Distrib Parallel Databases (2013) 31:71–109
DOI 10.1007/s10619-012-7112-4

ART: sub-logarithmic decentralized range query processing with probabilistic guarantees

S. Sioutas · P. Triantafillou · G. Papaloukopoulos · E. Sakkopoulos · K. Tsichlas · Y. Manolopoulos

Published online: 3 October 2012

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

Communicated by: Beng Chin Ooi.

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 [26].

This work was partially supported by Thales Project entitled "*Cloud9: A multidisciplinary, holistic approach to internet-scale cloud computing*". For more details see the following URL: https://sites.google.com/site/thaliscloud9/home.

S. Sioutas (✉)
Department of Informatics, Ionian University, Corfu, Greece
e-mail: sioutas@ionio.gr

P. Triantafillou · G. Papaloukopoulos · E. Sakkopoulos
CTI and Dept. of Computer Engineering & Informatics, University of Patras, Patras, Greece

P. Triantafillou
e-mail: peter@ceid.upatras.gr

G. Papaloukopoulos
e-mail: papaloukg@ceid.upatras.gr

E. Sakkopoulos
e-mail: sakkopul@ceid.upatras.gr

K. Tsichlas · Y. Manolopoulos
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

K. Tsichlas
e-mail: tsichlas@csd.auth.gr

Y. Manolopoulos
e-mail: manolopo@csd.auth.gr

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 we go one step further and achieve a sub-logarithmic complexity. We contribute the ART (Autonomous Range Tree) 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

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, etc. 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 (trillions 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 [13], a multiple dimensional grid as in CAN [20], a multiple list as in SkipGraph [2, 8], or a tree as in PHT [22], BATON [11] and BATON* [12]. 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. We don't compare our solution to DHT-based structures like Chord, since the latter destroys the locality and cannot support range queries.

In Sect. 2 we present more thoroughly key previous work. Section 3 describes a terminology table, so that the paper is easier to follow. Section 4 presents a combinatorial game based analysis, which is the foundation of the theoretical analysis of our solution. Section 5 describes the ART structure and analyzes its basic functionalities. Section 6 presents a thorough experimental evaluation; Sect. 7 presents some interesting heuristics and thresholds, whereas Sect. 8 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 [13], CAN [20], Pastry [21], Tapestry [29], Kademlia [17] and Kelips [7]. 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 [1, 6, 23, 27]).

Skip list based systems such as Skip Graph [2, 8] and Skip Net [10] 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 [16], 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 [11] utilizes a binary balanced tree and as a consequence, it can control the tree height and avoid the problem of P-Grid.

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

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	$O(\log_b^2 \log N)$	$O(N^{1/4}/\log^c N)$	$O(\log\log N)$[a]

[a]expected with high probability case

Nevertheless, similarly to other P2P systems, BATON's search cost is bounded by $O(\log_2 N)$. BATON* [12] 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 [15] 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 [12].

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(α) [24], LPRS-Chord [28]), BATON [11] and its newest variation BATON* [12]. It is noted that $c > 0$ is a positive constant.

3 List of symbols for the proposed solution

Table 2 depicts a list of symbols (acronyms) used in the proposed solution, so that the paper is easier to follow.

4 Balls in bins combinatorial game based analysis

In the following we will briefly describe the combinatorial game of bins and balls presented in [14], which is the foundation of the theoretical analysis of our solution.

In particular we describe a balls and bins random process that models each update operation in a set of elements. Let S_0 be a set of n elements, which are drawn randomly according to the distribution $\mu(\cdot)$ from the interval $[a, b]$. We consider the next rn update operations, where r is a constant. Each update operation is either a uniformly at random deletion of an existing element, or a μ-random insertion of a

Table 2 A List of Acronyms

LRT	Level Range Tree
ART	Autonomous Range Tree
LSI	Left Spine Index
CI	Collection Index
RSI	Random Spine Index
b	A double-exponentially power of two (e.g. $2, 4, 16, \ldots$)
$d(i)$	The fanout or branching factor at level i
$t(i)$	The number of nodes (peers) at level i
n	The total number of w-bit keys
N	The total number of peers
$label(x)$	The numbered label of peer x
N'	The total number of cluster_peers
c	A big positive constant

new element from $[a, b]$. To model the update operations as a balls and bins random process, we do the following:

