# Roogle: Supporting Efficient High-Dimensional Range Queries in P2P Systems

Di Wu, Ye Tian and Kam-Wing Ng

Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{dwu, ytian, kwng}@cse.cuhk.edu.hk

**Abstract.** Multi-dimensional range query is an important query type and especially useful when the user doesn't know exactly what he is looking for. However, due to improper indexing method and high routing latency, existing schemes cannot perform well under high-dimensional situations. In this paper, we propose Roogle, a decentralized non-flooding P2P search engine that can efficiently support high-dimensional range queries in P2P systems. Roogle makes improvements on both indexing and routing. The high-dimensional data is indexed based on the maximum or minimum value among all dimensions. This simple indexing method performs rather well under high-dimensional situations and tolerates data points with missing values or different dimensionality. To speed query routing, Roogle is built on top of our proposed structured overlay - Aurelia, which has better routing performance by exploiting node heterogeneity. Aurelia also guarantees the data locality and efficiently support range queries. Experimental results from simulation validate the scalability and efficiency of Roogle.

## 1 Introduction

To fully exploit the gigantic amount of structured data (e.g., multimedia data, computation resources, scientific dataset, etc) shared in the P2P systems, which are inherently spatial, it is expected that advanced query types could also be efficiently supported.

Among advanced queries, the multi-dimensional range query [1] is an important query type and particularly useful for discovery purposes. When the user doesn't know exactly the properties of desired objects, range queries on multiple attributes can be issued to perform search. For instance, in order to locate the movies published in recent two months and with the size smaller than 400MB, one 2-dimensional range query can be issued: *(20060101≤ PublishDate< 20060201) AND (0M< FileSize < 400MB))?*. However, it should be noticed that the query dimensionality varies with different data types, and may be very high in some scenarios (e.g., color histogram, stock dataset).

[1] Also known as multi-attribute range query.

To date, there have been quite a few schemes (e.g., [1],[2],[3],[4],etc) being proposed to enable multi-dimensional range queries in P2P systems, but most of them can only function well under the low-dimensional situations. In case of high dimensionality, their performance will deteriorate greatly due to the "*curse of dimensionality*" [5]. Additionally, their query routing is also inefficient. Many approaches are built on top of traditional structured overlays (e.g., Chord, CAN), which can only provide O(log$N$)-hop routing performance. Considering the large query range of high-dimensional queries, the latency will be intolerable for users.

In this paper, we address the above issues by proposing Roogle, a decentralized non-flooding P2P search engine that can efficiently support high-dimensional range queries. Due to the characteristics of high-dimensional data, it is safe to index data simply based on the maximum or minimum values among all dimensions. Such indexing method performs rather well under high-dimensional situations, and also tolerates data points with missing values or different dimensionality. At the same time, a locality-preserving structured overlay called *Aurelia* is proposed as the underlying overlay. The routing performance of Aurelia is greatly improved by exploiting node heterogeneity. The size of routing table on each node is proportional to the node capacity, and multicasting is adopted for scalable routing table maintenance. The size of data index on each node is also adaptive to node capacity, and the data key is allowed to be registered into multiple nodes to guarantee the reliability. Finally, the performance of Roogle is evaluated via simulation, and the experimental results confirm the scalability and efficiency of our design.

The rest of the paper is organized as follows. Section 2 provides an overview of related work. Section 3 describes the detailed design of Roogle. Experimental results are presented and analyzed in Section 4. Section 5 concludes the paper.

## 2 Related Work

In unstructured P2P systems, flooding-based approaches are widely adopted for multi-dimensional range queries, but huge traffic will be incurred. Therefore, most of the existing non-flooding schemes are built on top of structured P2P systems in order to limit the traffic.

In [1], a wide-area resource discovery engine called SWORD is implemented to answer multi-attribute range queries so as to locate computation resources. In SWORD, nodes participate in multiple DHTs, one per attribute. A query is routed in the DHT overlay corresponding to its most selective attribute. In MAAN [2], order-preserving hash is adopted to preserve data locality, and query selectivity is used to identify the relevant DHT for query routing. Similar schemes also include Mercury [4], PHT [6], etc. They all require prior knowledge of attribute selectivity, which is a kind of drawback in the design.

