Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

ALEXANDER REINEFELD, FLORIAN SCHINTKE, AND
THORSTEN SCHÜTT

P2P Routing of Range Queries in Skewed Multidimensional Data Sets

ZIB-Report 07-23 (August 2007)

P2P Routing of Range Queries in Skewed Multidimensional Data Sets*

Alexander Reinefeld, Florian Schintke, and Thorsten Schütt

{reinefeld,schintke,schuett}@zib.de

Zuse Institute Berlin

Abstract. We present a middleware to store multidimensional data sets on Internet-scale distributed systems and to efficiently perform range queries on them. Our structured overlay network *SONAR (Structured Overlay Network with Arbitrary Range queries)* puts keys which are adjacent in the key space on logically adjacent nodes in the overlay and is thereby able to process multidimensional range queries with a single logarithmic data lookup and local forwarding. The specified ranges may have arbitrary shapes like rectangles, circles, spheres or polygons.
Empirical results demonstrate the routing performance of SONAR on several data sets, ranging from real-world data to artificially constructed worst case distributions. We study the quality of SONAR's routing structure which is based on local knowledge only and measure the indegree of the overlay nodes to find potential hot spots in the overlay. We show that SONAR's routing table is self-adjusting, even under extreme situations, keeping always a maximum of $\lceil \log N \rceil$ routing entries.

Key words: structured overlays, range queries, routing, multidimensional data sets

1 Introduction

Data lookup in peer-to-peer (P2P) networks is an important research issue. Many P2P protocols like CAN [20], Chord [28], Kademlia [16], Pastry [22], P-Grid [1], or Tapestry [29] use *consistent hashing* [13] to store (key,value)-pairs in a distributed hash table (DHT). DHTs spread the load in such a way that each participating node holds approximately the same number of objects. A data lookup needs with high probability $\log N$ communication hops in networks of N nodes. While the hashing is important for the logarithmic routing performance, it does not allow queries with partial keywords, wildcards, or ranges, because adjacent identifiers are spread over non-adjacent nodes.

We present a distributed algorithm, named SONAR, that is capable of storing and retrieving multi-attributed data with arbitrary range queries. SONAR

* This paper complements our Europar 2007 paper [25]. It contains a more detailed analysis with additional empirical results. The focus here is on the performance of SONAR on skewed multidimensional data sets.

[25] is a generalization of Chord$^{\#}$ [24] to support multidimensional datasets. In SONAR, the number of attribute domains is not limited; new domains can be added at runtime. Compared to other approaches, SONAR also supports non-rectangular range queries over multiple dimensions with a logarithmic number of routing hops. Non-rectangular range queries are, for example, important for geo-information systems, where locations (objects) are sought that lie within a given distance from a specified position. Another domain of applications are Internet games with thousands or millions of online-players concurrently interacting in a virtual world. In a broader context, SONAR can also be used to build a content-based publish/subscribe system [9], where data items are categorized according to their attribute domain and consumers subscribe to data based on their vicinity or other attributes. Instead of storing data points and querying for subspaces, the subscriptions are stored as subspaces and the publish will become a point query.

In this paper we focus on the routing performance with various data sets, two real-world data sets, one exponential distribution and one artificially constructed pattern that tests the routing algorithm under worst case conditions. We present an adaptive monitoring scheme that detects routing anomalies without using global information. It works by comparing the size of the local routing table, which should always be $\lceil \log N \rceil$, to the observed number of routing hops, which should also be $O(\log N)$. This scheme is used to improve the routing table entries so that—even with highly skewed attributes—logarithmic routing performance is obtained.

1.1 Background

In the literature, two different approaches have been proposed for range queries on structured overlays: space-filling curves on DHTs and key-order preserving mapping functions for various network topologies. The methods based on space-filling curves [3,8,12] incur higher maintenance costs, because they are built on top of a DHT and therefore require an additional mapping. Moreover, space-filling curves cover the key-space by discrete non-overlapping patches, which makes it difficult to support large multidimensional range queries covering several patches in a single lookup. Depending on the patch size, such queries require several independent lookups (ref. Fig. 4a). Chawathe et al. [8] report a query latency of 2 to 3 seconds in networks of 24 to 30 nodes with a system using z-curves on top of OpenDHT.

The second method for range queries on structured overlays maps adjacent key ranges to contiguous ranges in the node space. These methods are key-order-preserving and therefore capable of supporting arbitrary range queries. Since the key distribution is not known *a priori*, one approach, Mercury [7], approximates a density function of the keys to adapt routing tables and thereby to ensure logarithmic routing. The associated maintenance and storage overhead is not negligible, despite a mediocre accuracy. MURK [12] is similar to our approach in that it also splits the data space into boxes which are managed by separate nodes. In contrast to our approach, MURK uses a heuristic based on skip graphs

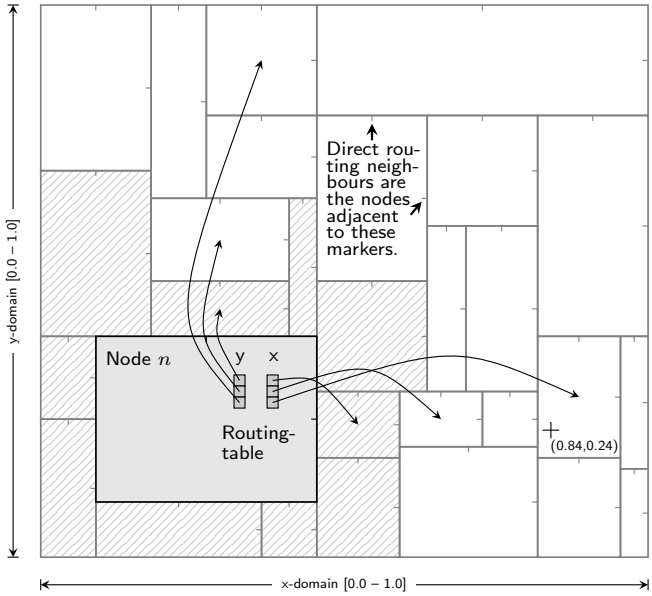


Fig. 1: Routing table for $d = 2$

whereas SONAR uses self-adapting routing tables with pointers to exponentially increasing node distances [24].

