One Torus to Rule them All: Multi-dimensional Queries in P2P Systems

Prasanna Ganesan Beverly Yang Hector Garcia-Molina

Stanford University
{prasannag,byang,hector}@cs.stanford.edu

ABSTRACT

Peer-to-peer systems enable access to data spread over an extremely large number of machines. Most P2P systems support only simple lookup queries. However, many new applications, such as P2P photo sharing and massively multi-player games, would benefit greatly from support for multi-dimensional range queries. We show how such queries may be supported in a P2P system by adapting traditional spatial-database technologies with novel P2P routing networks and load-balancing algorithms. We show how to adapt two popular spatial-database solutions – kd-trees and space-filling curves – and experimentally compare their effectiveness.

1. INTRODUCTION

Peer-to-peer systems have become a key medium for publishing and finding information on the internet today. Their popularity stems from their ease of use, self-administering nature, scalable support for large numbers of users, and their relatively anonymous, privacy-preserving content-publishing model. Content in a P2P system can be modelled as a horizontally partitioned relation, just as in a parallel database, with the system possessing control over how data is partitioned across nodes [17, 16, 7].

Most P2P systems thus far, both deployed and proposed in literature, support only simple lookup queries over such a relation, i.e., queries that retrieve all tuples with a particular key value [17, 15, 16]. Some recent work has extended this functionality to support efficient *range queries* over a single attribute [7, 11]. However, many interesting P2P applications require more powerful multi-dimensional range queries, i.e., conjunctive queries containing range predicates on two or more attributes of the relation.

For example, consider a P2P photo-sharing application where each user publishes photographs tagged with metadata such as GPS location information, the time the picture was taken, keywords associated with the picture, and so on. A typical query in such a system would contain range predicates on multiple attributes; a user may request all photographs taken within the last year, whose location is within 100 metres of a specified place. As another example, massively multi-player online games involve large sets of users moving about in a "virtual space". Each user continuously queries the P2P system to locate all objects, and other users, within a certain distance of her own position in a two-dimensional, or three-dimensional, world [12].

Many solutions for multi-dimensional queries are available in the world of databases. However, adapting these solutions to the P2P world presents four challenges:

Distribution Data needs to be partitioned across a large number of nodes while ensuring both load balance across nodes and efficient queries.

Dynamism Nodes in a P2P system may join and leave frequently. Therefore, the data partitioning needs to be over a dynamic set of nodes, while still retaining good balance and efficient queries.

Data Evolution Data distributions may change over time, and can cause load imbalance even if nodes remain stable. Thus, data may need to be re-partitioned across nodes frequently to ensure load balance.

Decentralization P2P systems do not have a central site that maintains a directory mapping data to nodes. Instead, a query submitted at any node must be efficiently transmitted to the relevant nodes by forwarding the query along on an *overlay network* of nodes. The overlay network is designed to ensure that both the cost of forwarding queries, and the cost of updating the network structure when nodes join and leave, are low.

The problem of supporting multi-dimensional queries in a P2P system can be broken into two components: partitioning and routing. A *partitioning strategy* distributes a relation R over a set of nodes S, supporting the insertion and deletion of tuples from R, as well as the joining of new nodes into S or the leaving of existing ones from S. Once a partitioning strategy is chosen to distribute data across nodes, we require a *routing strategy* to transmit a query to the relevant nodes. As discussed earlier, nodes are interconnected in an overlay network, with each node having communication links to a small number of "neighbor" nodes; queries are routed on this overlay network to be delivered to the relevant nodes.

In this paper, we adapt two different database approaches for multi-dimensional queries – space-filling curves and kd-trees – to the P2P setting while tackling the above challenges (Sections 3 and 4). We then compare the two resulting solutions in order to understand the strengths and weaknesses of each approach in the P2P context (Section 5).

Seventh International Workshop on the Web and Databases (WebDB 2004), June 17-18, 2004, Paris, France.

