# ART : Sub-Logarithmic Decentralized Range Query Processing with Probabilistic Guarantees

S. Sioutas[1], P. Triantafillou[2], G. Papaloukopoulos[2],
E. Sakkopoulos[2], K. Tsichlas[3], and Y. Manolopoulos[3]

[1] Ionian University, Department of Informatics, sioutas@ionio.gr
[2] CTI and Dept. of Computer Engineering & Informatics, University of Patras, (peter, papaloukg, sakkopul)@ceid.upatras.gr
[3] Aristotle University of Thessaloniki, Department of Informatics, (tsichlas, manolopo)@csd.auth.gr

**Abstract.** We focus on range query processing on large-scale, typically distributed infrastructures, such as clouds of thousands of nodes of shared-datacenters, of p2p distributed overlays, etc. In such distributed environments, efficient range query processing is the key for managing the distributed data sets per se, and for monitoring the infrastructure's resources. We wish to develop an architecture that can support range queries in such large-scale decentralized environments and can scale in terms of the number of nodes as well as in terms of the data items stored. Of course, in the last few years there have been a number of solutions (mostly from researchers in the p2p domain) for designing such large-scale systems. However, these are inadequate for our purposes, since at the envisaged scales the classic logarithmic complexity (for point queries) is still too expensive while for range queries it is even more disappointing. In this paper [4] we go one step further and achieve a sub-logarithmic complexity. We contribute the ART [5] structure, which outperforms the most popular decentralized structures, including Chord (and some of its successors), BATON (and its successor) and Skip-Graphs. We contribute theoretical analysis, backed up by detailed experimental results, showing that the communication cost of query and update operations is $O(\log_b^2 \log N)$ hops, where the base $b$ is a double-exponentially power of two and $N$ is the total number of nodes. Moreover, ART is a fully dynamic and fault-tolerant structure, which supports the join/leave node operations in $O(\log\log N)$ expected w.h.p number of hops. Our experimental performance studies include a detailed performance comparison which showcases the improved performance, scalability, and robustness of ART.

**Keywords**:Distributed Data Structures, P2P Data Management.

[4] A limited and preliminary version of this work has been presented as brief announcement in Twenty-Ninth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Zurich, Switzerland July 25-28, 2010 [28]

[5] **A**utonomous **R**ange **T**ree

# 1 Introduction and Motivation

Decentralized range query processing is a notoriously difficult problem to solve efficiently and scalably in decentralized network infrastructures. It has been studied in the last years extensively, particularly in the realm of p2p, which is increasingly used for content delivery among users. There are many more real-life applications in which the problem also materializes. Consider the (popular nowadays) cloud infrastructures for content delivery. Monitoring of thousand of nodes, where thousands of different applications from different organizations execute, is an apparent requirement. This monitoring process often requires support for range queries over this decentralized infrastructure: consider range queries that are issued in order to identify which of the cloud nodes are under-utilized, (i.e., $utilization < threshold$) in order to assign to them more data & tasks and better exploit all available resources, increasing the revenues of the cloud infrastructure, or to identify overloaded nodes, ($load > threshold$) in order to avoid bottlenecks in the cloud, which hurts overall performance, and revenues.

Each node in the cloud maintains a tuple with attributes: utilization, OS, load, NodeId, e.t.c. Collectively, these makeup a relation, CloudNodes, and we wish to execute queries such as:

SELECT NodeId
FROM Cloudnodes
WHERE low < utilization < high

or point and range queries, e.g.

SELECT NodeId
FROM Cloudnodes
WHERE low < utilization < high and OS=UNIX

An acceptable solution for processing range queries in such large-scale decentralized environments must scale in terms of the number of nodes as well as in terms of the number of data items stored. The available solutions for architecting such large-scale systems are inadequate for our purposes, since at the envisaged scales (trilions of data items at millions of nodes) the classic logarithmic complexity (for point queries) offered by these solutions is still too expensive. And for range queries, it is even more disappointing. Further, all available solutions incur large overheads with respect to other critical operations, such as join/leave of nodes, and insertion/deletion of items. Our aim with this work is to provide a solution that is comprehensive and outperforms related work *with respect to all major operations, such as lookup, join/leave, insert/delete, and to the required routing state that must be maintained in order to support these operations*. Specifically, we aim at achieving a sub-logarithmic complexity for all the above operations!

Peer-to-peer (P2P) systems have become very popular, in both academia and industry. They are widely used for sharing resources like music files etc. Search for a given ID, is a crucial operation in P2P systems, and there has been considerable recent work in devising effective distributed search (a.k.a. lookup) techniques. The proposed structures include a ring as in Chord [15], a multiple dimensional grid as in CAN [22], a multiple list as in SkipGraph [2, 10], or a tree as in PHT [24], BATON [13] and BATON* [14]. Most search structures (including all the ones just mentioned except for BATON* and PHT) bound the search cost to a base 2 logarithm of the search space: for a system with $N$ peer nodes, the search cost is bounded by $O(\log N)$. Relative to tree-based indexes, a disadvantage of PHTs (Prefix Hash Trees) is that their complexity is expressed in terms of the log of the domain size, $D$, rather than the size of the data set, $N$ and depends on distribution over $D - bit$ keys. BATON* is a multi-way search tree, which reduces the search cost to $O(\log_m N)$, where $m$ is the tree fanout. The penalty paid is a larger update cost, but no worse than linear in $m$. One of the distributed indexes with high fanout is the P-Tree [5], where each peer maintains a B$^+$-tree leaf and a path of virtual index nodes from the root to the specific leaf. Search is very effective, but updates are expensive, possibly requiring substantial synchronization effort. BATON* extends BATON by allowing a fanout of $m > 2$. Thus, the search cost becomes $O(\log_m N)$, as expected. Moreover, the cost of updating routing tables is $O(m \log_m N)$ only, as compared to $O(\log_2 N)$ in BATON - an improvement that is better than linear in $m$. Furthermore, BATON* has better fault tolerance properties than BATON, and supports load balancing more efficiently. In fact, the system's fault tolerance, measured as the number of nodes that must fail before the network is partitioned, increases linearly with $m$. Similarly, the expected cost of load balancing decreases linearly with $m$.

**Our Results:** In this paper we present the ART structure, which outperforms the most popular decentralized structures, including Chord (and some of its successors), BATON and BATON* and Skip-Graphs. ART is an exponential-tree structure, which remains unchanged w.h.p., and organizes a number of fully-dynamic buckets of peers. We provide and analyze all relevant algorithms for accessing ART. We contribute theoretical analysis, backed up by detailed experimental results, showing that the communication cost of query and update operations is $O(\log_b^2 \log N)$ hops, where the base $b$ is a double-exponentially power of two. Moreover, ART is a fully dynamic and fault-tolerant structure, which supports the join/leave node operations in $O(\log \log N)$ expected w.h.p number of hops. Since ART is a tree based system, our experimental performance studies include our development of BATON* (the best current tree based system), and a detailed performance comparison which showcases the improved performance, scalability, and robustness of ART.

