),each of which has one key (not shown in the figure). For simplicity, membership vectors of nodes are distinct, which are chosen from 000 to 111 (in implementation, the membership vectors are randomly generated, which could be same). At skip-level 0, all nodes are doubly-linked in sequence by their keys. At skip-level 1, there are two lists: L_0, L_1. Since the first bit of the membership vectors of nodes A, C, E, G is 0, these nodes belong to L_0. So are nodes for other lists.

Each node (except the first and the last node) in a list has two neighbors: left neighbor and right neighbor. For example, in Figure 2, node C has two neighbors B, D at skip-level 0. At skip-level 1, in L_0, it also has two neighbors A, E. At skip-level 2, it has only one neighbor G. All neighbors of a node form the node's routing table. When searching, a node will first check its neighbors at the highest skip-level. If there is a neighbor whose key is not past the search key, the query will be forwarded to the neighbor; otherwise, neighbors at a lower skip-level are checked. For example, in Figure 2,

suppose node C receives a query, whose destination is F. C will first check its neighbor at skip-level 2, G. Since G's key is larger than F's key, searching will go down to skip-level 1. Among C's neighbors at skip-level 1 (A and E), E is qualified, whose key is between C's key and the search key F. The query will be forwarded to E, and so on. The search operation in a skip graph with N nodes takes expected O(log N) time. Note each node has only one key.

Query Routing In ZNet Skip graph was proposed to handle range queries with one key per node, thus each node needs to maintain $O(logm)$ state, where m is the number of keys. In addition, in skip graph, there is no description about how keys are assigned to nodes in the system, thus making no guarantee about system-wide load balancing.

In ZNet, since zones mapped to each node are continuous (each node covers continuous z-addresses), and also, load balancing is ensured among nodes, ZNet can extend skip graph for query routing while not having its problems. In ZNet, each node maintains only O(log N)state, where N is the number of nodes.

When given a search key (a point), a node will first transform the point to a z-address, which is then compared to z-addresses covered by the node's neighbors as defined in skip graph. Complexities for routing in ZNet rise in that the z-address of a search point may not be able to be fully resolved initially due to uneven space partitioning.

For example, in Figure 3, the space is partitioned among 8 nodes, A(00,01)(A contains z-addresses 00 and 01), B(1000,1001), C(101000), D(101001), E(101010), F(101011), G(1011), J(11). The membership vector of each node is shown as in the figure 2, thus, A has 3 neighbors B, C, E; B has 4 neighbors, and so on. 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), two of A's neighbors are qualified: C, E. At current implementation, we just randomly choose one. Suppose E is chosen, and 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 same z-level). By choosing D from E's neighbors as the forwarding node, the query is finally resolved.

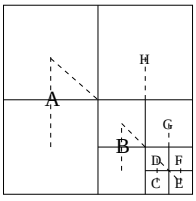

Fig. 3. A routing example

Therefore, given a search point, a node may only get a prefix of the point's full z-address, due to incomplete knowledge about space partitioning. With each routing step, however, the point's z-address will become more refined. The routing convergence can be ensured, since with each routing, the query is routed closer to the destination. However, the routing cost in ZNet may be a little worse than O(logN), where N is the number of nodes. In the worst case, the cost could be O(l*logN), where l is the deepest z-level in the space.

Range Query Resolving In ZNet, range queries are resolved in a recursive way. The Algorithm is shown in Figure 5, including two parts(A and B).

For d-dimensional space, a range query QR will be like $([l_0, h_0], ..., [l_{d-1}, h_{d-1}])$. When a node receives such a query, it will first decide the routing z-level (l), whose space covers the query range (line 1 in A). For a node, besides its own space (it is responsible for), it also has knowledge about the spaces which cover its space. For example, figure 4 shows the network status for a 2-d space after 5 nodes joining. In the figure, nodes A, B are at z-level 2 (zones covered by them are at z-level 2). So are for nodes C, D. However, nodes A, B and nodes C, D have different knowledge about spaces. For node A, its own space, level 2 space, level 1 space are shown respectively in (a)'s, (b)'s, (c)'s shaded area. Node B's level 1 space and level 2 space are same as node A's.