1.1 Desiderata

Any P2P solution for supporting multi-dimensional queries should ideally have certain properties. First, a good partitioning strategy should have the following characteristics:
(1) **Locality** The cost of executing a query in a P2P system is often proportional to the number of nodes that need to process the query; hence, each query should ideally execute at as few nodes as possible. For multi-dimensional range queries, this implies that the partitioning must have *locality*, i.e., tuples nearby in the data space should ideally be stored at the same node.
(2) **Load Balance** The amount of data stored by each node should be roughly the same, to ensure load balance[1]. This load balance should be ensured even as (a) data evolves with tuple insertions and deletions, and (b) nodes join and leave the system.
(3) **Minimal Metadata** We define partition metadata as a directory that maps each data point to the node managing the partition containing the point. In a P2P system, there is no central site that can maintain this directory, and metadata will be distributed across the participant nodes themselves. The more the metadata, the more work that needs to be performed in updating it across multiple nodes when nodes join and leave the system. Therefore, we wish to keep the metadata as small and simple as possible.

We note that these properties are desirable even when partitioning data in a parallel database. The P2P environment merely makes them even more crucial.

In addition, the routing algorithm and overlay network should have the following characteristics:
(1) **Low per-node state**: The number of links maintained by each node should be small. This is necessary since links need to be updated every time a new node joins, or an old node leaves.
(2) **Efficient Routing**: The number of messages required to send a query to the relevant nodes should be small.
(3) **Routing Load Balance**: The number of routing messages forwarded per second should be roughly equal across nodes. This rules out the use of tree-like overlay networks, since much traffic would have to pass through the root of the tree.

2. RELATED WORK

Partitioning Single-Dimensional Data Hash partitioning can be used to distribute tuples across a set of disks. When using a relational key as the hash attribute, this approach ensures load balance, and minimal meta-data (just the hash function). However, hashing destroys data locality, and range queries are very expensive [2].

Range partitioning designates each node responsible for one contiguous range of attribute values, and thus provides good locality. The amount of meta-data is fairly small, requiring just the attribute values at the partition boundaries. However, ensuring load balance across partitions as data evolves is a non-trivial problem. Recently, we have shown how such load balance may be achieved efficiently [7].

Partitioning Multi-Dimensional Data When data is multi-dimensional, one could still partition it based on just one dimension; this approach becomes very expensive when queries involve ranges on a non-partitioning attribute, since the query would have to be forwarded to a large number of nodes. The BERD declustering strategy used in the Bubba parallel machine [5] improves on this idea by enhancing range partitioning with secondary indexes, but even short range queries continue to be expensive [8].

The MAGIC declustering strategy [8] fragments the data space into a grid of rectangular fragments, using a set of partitioning values on each dimension. Each fragment is allocated to a node, while ensuring that all nodes manage roughly the same number of fragments. The strategy requires prior knowledge of the total number of nodes, and it is unclear whether it can be adapted to support dynamic node joins and leaves. Moreover, there are serious load-balancing issues when the data distribution is not uniform [8].

Reference [6] partitions data using space-filling curves, as does our approach in Section 3. However, our use of space-filling curves is very different from [6], where the objective was to destroy data locality rather than to preserve it.

Routing As we will discuss later, our solutions are adaptations of routing structures used in distributed hash tables [17, 15, 9, 3], but have to deal with added complexities arising from the multi-dimensional nature of the data space, and non-uniformity of node partitions in the data space.

3. SCRAP: SPACE-FILLING CURVES WITH RANGE PARTITIONING

Our first approach to supporting multi-dimensional queries, *SCRAP*, uses a two-step solution to partition the data: (a) Data is first mapped down into a single dimension using a space-filling curve; (b)This single-dimensional data is then range-partitioned across a dynamic set of participant nodes.