In Squid [3] and SCRAP [7], Space-Filling Curves(SFC) is introduced for dimension reduction. Multi-dimensional data is mapped to single-dimensional data and then is range-partitioned across peers. However, the data locality of SFC will become worse with the increase of dimensionality. To improve the data

locality, MURK [7] and SkipIndex [8] adopt KD-tree to partition the multi-dimensional space into hypercuboids, each of which is assigned to a node. They use hashless CAN [9] and SkipGraph [10] as the underlying overlay respectively. However, in dynamic environments, both of them suffer from the problem of load balancing. Additionally, it is hard for them to index data points with missing values in some dimensions, or with different dimensionality.

Different from the above approaches, our proposed Roogle doesn't require prior knowledge of attribute selectivity, and works efficiently under the high-dimensional situation. It avoids the poor performance of range queries caused by awkward DHT-based designs. By exploiting node heterogeneity and guaranteeing data locality, both indexing robustness and routing performance are much improved in Roogle.

## 3 System Design of Roogle

In this section, we will introduce Roogle from four aspects: overlay structure, data indexing, query processing and load balancing.

### 3.1 Overlay Structure

The underlying overlay provides the basic routing function for uplevel applications. To improve routing performance, Aurelia is proposed as the underlying overlay structure to support Roogle.

Like Chord, Aurelia organizes nodes into a circular ring that corresponds to the ID space $[0, 2^{128} - 1]$. Initially, each node is assigned a unique node ID by uniform hashing, which consists of three parts: *RangeID*, *LevelID* and *RandomBits*. For example, for the node $\underline{001}010...10\underline{10}$, its *RangeID* and *LevelID* are $\underline{001}$ and $\underline{10}$ respectively, and the bits in between are *RandomBits*.

The *RangeID* is the first $r$-bit prefix of the node ID, which defines the value range that the node is responsible for. Aurelia abandons the using of hashing to distribute the data objects across the nodes, for hashing destroys the data locality. Instead, Aurelia simply maps the data object to the node according to its normalized value. All the data keys whose first $r$-bit prefix is the same as the *RangeID* will be placed on that node. The length of *RangeID* determines the size of data index if the objects are uniformly distributed. The nodes can choose a suitable length of *RangeID* based on its capacity. Formally, supposing the range of the ring is mapped to [0, 1], the node with *RangeID* of $b_0 b_1 \dots b_{r-1}$ will take charge of the range:

$$Range(b_0 b_1 \dots b_{r-1}) = \left[ \frac{\sum_{i=0}^{r-1} b_i \times 2^{r-i-1}}{2^r}, \frac{\sum_{i=0}^{r-1} b_i \times 2^{r-i-1} + 1}{2^r} \right)$$

The *LevelID* is the $l$-bit suffix of the node ID, which determines the routing table size together with "*Routing Regions*". Here, "*Routing Region*" refers to the region where the routing pointers take effects. Under uniform node distribution,

there is only one routing region, i.e., the whole ID space. In case of non-uniform node distribution, the whole ID space is divided into multiple routing regions based on the node density. The node can have different *LevelID*s for different routing regions. For a node $A$, only the nodes whose node ID is within the routing region and the last $l$ bits are the same as $A$'s *LevelID* appear in the routing table of node $A$. The shorter the *LevelID*, the bigger the routing table.

In case of uniform node distribution in the ring, the routing scheme of Aurelia will be similar to SmartBoa [11]. However, with node dynamism or load balancing reasons(e.g., under-loaded nodes quit their current position and rejoin the heavy-loaded region), the node distribution in the ring may become non-uniform. In such scenarios, SmartBoa's routing scheme will become inefficient. For the region with dense node distribution, more hops will be required than expected(as illustrated in Fig. 1)). This drawback leads to the design of Aurelia.