In the following section, we introduce the SONAR algorithm and then present performance results on different data sets and provide a more detailed discussion on related work in Sect. 4. We conclude the paper with a summary and outlook.

2 SONAR: Structured Overlay Network with Arbitrary Range Queries

We first present SONAR in its simplest form and then introduce additional features that make the system robust under churn.

SONAR operates on a d-dimensional Cartesian coordinate space as illustrated in Fig. 1. Each dimension represents the domain of one attribute of the keys. Fig. 1 shows a two-dimensional data space with x- and y-intervals of $[0, 1]$. The coordinate space is built on an overlay network which in turn is mapped onto arbitrarily located storage nodes.

The key space is dynamically partitioned among the nodes such that each node is responsible for roughly the same amount of objects. Load-balancing [14] allows to add and remove nodes when the number of objects increases or shrinks or when additional storage space becomes available. This process, called *node join* and *node leave* is detailed in Sect. 2.4. New attributes may be added at any time by adding a new dimension to the overlay.

```
// checks whether the resp. coordinate of b lies between a and c
bool IsBetween(Dim dim, Key start, Key end, Key pos);

void updateRoutingTable(int dim) {
  int  i = 1;
  bool done = false;

  rt[dim][0] = this.Successor[dim];

  while (!done) {
    Node candidate = rt[dim][i - 1].getPointer(dim, i - 1);
    if (IsBetween(dim, rt[dim][i - 1].Key, candidate.Key, this.Key)){
      rt[dim][i] = candidate;
      i++;
    } else
    done = true;
  }
}
```

Fig. 2: Calculation of routing pointers for one dimension **dim**.

2.1 Overlay

Multi-attributed data is stored in a virtual d-dimensional torus topology. Fig. 1 gives an example of geospatial objects arranged in a 2D data space. One data object, marked '+', is located at coordinate (0.84,0.24). Each box is handled by one node of the overlay, and all together they cover the complete key space. Since the keys are usually not uniformly distributed, the boxes have different sizes to balance the load.

Because of the different sizes, a box may have more than one direct neighbor in each direction. The pointers to all neighbors of a node are stored in a *neighbor list*. SONAR uses the neighbor list only for accessing direct neighbors; the routing to distant nodes is done with *routing tables* (Sect. 2.2).

In the one-dimensional case the successors form a ring topology. Following one successor after the other, the ring can be traversed similar as in Chord [28]. Doing the same in the d-dimensional case does not guarantee to end in the starting node, because the rings are skewed due to different node sizes. There is no need to care, however, because queries never need a full round.

2.2 Routing in the Node Space

The neighbor list is—in principle—sufficient for routing, as long as its links are up-to-date. But due to the lack of far-reaching routing information, the routing performance of such a system would be $O(\sqrt[d]{N})$ on average and $O(N)$ in the worst case.

For improved routing, SONAR maintains additional *routing tables* in each node which contain pointers to nodes at exponentially increasing distances in each dimension. With a total of $\log N$ routing pointers per node, the average number of hops is reduced to $O(\log N)$. For a proof see [24].

Compared to Chord, which uses a DHT, SONAR does not compute the routing pointers in the key space, but does this in the node space: When Chord jumps over half of the *key entries* in the ring, SONAR jumps over half of the *nodes* in the ring, which is actually the goal of Chord's behavior. To calculate its i^{th} pointer in the routing table, a node looks at its $(i-1)^{th}$ pointer and asks the remote node listed there, for its $(i-1)^{th}$ pointer. In general, the pointers at level i are set to the pointers' neighbors in the next lower level $i-1$. At the lowest level, the pointers refer to the direct successor [24]:

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1}.getPointer(i-1) & : i \neq 0 \end{cases}$$

The shortest pointer leads to the direct successor and each following pointer doubles the distance by combining two shorter pointers. Such a routing table exists for each dimension. Fig. 2 sketches the pointer update algorithm in pseudo code.

Since there may be several neighbors in each direction (ref. Sect. 2.1), we define the one at the center of the respective side as the neighbor which is used for the first (shortest) routing table entry. It is marked by a small tick on the edges in Fig. 1.

Note that our pointer placement algorithm calculates each pointer in constant time $O(1)$, whereas Chord needs $O(\log N)$ for the same operation because Chord does a key-lookup for calculating a pointer, which needs $\log N$ hops.

2.3 Size of the Routing Table

Given that the total number of nodes is not known locally, how do we limit the number of routing table entries to $\log N$? We could estimate the size of the system (like Mercury) with a density function, but that would cause unnecessary effort while still being imprecise. In SONAR, the maximum number $\log N$ of routing table entries is given implicitly by successively filling the routing table with pointers of exponentially increasing length:

> Starting with the neighbor that is adjacent to the center of node n in routing direction s, we insert an additional $pointer_i$ into the table as long as the following equations holds true:
>
> $$pointer_{i-1}[d] < pointer_i[d] < n[d]$$

When this condition fails, it is clear that the pointer would overspan the current node n (i.e. making more than one round) and hence would not be included in the routing table. Fig. 2 shows pseudo-code for the pointer update algorithm and Sect. 3 shows its accuracy.