For the first step, we may map multi-dimensional data down to a single dimension using a space-filling curve such as *z-ordering* [14] or the *Hilbert* curve (e.g., [10]). For example, say we have two-dimensional data that consists of two 4-bit attributes, X and Y. A two-dimensional data point $\langle x, y\rangle$ can be reduced to a single 8-bit value z, by interleaving the bits in x and y. Thus, the two-dimensional point $\langle 0100, 0101\rangle$ would be mapped to the single-dimensional value 00110001. (This mapping corresponds to the use of z-ordering as the space-filling curve.) Note that this mapping is *bijective*, i.e., every two-dimensional point maps to a unique single-dimensional value and vice versa.

In the second step, once data has been reduced to one dimension using the space-filling curve, relation R can then be range partitioned across the available S nodes; i.e., each node manages data in one contiguous range of z values. Maintaining this range partitioning when nodes join and leave is easy: When a new node joins, it simply splits the range of some existing node; when a node leaves, one of its "neighbors" takes over its range[2].

Routing To execute queries over a SCRAP network, the multi-dimensional query must be sent to the set of nodes that contain data relevant to the query. We may again visualize routing as a two-step process: (In an actual implementation, the two steps are interleaved for efficiency.) (a) The

[1] More generally, we may require the number of queries executing at each node to be equal, when access patterns are not uniform across data. Our subsequent discussion and algorithms generalize directly to this case.

[2] Since nodes may not leave gracefully, data may need to be replicated to avoid loss. We will ignore this issue here as standard P2P techniques take care of this problem.

multi-dimensional range query is first converted to a *set* of 1-d range queries, which together are guaranteed to contain all query answers. (b) We route each of the 1-d range queries acquired in step (a) to the appropriate nodes for that query, i.e., those nodes whose ranges intersect the query range.

Step (a) is performed using well-known algorithms for query-mapping with space-filling curves, e.g., [14, 4]. We note that the query mapping algorithms use simple heuristics to ensure that the number of 1-d ranges returned is not too large. These heuristics may result in "false positives" – portions of the resulting 1-d ranges that do not actually map to a point in the native query range. For example, if the correct set of 1-d ranges is {[0,4), [6,9)}, the algorithm may return {[0,9)}. False positives may result in a non-relevant node receiving and processing the query.

Step (b) is performed using a well-known routing network known as the Skip graph [3, 9]. A skip graph is a circular linked list of nodes, arranged in order of their partition boundaries, enhanced with additional *skip pointers* to enable faster routing. The pointers are constructed using a randomized protocol by which each node establishes $O(\log n)$ pointers (n is the number of nodes) at exponentially increasing distances from itself on the list, i.e., the i^{th} skip pointer is expected to point to a node that is 2^i positions away on the list. With a normal doubly-linked list, locating the node containing a specific data point will take $O(n)$ messages. With skip pointers, only $O(\log n)$ messages are required.

Finding all relevant nodes for a query consists of first finding the node containing the minimum point in the query range, via the skip-graph routing protocol; all remaining relevant nodes can then be reached via neighbor links to the successor nodes on the list.

Discussion The SCRAP approach meets many of our desiderata defined earlier. **Locality** is achieved since the space-filling curve attempts to ensure that nearby data points in the multi-dimensional space are also adjacent in the single dimension. However, as the number of dimensions increases, locality becomes worse since space-filling curves are afflicted by the curse of dimensionality. **Load Balance** in SCRAP can be achieved using recent techniques we have developed for single-dimensional range partitioning [7]. The key idea behind the techniques is to perform "local" alterations to the range partitioning using two operations: (a) NbrAdjust adjusts the partition boundary between two nodes managing neighboring ranges, to transfer load from one node to the other; (b) Reorder uses a node with an empty partition to split the range of a heavily loaded node. We show in [7] that a judicious use of these two operations leads to guaranteed load balance, while guaranteeing that the *cost* of achieving load balance is very small. The **metadata** required to describe the partitioning is also small: all that is required for each node is the partition boundaries of itself and its neighbors.