Let ρ be the set of bins. Each bin $\mathcal{B}_i$, $1 \leq i \leq \rho$, stores a subset of elements in S_0 and is represented by the element $rep(i) = \max\{x : \widetilde{x} \in \mathcal{B}_i\}$. The set of elements stored in the bins constitute an ordered collection $\mathcal{B}_1, \ldots, \mathcal{B}_\rho$ such that $\max\{x : \widetilde{x} \in \mathcal{B}_i\} < \min\{y : \widetilde{y} \in \mathcal{B}_{i+1}\}$ for all $1 \leq i \leq \rho - 1$. In other words, $\mathcal{B}_i = \{\widetilde{x} : x \in (rep(i-1), rep(i)]\}$, for $2 \leq i \leq \rho$, and $\mathcal{B}_1 = \{\widetilde{x} : x \in [rep(0), rep(1)]\}$, where $rep(0) = a$ and $rep(\rho) = b$.

We represent each selected element from $[a, b]$ as a *ball*. We partition the interval $[a, b]$ into $\rho = \frac{n}{\ln n}$ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \cdots \cup (rep(\rho - 1), rep(\rho)]$, where $rep(0) = a$, $rep(\rho) = b$, and $\forall i = 1, \ldots, \rho - 1$, the elements $rep(i) \in [a, b]$ are those previously defined. We represent each of these ρ parts as a distinct *bin*.

During each of the rn insertion/deletion operations, a μ-random ball $x \in [a, b]$ is inserted in (deleted from) the i-th bin $\mathcal{B}_i$ iff $rep(i-1) < x \leq rep(i)$, $i = 2, \ldots, \rho$, otherwise x is inserted in (deleted from) $\mathcal{B}_1$.

If we knew the distribution $\mu(\cdot)$, then we could partition the interval $[a, b]$ into $\rho = \frac{n}{\ln n}$ distinct bins (parts), $[rep_\mu(0), rep_\mu(1)] \cup (rep_\mu(1), rep_\mu(2)] \cup \cdots \cup (rep_\mu(\rho - 1), rep_\mu(\rho)]$, with $rep_\mu(0) = a$ and $rep_\mu(\rho) = b$, such that a μ-random ball x would be equally likely to belong into any of the ρ corresponding bins. In other words, since these ρ bins have equal probability to receive ball x, we have that $\forall x \in [a, b]$ it holds:

$$\begin{aligned}&\Pr\big[x \in \big(rep_\mu(i-1), rep_\mu(i)\big]\big]\\ &\quad = \int_{rep_\mu(i-1)}^{rep_\mu(i)} \mu(t)dt = \frac{1}{\rho} = \frac{\ln n}{n}, \quad i = 1, \ldots, \rho = \frac{n}{\ln n}.\end{aligned}$$

Remark 1 The above expression implies that the unknown sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$ makes the event "insert (delete) a μ-random (random) element x" equivalent to the event "throw (delete) a ball uniformly at random into (from) one of ρ distinct bins". Such a uniform distribution of balls into bins is well understood and it is folklore to find conditions such that no bin remains empty and no bin gets more than $O(\ln n)$ balls.

Unfortunately, the probability density $\mu(\cdot)$ is unknown. Consequently, the major goal is to *approximate* the unknown sequence $rep_\mu(0), \ldots, rep_\mu(\rho)$ with a sequence $rep(0), \ldots, rep(\rho)$, that is, to partition the interval $[a, b]$ into ρ parts $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \cdots \cup (rep(\rho - 1), rep(\rho)]$, aiming to prove that each bin (part) will have the element property:

$$\begin{aligned}&\Pr\bigl[x \in \bigl(rep(i-1), rep(i)\bigr]\bigr]\\ &\quad = \int_{rep(i-1)}^{rep(i)} \mu(t)dt = \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right), \quad i = 1, \ldots, \rho.\end{aligned}$$

Remark 2 The sequence $rep(0), \ldots, rep(\rho)$ makes the event "insert (delete) a μ-random (random) element x" equivalent to the event "throw (delete) a ball *almost* uniformly at random into one of ρ distinct bins". This fact will become the cornerstone in our subsequent proof that no bin remains empty and almost no bin gets more than $\Theta(\ln n)$ balls.

An illustration of Remarks 1 and 2 is given in Fig. 1.