In Section 2 we present more thoroughly key previous work. Section 3 describes the ART structure and analyzes its basic functionalities. Section 4 presents a thorough experimental evaluation; Section 5 presents some interesting heuristics and thresholds, whereas Section 6 concludes the paper.

## 2 Previous Work

Existing structured P2P systems can be classified into three categories: distributed hash table (DHT) based systems, skip list based systems, and tree based systems. There are several P2P DHT architectures like Chord [15], CAN [22], Pastry [23], Tapestry [31], Kademlia [20] and and Kelips [9]. Unfortunately, these systems cannot easily support range queries since DHTs destroy data ordering. This means that they cannot support common queries such as "find all research papers published from 2004 to 2008". To support range queries, inefficient DHT variants have been proposed (for details see [8], [25], [1], [29]).

Skip list based systems such as Skip Graph [2, 10] and Skip Net [12] are based on the skip-list structure. To provide decentralization they use randomized techniques to create and maintain the structure. Moreover, they can support both exact match queries and range queries by partitioning data into ranges of values. However, they cannot guarantee data locality (which hurts efficient range query processing) and load balancing in the system.

Tree based systems also carry their own disadvantages. P-Grid [5] utilizes a binary prefix tree. It can neither guarantee the bound of search steps since it cannot control the tree height. An arbitrary multi-way tree was proposed in [19], where each node maintains links to its parent, children, sibling and neighbors. It also suffers from the same problem. P-Tree [5] utilizes a B$^+$-tree on top of the CHORD overlay network, and peers are organized as a CHORD ring, each peer maintaining a data leaf and a left most path from the root to that B$^+$-tree node. This results in significant overhead in building and maintaining the consistency of the B$^+$-tree. In particular, a tree has been built for each joining node, and periodically, peers have to exchange their stored B$^+$-tree for checking consistency. BATON [13] utilizes a binary balanced tree and as a consequence, it can control the tree height and avoid the problem of P-Grid. Nevertheless, similarly to other P2P systems, BATON's search cost is bounded by $O(\log_2 N)$. BATON* [14] is an overlay multi-way tree based on B-trees, with better searching performance. The penalty paid is a marginally larger update cost.

Systems like MAAN [4], Mercury [3] and DIM [18] support multi-attribute queries in a multi-dimensional space. BATON* can also effectively support queries over multiple attributes. In addition to supporting the use of multiple attributes in a single index, BATON* further introduces the notion of attribute classification, based on the importance of the attribute for querying, and the notion of attribute groups. In particular, BATON* relies on the construction of multiple independent indexes for groups of one or more attributes. For further details about the suggested techniques for partitioning attributes into such groups, see [14].

| P2P architectures | Lookup, Insert, Delete key | Maximum Size of routing table | Join/ Depart peer |
|---|---|---|---|
| CHORD | $O(\log N)$ | $O(\log N)$ | $O(\log N)$ w.h.p. |
| H-F-Chord(a) | $O(\log N/\log\log N)$ | $O(\log N)$ | $O(\log N)$ |
| LPRS-Chord | $O(\log N)$ | $O(\log N)$ | $O(\log N)$ |
| Skip Graphs | $O(\log N)$ | $O(1)$ | $O(\log N)$ amortized |
| BATON | $O(\log N)$ | $O(\log N)$ | $O(\log N)$ w.h.p. |
| BATON* | $O(\log_m N)$ | $O(m \log_m N)$ | $O(m \log_m N)$ |
| ART-tree | $O(\log_b^2 \log N)$ | $O(N^{1/4}/\log^c N)$ | $O(\log\log N)$ expected w.h.p. |

**Table 1.** Performance comparison between ART, Chord, BATON and Skip Graphs.

For comparison purposes, in Table 1 we present a qualitative evaluation with respect to elementary operations between ART, Skip-Graphs, Chord and its newest variations (F-Chord($\alpha$) [26], LPRS-Chord [30]), BATON [13] and its newest variation BATON* [14]. It is noted that $c$ is a big positive constant.

## 3 Our Solution

First, we build the LRT (**L**evel **R**ange **T**ree) structure, one of the basic components of the final ART structure. LRT will be called upon to organize collections of peers at each level of ART.

### 3.1 Building LRT structure

LRT is built by grouping peers having the same ancestor and organizing them in a tree structure recursively. The innermost level of nesting (recursion) will be characterized by having a tree in which no more than $b$ peers share the same direct ancestor, where $b$ is a double-exponentially power of two (e.g. 2,4,16,...). Thus, multiple independent trees are imposed on the collection of peers. Figure 1 illustrates a simple example, where $b = 2$.

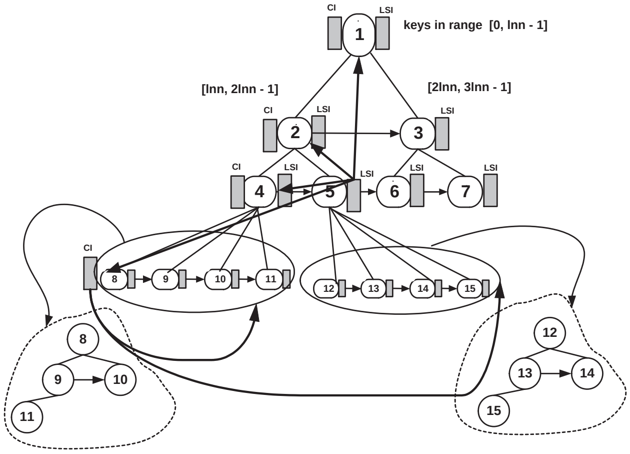

**Fig. 1.** The LRT structure for b=2

The degree of the peers at level $i > 0$ is $d(i) = t(i)$, where $t(i)$ indicates the number of peers at level $i$. It holds that $d(0)$=b and $t(0)$=1. Let $n$ be $w$-bit keys. Each peer with label $i$ (where $1 \leq i \leq N$) stores ordered keys that belong in the range $[(i-1)\ln n, i \ln n{-}1]$, where $N = n/lnn$ is the number of peers. Note here that the $lnn$ (and not $logn$) factor is due to a specific combinatorial game ([16]) we invoke in the next subsection.

We also equip each peer with a table named *Left Spine Index* (LSI), which stores pointers to the peers of the left-most spine (see pointers starting from peer 5).

Furthermore, each peer of the left-most spine is equipped with a table named *Collection Index* (CI), which stores pointers to the collections of peers presented at the same level (see pointers directed to collections of last level). Peers having the same father belong to the same collection. For example, in Figure 1, peers 8, 9, 10, and 11 constitute a certain collection.