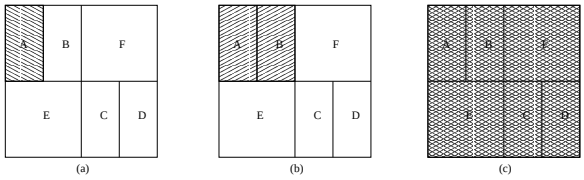

Fig. 4. Level Spaces

Based on the z-level l, a node can compute the smallest and largest z-address of QR: the smallest z-address of QR (z*L*) is the z-address of point$(l_1, ..., l_d)$, and the largest z-address of QR (z*H*) is the z-address of point $(h_1, ..., h_d)$. (line 2-3 in A)

Then the query is routed at l, checking nodes whether their l level space overlaps with QR. If a node whose l level scope space does not overlap with QR, it needs to find a neighbor from its routing table which is closer than itself to the lowest z-address (line 1-3 in B); else the node checks whether it is at z-level l, if it is at l (this means that the node's own space overlaps with QR), it will send a message to the query initiator (line 5-6 in B); if it is not at l , the query will go down one lower z-level and repeat the whole process (line 8-10 in B).

Two methods are employed about how a query is routed at a certain z-level when a node's level space overlaps with the range. One is *one-way*, that is, the query is always routed from the smallest z-address to the largest z-address. For this method, unnecessary

visits may be incurred. Another method is *two-way*, the query is partitioned into two parts which will be routed at a z-level along opposite directions to avoid unnecessary visits: in one direction, the query is always forwarded to nodes which contain larger z-addresses than the current node's at the z-level; in the other direction, the query is always forwarded to nodes which contain smaller z-addresses than the current node's at the z-level. Figure 5 only shows two-way method (line 11-18 in B).

Part A: N**.RangeSearch(QR)**

1. l=DecideRouteZLevel(QR);
   // decide the lowest and highest z-address of QR
2. zL = LowestZAddress (QR, l)
3. zH = HighestZAddress (QR, l)
4. RangeSearch1(QR, zL, zH, l)

Part B: N**.RangeSearch1(QR,**zL**,**zH**,**l**)**
   // QR is the query range;
   // zL and zH are the lowest and highest
   // z-addresses of QR at l;

1. If N at z-level scope space doesn't overlap $zL - zH$
2. M = FindCloserNode(zL)
3. M.RangeSearch1(QR, zL, zH, l)
4. else
5. If N is at l
6. send message to query initiator
7. else
8. l=l+1;
9. QR = QR ∩ N' l level space
10. N.RangeSearch(QR)
   // route at two-way;
11. If $zL - zH$ contains the largest z-address covered by N at l
12. reset zL
13. M = FindCloserNode(zL)
14. M.RangeSearch1(QR, zL, zH, l)
15. If $zL - zH$ contains the smallest z-address covered by N at l
16. reset zH
17. M = FindCloserNode(zH)
18. M.RangeSearch1(QR, zL, zH, l)

Fig. 5. Range Search in ZNet.

3.3 Node Join and Leave

When a new node joins the network, it needs to find an existing node X in the network to get some space it is responsible for (How X is decided is described in next subsection). After X splits its space, the new node will build its routing table by selecting neighbors in the network, according to its membership vector which is generated randomly (maybe same as another node's membership vector). The join cost of a node is at $O(logN)$.

For example, in Figure 3, suppose node J joins the network and node B is chosen to split the space. J will first insert itself in skip-level 0 (in the skip graph). Suppose z-addresses covered by B are smaller than ones covered by J, J will choose B and C as its skip-level 0 neighbors. Neighbors at upper skip-levels are decided by J's membership vector. Suppose J's initial generated membership vector is 110 (which is the same as D's), it will choose B as its neighbor at skip-level 1, D as its neighbor at both skip-level 2 and 1. At this time, a new skip-level (3) will be generated, D is its only neighbor at skip-level 3.

ZNet can route correctly as long as the bottom skip-level neighbors of each node are maintained, since all other neighbors contribute only to routing efficiency, not routing correctness. Thus, each node in ZNet maintains redundant neighbors (in the right neighbor-list and the left neighbor-list) which include the closest (right and left) neighbors along the bottom skip-level list to deal with node failure or departure. A background stabilization process runs periodically at each node to fix neighbors at upper skip-levels.

3.4 Load balancing

In ZNet, we only consider load balancing from storage perspective, since routing load balance can be achieved with the symmetric nature of skip graphs.