Consider the part of the horizontal axis spanned by $[a, b]$, which will be referred to as the $[a, b]$ *axis*. Suppose that only a wise man knows the positions on the $[a, b]$

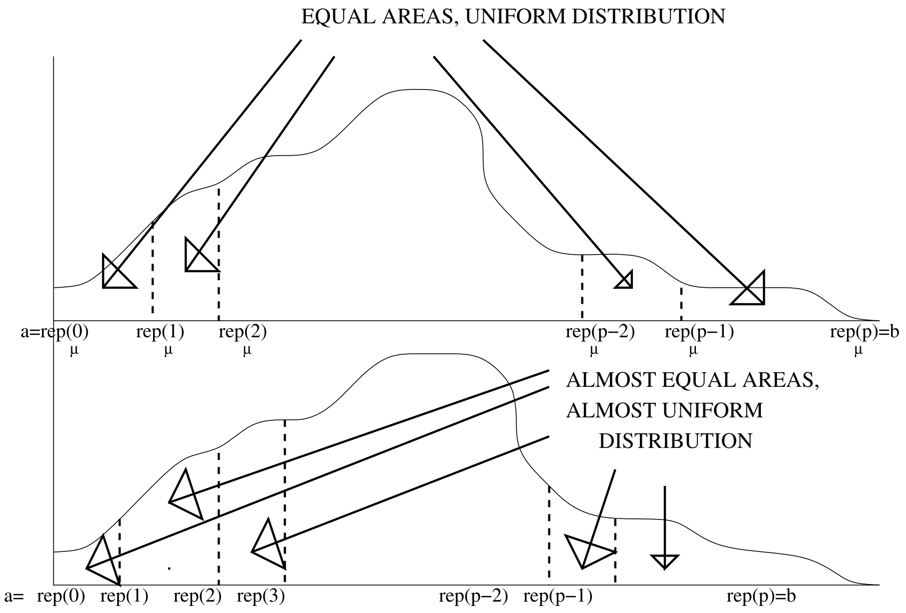


Fig. 1 Plot of an unknown probability density $\mu(x), x \in [a, b]$. The *upper graphic* represents the uniform bins defined by: $rep_\mu(0), rep_\mu(1), \ldots, rep_\mu(\rho)$. The *lower graphic* represents the *almost* uniform bins defined by: $rep(0), rep(1), \ldots, rep(\rho)$

axis of the sequence $rep_\mu(0), \dots, rep_\mu(\rho)$, referred to as the *red dots*. Next, perform n independent insertions of μ-random elements from $[a, b]$ (this is the role of the set S_0). In each insertion of an element x, we add a *blue dot* in its position on the $[a, b]$ axis. At the end of this random game we have a total of n blue dots in this axis. Now, the wise man reveals the red dots on the $[a, b]$ axis, i.e., the sequence $rep_\mu(0), \dots, rep_\mu(\rho)$. If we start counting the blue dots *between* any two consecutive red dots $rep_\mu(i-1)$ and $rep_\mu(i)$, we almost always find that there are $\ln n + o(1)$ blue dots. This is because the number X_i^μ of μ-random elements (blue dots) selected from $[a, b]$ that belong in $(rep_\mu(i-1), rep_\mu(i)]$, $i = 1, \dots, \rho$, is a Binomial random variable, $X_i^\mu \sim B(n, \frac{1}{\rho} = \frac{\ln n}{n})$, which is sharply concentrated to its expectation $E[X_i^\mu] = \ln n$.

The above discussion suggests the following procedure for constructing the sequence $rep(0), \dots, rep(\rho)$. Partition the sequence of n blue dots on the $[a, b]$ axis into $\rho = \frac{n}{\ln n}$ parts, each of size $\ln n$. Set $rep(0) = a$, $rep(\rho) = b$, and set as $rep(i)$ the $i \cdot \ln n$-th blue dot, $i = 1, \dots, \rho - 1$. Call this procedure `Red-Dots`.

Remark 3 The above argument stresses on the crucial fact that the probability measure enclosed in the random interval $(rep(i-1), rep(i)]$, $i = 1, \dots, \rho$, must be of order $\Theta(\frac{1}{\rho}) = \Theta(\frac{\ln n}{n})$, regardless of the particular distribution density $\mu(\cdot)$.