**Lookup Algorithm** Assume we are located at peer $s$ (we mean the peer labeled by integer number $s$) and seek a key $k$. First, we find the range where $k$ belongs in. Let say $k \in [(j-1)\ln n, j\ln n - 1]$. The latter means that we have to search for peer $j$. The first step of our algorithm is to find the LRT level where the desired peer $j$ is located. For this purpose, we exploit a nice arithmetic property of LRT. This property says that for each peer $x$ located at the left-most spine of level $i$, the following formula holds:

$$label(x) = label(father(x)) + b^{2^{i-2}} \tag{1}$$

For example, peer 4 is located at level 2, thus $4 = father(4) + 2$ or peer 8 is located at level 3, thus $8 = father(8) + 4$ or peer 24 (not depicted in the Figure 1) is located at level 4, thus $24 = father(24) + 16$. The last equation is true since $father(24) = 8$.

Thus, for each level $i$ (in the next subsection we will prove that $0 \leq i \leq \log\log N$), we compute the label $x$ of its left most peer by applying Equation (1). Then, we compare the label $j$ with the computed label $x$. If $j \geq x$, we continue by applying Equation (1), otherwise we stop the loop process with current value $i$. The latter means that peer $j$ is located at the $i$-th level. So, first we follow the $i$-th pointer of the LSI table located at peer $s$ so as to reach the leftmost peer $x$ of level $i$. Then, we compute the collection in which the peer $j$ belongs. Since the number of collections at level $i$ equals the number of peers located at level $(i-1)$, we divide the distance between $j$ and $x$ by the factor $t(i-1)$. Let $m$ (in particular $m = \left\lceil \frac{j-x+1}{t(i-1)} \right\rceil$) be the result of this division. The latter means that we have to follow the $(m+1)$-th pointer of the CI table so as to reach the desired collection. Since the collection indicated by the CI[$m$+1] pointer is organized in the same way at the next nesting level, we continue this process recursively.

**Analysis** The degree of the peers at level $i > 0$ is $d(i) = t(i)$, where $t(i)$ indicates the number of peers at level $i$. It is defined that $d(0)$=b and $t(0)$=1. It is apparent that $t(i) = t(i-1)d(i-1)$, and, thus, by putting together the various components, we can solve the recurrence and obtain $d(i) = t(i) = b^{2^{i-1}}$ for $i \geq 1$. This double exponentially increasing fanout guarantees the following lemma:

**Lemma 1**: The height (or the number of levels) of LRT is $O(\log\log_b N)$ in the worst case.

The size of the LSI table equals the number of levels of LRT. Moreover, the maximum size of the $CI$ table appears at last level. It is apparent from the building of the LRT structure that at last level $h$, $t(h) = O(N)$. It holds that $t(h) = b^{2^{h-1}}$, thus $b^{2^{h-1}} = O(N)$ or $h-1 = O(loglog_b N)$ or $h = O(loglog_b N)+1$. Since the number of collections at level $h$ equals the number of peers located at level $(h-1)$ we take $t(h-1) = b^{2^{h-2}} = b^{2^{(O(loglog_b N)+1)-2}}$ or $b^{2^{O(loglog_b N)-1}} = b^{2^{O(loglog_b N)}2^{-1}} = \left(b^{2^{O(loglog_b N)}}\right)^{1/2}$ and the lemma 2 follows:

**Lemma 2**: The maximum size of the $CI$ and $LSI$ tables is $O(\sqrt{N})$ and $O(\log\log N)$ in worst-case respectively.

We need now to determine what will be the maximum number of nesting trees that can occur for $N$ peers. Observe that the maximum number of peers with the same direct ancestor is $d(h-1)$. Would it be possible for a second level tree to have the same (or bigger) depth than the outermost one?

This would imply that $\sum_{j=0}^{h-1} t(j) < d(h-1)$.

As otherwise we would be able to fit all the $d(h-1)$ peers within the first $h-1$ levels. But we need to remember that $d(i) = t(i)$, thus $d(h-1) + \sum_{j=0}^{h-2} d(j) < d(h-1)$.

This would imply that the number of peers in the first $h-2$ levels is negative, clearly impossible. Thus, the second level tree will have depth strictly lower than the depth of the outermost tree.

The innermost (let say $j^{th}$) level of nesting (recursion) is characterized by having a tree in which no more than $b$ nodes share the same direct ancestor, where $b$ is a double-exponentially power of two (e.g. 2,4,16,...). In this case $b = N^{1/b^j}$ and the lemma 3 follows:

**Lemma 3**: The maximum number of possible nestings in LRT structure is $O(\log_b \log N)$ in the worst case.

At each peer we pay an extra processing cost by repeating the equation (1) $O(\log\log N)$ times at most in order to locate the desired LSI pointer. Then, we need $O(1)$ hops for locating the left-most peer $x$ of the desirable level. We must note here that the processing overhead compared to communication overhead is negligible, thus we can ignore the $O(\log\log N)$ processing factor at each peer. Finally we need $O(1)$ hops for locating the desirable collection of peers via the CI[$m$+1] pointer. Since, the collection indicated by the CI[$m$+1] pointer is organized in the same way at a next nesting level, we continue the above process recursively. According to lemma 2 the maximum number of nesting levels is $O(\log_b \log N)$, and the theorem follows:

**Theorem 1**: Exact-match queries in the LRT structure require $O(\log_b \log N)$ hops or lookup messages in the worst case.

### 3.2 Building ART Structure

We define as cluster_peer a bucket of $\Theta$(polylog $N'$) ordered peers, where $N'$ is the number of cluster_peers.

At initialization step we choose as cluster_peers the 1st peer, the $(\ln n+1)$-th peer, the $(2\ln n+1)$-th peer and so on. This means that each cluster_peer with label $i'$ (where $1 \leq i' \leq N'$) stores ordered peers with sorted keys belonging in the range $[(i'-1)\ln^2 n, \ldots, i'\ln^2 n - 1]$, where $N' = n/\ln^2 n$ is the number of cluster_peers.

ART stores cluster_peers only, each of which is structured as an independent decentralized architecture. The backbone-structure of ART is exactly the same with LRT (see Figure 2). Moreover, instead of the **L**eft-most **S**pine **I**ndex (LSI), which reduces the robustness of the whole system, we introduce the **R**andom