If two nodes contain nearly the same number of indices, then loads on these two nodes will be nearly the same. We try to balance data distribution among nodes by choosing appropriate nodes and splitting their space when new nodes join the network. Currently, two strategies are employed in ZNet: In the first strategy, when a node joins the network, it randomly chooses one data object *which has already published to the network*, and uses the point which corresponds to the data object as the join destination. And then the join request is routed to the node whose space covers the point. In the second strategy, m such candidates are used, the one whose corresponding destination node has the heaviest load is chosen as the joining destination. The second strategy could achieve better load balancing than the first one, however, the join cost is m times higher.

With large number of nodes, nodes should be distributed in a way which is approximately proportional to the data distribution. A large number of nodes will be clustered in the space which is densely populated. Also, another benefit from this kind of joining is that the publishing cost of a new joining node could be saved (suppose that most data objects in a node are similar, they will be published into nearby nodes).

One problem with above load balancing is that it only considers static data distribution. Thus, when there is data evolution, the load will not be balanced anymore. We

use the following method to address this problem. Each node first estimates the average load $\overline{L}$ by sampling loads on its neighbors. Since the high-level links in the skip graph approximate a random graph, a simple procedure that samples load on neighbors in the skip graph performs well. A node is said to be lightly loaded if its load is less than $\overline{L} - \delta$ and heavily loaded if its load is larger than $\overline{L} + \delta$, where δ is a variable which represents a tradeoff between the amount of load moved and the quality of balance achieved. Next, each node periodically exchanges its load information with its neighbors (We differentiate two kinds of a node's neighbors: neighbors whose z-addresses are continuous with the node's z-addresses are the node's *close neighbors*, in fact, these neighbors are the node's neighbors at the bottom skip level; 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 partial of its load to its close neighbors. In case it is not successful (e.g., all 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 taken to make requests be sent to less loaded parts in the network: 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 itself is lightly loaded or it is close to lightly loaded nodes (as lightly loaded nodes always average their load with their close neighbors). With randomness of a skip graph, 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 our system via simulation. We first measure how index distribution are balanced in the network, followed by the test of average lookup cost of routing in ZNet; Then we focus on range queries.

The set of experiments are done on synthetic datasets of increasing dimensionality, which are generated based on normal distribution. By default, we use data sets with skewed 8-dimensional 300,000 data points, and 6,000 nodes in the network. The dimensionality of data in the experiments is varied from 4 to 20, and the number of nodes is varied from 2,000 to 10,000.

4.1 Load Balancing

We measure load balancing mainly in data distribution among the nodes in network. Two approaches are employed to balance the load: one is LB-1, the other is LB-x. In LB-1, when a node joins, a random point of a data object (published by the node) is chosen as a representative for the node to decide the join destination; In LB-m ($m > 1$), m such candidates are used and tried, the one whose corresponding destination has the heaviest load is finally chosen for node joining. In the experiment, we choose m to be 5. Larger m is also tried, however, no further improvement is observed.

Our approaches are compared with two cases: One is an ideal case, where each node contains the same amount of data; The other is an extreme case, where the data space is partitioned and assigned to nodes *evenly*.

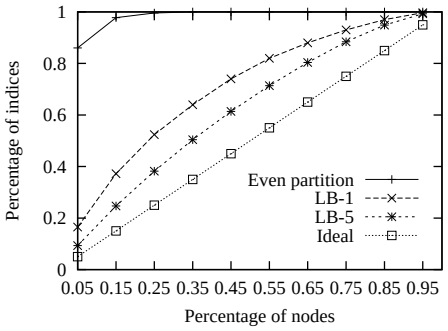

Fig. 6. Load Balance At Join

In Figure 6, nodes are sorted in decreasing order according to the number of data contained by them. From the figure, we can see that, when space is assigned to nodes evenly, 80% of indies are inserted in 5% of nodes, the data distribution among nodes is severely unbalanced. Both LB-5 and LB-1 achieve good load balancing, with LB-5 close to the ideal case. LB-5 is better than LB-1, since it makes the decision according to the current load distribution in the network. This figure shows that our approaches, esp., LB-5, follow the data distribution well. All following experiments are done with LB-5 assumed.

