

# Supporting High Dimensional Range Queries in Peer-to-Peer Systems

Sai Wu<sup>1</sup> and Hong Gao<sup>2</sup> and Bei Yu<sup>1</sup>

<sup>1</sup> School of Computing,  
National University of Singapore,  
Singapore

<sup>2</sup> Department of Computer Science and Technology,  
Harbin Institute of Technology,  
China

{wusai, yubei}@comp.nus.edu.sg{honggao}@hit.edu.cn

**Abstract.** A large number of works have been proposed to improve the search efficiency in peer-to-peer (P2P) systems. However, most of them focus on exact key match queries, whereas few schemes are proved efficient and practical for the range queries. Actually, high dimensional data and range queries need to be supported in lots of real applications, such as database management systems (DBMS) and image retrieval systems. In this paper, we present a scalable scheme for efficiently performing high dimensional range queries in P2P systems in terms of both dimensionality and network size. Our data space partition strategy also forces load balancing among participating peers. We compare our schemes with previous work. Experiments with various synthetic and real datasets show the effectiveness of the proposed scheme.

**Key words:** P2P, BATON, Range Query

## 1 Introduction

Various resources have been shared in today's peer-to-peer (P2P) systems, and more people are willing to participate in the P2P network to enjoy the service of abundant, yet free resources. Some research work shows that more than 70 percent of the traffic on the Internet is P2P related [8]. Indeed, P2P systems are becoming an increasingly important applications in the Internet. However, explosive amount of data makes it difficult to search for a desired object in a P2P system. Early P2P systems, such as Gnutella, flood the user's request in the network. This approach incurs high message overhead and has no performance guarantee. Later, DHT-based (Distributed Hash Table) P2P systems [13][11][10] are proposed to resolve this problem by enforcing some special overlay structures. They are efficient for the exact key match query with  $O(\log(n))$  search cost, but unfortunately not good for complex queries.

Some proposals [9][6][1] extend existing P2P systems to support range queries. They only focus on the single dimensional case, while real applications, such as database management system (DBMS) and image retrieval system, require the support for high-dimensional range queries. For example, images are always represented by color histograms (e.g. 30 bins for different color sets). The color histograms of an image are

always formalized as a high-dimensional vector, and thus searching similar images is performed by comparing high-dimensional vectors of images via range queries.

To cope with high-dimensional range queries in a P2P network, the data space is partitioned into subspaces and disseminated to different peers. The range query is processed by checking all the overlapped subspaces. Specifically, three issues must be addressed:

- *Space partition and mapping.* How the data space is partitioned and mapped to peers? Peers’ joining or leaving should not affect the correctness of the partition and only incur little overhead to fix the mapping.
- *Routing.* The range queries will be partitioned and issued to corresponding peers. We must consider how to route query messages to correct peers and simultaneously reduce the routing cost as much as possible. Routing strategy also relies on the characteristics of P2P overlays.
- *Load balance.* In DHT systems, we rely on consistent hashing to generate data IDs that are uniformly distributed in the key space. However, to answer range queries, the usage of hash function should be avoided to preserve data locality. Consequently, skewed data will cause imbalanced load among peers. A new scheme is required to re-balance the load.

In this paper, we present an innovative scheme to process high dimensional range queries. We adopt a partition strategy similar to K-D tree to partition the high-dimensional space into subspaces, i.e., high-dimensional rectangles. But different from K-D tree that splits the space evenly, our approach partitions the space in a way to balance the query load as much as possible. BATON [7], a balanced tree structure, is employed as our P2P overlay, since it supports one dimensional range query and also guarantees the load balance. We extend BATON by mapping the subspaces into different nodes (if not specified, “peer” and “node” are used interchangeably). We experimentally compare our scheme with MURK [5] and ZNet [12], which shows the superiority of our scheme in terms of both search efficiency and load balance.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 presents our space partition and range query processing schemes. Section 4 shows the experimental validation, and finally we conclude the paper in Section 5.

## 2 Related Work

Current approaches for supporting range queries in P2P networks can be classified into two categories — to extend the existing P2P overlays or to invent new overlays. [9][6][1] modify the strategy of mapping data to peers in existing overlays. Both [9] and [6] apply the locality sensitive hash function to map similar data into a continuous range. [1] expels the hash function and uses original value of the data as the key. However, it is unclear how to extend these schemes to support multi-dimensional search. Skip-graph [3], BATON [7] and P-Grid [2] are P2P overlays that are invented to support range queries. Compare to the former schemes, these overlays are easily extended to cope with complex queries.