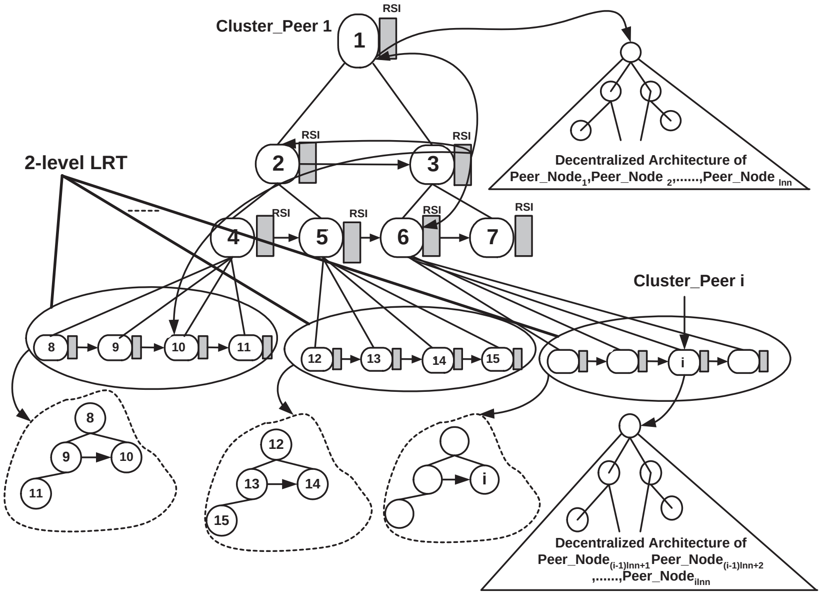
Cluster_Peer 1
RSI
1
2-level LRT
2
RSI
3
RSI
Decentralized Architecture of
Peer_Node$_1$,Peer_Node $_2$,......,Peer_Node $_{lnn}$
4
RSI
5
RSI
6
RSI
7
RSI
Cluster_Peer i
8
9
10
11
12
13
14
15
i
8
9
10
11
12
13
14
15
i
Decentralized Architecture of
Peer_Node$_{(i-1)lnn+1}$Peer_Node$_{(i-1)lnn+2}$
,......,Peer_Node$_{ilnn}$

**Fig. 2.** The ART structure for b=2

**S**pine **I**ndex (RSI) routing table, which stores pointers to randomly chosen (and not to left-most) cluster_peers (see in Figure 2 the pointers starting from peer 3). In addition, instead of using fat $CI$ tables, we access the appropriate collection of cluster_peers by using a 2-level LRT structure. The 2-level LRT is an LRT structure over $\log^{2c} Z$ buckets each of which organizes $\frac{Z}{\log^{2c} Z}$ collections in a LRT manner, where $Z$ is the number of collections at current level and $c$ is a big positive constant (see Figure 3)

**Load Balancing** We model the join/leave of peers inside a cluster_peer as the combinatorial game of bins and balls presented in [16] and the lemma 4 follows:

**Lemma 4**: Given a $\mu(\cdot)$ random sequence of join/leave peer operations, the load of each cluster_peer never becomes zero and never exceeds $\Theta(\text{polylog } N')$ size in expected w.h.p. case.

**Routing Overhead** ART stores cluster_peers, each of which is structured as an independent decentralized architecture (be it BATON*, Chord, Skip-Graph, e.t.c.) (see Figure 2). Here, we will try to avoid the existence of CI routing tables, since these tables may become very large ($O(\sqrt{N})$) in the worst case as well as the occurrence of local hot spots in the left-most spine results in a less robust decentralized infrastructure. Thus, instead of the **L**eft-most **S**pine **I**ndex (LSI), we introduce the **R**andom **S**pine **I**ndex (RSI) routing table. The latter table stores pointers to the cluster_peers of a random spine (for example, in Figure 2 the randomly chosen cluster_peers 1, 2, 6 and 10 are pointed to by the RSI table of cluster_peer 3). Furthermore, instead of CI tables, we can access the appropriate

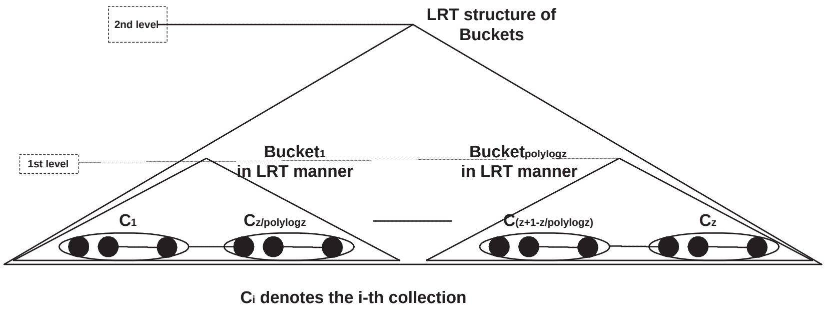

**Fig. 3.** The 2-level LRT structure

collection of cluster_peers by using the 2-level LRT structure discussed above (see Figure 3). Since the larger number of collections is $Z = O(N^{1/2})$ (it appears in the last level), the overhead of routing information is dominated by the second level structures in each of which we have an $O(\sqrt{\frac{Z}{\log^{2c} Z}}) = O(N^{1/4}/\log^c N)$ routing overhead. Thus, Theorem 2 follows:
**Theorem 2**: The overhead of routing information in ART is $O(N^{1/4}/\log^c N)$ in the worst case.
**Remark 1**: If we use a k-level LRT structure, the routing information overhead becomes $O(N^{1/2^k}/\log^c N)$ in the worst case.

**Lookup Algorithms** Let us explain the lookup operations in ART. For example, in Figure 4 suppose we are located at cluster_peer 3 and we are looking for two keys, which are located at cluster_peers 19 and 119 respectively. The first step of our algorithm is to find the levels of the ART where the desired cluster_peers (e.g. 19 and 119) are located. In our example, the fourth and fifth levels are the desired levels. By following the RSI[4] and RSI[5] pointers we reach the cluster_peers 10 and 87 respectively. Now, we are starting from peers 10 and 87 to lookup the peers 19 and 119 respectively in the 2-level LRT structures of the collections in respective levels.
Generally speaking, since the maximum number of nesting levels is $O(\log_b \log N)$ and at each nesting level $i$ we have to apply the standard LRT structure in $N^{1/2^i}$ collections, the whole searching process requires $T_1(N)$ hops or lookup messages to locate the target cluster_peer, where:

$$T_1(N) = \sum_{i=0}^{\log_b \log N} \log_b \log(N^{1/2^i}) = \log_b(\prod_{i=0}^{\log_b \log N} \log(N^{1/2^i})) \qquad (2)$$

where

$$\prod_{i=0}^{\log_b \log N} \log(N^{1/2^i}) < (\log N)^{\log_b \log N}$$

from which we get:

$$T_1(N) < \log_b((\log N)^{\log_b \log N}) = O(\log_b^2 \log N)$$

Then, we have to locate the target peer by searching the respective decentralized structure, requiring $T_2(N)$ hops. Since each of the known decentralized architectures requires a logarithmic number of hops, the total process requires $T(N) = T_1(N) + T_2(N) = O(\log_b^2 \log N)$ hops or lookup messages and the theorem follows.

**Theorem 3**: Exact-match queries in the ART structure require $O(\log_b^2 \log N)$ hops or lookup messages.

Having located the target peer for key $k_\ell$ and exploiting the order of keys on each node, range queries of the form $[k_\ell, k_r]$ require an $O(\log_b^2 \log N + |A|)$ complexity, where $|A|$ is the number of node-peers between the peers responsible for $k_\ell$, $k_r$ respectively. The theorem follows.

**Theorem 4**: Range queries of the form $[k_\ell, k_r]$ in the ART structure require an $O(\log_b^2 \log N + |A|)$ complexity, where $|A|$ is the answer size.