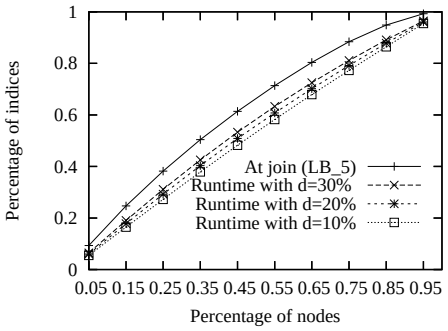

Fig. 7. Load Balance At Runtime

The result are not bad, for x=10, the system needs only 18 rounds, and the lowest load in the network is 133 and the highest load is 166 (for AvgLoad=150), however, there are 220 nodes leave and rejoin. When x is larger, the number of node leave and rejoin will be less. for x=30, there are 23 nodes leave and rejoin altogether. Node leave and rejoin is unavoidable. There is a tradeoff between load balance and the cost of range searches. Better load balancing may result in higher cost on range searches. Before runtime load balancing, for networksize (2000) and query range size (0.1), the number of processing and routing nodes are about 15 and 26 respectively; After runtime load balancing, the number of processing and routing nodes are about 19 and 28 respectively ; With larger query range size, the difference is more. The increase in the number of processing nodes is due to better load balancing among nodes.

4.2 Average Lookup Cost

In this set experiment, we measure average lookup cost for point queries in ZNet.

The lookup cost is measured by the number of hops between two random selected nodes, averaged over 10 times the network size. Figure 8 shows the effect of network size on lookup cost. As shown in the figure, the average lookup cost increases with the network size (which is a little more than $0.5 * logN$).

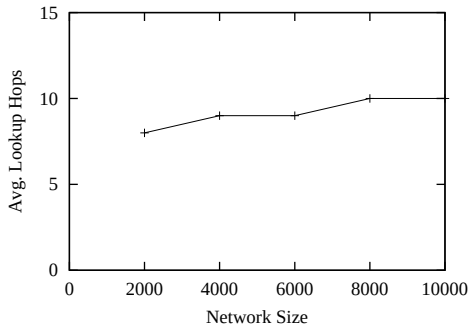

Fig. 8. Average Lookup Cost for point queries

4.3 Range Search Cost

We focus on two metrics for measuring range searches:

- Processing Cost: The number of nodes whose spaces overlap the query range. These nodes are needed to search their virtual databases for query results;
- Routing Cost: The number of nodes for routing the query *only*. These nodes are visited for routing the query to nodes whose spaces overlap the query range;

Three factors are involved for range searches: network size, the dimensionality, and the query range size. Thus, to measure the cost for range queries, we vary these three factors respectively at each time. All range queries are generated according to the data distribution: queries are clustered in dense data area, 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 size, each is initiated from 100 random nodes in the network.

Two methods (one-way and two-way) are compared in terms of the routing cost with the network size varied (same trends are observed with dimensionality and query range size varied).

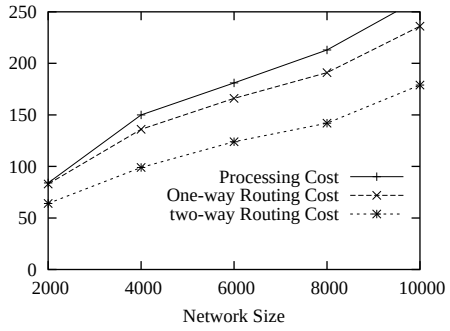

Fig. 9. The Effect Of Network Size On Range Queries

Effect of Network Size To measure the effect of network size over the cost, we fix query range size at each dimension to be 0.2. Data set is the same for all network sizes.

As shown in Figure 9, the processing cost increases with the network size, since more nodes are clustered in the dense data area according to our space partitioning method. For the routing cost, the routing costs of both one-way and two-way increase with the network size also, however, two-way method visits less nodes than one-way method.

High processing and routing costs in Figure 9 are mainly due to the relatively high dimensionality (8) we used in the experiments. With higher dimensionality, the clustering of z-ordering becomes worse. When the dimensionality is low, the cost is much lower. The effect of dimensionality on cost is tested next.