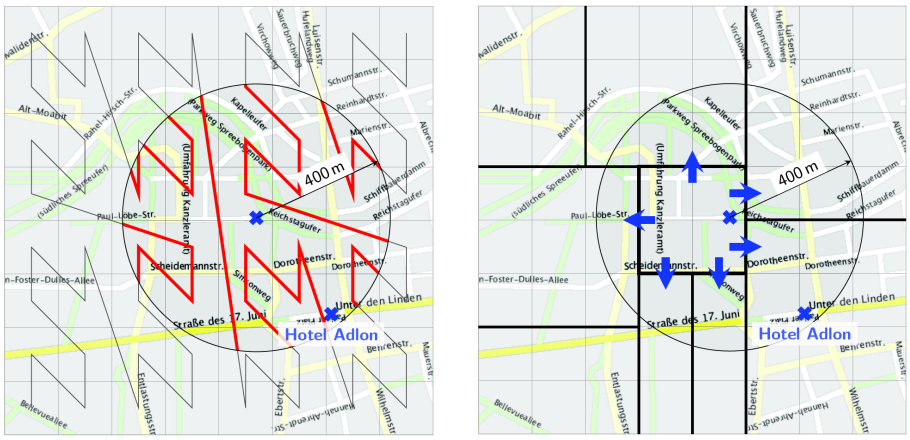
```
double getDistance(Node a, Node b);

Node findNextHop(Point target)
{
    Node candidate = this;
    double distance = getDistance(this, target);

    if (distance == 0.0)
        return this;  // found target

    for (int d = 0; d < dimensions; d++) {
        for (int i = 0; i < rt[d].Size; i++) {
            double dist = getDistance(rt[d][i], target);
            if (dist < distance) {
                candidate = rt[d][i];  // new candidate
                distance = dist;
            }
        }
    }
    Assert(candidate != this);
    return candidate;
}

Node find(Point target)
{
    Node nextHop = findNextHop(target);
    if (nextHop == this)
        return this;
    else
        return nextHop.find(target);
}
```

Fig. 3: Routing to a target.

When all routing tables are filled with the appropriate routing information using the code in Fig. 2, the tables can be used to route queries to their target node(s). Just as in other DHTs, we use a greedy routing strategy. In each node the pointer is taken which maximally reduces the euclidean distance to the target (see Fig. 3).

2.4 Handling Churn

When a node wants to join the network, the area covered by an existing node has to be divided into two parts and the responsibilities split over the two nodes. Two things have to be done:



(a) Z-curve (8 line segments): $8 \cdot log_2(N)$ (b) Neighborhood broadcast: $log_2(N)+6$

Fig. 4: Circular range query.

1. Find a suitable target node with a high load in terms of query- or item-load: This can be done by randomly choosing some nodes (as in Cyclon [27]) and selecting a candidate from the list of resulting nodes.
2. Split the data of the selected node and transfer one portion to the new node: Splits are performed parallel to one of the coordinate system axes. Care must be taken to select the 'right' axis: If a dimension is always favored over the others, the number of nodes contacted in a range query may become disproportionately high and leave operations may become more expensive.

The nodes and their splitting planes form a kd-tree [6]. In contrast to traditional databases, the kd-tree is here not used as an indexing structure, but only for maintaining the topology—similar as in MURK [12]. When a node leaves the system, the occupied space must not be merged with an arbitrary neighbor, but only with a neighboring node which is also a sibling in the kd-tree. By keeping the tree balanced the probability of having a sibling as a neighbor increases.

2.5 Range Queries

SONAR supports multi-attributed range queries. Given d intervals over d attribute domains, the range query will return all values between the lower and upper bounds for each domain. Because of their shape, such range queries are named d-dimensional *rectangular* range queries.

In practice, circles or polygons often better fit user requests. Fig. 4 illustrates a two dimensional circular range query where the query is defined by a center and a radius. For this example, we assume a person located in the governmental district of Berlin searching for a hotel. The center is the location of the person