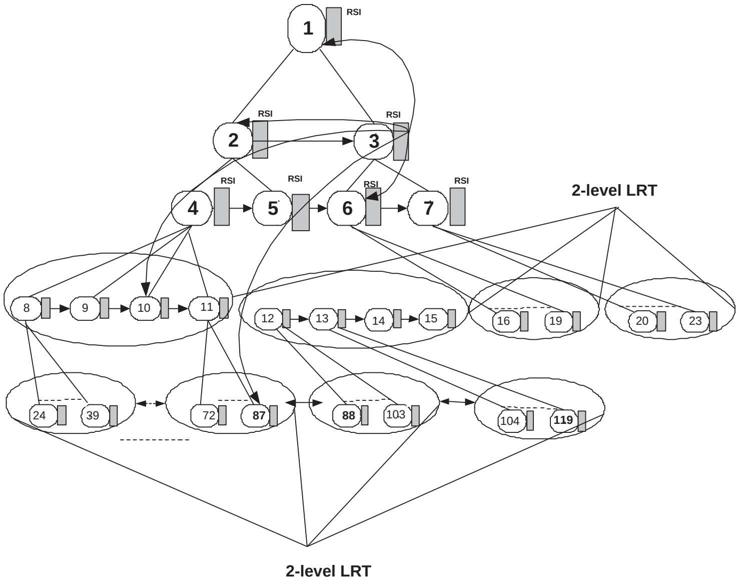

**Fig. 4.** An example of Lookup Steps via RSI[ ] tables and 2-level LRT structures

**Query Processing, Data Insertion and Data Deletion, Peer Join and Peer Departure** In the following we briefly present the basic routines for query processing, data insertion and data deletion, peer join and peer departure.

The *Range_Search*$(s, k_\ell, k_r)$ routine (Algorithm 1) gets as input the peer $s$ in which the query is initiated and the respective range of keys $[k_\ell, k_r]$ and returns as output the *id* of the cluster_peer $S$, which contains peer $s$ as well as the cluster_peer $W$ in which the key $k_\ell$ belongs. Then, it calls the basic *ART_Lookup*$(T, S, idS, W, idW)$ routine, in order to locate the target peer responsible for key $k_\ell$, and then, exploiting the order of keys on each peer performs

**Algorithm 1** Range_Search(s,$k_\ell$,$k_r$,A)

```
Input: $s, k_\ell, k_r$ (we are at peer $s$ and we are looking for keys in range $[k_\ell, k_r]$)
Output: $idW$ (the identifier of cluster-peer $W$, which stores $k_\ell$ key), $A$ (the answer)
BEGIN
We compute $idS$:the identifier of Cluster_peer $S$, which contains peer $s$;
We compute $idW$:let $j$ be the identifier of target Cluster_peer $W$, which stores $k_\ell$
  key;
Let $T$ the basic ART structure of cluster-peers;
W=ART_Lookup($T, S, idS, W, idW$); {call of the basic routine}
A=Linear_Scan of all Cluster_peers located in and right to $W$ until we find a
  $key > k_r$;
END
```

a right linear_scan till it finds a $key > k_r$.
The *ART_Lookup($T, S, idS, W, idW$)* routine (Algorithm 2) gets as input the cluster_peer $S$ (with identifier $idS$) in which the query is initiated and returns as output the $id$ ($idW$) of the cluster_peer $W$ in which the key $k_\ell$ belongs. $T$ denotes the ART-tree structure. Moreover, Algorithm 2 requires $O(\log_b^2 \log N)$ hops, according to first part ($T_1(N)$) of Theorem 3. Obviously, the same complexity holds for insert/delete key operations (see Algorithms 3 and 4), since we have to locate the target peer into which the key must be inserted or deleted.

For join (depart) peer operations (for details see Algorithm 5), we need $O(\log_b^2 \log N) + T_{join}(N)$ $(O(\log_b^2 \log N) + T_{depart}(N))$ lookup messages, where $T_{join}(N)$ $(T_{depart}(N))$ is the number of hops required from the respective decentralized structure for peer-join (peer-departure).

In the peer join algorithm we assumed that the new peer is accompanied by a key, and this key designates the exact position in which the new peer must be inserted. If an empty peer $u$ makes a join request at a particular peer $v$ (which we call *entrance peer*) then there is no need to get to a different cluster peer than the one in which $u$ belongs. Similarly, the algorithm for the departure of a peer $u$ assumes that the request for departure of peer $u$ can be made from any peer in the ART-structure. This may not be desirable, and in many applications it is assumed that the choice for departure of peer $u$ can be made only from this peer. Of course, in this way the algorithm for peer departure is simplified since there is no need to traverse the ART structure but only the cluster peer in which $u$ belongs. In order to bound the size of each cluster_peer we assume that the probability of picking an entrance peer is equal among all existing peers, and that the probability of a peer departing is equal among all existing peers in the ART. Since the size of the cluster_peer is bounded by *polylogN* expected w.h.p., the following theorem is established:

**Theorem 5**: The peer join/departure can be carried out in $O(loglogN)$ hops or lookup messages.

**Node Failure, Fault Tolerance, Network Restructuring and Load Balancing** Since we have modeled the join/leave of peers inside a cluster_peer as the combinatorial game of bins and balls presented in [16], each cluster_peer of an ART structure (according to lemma 4) never exceeds a polylogarithmic number

**Algorithm 2** ART_Lookup($T, S, idS, W, idW$)

```
Input: We are at cluster-peer $S$ with identifier $idS$
Output: We are looking for the cluster-peer $W$ with identifier $idW$
BEGIN
If ($S$ is responsible for $k_\ell$)
    Return $S$;
Else
    If $W$=1 then $i$=0;
    Else if $W \in \{2, 3, \ldots, b + 1]$ then $i$=1;
Else
    $x$=b+2;
    For $(i = 2; i < c_1 log \log_b N; + + i)$
        $x = father(x) + b^{2^{i-2}}$;
        If $j < x$ then break( );
Follow the RSI[$i$] pointer of cluster_peer $S$;
Let $X$ the correspondent cluster_peer;
Search for $W$ the 2-level LRT structure starting from $X$;
Let $Y$ the first cluster-peer of the correspondent collection;
Let $T'$ the ART structure of the collection above at next level of nesting with root the cluster-peer $Y$;
$S = Y$;
ART_Lookup($T', S, idS, W, idW$); {recursive call of the basic routine}
Return $W$;
END
```

**Algorithm 3** ART_insert($T, s, k$)

```
Input: We are at peer $s$ and we want to insert the key $k$
Output: The peer $w$ in which $k$ must be inserted
BEGIN
We compute $idS$:the identifier of Cluster_peer $S$, which contains peer $s$;
We compute $idW$:let $j$ be the identifier of target Cluster_peer $W$, which stores the $k$ key;
ART_Lookup($T, S, idS, W, idW$);
Let $W$ the target cluster_peer;
Search $W$ for peer $w$ containing $k$;
If $k$ does not exist into $w$, then insert $k$ into it;
END
```