Mercury [4] adopts a Chord [13] like ring structure. Data are mapped to peers by their values, in order that the range query can be answered by the peers in a continuous region of the ring. To support multi-attribute queries, Mercury creates an individual ring for each attribute. A multi-attribute query is first processed based on the single arbitrarily chosen attribute in the corresponding ring, and then the other attributes are used as filters for retrieving final results. The load balance is guaranteed by periodically migrating peers from lightly loaded areas to heavily loaded areas in the ring. The major drawback of Mercury is the expensive cost to maintain multiple ring structures.

In [5], Ganesan et. al propose and compare two different approaches to support multi-dimensional range queries. One approach, called SCRAP, employs the space filling curve, such as z-order curve, to map high dimensional data to single dimensional points. The mapped data are disseminated into peers in Skip-graph. A multi-dimensional range query is partitioned into a series of single dimensional range queries, which can be resolved by Skip-graph efficiently. However, the space filling curve is not scalable when the dimensionality is high, because a high dimensional region may be partitioned into a large number of one dimensional ranges. All peers overlapped with the ranges are forced to process the queries. Different from SCRAP, the other approach, MURK, partitions the high dimensional search space into hypercubes, and each hypercube is mapped to one peer in CAN [10]. The proposed partition strategy is similar to that of the K-D tree, where the dimensions are used cyclically for splitting. To optimize the routing, each peer is required to keep some skip pointers to other peers, which are picked randomly or by certain specialized strategy. Both SCRAP and MURK partition the data space in a static manner, hence they cannot adapt to dynamic changes in P2P networks, such as new data objects are inserted or query loads distribution evolves.

ZNet [12] also addresses the multi-dimensional range queries based on Skip-graph. In ZNet, z-order curve is used to partition the data space. Each subspace is assigned a unique z-order identifier. Given a query, the peer can estimate the z-ID of a super zone that covers the search range, because the space is partitioned evenly. Then, ZNet forwards the query message to the estimated zone. In the course of routing query among peers, the zone is refined to contain the querying content more accurately. The static partition approach in ZNET causes the estimated superset to overlap with many irrelevant zones. Our approach adopts more dynamic partition and routing scheme for resolving this problem.

In [14], a space partition strategy similar to the one used in our scheme is applied to answer skyline queries, which is also based on BATON. But range queries are required to be split in a different way from skyline queries. Therefore, new routing and pruning approaches should be employed to reduce the message overhead for query processing.

### 3 The System Design

In this section, we will start from a brief introduction of BATON [7] (Section 3.1), and following it, we present our space partition and mapping strategy (Section 3.2). Finally, we show how to implement the partition strategy in BATON and how to answer high dimensional range queries based on the proposed partition strategy (Section 3.3).

### 3.1 Background about BATON

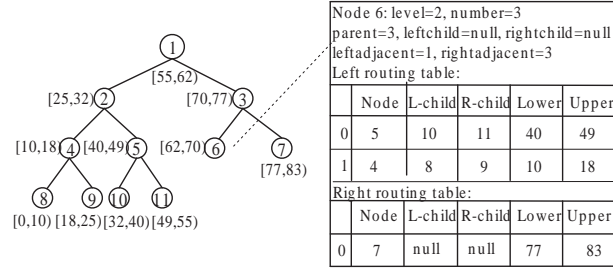


Fig. 1. BATON Overlay

BATON (BALanced Tree Overlay Network) [7] is a distributed tree structure based overlay network, where each tree node is maintained by a peer. In BATON, the height difference of any subtrees is at most one. Each node keeps the information about its parent node, child nodes and adjacent nodes (in order). In addition, the node also keeps the routing links to the nodes at distance of power of two in the same level. Figure 1 illustrates the BATON overlay and the routing table maintained at each node. The balance characteristics and the routing links of BATON guarantee the exact match query to be resolved in  $O(\log N)$  routing hops.