For the overlay network, **low state** is achieved with only $O(\log n)$ links per node. **Routing load balance** is achieved due to the symmetric nature of skip graphs. **Query routing** for a single 1-d range is efficiently performed in $O(\log n)$ hops. However, as the native dimensionality of the data increases, the number of "relevant" nodes for the original multi-dimensional query may increase dramatically, leading to an increased routing cost.

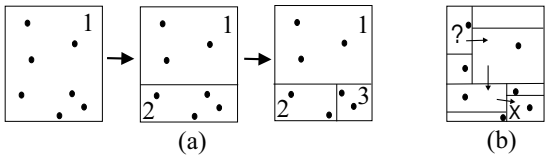

Figure 1: (a) Evolution of partitions as nodes join the network. Each partition represents a node in the network. (b) Routing example over a MURK network.

4. MURK: MULTI-DIMENSIONAL RECTANGULATION WITH KD-TREES

Our second approach, MURK, partitions the data *in situ* in the high-dimensional space, breaking up the data space into "rectangles" (hypercuboids in high dimensions), with each node managing one rectangle. One way to achieve this partitioning is to use kd-trees, in which each leaf corresponds to a rectangle being stored by a node.

We illustrate this partitioning in Figure 1(a). Imagine there is initially one node in the system that manages the entire 2-d space, corresponding to a single-node kd-tree. When a second node arrives, the space is split along the first dimension into two parts of equal load, with one node managing each part; this corresponds to splitting the root node of the kd-tree to create two children. As more nodes arrive, each node splits the partition managed by an existing node, i.e., an existing leaf in the kd-tree. The dimensions are used cyclically in splitting, to ensure that locality is preserved in all dimensions.

When a node leaves, its space needs to be taken over by existing nodes. The simple case is when the node's sibling in the tree is also a leaf, e.g., node 3 in figure; in this case, the sibling node 2 takes over 3's space. The more complex case arises when the node's sibling is an internal node, e.g., node 1 in figure. When node 1 leaves, a lowest-level leaf node in its sibling subtree, say node 2, hands over data to its sibling node 3, and takes over the position of node 1.

We note that the above means of partitioning is very similar to that employed in an existing P2P system called CAN [15], with one crucial difference: CAN *hashes* data into a multi-dimensional space and, since data is expected to be uniformly distributed, a new node splits an existing node's *data space* equally, rather than splitting the *load* equally. At a philosophical level, another key difference in CAN is that the number of dimensions used by CAN is governed not by the dimensionality of the data, but by *routing* considerations. We will see (Section 5) that for low dimensionality (e.g., 2-d), routing in CAN performs very poorly.

Routing First, we must interconnect nodes so that a multi-dimensional range query can be sent along to all the relevant nodes. One simple way to interconnect nodes is to create a link between all "neighboring" nodes, i.e., nodes that share a boundary, resulting in a grid-like structure. (CAN [15] uses this very structure for routing, albeit with all rectangles being roughly the same size.) Observe that this structure is the multi-dimensional analogue of the linked list.

Routing over these "grid" links proceeds by the use of greedy routing. Say the multi-dimensional query requires data in the "rectangle" Q. We define the *distance* from a node N to rectangle Q as the minimum Manhattan distance (L_1 distance) from any point in N's rectangle to any point in Q. The routing protocol forwards the query from a node

to its grid neighbor that reduces the Manhattan distance to Q by the largest amount. An example of query routing is shown in Figure 1(b), where a query is routed from the node labeled '?' to its destination marked 'X'. Once the query has reached one of the nodes with relevant data, say D, node D sends the query along to those of its neighbors that also have relevant data, proceeding recursively until all relevant nodes are reached. Note that each node must know the partition boundaries of each of its neighbors.