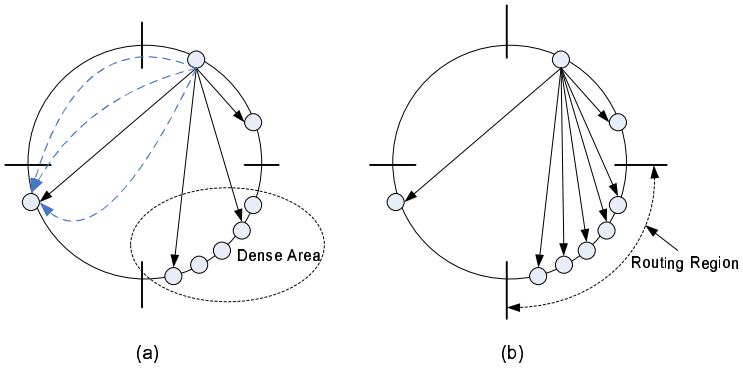


**Fig. 1.** Routing under non-uniform node distribution (a) SmartBoa's routing pointers; (b) Aurelia's routing pointers

The basic idea of Aurelia is to adjust the density of routing pointers so as to adapt to the node distribution. In the sparse routing region, fewer routing pointers are allocated; while in the dense region, more routing pointers will be allocated for fast routing. The density of routing pointers is controlled by adjustment of *LevelID*. The node seems like hosting multiple virtual nodes with *LevelID* in different lengths, each of which corresponds to a different routing region. By adjusting the length of *LevelID* in different routing region, we can make the density of routing pointers congruous to the node distribution. Note that all these virtual nodes share the same node ID.

As to the routing strategy, Aurelia performs in a greedy-like style. Given a target key, the node always selects the nearest one in the routing table for forwarding. For the powerful nodes, the large routing table enables them to complete the routing even in 1 hop.

To build such kind of routing table, a big challenge is to maintain the node-count distribution in a decentralized way. Aurelia adopts the technique of sampling to collect the statistical information and build the approximate node-count histogram locally. In addition to producing local estimate, the node also periodically makes sampling uniformly in the ring. According to the collected information, the node chooses the most recent statistical data to produce the node-count histogram. After the histogram is built and normalized, the node can adjust its routing pointers in the following way:

Each block in the histogram represents a continuous region with the size of $1/2^k$ of the whole ID space. Based on the ratio between the block's density level and the average density level, the length of virtual nodes' *LevelID* will be increased or decreased accordingly. Such adjustment will impact the density of routing pointers in different routing regions. For sparse routing region, fewer routing pointers are needed, so the *LevelID* of the virtual node in that region can be extended. On the contrary, for dense routing region, more routing pointers are included by shortening the *LevelID*. However, the extension and reduction of the *LevelID* cannot deviate too much from its real level.

To maintain the routing pointers in a scalable fashion, Aurelia adopts multicasting to distribute the events of node join, leave or status change. The multicast tree doesn't require explicit management. It is based on the *LevelID* and routing region to disseminate the event from high-level nodes to low-level nodes. The details of the multicast algorithm are as follows: Each node maintains a list of "*top nodes*", whose *levelID* is the shortest suffix of current nodes's *levelID*. "*Top nodes*" are often powerful nodes that hold more routing pointers. When a node joins or changes its status, it first forwards the event with the node ID to one of its top nodes randomly. Then this top node disseminates the event to all the nodes whose *levelID* is the suffix of the reported node ID. But for the virtual node, if an announced node ID is not within the routing region it wants to be notified, the event is ignored. In every step of the multicast process, the node that receives the event first sends the event to the next lower level, then notifies the other nodes with the same *levelID*. By this approach, we can guarantee the event to be exactly delivered from the high-level nodes to the low-level nodes. When a node leaves, its predecessor will detect the event and help to notify the top nodes and propagates the change information to all the related nodes.

### 3.2 Data Indexing

Generally, for a multi-dimensional range query, all values of all dimensions must satisfy the query range along each dimension. If any of them fails, the data point will not be qualified. Therefore, a straightforward approach is to index on a small subset of the dimensions. However, the effectiveness of such an approach depends on the data distribution of the selected dimensions and requires prior knowledge.

To avoid this drawback, we index the data based on the maximum or minimum value among all dimensions of the data point. As the proportion of data

points with a very big or small value in one dimension will increase with dimensionality, so it is safe to index the data points based on their edges. The feasibility of such indexing method has been validated by [12] in traditional database field.

The transformation process is a simple mapping, which is computationally inexpensive. Let $x_{min}$ and $x_{max}$ be respectively the smallest and biggest values among all the $d$ dimensions of data point $< x_1, x_2, ..., x_d >$, $x_j \in [0, 1]$, $1 \leq j \leq d$. Let the corresponding dimensions of $x_{min}$ and $x_{max}$ be $d_{min}$ and $d_{max}$ respectively, then the high-dimensional point $x$ can be mapped to a point $y$ in the single-dimensional space through the following function,

$$y = \begin{cases} d_{min} \times c + x_{min}, \text{ if } x_{min} + \theta < 1 - x_{max}; \\ d_{max} \times c + x_{max}, \text{otherwise.} \end{cases}$$

where $c$ is a positive constant to stretch the range of index keys(normally assigned the value 1) and $\theta$ is the tuning parameter to adjust the data distribution. In case of distributed systems, the value $\theta$ should be negotiated among peers. By random sampling, the "*top nodes*" collect the information of data distribution. Periodically, they negotiate the value of $\theta$ through distributed voting [13] and then multicast the decision to the low-level nodes.