### 3.2 Space Partition and Mapping

In P2P systems, the whole data space is partitioned and disseminated to peers. Each peer maintains the data in a subspace. To answer a high-dimensional range query, the search space is split into subranges in the same way as we partition the data space. The query will be routed to peers whose spaces overlap with the search ranges. Therefore, the number of messages incurred by range queries is determined by the partition and routing strategy. We partition the data space based on the partition strategy of K-D tree, since it offers flexibility to the partition method — the space can be split based on the space size, number of data within the space or any other metrics. Because load balance is a major concern in our system, our partition criteria is based on the load of each node. Formally, the load of peer  $i$  is modeled as

$$\alpha S_i + \beta Q_i$$

where  $S_i$  denotes the storage cost for the data in the subspace and  $Q_i$  represents the query load (the frequency of the subspace being queried). With our approach, the storage cost in each peer is very small (only indices are recorded). Therefore, we set  $\alpha$  to 0 and only focus on balancing the query load.

Figure 2(a) demonstrates the partition strategy in two dimensional case. The partition procedure is invoked when a new node joins the network and wants to share the

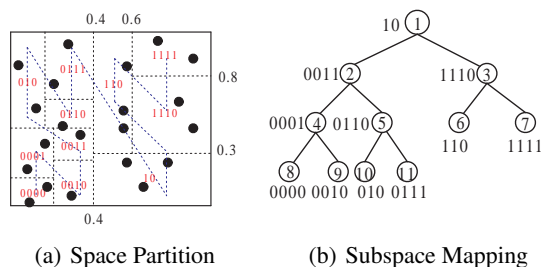


Fig. 2. Data Space Partition and Mapping

load of an existing one. Initially, only one node (root node) exists, taking over the whole data space. When a second node joins, the entire space is split into two subspaces along the  $X$ -axis, i.e., data points are disseminated to different nodes according to their  $X$ -values. In the following partitions, the  $X$ -axis and  $Y$ -axis are used alternately. When the third node joins the network, we select the node with heavier load to share with the new node. In our proposal, if a node receives a join request, it will check all the qualified nodes that can accept the new node and pick the one with heaviest load.

As mentioned, we want to partition the space in a way that each resulted subspace will have almost the same query load. This can be achieved by assigning weight to each of the data points, indicating how many times it has been queried. Initially, all data points are assigned the same weight. When a data point is returned as a query result, its weight is increased. We adopt an aging weights computation (original aging weights have been applied to cache management) for calculating the weights of data points, which is shown as follows

$$\frac{W'}{T} + C$$

where  $W'$  represents the weight in previous time unit,  $T$  is the aging factor and  $C$  denotes the number of times being queried in current time unit. The query load of a subspace is calculated as the total weight of all data points within the subspace. This metric, though rough, is a good estimation for the query load distribution.

The partition scheme is described in Algorithm 1. In order to distinguish the subspaces, we assign a unique ID to each subspace. Suppose the original ID of the space is  $x$ , the ID of its subspace resulted in partition will be  $x0$  if it lies in the left or lower part, or  $x1$  if it lies in the right or upper part. Note that the subspaces' IDs actually follow  $z$ -order. However, different from other approaches relying on  $z$ -curve to partition the space, our scheme partitions the space dynamically based on query loads. We also record for each subspace the positions where the previous partitions are performed, as its *split history*. For example, the split history of the node maintaining subspace "1110" in Figure 2(a) is  $\{(0.4, 1), (0.3, 1), (0.6, 1), (0.8, 0)\}$ . The split history represents the node's knowledge about the partition of whole data space. It can be used to direct the routing of queries.

With the subspace's unique ID, we map a high-dimensional hypercube into a one dimensional point, which can be routed in BATON. Figure 2(b) illustrates the mapping

result from partitioned space in Figure 2(a) to BATON tree in Figure 1. As nodes join the network, the data space is partitioned and disseminated to different nodes. If we travel BATON nodes in inorder, the subspaces are visited in an increasing order of their IDs. For example, the subtree rooted at node 3 maintains the subspaces with ID "1\*". Node 6, 3 and 7 are responsible for subspaces "110", "1110" and "1111" respectively.