Theorem 1 *Let* $rep(0), rep(1), \dots, rep(\rho)$ *be the output of procedure* `Red-Dots`, *and let* $p_i(n) = \int_{rep(i-1)}^{rep(i)} \mu(t)dt$. *Then*:

$$\Pr\left[\exists\, i \in \{1, \dots, m\} : p_i(n) \neq \Theta\left(\frac{1}{\rho}\right) = \Theta\left(\frac{\ln n}{n}\right)\right] \to 0.$$

Proof For details see [14]. □

The above discussion and Theorem 1 imply the following.

Corollary 1 *If n elements are μ-randomly selected from $[a, b]$, and the sequence $rep(0), \dots, rep(\rho)$ from those elements is produced by procedure* `Red-Dots`, *then this sequence partitions the interval $[a, b]$ into ρ distinct bins (parts) $[rep(0), rep(1)] \cup (rep(1), rep(2)] \cup \cdots \cup (rep(\rho-1), rep(\rho)]$ such that a ball $x \in [a, b]$ can be thrown (deleted)* independently *of any other ball in $[a, b]$ into (from) any of the bins with probability $p_i(n) = \Pr[x \in (rep(i-1), rep(i)]] = \frac{c_i \ln n}{n}$, where $i = 1, \dots, \rho$ and c_i is a positive constant.*

Definition 1 Let $c = \min_i\{c_i\}$ and $C = \max_i\{c_i\}$, $i = 1, \dots, \rho$, where $c_i = \frac{n p_i(n)}{\ln n}$.

Definition 2 Let the random variable $M(j)$ denote the number of balls existing at the end of the j-th insertion/deletion operation, $j = 0, \dots, rn$. Initially, $M(0) = n/c$.

All the above establish the following result.

Theorem 2 *Consider the aforementioned random process of n balls and $n/\ln n$ bins modeling the update operations, where during each operation $j = 0, \ldots, rn$ (r constant) with probability $p > 1/2$ a μ-random ball $x \in [a, b]$ is inserted into an appropriate bin and with probability $1 - p$ an existing ball is deleted (uniformly at random) from the current number of $M(j)$ balls. Then, with high probability, there is no sequence of empty bins and each bin receives $\Theta(\log n)$ balls.*

Proof For details see [14]. □

5 Our solution

First, we build the LRT (Level Range Tree) 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.

5.1 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 2 illustrates a simple example, where $b = 2$.

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 $1 \le i \le N$) stores ordered keys that belong in the range $[(i-1)\ln n, i \ln n - 1]$, where $N = n/\ln n$ is the number of peers. Note here that the $\ln n$ (and not $\log n$) factor is due to the specific combinatorial game we presented in Sect. 4.

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 Fig. 2, 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:

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

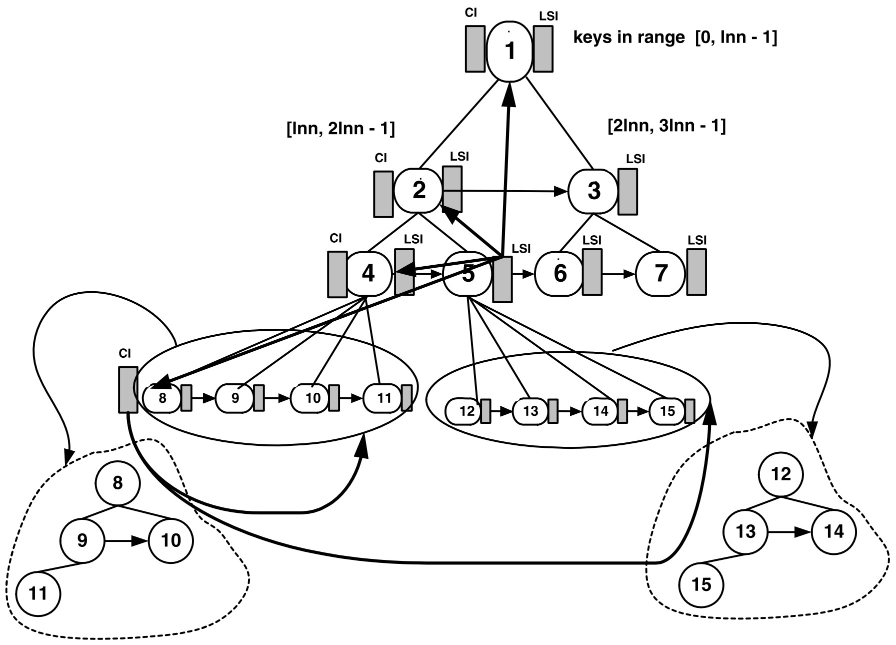


Fig. 2 The LRT structure for $b = 2$

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

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 $m = \lceil \frac{j-x+1}{t(i-1)} \rceil$) 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 $h - 1 = O(\log\log_b N)$ or $h = O(\log\log_b N) + 1$. Since the number of collections at level h equals the number of peers located at level $(h - 1)$ we take $t(h-1) = b^{2^{h-2}} = b^{2^{(O(\log\log_b N)+1)-2}}$ or $b^{2^{O(\log\log_b N)-1}} = b^{2^{O(\log\log_b N)}2^{-1}} = (b^{2^{O(\log\log_b N)}})^{1/2}$ 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 jth) 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. $2, 4, 16, \ldots$). 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 Eq. (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 3 *Exact-match queries in the LRT structure require* $O(\log_b \log N)$ *hops or lookup messages in the worst case.*

5.2 Building ART structure

We define as cluster_peer a bucket of ordered peers. At initialization step and according to *Red-Dots* procedure presented in Sect. 4, we choose as bucket representatives 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 $N' = n/\ln^2 n$ or $N' = N/\ln n$ 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 Fig. 3). Moreover, instead of the Left-most Spine Index (LSI), which reduces the robustness of the whole system, we introduce the Random Spine Index (RSI) routing table, which stores pointers to randomly chosen (and not to left-most) cluster_peers (see in Fig. 3 the pointers starting from peer 3). Let W be the cluster_peer pointed by $RSI_S[h]$ link, $1 \leq h \leq O(\log\log N)$, of the RSI routing table located at node S. In particular, $RSI_S[h]$ points the root of the decentralized structure of W. With other words, $RSI_S[h] = Root(W)$. If the $Root(W)$ peer (node) fails or departs, then we have to execute the respective $failure\_departs\_Repairing(W)$ routine, which is according to the decentralized structure we use and when the recovering has been occured we update the $RSI_S[h]$ link with the new root of W. So, the total cost of RSI maintenance in ART structure is the cost of running the appropriate $failure\_departs\_Repairing(W)$ routine. In our experiments $W = BATON^*$, meaning

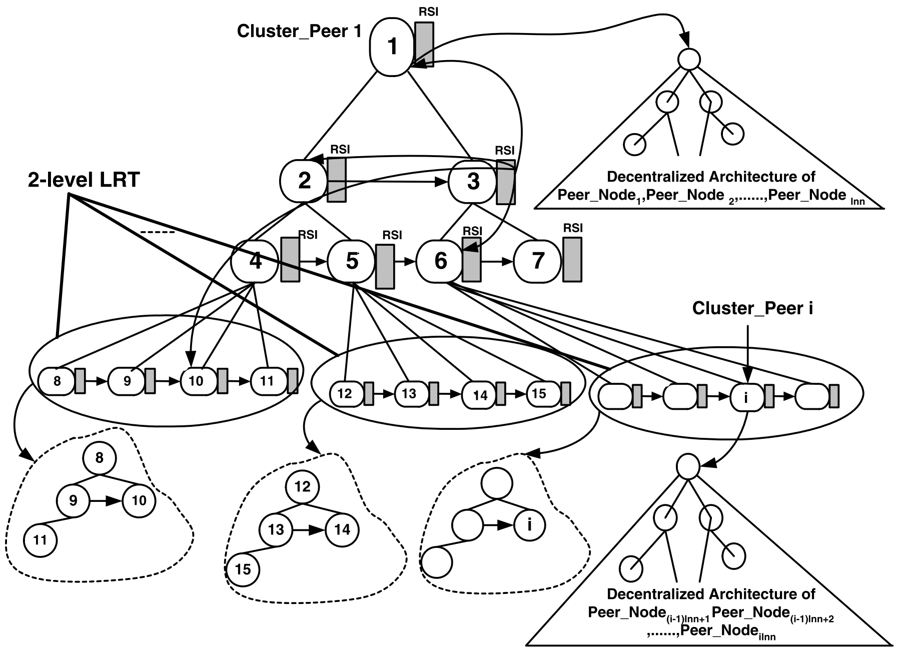

Cluster_Peer 1
RSI
1
2
3
4
5
6
7
2-level LRT
Cluster_Peer i
8
9
10
11
12
13
14
15
i
Decentralized Architecture of Peer_Node$_1$,Peer_Node $_2$,......,Peer_Node $_{lnn}$
Decentralized Architecture of Peer_Node$_{(i-1)lnn+1}$ Peer_Node$_{(i-1)lnn+2}$,......,Peer_Node$_{ilnn}$