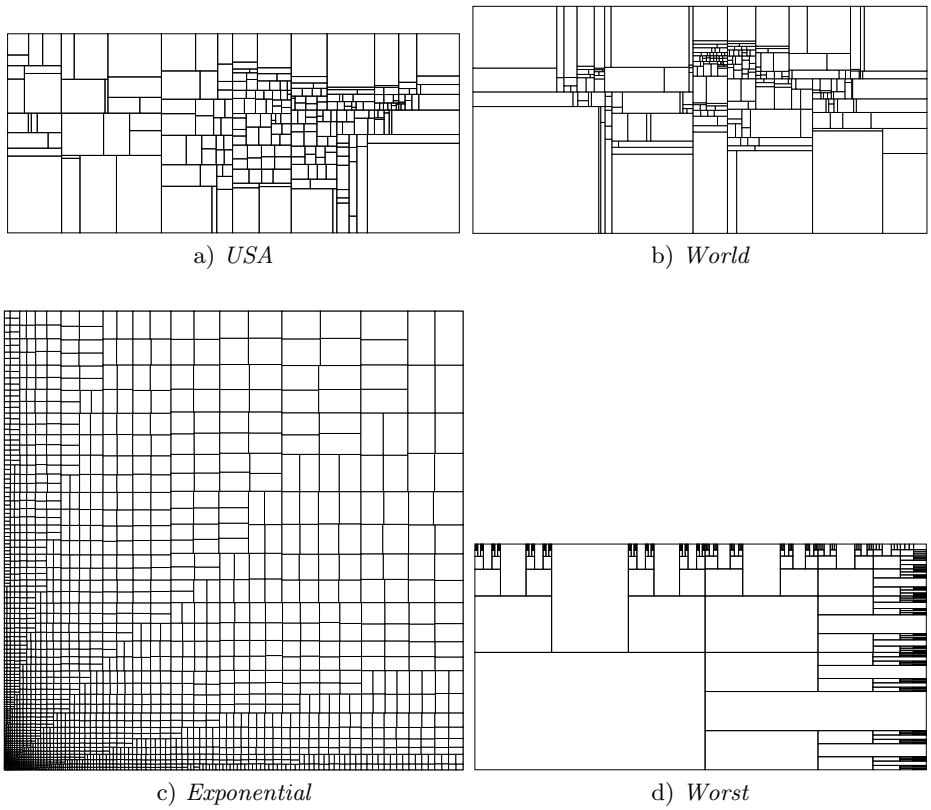
and the radius is the acceptable 'walking distance'. In a first step (Fig. 4b) the query is routed to the node responsible for the center of the circle. Thereafter the query is forwarded to all neighbors that partially cover the circle. The query is checked against their local data and the results are returned to the requesting node. Fig. 5 shows the pseudo code for this range query algorithm. To avoid redundant operations, the query is only forwarded to nodes that have not been asked before.

In contrast to overlays with space-filling curves, SONAR just needs to route to the node in the center of the circle. When space-filling curves are used (Fig. 4a), several line segments of the curve may be responsible for the range and for each segment a complete routing (each in $O(\log N)$) must be performed to retrieve all results.

```
void doRangeQuery(Range r, Operation op, Id id)
{
    // avoid redundant executions
    if (pastQueries.Contains(id))
        return;
    pastQueries.add(id);

    foreach (Node neighbor in this.Neighbors)
        if (r ∩ neighbor.Range != ∅)
            neighbor.doRangeQuery(r \ this.Range, op, id);

    // execute operation locally
    op(this, r);
}

void queryRange(Range r, Operation op)
{
    Node center = find(r.Center);
    center.doRangeQuery(r, op, newId());
}
```

Fig. 5: Range query algorithm.

In SONAR the neighborship is given by the key space and queries may take arbitrary shapes. The query is sent to the center node and from there forwarded to all neighbors which possibly may have a subset of the query range. These nodes are then responsible for the cut-off. A more sophisticated implementation could route to several points on the outline of the shape and start the flooding towards the center.



a) *USA* b) *World*

c) *Exponential* d) *Worst*

Fig. 6: The four data sets used in the performance evaluation.

3 Evaluation

We used four different data sets to evaluate the performance of SONAR: two with real geographic coordinates and two with synthetically generated objects. For each data set, we did several experiments with varying data sizes. Each experiment was started by partitioning the two-dimensional space into non overlapping rectangular patches so that each patch contains about the same amount of data items. This was done by recursively splitting the patches at alternating sides until the number of data items in the area dropped below a given threshold. Fig. 6 illustrates the four data sets:

a) ***USA*** contains the locations of the 13,509 cities in the United States with a population of at least 500 inhabitants. This data set is taken from the TSPLIB, see http://www.tsp.gatech.edu/.
b) ***World*** is also taken from the TSPLIB. It contains the 1,904,711 largest cities in the world. Each city location (longitude and latitude) was mapped onto a doughnut-shaped torus rather than a sphere, because the poles of a

sphere would become a bottleneck and the routing direction in the western hemisphere would interfere with that of the eastern hemisphere (southwards vs. northwards).

c) ***Exponential*** was generated by running a random number generator with an exponential distribution and placing the data points on the 2D plane.

d) ***Worst*** is a synthetically constructed worst case pattern that bends the direct routing neighbors to nodes with few successors in the same routing direction. This makes it difficult for SONAR to find a short routing path, because for each dimension i, the routing table holds less than $\log N_i$ pointers (with N_i being the number of keys in this direction only).
The network was constructed by recursively splitting the rectangular area into four patches: south-west (SW), south-east (SE), north-west (NW), and north-east (NE).
   - The SW part is slightly larger in both directions than the other parts and it is not further subdivided.
   - The NW part is split into three equally-sized vertical slices: The middle one fills the whole space, while the outer ones fill just a bit more than half of the available space, and the remainder is recursively split as before—up to the specified recursion depth.
   - The SE part follows the same pattern as NW, but is splitted horizontally.
   - For the NE part, the described global pattern with the splitting into four parts is applied recursively until the recursion depth is reached. If the recursion depth is reached, the remaining part is added as a whole. We used a recursion depth of 5 iterations, which lead to 348 rectangles.

The two TSPLIB data sets have been selected because their Zipf-like distribution [30] is typical for many applications. *Exponential* was used to check the routing behavior with a skewed distribution where the boxed have very different sizes and *Worst* is an artificially constructed data pattern that causes a worst case routing behavior.

We did not include results of a uniform data distribution, because it partitions the data space into a regular grid. Each node then has a routing table with exactly $\log N$ entries, that is, $\frac{logN}{i}$ pointers for each dimension $0 < i \leq d$. In this ideal data distribution, SONAR's pointer placement algorithm calculates precisely exponentially spaced pointers, which results obviously in an optimal logarithmic routing performance.