---

**Algorithm 1** PartitionSpace(node  $n$ )
 

---

```

1:  $S = \text{BATON\_Join}(n)$  //  $S$  are qualified nodes that can accept  $n$  as child
2:  $m = \text{SelectHeaviestLoad}(S)$ 
3:  $\text{BATON\_Insert}(m, n)$  //  $m$  accepts  $n$  as its child
4: if  $n$  is the left child of  $m$  then
5:    $n.\text{Space} = \text{EqualLoadSplit}(m.\text{NextPartitionDimension} \% \text{Dim}, \text{left})$ 
6:    $n.\text{SpaceID} = m.\text{SpaceID} + "0"$ 
7:    $n.\text{SplitHistory} = m.\text{SplitHistory} + (\text{SplitPosition}, 0)$ 
8: else
9:    $n.\text{Space} = \text{EqualLoadSplit}(m.\text{NextPartitionDimension} \% \text{Dim}, \text{right})$ 
10:   $n.\text{SpaceID} = m.\text{SpaceID} + "1"$ 
11:   $n.\text{SplitHistory} = m.\text{SplitHistory} + (\text{SplitPosition}, 1)$ 
12:   $m.\text{NextPartitionDimension}++$ 
13:   $n.\text{NextPartitionDimension} = m.\text{NextPartitionDimension}$ 

```

---

When a node leaves the network, we need to combine its space with that of its neighbors. Depending on the node's split history, two possible combinations are considered. In one case, if node 7 leaves, its space "1111" can be combined with space "1110" (node 3), which happens to be its parent node's space. The new space for node 3 is assigned ID "111" and all data in space "1111" are transferred to it. In the other case, if node 6 leaves, its space "110" should be combined with space "111". But space "111" has already been further partitioned into two subspaces (node 3 and node 7). So the combination can not be performed, and we have to find another node to replace node 6. Node 7 is selected since its space can be combined with its parent node. Hence, node 7 is forced to leave and rejoin the network to take over space "110", and its own space "1111" is transferred to its former parent node. The tree rotation process may be invoked if the balanced characteristic is broken after node replacement.

Our scheme tries to balance the query load amongst nodes in two steps. First, when a new node joins the network, we find a heavily loaded node and partition its search space to share half of the load with the new one. Second, the node continuously monitors the work load of its neighbors. If its load is more than  $\alpha$  times of the average ( $\alpha = 2$  in this paper), it will find a light loaded node to share its load. The light loaded node will leave the network and rejoin as a child of the heavily loaded one. In BATON, the node will periodically send ping messages to nodes in its routing table. We now require the nodes to piggy back their load in the pong messages. So each node can find a relatively heavy or light load node from its routing table.

### 3.3 Range Querying Processing

The general idea for preprocessing range queries is to partition the queries in the same way as we partition the data space, and then send the subqueries to corresponding nodes. A

single node has no global knowledge about how the data space is partitioned, because the partition is processed in a distributed manner. However, it is aware of how its own space is created, which is recorded in the split history. Based on its split history, it can partition the queries by estimation. The accuracy of estimation is inversely proportional to the distance between the search space and the node's space. As an example, Figure 3 shows how a 2-dimensional range query  $\{(0.3, 0.75], (0.55, 0.9]\}$  is partitioned by node "1111" of Figure 2(a). The first entry of the split history tells node "1111" that if the X-value is less than 0.4, the query will be covered by the spaces with prefix "0\*". Therefore, node "1111" splits the search space into two parts:  $\{(0.3, 0.4], (0.55, 0.9]\}$  and  $\{(0.4, 0.75], (0.55, 0.9]\}$ . The former query space is covered by the nodes with ID "0\*" and the later should be handled by nodes with ID "1\*". So we assign query ID "0" to the first subquery  $Q_1$  and "1" to the second one. Query ID denotes a coarse estimation for the query destination. Node "1111" has no detailed knowledge about how the space "0\*" is partitioned, but it can further partition the query space in "1\*" region in a similar way based on its split history. As Figure 3 shows, the search space is finally partitioned into four subqueries. Observe that query  $Q_1$  needs further partition, which can be completed by other nodes.