Through the above transformation, we get an 1-dimensional value $y$, $y \in [1, d + 1]$, which can be further normalized as a binary string $b_0 b_1 ... b_{n-1}$ (n is the maximum ID length). The string $b_0 b_1 ... b_{n-1}$ is the key of the data object to be published in the Aurelia overlay.

The key is registered into all of the Aurelia nodes whose *RangeID* is a prefix of that key. In this way, one data object has multiple replicas and makes the indexing service more robust. Even when some nodes that host the data key leave the system, the lookup may still be successful.

The detailed publishing process is as follows: Besides "*top nodes*", every node also maintains a list of "*top index nodes*", whose *RangeID* is the shortest among nodes whose *RangeID* is a prefix of the current node's *RangeID*. During data publishing, the registration message is firstly routed to the node whose node ID is nearest to the data key; then the node will select a "*top index node*" randomly and forward the message to this *"top index node"*, which is responsible for multicasting the message to all the nodes whose index range is covered by it. The multicast algorithm is similar to the above-mentioned multicast algorithm for routing table maintenance, except that it is based on the *RangeID* for multicasting.

During implementation, although the index key is only single-dimensional, the index entry can contain the complete $d$-dimensional data point. Accordingly, if the query also contains the full original query, then the results can be directly filtered at the side of index nodes. Only the qualified results are returned to the query initiator. In this way, the network traffic is further reduced.

### 3.3 Query Processing

To perform query, the $d$-dimensional query in the original data space should be transformed into 1-dimensional queries first. For a $d$-dimensional range query

$q = ([x_{11}, x_{12}], [x_{21}, x_{22}], ..., [x_{d1}, x_{d2}])$, it will be mapped to $d$ subqueries $sq_j = [l_j, h_j]$, $1 \leq j \leq d$, in the single dimension, with:

$$sq_j = \begin{cases} [j + max_{i=1}^{d} x_{i1}, j + x_{j2}], & \text{if } min_{i=1}^{d} x_{i1} + \theta \geq 1 - max_{i=1}^{d} x_{i1}; \\ [j + x_{j1}, j + min_{i=1}^{d} x_{i2}], & \text{if } min_{i=1}^{d} x_{i2} + \theta < 1 - max_{i=1}^{d} x_{i2}; \\ [j + x_{j1}, j + x_{j2}], & \text{otherwise.} \end{cases}$$

The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained. Among the $d$ subqueries, some subqueries are not necessary to be evaluated. The subqueries will be eliminated from the query set if any of the following conditions is satisfied:

(i) $min_{i=1}^{d} x_{i1} + \theta \geq 1 - max_{i=1}^{d} x_{i1}$ and $h_j < max_{i=1}^{d} x_{i1}$

(ii) $min_{i=1}^{d} x_{i2} + \theta < 1 - max_{i=1}^{d} x_{i2}$ and $l_j > min_{i=1}^{d} x_{i2}$

In this way, at most $d$ subqueries are required to be evaluated. Suppose that the subquery $[l_j, h_j]$ corresponds to the ID range $[lkey_j, hkey_j]$. To perform range query, the query initiator should first compute the common prefix $l_r$ of $lkey_j$ and $hkey_j$, and then check whether there exists a node in the routing table whose *RangeID* is a prefix of $l_r$. If existing, the range query is directly forwarded to that node. To avoid causing too much traffic on the powerful nodes, the prefix matching is based on the best-matching rule. If no entry exists, the node selects the nearest node ID to $lkey_j$ for forwarding.

The node that receives the query checks whether its range intersects with the query range, if its responsible index range covers the whole query range, it returns the results directly; otherwise, the node answers the part it knows and forwards the left part to the node whose ID is nearest to the lowest bound of the range.