**Algorithm 4** ART_delete($T, s, k$)

```
Input: We are at peer $s$ and we want to delete the key $k$
Output: The peer $w$ in which $k$ must be deleted
BEGIN
We compute $idS$:the identifier of Cluster_peer $S$, which contains peer $s$;
We compute $idW$:let $j$ be the identifier of target Cluster_peer $W$, which stores the $k$ key;
ART_Lookup($T, S, idS, W, idW$);
Let $W$ the target cluster_peer;
Search $W$ for peer $w$ containing $k$;
If $k$ exists into $w$, then delete it;
END
```

**Algorithm 5** ART_join/leave_peer($T, s, w$)

```
Input: We are at peer $s$ and we want to insert/delete the new peer $w$
Output: The cluster_peer $W$ in which the peer $w$ must be inserted/deleted
BEGIN
We compute $idS$:the identifier of Cluster_peer $S$, which contains peer $s$;
We compute $idW$:let $j$ be the identifier of target Cluster_peer $W$, which contains peer $w$;
ART_Lookup($T, S, idS, W, idW$); {call of the basic routine}
Let $W$ the target cluster_peer;
Insert/delete $w$ into/from $W$;
END
```

of peers and never becomes empty in expected case with high probability. The latter means that the skeleton ART structure of cluster_peers remains unchanged in the expected case with high probability as well as in each cluster_peer the algorithms for peer failure, network restructuring and load balancing are according to the polylogarithmic-sized decentralized architecture we use.

**Multi-attribute Queries** As in [14], we divide the whole range of attributes into several sections: each section is used to index an attribute (if it appears frequently in queries) or a group of attributes (if these attributes rarely appear in queries). Since ART can only support queries over one-dimensional data, if we index a group of attributes, we have to convert their values into one-dimensional values (by choosing Hilbert space filling curve or other similar methods). For example, if we have a system with 12 attributes: $a_1, a_2, \cdots, a_{12}$ in which only 4 attributes from $a_1$ to $a_4$ are frequently queried (i.e. 90% of all queries), we can build 4 separate indexes for them. The remaining attributes can be divided equally into two groups to index, four attributes in each group. This way, the number of replications can be significantly reduced from 12 down to 6.

## 4 Evaluation

For evaluation purposes we used the Distributed Java D-P2P-Sim simulator presented in [27]. The D-P2P-Sim simulator is extremely efficient delivering $> 100,000$ cluster peers in a single computer system, using 32-bit JVM 1.6 and 1.5 GB RAM and full D-P2P-Sim GUI support. When 64-bit JVM 1.6 and 5 RAM is utilized the D-P2P-Sim simulator delivers $> 500,000$ cluster peers and full D-P2P-Sim GUI support in a single computer system. When D-P2P-Sim simulator acts in a distributed environment with multiple computer systems with network connection delivers multiple times the former population of cluster peers with only 10% overhead.

Our experimental performance studies include a detailed performance comparison with BATON*, one of the state-of-the-art decentralized architectures. In particular, we implemented each cluster_peer as a BATON* [14], the best known decentralized tree-architecture. We tested the network with different numbers of peers ranging up to 500,000. A number of data equal to the network size multiplied by 2000, which are numbers from the universe [1..1,000,000,000] are inserted to the network in batches. The synthetic data (numbers) from this universe were produced by the following distributions: beta, uniform and power-law. For each test, 1,000 exact match queries and 1,000 range queries are executed, and the average costs of operations are taken. Searched ranges are created randomly by getting the whole range of values divided by the total number of peers multiplies $\alpha$, where $\alpha \in [1..10]$. Note that in all experiments the default value of parameter $b$ is 4. The source code of the whole evaluation process is publicly available [6].

### 4.1 Single- and Multi-attribute Query Performance

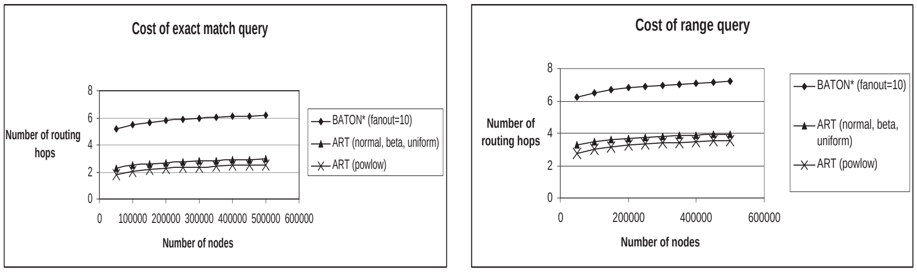

**Fig. 5.** Cost of exact match query (left) and cost of range query (right).

As proved previously, the whole query performance of ART is $O(\log_b^2 \log N')$ where the $N'$ cluster_peers structure their internal peers according to the BATON* architecture. For normal, beta and uniform distributions each cluster_peer contains $0.75 \log^2 N$ peers on average and for power-law distributions each cluster_peer contains $2.5 \log^2 N$ peers on average. Thus, in the former case the average number of cluster_peers is $N' = \frac{N}{0.75 \log^2 N}$, whereas in the latter case the number of cluster_peers becomes $N' = \frac{N}{2.5 \log^2 N}$ on average. In all cases, ART outperforms BATON* by a wide margin. As depicted in Figure 5 (up), our method is almost 2 times faster and as a consequence we have a 50% improvement. The results are analogous with respect to the cost range queries as depicted in Figure 5 (down).

Figure 6 (up) depicts the cost of updating routing tables. Since each cluster_peer structures $O(N/\text{polylog } N)$ (and not $O(N)$) peers according to BATON* architecture, the results are as expected. We remark that BATON* requires $m \log_m N$ hops, whereas $m \log_m \text{polylog } N$ hops are required by ART. In particular and as depicted in Figure 6 (up), our method updates the routing tables 3 or 4 times faster. Figure 6 (down) depicts the insertion cost in multi-attribute case, where we have 6 separate indexes. BATON* requires $6 \log N$ hops and ART requires $6 \log_b^2 \log(N/\text{polylog } N) + 6 \log(\text{polylog } N)$ hops. We observe that the insertion cost of ART is the lowest for any distribution. Again, our method is almost 2 times faster. Finally, the results are analogous for multi-attribute exact-match and range queries respectively (see Figures 7 (up) and 7 (down)).