Optimized Routing The above "naive" approach is simple; however, it has two major shortcomings: (1) **Non-uniformity**: The number of grid neighbors that a node has is no longer uniform, unlike in a linked list where each node has two neighbors. Unless the distribution of data points is uniform, nodes will have to manage partitions of unequal space. Nodes that manage partitions covering a large space are likely to have more grid neighbors, compared to nodes managing small spaces. This is a problem because we would like the number of neighbors to be balanced among peers, and because it may translate into unbalanced routing load. (2) **Inefficiency**: "Grid" pointers are not very effective in improving query efficiency, especially when dimensionality is low. For example, in a uniform 2-d grid of n nodes, with each node having 4 neighbors, query routing still requires $\Theta(\sqrt{n})$ messages. On the other hand in a linked list with each node having just one skip pointer in addition to its two list pointers, routing requires only $\Theta(\log^2 n)$ messages. Therefore, despite the presence of grid pointers, additional skip pointers are necessary to ensure efficient routing.

We do not attempt to combat the first problem of non-uniform numbers of grid pointers; it is unclear whether it can be overcome while preserving routing load balance. For the second problem, we can have each node maintain "skip pointers" to a few other nodes in order to speed up routing. Note that we maintain the same greedy routing protocol: each node would forward the query to the neighbor closest to the destination, whether the neighbor be a "grid" neighbor, or a neighbor through a skip pointer.

The key is to determine how skip pointers are chosen for each node. In the 1-d case, we simply used skip graphs to assign these skip pointers. However, there is no obvious analogue of skip graphs in higher dimensions[3]. Our approach, therefore, is to develop heuristic methods for establishing skip pointers, and evaluate their performance through experiments. We consider two different approaches for establishing skip pointers:

(a) Random: Each node maintains skip pointers to s nodes chosen at random from the set S of all nodes. In practice, finding a random node is implemented using random walks on the overlay network. Such random skip pointers allow the same kind of query efficiency and routing robustness as multiple *realities* in CAN, without requiring that data be replicated [15].

(b) Space-filling Skip graph: Recall that the key idea in a skip graph was to ensure that each node maintained skip pointers at exponentially increasing distances from it. To emulate such an exponential distribution of skip pointers, we use the following strategy: a linear ordering of nodes is created, such that the distance between nodes in the linear ordering approximates the grid distance between nodes in the native space.

This linear ordering is achieved as follows: The *ID* of a node is defined to be the coordinate of the centroid of its partition, mapped down to one dimension using a space-filling curve, such as the z-curve. Nodes are ordered linearly using this node ID, and a skip graph is built on this linear ordering of nodes, i.e., nodes maintain a linked list sorted by node ID along with additional skip pointers just as dictated in the skip graph. (Note that the skip graph construction continues to occur in a completely decentralized fashion.) Intuitively, this structure is a multi-dimensional approximation of the skip graph, and the skip pointers of nodes are distributed in an exponential fashion.

Discussion The MURK solution offers good data **locality**, since it partitions the data space into exactly as many rectangles as there are nodes. The **metadata** necessary to describe the partitioning is simply the kd-tree itself, whose size is fairly small (proportional to the number of nodes in the system).

However, **load balancing** across partitions is tricky. If the data distribution is static, and does not change over time, it is possible to also obtain good load balance across nodes, using simple techniques developed for load balancing in distributed hash tables, e.g. [1, 13]. When the data distribution is itself dynamic, however, the insertion and deletion of tuples can make the partitions unbalanced even if the set of nodes is fixed. As ongoing work, we are investigating the use of the NbrAdjust and Reorder primitives discussed in Section 3 to achieve dynamic load balancing.

Since the routing properties of MURK are difficult to formalize, we defer a discussion of routing performance to our evaluation in Section 5.

5. EVALUATION

We now evaluate the effectiveness of our proposed solutions for multi-dimensional queries in P2P systems.

Setup Our experiments compare the following approaches:

- **SCRAP** – A SCRAP network using the *z-ordering* space-filling curve [14].
- **MURK-Ran** – A MURK network in which each node maintains $2 \log n$ skip pointers, chosen at random.
- **MURK-SF** – A MURK network in which skip pointers are determined by a space-filling skip graph over nodes' partition centroids.
- **MURK-CAN** – As a baseline for comparison, we will evaluate multi-dimensional range queries over a standard CAN [15] (grid) network with no skip pointers.