In case that the query initiator wants to get the complete answer, all the subqueries should be executed. There are two approaches for execution: sequential or parallel. In the former, the subqueries are executed sequentially from the lowest bound to the highest bound. With each range query being answered, the whole range is increasingly reduced. In the latter, the subqueries are sent in parallel. For efficiency, before issuing subqueries, some subqueries can be combined into one if one node whose index range can answer them all is found.

### 3.4 Load Balancing

The problem of load balancing is a big issue in case of non-uniform query distribution. Roogle achieves system-wide load balancing through node self-adaptation.

When a node feels overloaded, it randomly selects one node from the uplevel index entries and forwards the later incoming queries to that node. Since the index range of up-level index node covers the range of low-level nodes, the queries can also be resolved. If the node itself is already the "*top index node*", it will firstly try to probe an underloaded node, and request this underloaded node to leave its current position and rejoin the "hot" region, so that the load can

be partitioned; however, in case that the probe fails within a time limit, the node reduces its responsible range by extending the *RangeID*. After making changes, the node should notify its predecessor and successor. At the same time, it should also multicast the event to all the low-level nodes. The low-level nodes then change their top index entries accordingly.

## 4 Experimental Evaluation

Currently, the performance of Roogle is validated by simulation. The hardware platform is a Sun Enterprise E4500 server with 12 UltraSPARC-II 400MHz CPUs, 8GB RAM and 1Gbps network bandwidth.

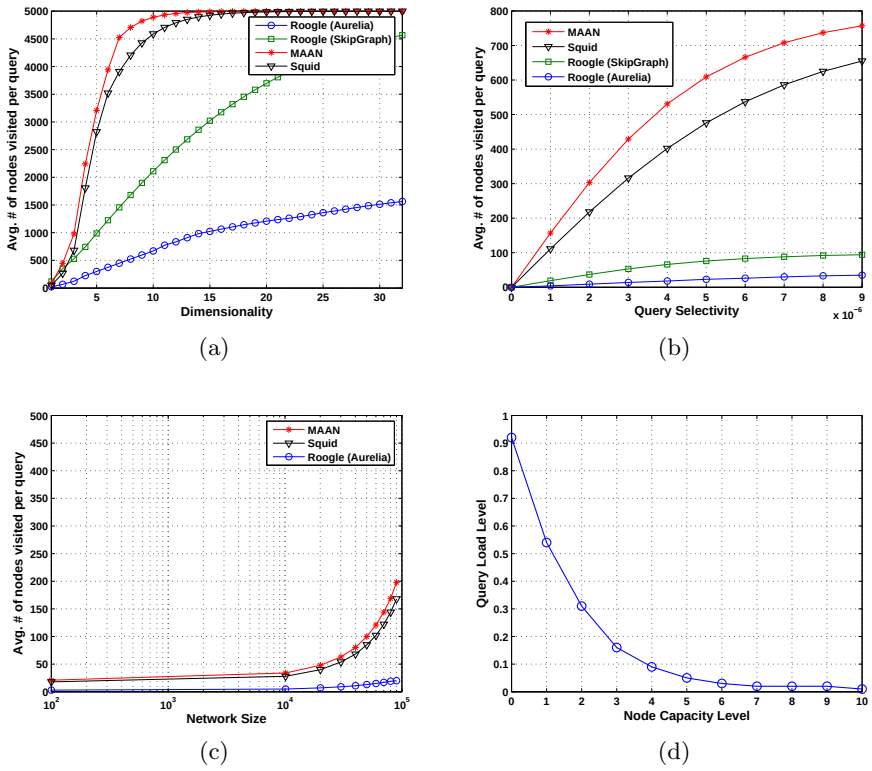


**Fig. 2.** (a) Effect of dimensionality on query performance; (b) Effect of query selectivity on query performance; (c) Effect of network size on query performance; (d) Query load level of nodes with different capacities.

Two datasets are used to evaluate the query performance of Roogle: one is Corel Image Features dataset [14], which contains 32-dimensional color his-

togram vectors extracted from 68,040 photo images; another is Movies dataset [15], which contains 11,435 30-dimensional movie records. In the Movies dataset, missing values are common. The node capacity distribution follows the measurement results of Gnutella in [16]. To simulate system dynamics, each node is associated with a lifetime satisfying Pareto distribution ($\alpha = 2.1, \beta = 2$) and the peer arrival follows a Poisson process.