3.1 Average Routing Performance

To evaluate SONAR's routing performance we conducted an all-to-all search by issuing a query from each node to the center of each other node, resulting in a total of N^2 queries.

Fig. 7 shows the average number of routing hops for various network sizes. As can be seen, the curves of *USA* and *World* lie within the expected average of $0.5 \log N$ hops depicted by the straight line, which indicates that the scheme works well with Zipf-distributed [30] real-world data.

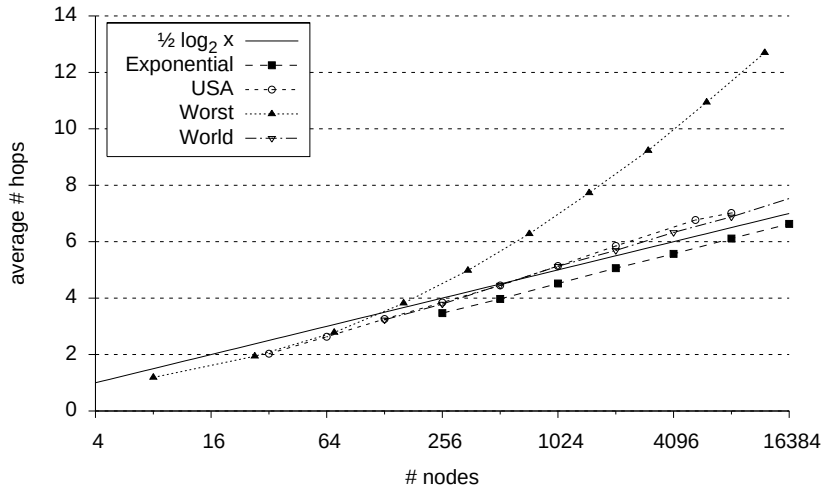


Fig. 7: SONAR routing performance on networks of various sizes for 4 data sets.

The results for the *Exponential* data set further support the claimed $\log N$ routing performance, despite the greater variation in the box sizes which makes the pointer placement more difficult. Only the *Worst* data set shows a degradation to $\log^2 N$ for the larger instances. In Sect. 3.4 we present a monitoring scheme that detects and eliminates such anomalies without using global information.

3.2 Routing Load per Node

In another set of experiments, we checked the distribution of the routing load over all participating nodes. The larger the spectrum of node sizes, so our conjecture, the higher the routing load at those nodes that are responsible for a greater part of the data space, because they are likely to have more incoming routing pointers.

Fig. 8 depicts the *USA* data set with a network of 128 nodes. Each box is handled by one node and contains about the same number of cities. The lines show all routing hops (including wrap-around hops) performed in an all-to-all search. We counted a total of 50,824 hops, resulting in an average of $50,824/16,256 = 3.1$ hops per query, which is slightly better than the expected $1/2 \log 128 = 3.5$ hops.

A closer inspection of the results in Fig. 8 reveals that some nodes have a much higher indegree than others, see for example the 'Florida Node' at the right bottom. This may result in a higher CPU load, because nodes with many incoming routing pointers will receive more forwarding requests from other nodes. To further investigate this aspect, we plotted the distribution of the node indegrees

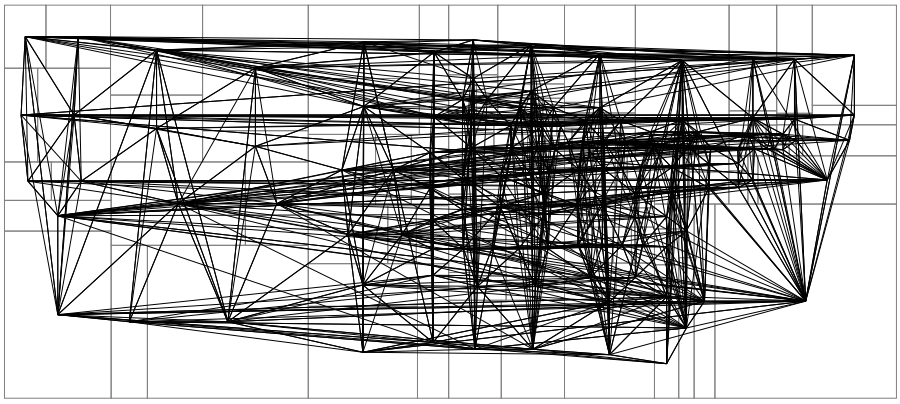


Fig. 8: All routing hops of an all-to-all search (USA, $N = 128$)

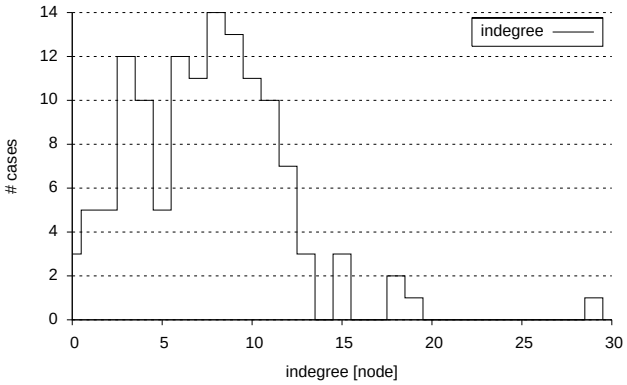


Fig. 9: Node indegree distribution for Fig. 8 (USA, $N = 128$)

in Fig. 9. It can be seen that the indegree of most nodes is near the expected value of $\log 128 = 7$. Only seven nodes have an indegree > 14. With an indegree of 29, the node covering Florida is the largest.