We evaluate each approach via simulation. For each high-level action, such as a query, we simulate the messages that are sent between nodes. We measure the performance of an approach with the following two metrics:

- **Locality** – the average number of nodes across which the answer to a given query is stored.
- **Routing Cost** – the average number of messages exchanged between nodes in order to route a given query.

Due to space constraints, we omit a description of our evaluation on other metrics, such as data and routing load distribution across nodes, and comment on these metrics in Section 6.

[3]Note that when node distribution is uniform, it is indeed possible to obtain exactly the same routing performance irrespective of the number of dimensions, using a generalization of skip graphs. We do not discuss this here.

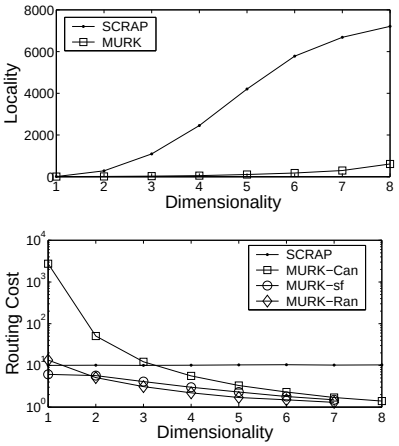

Figure 2: Performance as dimensionality of data is varied

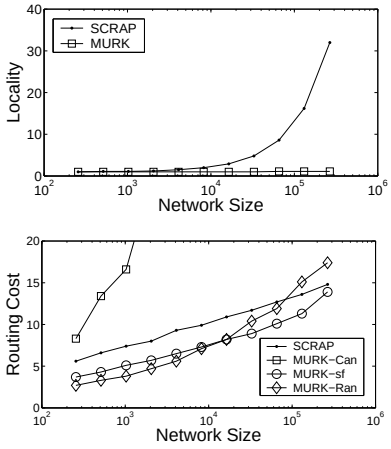

Figure 3: Performance as size of network is varied

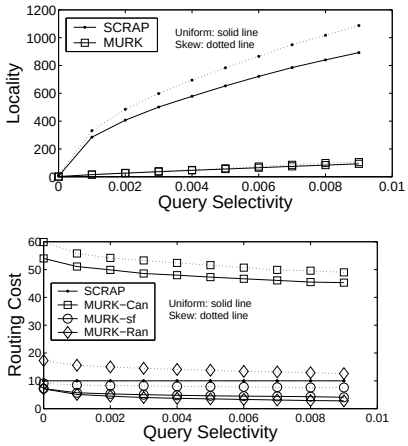

Figure 4: Performance as selectivity of range query is varied

Note that when a query is sent out to the network, some nodes process the query to "route" it to nodes with answers, while other nodes process it because their partitions overlap with the query (and hence, potentially have answers to the query). We differentiate between these two types of work – the first type is captured by Routing Cost, while the second is captured by Locality.

Our workload consists of hypercubic region queries of a given *selectivity* s over the entire data space. We define the selectivity to be s, $0 \leq s \leq 1$, if a fraction s of the entire data space is covered by the query region. Note that selectivity does not determine the number of data points in the region, if data distribution is not uniform. Selectivity may be varied to study how each approach performs under various range sizes. We evaluate queries over two types of data: (1) uniformly random data (UNIFORM) which we generate in any dimensionality desired, and (2) highly skewed/clustered data (CLUSTERED), consisting of 2-d GPS coordinates over a real-life digital photo collection.

We present experiments on a *static* network, constructed by initially inserting all data into a single node, and progressively allowing nodes to join the system to expand it to the desired size. Experiments with dynamic networks are omitted due to space constraints.