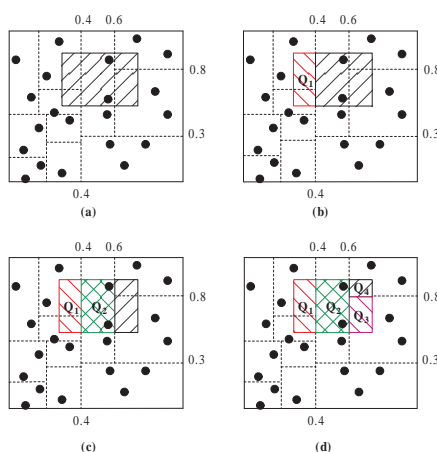


Fig. 3. Query Partitioning

After the node partitions the query, it will check the routing table and disseminate the resulted subqueries to corresponding nodes. For each subquery, it is forwarded to the first node in the routing table, whose space overlaps with the query. Supposing the query ID is  $QID$  and the node's ID is  $NID$ , the node overlaps with the query if and only if  $QID$  is a prefix of  $NID$ . For example, node 7 in Figure 2 answers  $Q_4$  with its local knowledge and forwards  $Q_1$ ,  $Q_2$  and  $Q_3$  to node 5, 6 and 3 respectively. Note that a node overlapping with the query does not indicate that it overlaps with the search range, because the query ID is generated by the message sender's estimation based on

incomplete knowledge. It actually represents a super-zone that contains the actual query range. When the query is routed to different nodes, its ID will be modified to reflect the query range more and more accurately. In our implementation, the algorithm does not send a real message until all queries' receivers have been confirmed. And then the messages to the same node are aggregated into one message to reduce the transmission overhead. The trade-off is that it may introduce a little latency.

---

**Algorithm 2** ProcessRangeQuery(node  $n$ , RangeQuery  $Q$ )

---

```

//n is the current node
//Q is the initial range query
1: subqueries=SplitQuery(n, Q)
2: if subqueries=NULL then
3:   return
4: result=localprocess(subqueries)
5: return result to the query initiator if it is not empty
6: Left=GetLeftQuery(subqueries)
7: Right=GetRightQuery(subqueries)
8: for each query  $Q_i$  in Left do
9:   forward  $Q_i$  to the first routing node overlapping with it or left adjacent node
10: for each query  $Q_i$  in Right do
11:   forward  $Q_i$  to the first routing node overlapping with it or right adjacent node

```

---

The algorithm for range query processing is described in Algorithm 2. After the query is partitioned, the subqueries that can be answered locally will be processed and the results are returned to the requester (line 4-5). Then the rest subqueries are grouped into two sets (line 6-7). The subqueries whose IDs are smaller than current node's ID are contained in one set, while the rest are in the other set. We forward each query to the first node in the routing table that overlaps with it. If no node in the routing table overlaps with the query, we will try to forward the request to the adjacent nodes .

## 4 Experimental Results

In this section, we evaluate our range query processing scheme called BATONRange by comparing it with MURK [5] and ZNet [12] in terms of network size, dimensionality and query load balance. The evaluation is conducted via simulations. Two kinds of synthetic datasets are generated with their dimensionality ranging from 2 to 20. One dataset follows uniform distribution, and the other is skewed, following zipfian distribution. Each set contains 1000000 data points. In addition, a real dataset from an image database is also used. The network size varies from 1000 to 10000 nodes. We consider the network as a "static" one, i.e., no node joins or leaves when the query is being processed. But after query is processed, the network becomes dynamic and accepts join and leave request.

Each experiment is repeated 10 times and the average result is measured. We issue 100 range queries in each test. And after every 20 queries, 5% of nodes join and 5% of

nodes leave the network. For uniform datasets, the selectivity of range queries are set to 0.01 in our experiments. We use the same set of queries for skewed datasets because it is difficult to estimate the query selectivity in it. Two metrics are employed to measure the performance: average number of processing nodes (the node whose subspace overlaps with the query) and average number of messages for processing a query.

#### 4.1 Effects of Dimensionality

We first examine the performance of BATONRange with different dimensionality. A network of 5000 nodes is created and 100000 data points are injected. Figure 4 shows the average number of routing messages for different datasets. Similar results are obtained for both uniform dataset and skewed dataset. With low dimensionality, MURK achieves comparative performance with ZNet and BATONRange or even outperforms them. However, when the dimensionality increases to 8 and higher, its gap to ZNet and BATONRange becomes significant.