Another interesting aspect is the frequency of the use of long versus short pointers. Ideally, a query would start with a long-distance hop and then successively reduce the hop distance until the target is found. Hops to direct neighbors should only occur at the very end of the lookup. A closer look at our experimental results reveals that 78% of all hops were indeed long-distance hops and only 22% were direct neighbor hops. More precisely: From the 3.13 hops, that were on the average required to find the target in our all-to-all search, only 0.68 hops were direct neighbor hops, while the other 2.45 hops were long-distance hops. This is another indication of SONAR's lookup efficiency.

3.3 Accuracy of the Routing Table Information

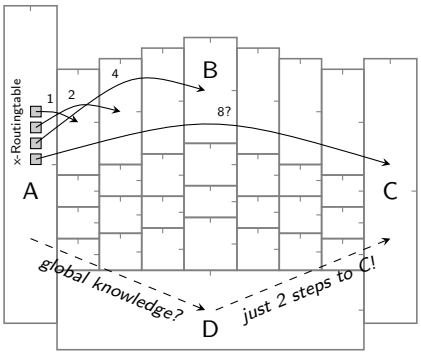


Fig. 10: Erroneous pointer lengths due to lack of global information: Node A believes that its routing pointers span distances of 1, 2, 4, and 8 hops respectively. From a global perspective, however, A's longest pointer spans only 2 hops, see the arc below.

SONAR has only local information for building the routing tables. Each node computes its pointers by recursively asking a remote node for its pointer information. This could lead to inaccuracies in the long pointers, because they are recursively constructed using local information of other shorter pointers.

Fig. 10 illustrates a situation, where local information misleads the pointer update algorithm to include pointers in the routing table which are too short. Each of the shown pointers is built by recursively concatenating the next shorter local pointers. This causes pointers of a believed length of 1, 2, 4, and 8 to be inserted into the routing table of node A. In reality, however, the longest pointer spans only two hops instead of eight. A global observer with an optimal routing strategy would just make two hops via node D to arrive at node C. This information is, unfortunately, not available in node A.

To check whether this situation occurs in practice, we plotted the actual pointer lengths versus their expected values. Fig. 11 and 12 show the deviation of the pointer lengths for the *World* and *Worst* data sets. The expected pointer length is always a power of two and can be derived from the pointer update algorithm. For the actual length we measured the number of hops an algorithm with global information would need to get from one node to the other by only moving east- or northwards as done by our greedy routing.

In both Figures, the large majority of the pointers is set correctly at their expected values, while only a small number of routing entries deviates. Since no direction is preferred, the deviations in both directions compensate each other. Interestingly, this also applies to the *Worst* data set with a slight tendency

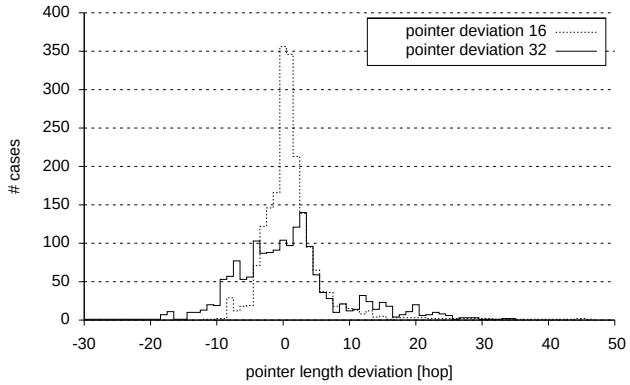


Fig. 11: Deviation of pointer lengths due to local information (*World*, $N = 1024$). The measured lengths are centered around their expected length of 16 resp. 32.

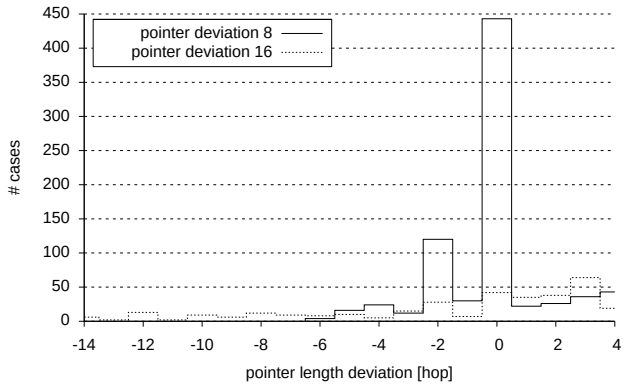


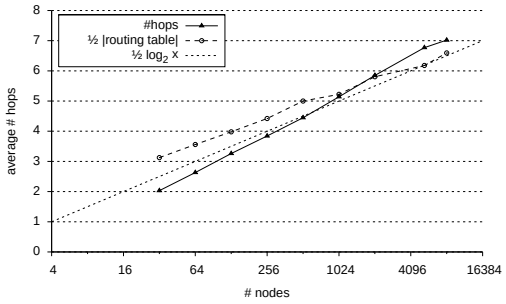
Fig. 12: Deviation of pointer lengths due to local information (*Worst*, $N = 1490$). The measured lengths are centered around their expected length of 8 resp. 16.

towards shorter pointers. This indicates that the $\log^2 N$ routing behavior of *Worst* is not caused by incorrect pointer lengths.