5.1 Results

Locality Our first set of experiments compares the merits of the SCRAP and MURK partitioning strategies, in terms of the locality obtained for queries, as the following parameters are varied: (a) the number of dimensions, (b) the number of nodes (c) query selectivity, and (d) data skew. Each experiment varies one of the above four parameters, while holding the others fixed. Locality results are shown in the upper set of graphs in Figures 2, 3, and 4. We will later discuss routing cost results, shown in the lower set of graphs. Recall that the total number of nodes that handle the query is equal to routing cost and locality added together.

Figure 2 depicts performance of each approach as the number of dimensions increases. All queries are of a fixed selectivity of 10^{-5}, executed on a network of 8192 nodes. As we can see, locality (top) degrades very little with increasing dimensions in MURK. However, query locality in SCRAP becomes very poor even for three dimensions; i.e., queries will have to visit many nodes to find all relevant data. Locality suffers in SCRAP because of the curse of dimensionality. As dimensionality increases, nearby points in native space become increasingly far apart in the mapped 1-d space. In addition, because the approximate query-mapping algorithms returns a fixed number of 1-d intervals in which the native region is contained (see Section 3 for discussion), the number of false positives introduced by the algorithm increases with dimensionality. With false positives, nodes that are not actually relevant will process the query; therefore, locality suffers.

Figure 3 depicts performance as the number of nodes in the system increases, for two-dimensional data and a fixed query selectivity. In the locality graph (top), we see that MURK scales well with increasing numbers of nodes; the query almost always hits just one node. SCRAP, on the other hand, does somewhat worse with the locality increasing roughly linearly with the number of nodes. (Note that the x-axis is in logarithmic scale.)

Finally, Figure 4 shows the impact of query selectivity and data skew on performance. The solid lines correspond to experiments with the UNIFORM dataset, while the dotted lines correspond to the CLUSTERED dataset. Along the x-axis we vary the selectivity of the query, executed over 2-dimensional data on a network of 8192 nodes. In the locality graph (top), we see that locality in MURK increases linearly with increasing query selectivity, even for clustered data, which suggests that the data is well balanced across nodes. Locality of SCRAP is much worse than for MURK, with some additional degradation being induced by clustered data. Locality degrades in SCRAP because of the imperfect ability of space-filling curves to map nearby points in native space to nearby points in the mapped 1-d space. As the query covers an increasingly large region in native space, a proportionally larger number of disjoint 1-d intervals can be needed to express this native region. We see in Figure 4 (top) that locality is in fact sublinear, largely due to the fact that many of the disjoint intervals map to the same node.

In summary, MURK far outperforms SCRAP in terms of locality, especially as dimensionality, selectivity, and net-

work size increase.

Routing Our second set of experiments compares the routing costs in SCRAP and MURK, while varying the same four parameters as in the earlier experiments.

In Figure 2 (bottom), we see the routing cost of each approach as the number of dimensions is varied. Not surprisingly, the cost of SCRAP routing is independent of the number of dimensions, since SCRAP routes in a 1-d space, and the number of 1-d intervals output by the query-mapping algorithm is fixed. For MURK, we see that both MURK-sf and MURK-ran are much better than MURK-CAN in low dimensions. For example, MURK-CAN requires over 2 orders of magnitude more messages to route a query in 1-d, compared to MURK-sf and MURK-ran. (Note that the y-axis is in logarithmic scale.) In high dimensions, there are so many grid pointers per node that the improvement obtained from skip pointers becomes marginal.

Figure 3 (bottom) plots the routing cost for 2-d data as the number of nodes is varied, with a fixed-selectivity query. We observe that MURK-CAN performs very poorly, which is expected since the routing cost is $\Theta(\sqrt{n})$. The routing cost of SCRAP increases logarithmically with the number of nodes, just as expected. MURK-sf performs consistently better than SCRAP, suggesting that the space-filling skip graph heuristic is an effective one; it performs better than SCRAP because nodes also have additional grid pointers in MURK, and because only a single query needs to be routed. In SCRAP, we must route one 1-d query for each 1-d interval output by the query-mapping algorithm.