Figure 5 demonstrates the average number of processing nodes (in percent) at different dimensionality. In MURK and ZNet, almost all nodes participate in the query processing, when dimensionality reaches 20. But for BATONRange, the number of nodes increases slowly because of the efficient partition strategy. Both Figure 4 and Figure 5 indicate that BATONRange can handle high dimensional queries more effectively, while MURK is incapable when the dimensionality is above 8.

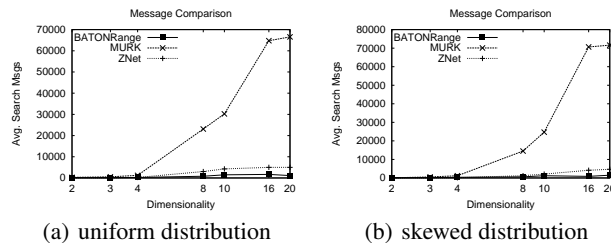
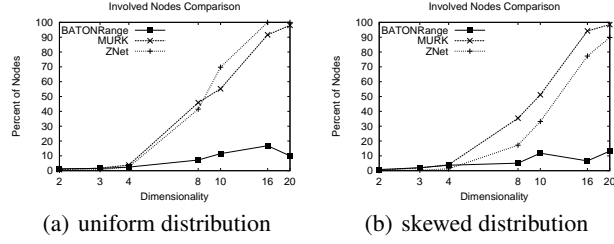


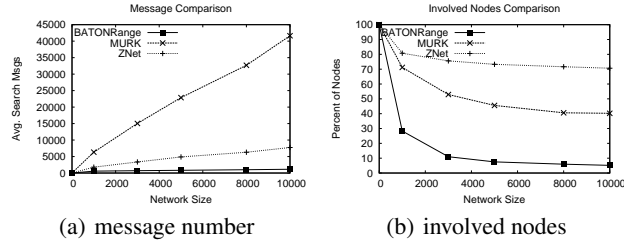
Fig. 4. Routing Cost as Dimensionality Varies

#### 4.2 Effects of Network Size

In this section, we study the effects of network size on the query performance. We generate P2P networks with node number ranging from 1000 to 10000. An 8-dimensional uniform dataset is used in this experiment. From Figure 6(a) we observe that the number of routing messages is proportional to the network size in MURK and ZNet. In BATONRange, the message number fluctuates slightly, which implies that the routing cost of BATONRange is not affected by the network size. Figure 6(b) shows the number of processing nodes (in percent) when the network size increases. Our approach scales quite well with increasing network size. The results for both uniform and skewed datasets are similar, and therefore we only show the results for uniform data set.



**Fig. 5.** Processing Node as Dimensionality Varies



**Fig. 6.** Search Cost as Network Size Varies

### 4.3 Effects of Query Selectivity

The selectivity of query is defined as the percentage of data items that match the query. Under uniform distribution, the query selectivity is proportional to the search range. Figure 7 shows the effect of query selectivity in a network of 5000 nodes, which are loaded with the uniform dataset consisting of 16-dimensional data points. Search cost of BATONRange increases slowly as the selectivity varies from 0.02 to 0.1. When selectivity is set to 0.1, less than 10000 messages are required (2 messages per node on average). For MURK and ZNet, when selectivity exceeds 0.02, almost all nodes participate in the query processing. The routing process actually degrades to a flooding approach. ZNet incurs less overhead than BATONRange, because it is based on SkipGraph [3], which provides an efficient approach to broadcast messages among nodes. For a network of  $N$  nodes, SkipGraph requires  $N - 1$  messages to flood messages to all nodes by exploiting the neighbor.

### 4.4 Load Balance

As mentioned previously, we focus on the query load, not the storage load. The query load of a node is measured by the number of queries that overlap with its subspace. We expect nodes in our system share balanced query load with our space partition method. For MURK and ZNet, the space is partitioned statically and never adjusted according to the query load. It assumes all data items have the same probability of being queried. However, this is not true for real systems, in which the queries always