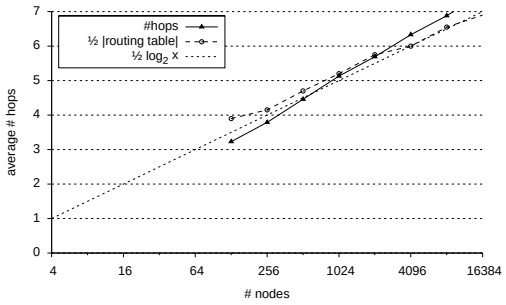
3.4 Dealing with the Worst Case: Self-Improving Routing Tables

SONAR exhibits a logarithmic routing performance in all but one of the four analyzed data sets. Only *Worst* needs $O(\log^2 N)$ hops on the average. Even though this artificially constructed data pattern is very unlikely to occur in practice, it must be dealt with. We conjecture that the degradation is caused by missing information in the routing table. This could happen if the table does not cover the whole key space with exponentially spaced pointers or if the table is not filled with $\log N$ pointers.

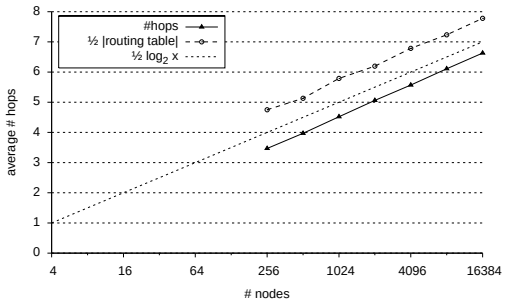
a) USA



b) World



c) Exponential



d) Worst

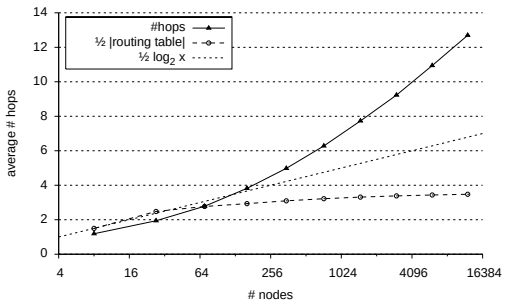


Fig. 13: SONAR performance on four data sets for various network sizes. The straight line shows the measured number of routing hops and the dotted line shows the number of hops predicted from the size of the routing table.

Fig. 13 shows the measured number of routing hops (straight lines) and the expected hops (dotted lines) for all four data sets. The expected hop number was computed by counting for all nodes the sizes of the routing table and dividing it by two. As can be seen, SONAR needs fewer hops than expected in the smaller networks of *USA* and *World*, while it needs slightly more for the larger networks. More pronounced is the gain in the *Exponential* data set, where SONAR saves one hop at all network sizes.

This situation is quite different for the *Worst* data set. Here, the average hop number (top curve) increases as the networks get larger. The degrade in the routing performance is paralleled by a reduction in the routing table size (bottom curve). The routing table does not grow with increasing network size because we constructed the *Worst* data set in such a way, that SONAR is not able to find enough pointers to fill its routing table with $\log N$ exponentially spaced targets.

Self-improving routing tables.

Two steps are necessary to fix the problem: First, a monitoring scheme must be employed to detect any degradation in the routing performance from the expected $\log N$-routing, and then measures for improvements must be taken. Since global information is not available in P2P systems, both steps must be done with local information only.

The monitoring is done by piggybacking the hop count on top of queries. Several hop counts are collected and averaged. From the construction of the routing table with exponentially spaced pointers (Sect. 2.2), we expect it to have $\log N$ entries. Comparing this to the observed hop number, which should be $0.5 \log N$ on average, we are able to assess the quality of the routing—based on local information.

When a routing degradation has been detected, the mapping of the key space to the nodes must be changed. This can be done, for example, by augmenting the load metric with a factor that takes the routing load (resp. the node indegree) into account. Nodes with a large indegree should split their key space to achieve a better workload balance and to improve the routing table.

4 Related Work

The first structured overlay networks like Chord [28] or CAN [20], published in 2001, allow to organize unreliable peers into a stable, regular structure, where each node is responsible for one part of the identifier space(ring segment in Chord, rectangle in CAN). Due to consistent hashing these system allow to distribute the nodes and keys equally over the system and to route with $O(\log N)$ resp. $O(\sqrt[d]{N})$ hops. Both handle one-dimensional keys but do not support efficient range queries, because adjacent keys are not mapped to adjacent nodes in the overlay.

One-dimensional range queries.

To allow efficient range queries, structured overlays without hashing have been developed which put adjacent keys to nodes adjacent in the overlay. With

one logarithmic lookup for the start of the range, the range query can be performed by that node and the nodes adjacent in the logical structure of the overlay. One major challenge for such systems is the uneven distribution of keys to nodes and the pointer maintenance for efficient routing. SkipGraphs [4], a distributed implementation of skip lists [19], for example, use probabilistically balanced trees and thereby allow efficient range queries with $O(\log N)$ performance.

Ganesan et al. [11] further improved SkipGraphs with an emphasis on load-balancing. Their load-balancing scheme maps ordered sets of keys on nodes with formally proven performance bounds, similar to [14]. For routing, they deploy a SkipGraph on top of the nodes, guaranteeing the routing performance of $O(\log N)$ with high probability.

Mercury [7] does not use consistent hashing and therefore has to deal with load imbalance. It determines the density function with random walk sampling, which generates additional traffic for maintaining the pointer table.

Multidimensional keys and range queries.

There exist several systems, that support multidimensional keys *and* range queries. They can be split into two groups:

a) Space filling curves.

Several systems [3,12,23] have been proposed that use space-filling curves to map multidimensional to one-dimensional keys. Space-filling curves are locality preserving, but they provide less efficient range queries than the space partitioning schemes described below. This is because a single range query may cover several parts which have to be queried separately (Fig. 4a).