[6] http://code.google.com/p/d-p2p-sim/

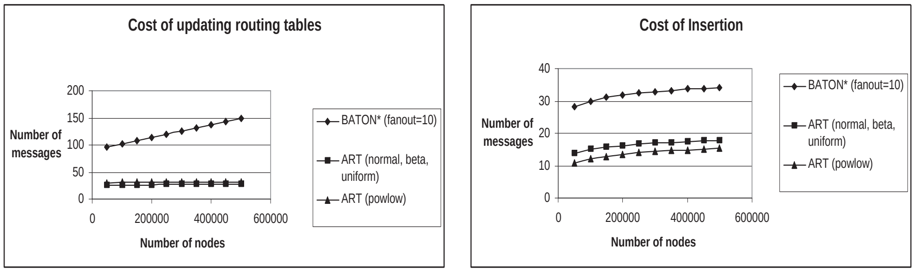

**Fig. 6.** Cost of updating routing tables (left) and cost of insertion (right).

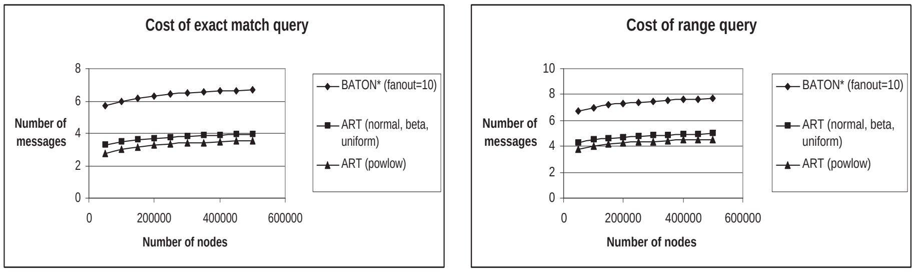

**Fig. 7.** Cost of multi-attribute exact-match (left) and range queries (right).

### 4.2 Load Balancing

ART not only reduces the search cost but also achieves better load balancing. To verify this claim, we test the network with a variety of distributions and evaluate the cost of load balancing. For simplicity, in our system, we assume that the query distribution follows the data distribution. As a result, the workload of a peer is determined only by the amount of data stored at that peer. In BATON*, when a peer joins the network, it is assigned a default upper and lower load limit by its parent. If the number of stored data at the peer exceeds the upper bound, it is considered as an overloaded peer and vice versa. If a peer is overloaded and cannot find a lightly loaded leaf peer, it is likely that all other peers also have the same work load; thus, it automatically increases the boundaries of storage capability. In ART the overlay of cluster_peer remains unaffected in the expected case with high probability when peers join or leave the network. Thus, the load-balancing performance is restricted inside a cluster_peer (which is a new BATON* structure) and as a result ART needs no more than 4 lookup-messages (instead of 1000 messages needed from BATON* in case of 500.000 nodes). For details see Figure 8 (up).

### 4.3 Fault Tolerance

To evaluate the system's fault tolerance in case of massive failure we initialized the system with 10,000 peers. In the sequel, we let peers randomly fail step by step without recovering. At each step, we check to see if the network is partitioned or not. With massive peer failures, we face a massive destruction of links

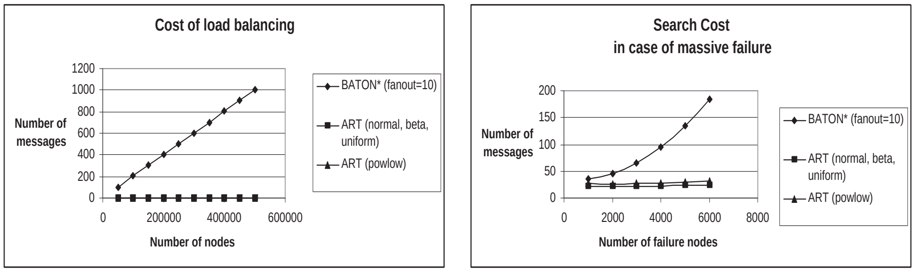

**Fig. 8.** Cost of load-balancing (left) and search cost in case of massive failure (right).

connected to failed peers. Since the search process has to bypass these peers, the search query has to be forwarded forth and back several times to find a way to the destination and as a result the search cost is expected that will increase substantially. Since the backbone of ART structure remains unaffected w.h.p., meaning that there is always a peer for playing the role of cluster representative, the search cost is restricted inside a cluster_peer (which is a BATON* structure) and as a result ART needs no more than 32 lookup-messages (instead of 180 messages needed from BATON* in case of 6.000 nodes). Figure 8 (down) illustrates this effect.

## 5 Trade-offs and Heuristics

If each collection of cluster_peers is organized individually as a BATON$^*$ structure (not the whole level of collections), then we can climb up the ART structure until we reach the nearest common ancestor of the cluster_peer we are located in, and the cluster peer we are searching. Then a downwards traversal is initiated to reach this cluster_peer. Since, each collection of $i^{th}$-level is organized according to BATON*, we can decide in $O(\log_m n^{1/2^i})$ hops the child we must follow for further searching. As a result, the total time becomes $O(\log_m n)$ and no improvement has been achieved.

In our solution, if we parameterize the size of the buckets (depicted in Figure 3) from $O(\log^{2c} N)$ to $O(\log^{2f(N)} N)$, where $f(N)$ is a function of the network size, then we can get an interesting trade-off between the routing data overhead and the number of hops for an operation. In particular, if $Z$ is the number of collections at the current level, then each bucket contains $O(\frac{Z}{\log^{2f(N)} N})$ collections. Thus, the first LRT layer organizes $O(\log^{2f(N)} N)$ bucket representatives and each second LRT layer organizes $O(\frac{Z}{\log^{2f(N)} N})$ collections. In this case, the routing overhead is dominated by the second layer LRTs which becomes $O(\frac{N^{1/4}}{\log^{f(N)} N})$. To achieve an optimal routing data overhead we would like the following: $O(\frac{N^{1/4}}{\log^{f(N)} N}) = O(1) \Leftrightarrow f(N) = O(\log N)$. In this case the first LRT layer contains $O(\log^{2f(N)} N)$ or $O(\log^{2\log N} N)$ bucket representative nodes. Therefore, a lookup operation in first layer requires $O(\log\log(\log^{2\log N} N))$ or $\omega(\log\log N)$ hops. Each of the second layer LRTs contains $O(\frac{Z}{\log^{2f(N)} N})$ collection representative nodes, where $Z$ is the number of collections at current level.

Therefore, the number of hops required by a lookup operation in second layer is $O(\log\log N)$. So, the total time becomes $\omega(\log\log N)$ and the sub-logarithmic complexity is not guaranteed. As a result, if we want an optimal routing overhead we cannot guarantee sub-logarithmic complexity. If we relax the routing overhead to be of polynomial size then we can achieve this.

In our solution the routing data overhead ($O(N^{1/4}/\log^c N)$) is a polynomial function. However, in reality even for an extremely large number of peers N=1.000.000.000, the routing data overhead is 6 for $c = 1$, which is less than the fanout of BATON* ($m = 10$) that we used to run our experiments. The latter demonstrates the significance of our result.

## 6 Conclusions

We presented a new efficient decentralized infrastructure for range query processing with probabilistic guarantees, the ART structure. Theoretical analysis showed that the communication cost of query, update and join/leave node operations scale sub-logarithmically expected w.h.p.. Experimental performance comparison with BATON*, the state-of-the-art decentralized structure, showed the improved performance, scalability and efficiency of our new method. Finally, we believe that ART will enable general purpose decentralized trees to support a wider class of queries, and then broaden the horizon of their applicability.