The metric in use is the average number of nodes visited per query. For the purpose of comparison, we also simulate another three schemes: MAAN [2], Squid [3] and Roogle with SkipGraph [10] as the underlying overlay. The simulation results are illustrated in Fig. 2.

Fig. 2(a) depicts the effect of the dimensionality on query performance. All the schemes perform worse with increasing of dimensionality. When the dimensionality is bigger than 10, the performance of MAAN and Squid is almost degraded to sequential scan, while Roogle has a slower decreasing rate no matter what kind of overlay structure is adopted. But Aurelia further improves the performance by exploiting the node heterogeneity.

In Fig. 2(b), we show the effect of query selectivity on query performance. By varying the query selectivity, we can observe that Roogle has better data locality compared with MAAN and Squid. The locality of MAAN and Squid deteriorates greatly with the increasing of query selectivity. The poor locality of MAAN/Squid is caused by bad indexing performance of locality-preserving hashing and Space-Filling Curve (SFC), in which nearby points in original data space are dispersed to a large region in the single-dimensional space.

We also measure the effect of network size on query performance (as shown in Fig. 2(c)). For a given query, it is observed that Roogle scales well with the increase of network size. On the contrary, for MAAN and Squid, the number of nodes required to visit in order to solve the query increases almost linearly with the network size. When the system size is bigger than 10,000, their performance degrades in exponential speed.

Fig. 2(d) depicts the query load distribution on nodes with different capacity levels. we find that, due to the large routing table and index range, high-level nodes are likely to have more query traffic. But they also enjoy quicker routing and querying than low-level nodes, thus the above cost gets well compensated. The results also show that load balancing is mostly achieved in the system, and all the nodes take a fair portion of load corresponding to their capacity.

## 5 Conclusion

Our focus in this paper is to efficiently support high-dimensional range queries in P2P systems. The problem is addressed from two aspects: indexing and routing. To overcome the curse of dimensionality, we index P2P shared data simply based on the maximum or minimum value along all dimensions. By exploiting the node heterogeneity, we speed up the query routing. Compared with previous schemes, the performance of our design is much improved.

The next step is to investigate additional query optimization and system resilience to failures, and deploy Roogle in PlanetLab [17] to verify its performance under more realistic environments.

## References


1. Oppenheimer, D., Albrecht, J., Patterson, D., Vahdat, A.: Scalable wide-area resource discovery. In: UC Berkeley Technical Report UCB CSD-04-1334. (2004)
2. Cai, M., Frank, M., Chen, J., Szekely, P.: Maan: A multi-attribute addressable network for grid information services. In: Proc. 4th International Workshop on Grid Computing (Grid'03), Phoenix, Arizona (2003)
3. Schmidt, C., Parashar, M.: Enabling flexible queries with guarantees in p2p systems. Internet Computing Journal **Vol.8, No.3** (2004)
4. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting scalable multi-attribute range queries. In: Proc. ACM SIGCOMM'04. (2004)
5. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proc. VLDB'98, St. Johns, Canada (1998)
6. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Brief announcement: Prefix hash tree. In: Proc. ACM PODC'04, St. Johns, Canada (2004)
7. Ganesan, P., Yang, B., Garcia-Molina, H.: One torus to rule them all: Multi-dimensional queries in p2p systems. In: Proc. WebDB'04. (2004)
8. Zhang, C., Krishnamurthy, A., Wang, R.Y.: Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. In: Princeton Technical Report, Submitted for publication. (2004)
9. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proc. ACM SIGCOMM'01, San Diego, CA (2001)
10. Aspnes, J., Shah, G.: Skip graphs. In: Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms. (2003)
11. Hu, J., et al.: Smartboa: Constructing p2p overlay network in the heterogeneous internet using irregular routing tables. In: Proc. 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04), San Diego, CA, USA (2004)
12. Yu, C., Bressan, S., Ooi, B.C., Tan, K.: Querying high-dimensional data in single-dimensional space. VLDB Journal **13(2)** (2004) 105–119
13. B., H., Kwiat, K., Upadhyaya, S.: Secure and fault-tolerant voting in distributed systems. In: IEEE Aerospace Conference. (2001)
14. http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html.
15. http://www-db.stanford.edu/pub/movies/main.html.
16. Saroiu, S.: Measurement and analysis of internet content delivery systems. Doctoral Dissertation, University of Washington (2004)
17. http://www.planet-lab.org/.