Intriguingly, MURK-Ran outperforms MURK-sf for small network sizes. This is because many skip-graph pointers in small networks are "too close" and do not aid efficient routing as much as in large networks. In fact, it turns out that the "threshold" network size at which MURK-sf outperforms MURK-ran increases with dimensionality. In 1-d, the threshold is at around 1000 nodes, in 2-d around 8000, and even higher values for higher dimensions. Similarly, as queries select more and more data, the threshold size increases; a query selecting lots of data only needs to reach one of the many relevant nodes through routing, which is intuitively akin to routing in a network with fewer nodes.

Figure 4 (bottom) depicts routing costs for uniform and clustered data, as the selectivity of the query is varied. Once again, solid lines depict the cost for uniform data, and dotted lines the cost for clustered data. The cost of SCRAP routing remains flat irrespective of the query range or data clustering, showing that it adapts well to data distribution skew. The cost of MURK-CAN is much higher than that of the other approaches, confirming the need for skip pointers in MURK. Both MURK-sf and MURK-ran perform better for uniform data than for clustered data. For clustered data, we see that MURK-sf performs considerably better than MURK-ran (by about a factor of 2), for all query ranges. This confirms once again that the space-filling skip graph heuristic used by MURK-sf performs well in practice, and is better than using random skip pointers.

In summary, we find that routing in the baseline MURK-CAN network is very expensive and unscalable. Skip pointers are effective in reducing routing cost in MURK, especially as network size increases. In particular, MURK-sf tends to outperform MURK-ran when network size is large, or data is skewed or in low dimensionality. SCRAP routing is also efficient; however, when looking at locality and routing costs together, we find that MURK-sf and MURK-ran are superior approaches across all the variables studied.

6. CONCLUSIONS AND FUTURE WORK

We have presented two approaches for supporting multi-dimensional queries. The first approach, SCRAP, uses space-filling curves with range partitioning, and performs well in low dimensions. It also allows for efficient load balancing across nodes even as tuples are inserted and deleted [7]. The second approach, MURK, partitions data into rectangles in the native space. In combination with a space-filling skip graph, MURK proves much more efficient than SCRAP, especially in high dimensions.

Preliminary experiments suggest that both SCRAP and MURK offer good data and routing load balance when data distribution is static. However, achieving efficient load balance in MURK under dynamic data distributions is potentially expensive, and is the subject of current research. In future work, we plan to investigate the use of SCRAP and MURK for the specific application of P2P photo sharing.

7. REFERENCES

[1] M. Adler, E. Halperin, R. M. Karp, and V. V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proc. STOC*, pages 575–584, June 2003.

[2] A.Silberschatz, H.F.Korth, and S.Sudarshan. *"Database System Concepts"*, chapter 17. McGraw-Hill, 1997.

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

[4] C. Bohm, G. Klump, and H. Kriegel. Xz-ordering: A space-filling curve for objects with spatial extension. In *Proc. Symposium on Large Spatial Databases*, 1999.

[5] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proc. SIGMOD*, 1988.

[6] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proc. Intl. Conf. on Parallel and Distributed Information Systems*, 1993.

[7] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. Technical report, Stanford University, 2004.

[8] S. Ghandeharizadeh and D. J. DeWitt. A performance analysis of alternative multi-attribute declustering strategies. In *Proc. SIGMOD*, 1992.

[9] N. J. A. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. USITS*, 2003.

[10] H. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. SIGMOD*, 1990.

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

[12] G. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proc. INFOCOM*, 2004.

[13] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. In *Proc. 15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA 2003)*, pages 50–59, June 2003.

[14] J. Orenstein and T. Merrett. A class of data structures for associative searching. In *Proc. PODS*, 1984.

[15] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp. A scalable Content-Addressable Network. In *Proc. SIGCOMM*, 2001.

[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, distributed object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, 2001.

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