## References

1. Andrzejak A. and Xu Z.: " Scalable, Efficient Range Queries for Grid Information Services", *Proceedings 2nd International Conference on Peer-To-Peer Computing (P2P)*, pp.33-40, Linkoping, Sweden, 2002.
2. Aspnes J. and Shah G.: "Skip Graphs", *Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp.384-393, Baltimore, MD, 2003.
3. Bharambe A.R., Agrawal M. and Seshan S.: "Mercury: Supporting Scalable Multi-attribute Range Queries", *Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp.353-366, Portland, OR, 2004.
4. Cai M., Frank M., Chen J. and Szekely P.: "Maan: a Multi-attribute Addressable Network for Grid Information Services", *Proceedings 4th International Workshop on Grid Computing (GRID)*, pp.184-191, Phoenix, AZ, 2003.
5. Crainiceanu A., Linga P., Gehrke J. and Shanmugasundaram J.: "Querying Peer-to-peer Networks Using P-Trees", *Proceedings 7th International Workshop on Web and Databases (WebDB)*, pp.25-30, Paris, France, 2004.
6. A. Carzaniga, E. Nitto, D. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In 3rd Intl Software Architecture Workshop, 1998.
7. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst., 19(3):332-383, 2001.
8. Gupta A., Agrawal D. and El Abbadi A.: "Approximate Range Selection Queries in Peer-to-peer Systems", *Proceedings 1st Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.
9. Indranil Gupta , Ken Birman , Prakash Linga , Al Demers AND Robbert van Renesse, "Kelips: Building an efficient and stable P2P DHT through increased memory

and background overhead", Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, 2003.

10. Goodrich M.T., Nelson M.J. and Sun J.Z.: "The Rainbow Skip Graph: a Fault-Tolerant Constant-Degree Distributed Data Structure", *Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp.384-393, Miami, FL, 2006.
11. Jun Gao, Peter Steenkiste, "An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems," 12th IEEE International Conference on Network Protocols (ICNP'04), pp.239-250, Berlin, Germany, 2004.
12. Harvey N.J.A., Jones M.B., Saroiu S., Theimer M. and Wolman A.: "Skipnet: a Scalable Overlay Network with Practical Locality Properties", *Proceedings USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.
13. Jagadish H.V., Ooi B.C. and Vu Q.H.: "Baton: a Balanced Tree Structure for Peer-to-peer Networks", *Proceedings 31st International Conference on Very Large Data Bases (VLDB)*, pp.661-672, Trondheim, Norway, 2005.
14. Jagadish H.V., Ooi B.C., Tan K.L., Vu Q.H. and Zhang R.: "Speeding up Search in P2P Networks with a Multi-way Tree Structure", *Proceedings ACM International Conference on Management of Data (SIGMOD)*, pp.1-12, Chicago, IL, 2006.
15. Karger D., Kaashoek F., Stoica I., Morris R. and Balakrishnan H.: "Chord: a Scalable Peer-to-peer Lookup Service for Internet Applications", *Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp.149-160, San Diego, CA, 2001.
16. Kaporis A., Makris Ch., Sioutas S., Tsakalidis A., Tsichlas K. and Zaroliagis Ch.: "Improved Bounds for Finger Search on a RAM", *Proceedings 11th Annual European Symposium on Algorithms (ESA)*, pp.325-336, Budapest, Hungary, 2003.
17. Kaporis A., Makris Ch., Sioutas S., Tsakalidis A., Tsichlas K. and Zaroliagis Ch.: "Dynamic Interpolation Search Revisited", *Proceedings 33rd International Colloquium Automata, Languages and Programming (ICALP)*, Part I, pp.382-394, Venice, Italy, 2006.
18. Li X., Kim Y.J., Govindan R. and Hong W.: "Multi-dimensional Range Queries in Sensor Networks", *Proceedings 1st International Conference on Embedded Networked Sensor Systems (SenSys)*, pp.63-75, Los Angeles, CA, 2003.
19. Liau C.Y., Ng W.S., Shu Y., Tan K.L. and Bressan S.: "Efficient Range Queries and Fast Lookup Services for Scalable P2P Networks", *Proceedings 2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing(DBISP2P)*, pp.93-106, Toronto, Canada, 2004.
20. Maymounkov P. and Mazieres D.: "Kademlia: a Peer-to-peer Information System Based on the XOR Metric", *Proceedings 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pp.53-65, Cambridge, MA, 2002.
21. S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers, The state of peer-to-peer simulators and simulations, SIGCOMM Comput. Commun. Rev., 2007 Vol 37, No 2, pp. 95-98
22. Ratnasamy S., Francis P., Handley M., Karp R. Shenker S.: "A Scalable Content addressable Network", *Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pp.161-172, San Diego, CA, 2001.
23. Rowstron A. and Druschel P.: "Pastry: a Scalable, Decentralized Object Location, and Routing for Large-scale Peer-to-peer Systems", *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (MIDDLEWARE)*, pp.329-350, Heidelberg, Germany, 2001.
24. Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, Scott Shenker: "prefix hash tree", Proceedings of the twenty-third annual ACM symposium on

Principles of distributed computing table of contents (Brief announcement), PP.368-368, Newfoundland, Canada, 2004.

25. Sahin O.D., Gupta A., Agrawal D. and El Abbadi A.: "A Peer-to-peer Framework for Caching Range Queries", *Proceedings 20th International Conference on Data Engineering (ICDE)*, pp.165-176, Boston, MA, 2004.
26. Stoica I., Morris R., Liben-Nowell D., Karger D.R., Kaashoek M.F., Dabek F. and Balakrishnan H.: "Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications", *IEEE/ACM Transactions on Networking*, Vol.11, No.1, pp.17-32, 2003.
27. S. Sioutas, G. Papaloukopoulos, E. Sakkopoulos, K. Tsichlas and Y. Manolopoulos "A novel Distributed P2P Simulator Architecture: D-P2P-Sim", ACM CIKM 2009, pp. 2069-2070, 2009.
28. S. Sioutas, G. Papaloukopoulos, E. Sakkopoulos, K. Tsichlas and Y. Manolopoulos "Brief announcement: ART–sub-logarithmic decentralized range query processing with probabilistic guarantees", ACM PODC 2010, pp. 118-119, 2010.
29. Triantafillou P. and Pitoura T.: "Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks", *VLDB 03 Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, 2003.
30. Zhang H., Goel A. and Govindan R.: "Incrementally Improving Lookup Latency in Distributed Hash Table Systems", *SIGMETRICS*, pp.114-125, San Diego, CA, 2003.
31. Zhao B.Y., Huang L., Stribling J., Rhea S.C., Joseph A.D. and Kubiatowicz J.D.: "Tapestry: a Resilient Global-scale Overlay for Service Deployment", *IEEE JSAC*, Vol.22, No.1, pp.41-53, 2004.