Effect of Dimensionality The effect of dimensionality is measured by fixing query selectivity. Because of small selectivity we choose, range queries are covered by only one node when dimensionality is 4 and 8 (in Figure 10). However, even with a small query selectivity, we can see from the figure that the number of processing nodes increases quickly when dimensionality increases. This is because the range size of a query with the same selectivity increases rapidly with higher dimension-

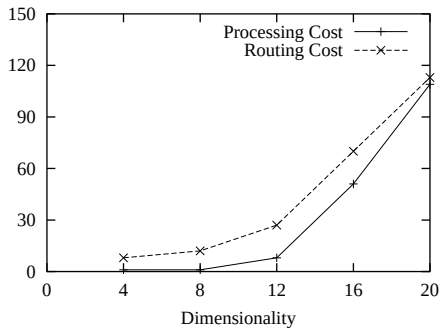

Fig. 10. The Effect Of Dimensionality On Range Queries

ality. Consequently, much bigger data space has to be searched, more nodes have to be visited for routing.

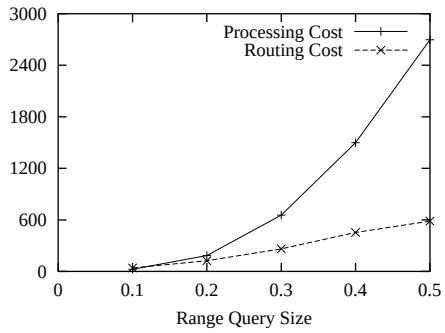

Fig. 11. The Effect Of Query Range Size On Range Queries

Effect of Range Size Finally, we test the effect of range size on costs. The query range size at each dimension is varied from 0.1 to 0.5. As shown in Figure 11, range size has much effect on the number of processing nodes and routing cost. With larger query range size, more nodes in ZNet need to be visited for processing or routing, since more nodes are clustered in dense data area and our queries are also clustered in the dense area.

5 Conclusion

In this paper, we described the design of ZNet, a distributed system for efficiently supporting multi-dimensional range searches. ZNet directly operates on the native data

space, which are partitioned dynamically and assigned to nodes in the network. By choosing appropriate subspaces to be split further, load imbalance could be reduced. By ordering subspaces in Z-curves of different granularity levels, we could extend skip graph to support efficient routing and range searches. Results from a simulation study show that ZNet is efficient in supporting range searches, esp. when dimensionality is not very high. In future work, we plan to address the load balancing problem when the data distribution is dynamic, and the efficiency problem when the dimensionality is high.

Acknowledgements The authors would like to thank Beng Chin Ooi and Yingguang Li for their helpful discussion.

References

1. A. Andrzejak, Z. Xu:Scalable, Efficient Range Queries for Grid Information Services. Proceedings fo the Second IEEE International Conference on Peer-to-Peer Computing(P2P2002)
2. J. Aspnes, G. Shah:Skip graphs. Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)
3. B.Awerbuch, C.Scheideler:Peer-to-Peer systems for Prefix Search. Proceedings of the Symposium on Principles of Distributed Computing(2003)
4. M. Cai, M. Frank, J. Chen, P.Szekely: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. 4th International Workshop on Grid Computing (Grid2003)
5. V. Gaede, O. Gnther: Multidimensional access methods. ACM Computing Surveys, Volume 30,Issue 2 (1998)
6. Gnutella Development Home Page: http://gnutella.wego.com/.
7. N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, A. Wolman: SkipNet: A Scalable Overlay Network with Practical Locality Properties. Fourth USENIX Symposium on Internet Technologies and Systems (USITS 2003)
8. X. Li, Y. J. Kim, R. Govindan, W. Hong: Multi-dimensional range queries in sensor networks. Proceedings of the first international conference on Embedded networked sensor systems(2003)
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: A scalable content-addressable network. Proceedings of the ACM Special Interest Group on Data Communication(SIGCOMM 2001).
10. A. Rowstron, P. Druschel: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)
11. O. D. Sahin, A. Gupta, D.Agrawal, A.El Abbadi:A Peer-to-peer Framework for Caching Range Queries. Proceedings of the 18th International Conference on Data Engineering (ICDE 2004)
12. C. Schmidt, M. Parashar: Flexible Information Discovery in Decentralized Distriubted Systems. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 2003)
13. I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan: Chord: A scalable peer-to-peer lookup service for internet applications. Proceedings of the ACM Special Interest Group on Data Communication(SIGCOMM 2001)

14. C. Tang, Z. Xu, S. Dwarkadas: Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. Proceedings of the ACM Special Interest Group on Data Communication(SIGCOMM 2003)
15. B. Y. Zhao, J. D. Kubiatowicz, A. D. Joseph: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. UC Berkeley Technical Report,UCB/CSD-01-1141 (2001)