Chawathe et al. [8] implemented a 2-dimensional range query system on top of OpenDHT using Z-curves for linearization. In contrast to many other publications, they report performance results from a real-world application. Due to the layered structure the query latency is only 2 s – 3 s for 24 – 30 nodes.

b) Space partitioning schemes.

Another approach is the partitioning and direct mapping of the key space to the nodes. SONAR belongs into this group of systems. The main differentiating factor between such systems is their indexing and routing structure.

SWAM [5] employs a Voronoi-based space partitioning scheme and uses a small-world graph overlay [15] with routing tables of size $O(1)$. The overlay does not rely on a regular partitioning like a kd-tree [6], but it must sample the network to place its pointers.

Multi-attribute range queries were also addressed by Mercury [7], but their implementation uses a large number of replicas per item to achieve logarithmic routing performance. SWORD [18] uses super-peers and query-caching to allow multi-attribute range queries on top of the Bamboo-DHT [21].

Ganesan et al. [12] proposed two systems for multidimensional range queries in P2P systems – SCRAP and MURK. SCRAP follows the traditional approach of using space-filling curves to map multidimensional data down to one dimension. Each range-query can then be mapped to several range queries on the one dimensional mapping.

MURK is more similar to our approach as it also divides the data space into hypercuboids with each partition assigned to one node. In contrast to SONAR, MURK implements a heuristic approach based on skip graphs.

5 Conclusion

We presented an algorithm for processing multi-attributed range queries in P2P networks. Our algorithm, named SONAR, handles range queries with an arbitrary number of attribute domains. The ranges do not need to be rectangular but may have arbitrary shapes like circles, spheres or polygons.

The recursive pointer placement algorithm used in SONAR needs just one hop for updating an entry in the routing table. This algorithm can also be used to speed up other P2P algorithms that use exponentially placed routing pointers. As an example, the $O(\log N)$ pointer placement used in Chord can be improved to $O(1)$.

We evaluated the routing performance on four different data sets: two data sets with real-world geographical data, one set with exponentially distributed two-dimensional data and one artificially constructed worst case data set. As shown in the paper, SONAR needs $O(\log N)$ hops for routing in the first three data sets, while the performance degrades to $O(\log^2 N)$ in the worst case data set—all this with local routing tables of size $\log N$.

Furthermore, we presented in Sect. 3.4 a performance monitoring scheme that detects routing inefficiencies without using global information. It works by comparing the local routing table size to the expected number of routing hops, thereby estimating the total number of nodes in the network N. This scheme can be used to improve the routing table so that—even with highly skewed data sets—logarithmic routing performance is obtained.

Our future work will focus on various strategies for the self-regulation of the routing process. Furthermore, SONAR will be implemented on PlanetLab to obtain performance results on Internet-distributed nodes.

Acknowledgements

Part of this research was supported by the EU projects SELFMAN and XtreemOS. Thanks to Slaven Rezić for the street map of Berlin (Fig. 4).

References

1. K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. *CoopIS*, Oct. 2001.
2. K. Aberer, L. Onana Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The Essence of P2P: A Reference Architecture for Overlay Networks. P2P 2005, 2005.
3. A. Andrzejak, and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. P2P 2002, 2002.

4. J. Aspnes and G. Shah. Skip graphs. *SODA*, Jan. 2003.
5. F. Banaei-Kashani and C. Shahabi. SWAM: a family of access methods for similarity-search in peer-to-peer data networks. *CIKM*, Nov. 2004.
6. J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, Vol. 18, No. 9, 1975.
7. A. Bharambe, M. Agrawal and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. *ACM SIGCOMM 2004*, Aug. 2004.
8. Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A Case Study in building Layered DHT Applications. SIGCOMM'05, Aug. 2005.
9. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many Faces of Publish/Subscribe. ACM Computing Surveys, Vol. 35, No. 2, pp. 114131, June 2003.
10. V. Gaede, and O. Günther. Multidimensional access methods. ACM Computing Surveys, 30 (2), 1998.
11. P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-t o-Peer Systems. *VLDB 2004.*
12. P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems. *WebDB 2004.*
13. D. Karger, E. Lehman, T. Leighton, R. Panigrah, M. Levine and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. *29th Annual ACM Sympos. Theory of Comp.*, May 1997.
14. D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. *IPTPS 2004*, Feb. 2004.
15. J. Kleinberg. The small-world phenomenon: An algorithmic perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000.
16. P. Maymounkov and D. Maziéres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *IPTPS*, March 2002.
17. M. Naor and U. Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. *SPAA 2003.*
18. D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. *14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, Jul. 2005.
19. W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, June 1990.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *ACM SIGCOMM 2001*, Aug. 2001.
21. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *Proceedings of the USENIX Annual Technical Conference*, Jun. 2004.
22. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Middleware*, Nov. 2001.
23. C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 19-26, May/June 2004.
24. T. Schütt, F. Schintke and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.
25. T. Schütt, F. Schintke and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.
26. Y. Shu, B. Chin Ooi, K. Tan, A. Zhou. Supporting Multi-dimensional Range queries in Peer-to-Peer Systems. P2P'05, Sep. 2005.

27. S. Voulgaris, D. Gavidia and M. van Steen: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13:197 217(21), Jun. 2005.
28. I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001.
29. B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, Vol. 22, No. 1, Jan. 2004.
30. G. Zipf. Relative Frequency as a Determinant of Phonetic Change. Reprinted from *Harvard Studies in Classical Philiology*, 1929.