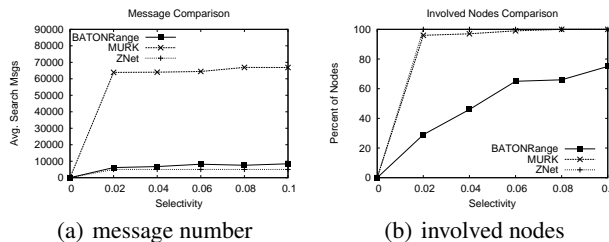


Fig. 7. Search Cost for Range Queries of Different Selectivity

follow some skewed distribution and some popular data items become hotspots. Our scheme, BATONRange, records the node's query load in a recent access log. Once detecting imbalanced load, it will dynamically repartition the space between neighbor nodes. Figure 8(a) and 8(b) illustrate the query load distribution in 4 dimensional case. Y axis represents the percent of queries that the nodes have processed. And x axis denotes the percent of nodes. We set dimensionality to 4, because as Figure 5 shows, MURK, ZNet and BATONRange require similar number of nodes to process a query in that case. We can see that BATONRange shows more balanced load than MURK and ZNet for both uniform dataset and skewed dataset. Furthermore, BATONRange also performs well in dealing with high dimensional data as indicated by Figure 8(c) and 8(d) (16-dimensional dataset is applied). We do not show the curve of MURK in Figure 8(c) and 8(d), because almost all nodes of MURK participate in the query processing, resulting in the same query load for all nodes.

## 5 Conclusion

In this paper, we have addressed the problem of processing high dimensional range queries in P2P environment. We develop our system based on BATON [7], a distributed tree structure supporting range queries. As nodes join or leave the system, the data space is partitioned or merged adaptively. Moreover, we adjust the partitioning space according to the evolvement of query load. To answer high dimensional range queries, a novel partition and routing scheme is devised. The experiments indicate that our approach has a significant improvement over existing approaches. Besides, with our partition scheme, nodes in the network can have similar query loads. In future work, we plan to apply our approach to some complex applications in P2P network, such as DBMS or Image Retrieval System, and study its performance in real systems.

## References

1. M. Abdallah and H. Le. Scalable Range Query Processing for Large-Scale Distributed Database Applications. In *Parallel and Distributed Computing and Systems*, 2005.
2. K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Puceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. In *SIGMOD Record*, 2003.

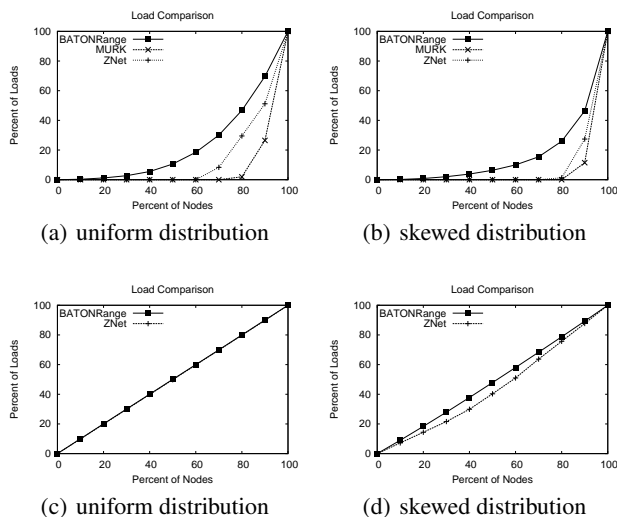


Fig. 8. Load Distribution

3. J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Jan. 2003.
4. A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, 2004.
5. P. Ganesan, B. Yang, and H. G. Molina. One Torus to Rule Them All: Multidimensional Queries in P2P Systems. In *WebDB*, pages 19–24, 2004.
6. A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.
7. H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *VLDB*, pages 661–672, 2005.
8. M. Meeker. Web 2.0 Summit presentation.
9. T. Pitoura, N. Ntarmos, and P. Triantafyllou. Load Balancing and Efficient Range Query Processing in DHTs. In *10th International Conference on Extending Database Technology (EDBT06)*, 2006.
10. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–17, Aug. 2001.
11. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218, 2001. [citeseer.nj.nec.com/rowstron01pastry.html](http://citeseer.nj.nec.com/rowstron01pastry.html).
12. Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting Multi-dimensional Range Queries. in Peer-to-Peer Systems. In *IEEE International Conference on Peer-to-Peer Computing*, 2005.
13. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, Aug. 2001.
14. S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient Skyline Query Processing on Peer-to-Peer Networks. In *ICDE*, 2007 (to be appear).