Fig. 3 The ART structure for $b = 2$

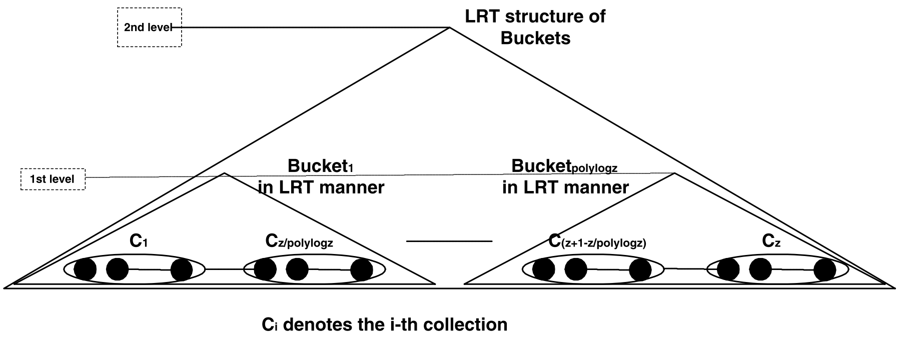


Fig. 4 The 2-level LRT structure

that we call the respective routine presented in [12]. Also, in [12] there is no complexity analysis for this operation (resistance after massive failures or load balancing after departs). Only heuristics have been applied. As a consequence, the same holds for the cost of *RSI* maintenance in our structure. *RSI* offers resistance in *ART* structure and Sect. 6.4 presents the respective experimental evaluation.

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 $c > 0$ is a positive constant (see Fig. 4).

Load balancing We model the join/leave of peers inside a cluster_peer (bucket) as the combinatorial game of bins and balls presented in [14] and according to Theorem 2, 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* $\Theta(\log N)$ *size in expected w.h.p. case.*

The complexity analysis of the combinatorial game of balls in bins presented in [14] is probabilistic and as a consequence the results hold for the expected with high probability scenario. In the worst-case scenario, the things are different. For example, in practice, the sequence of join/leave peer operations may become skewed. In this case the load distribution becomes skewed, meaning that a majority of peers ($\Theta(N)$) are possibly located in a cluster_peer and Lemma 5 follows:

Lemma 5 *For a skew sequence of join/leave peer operations, the load of each cluster_peer never becomes zero and never exceeds* $\Theta(\log N)$ *size in expected w.h.p. case but may become* $\Theta(N)$ *in the worst-case.*

Routing overhead ART stores cluster_peers, each of which is structured as an independent decentralized architecture (be it BATON*, Chord, Skip-Graph, etc.) (see

Fig. 3). 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 Left-most Spine Index (LSI), we introduce the Random Spine Index (RSI) routing table. The latter table stores pointers to the cluster_peers of a random spine (for example, in Fig. 3 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 collection of cluster_peers by using the 2-level LRT structure discussed above (see Fig. 4). Let $2lLRT_i$ be the 2-level-LRT structure, which organizes the ith level of ART. The overhead of *CI* and *LSI* routing information in $2lLRT_i$ structure is dominated by the second level structures in each of which, from Lemma 2, we need $O(\sqrt{\frac{Z}{\log^{2c} Z}})$ and $O(\log\log Z)$ space for *CI* and *LSI* tables respectivelly, where $Z = O(N^{1/2^i})$ the number of organized collections at ith level of ART. Obviously, the maximum overhead of routing information appears at first level, where we have the biggest number of $Z = O(N^{1/2})$ collections. In this case, we need $O(\sqrt{\frac{Z}{\log^{2c} Z}} + \log\log Z) = O(\sqrt{\frac{N^{1/2}}{\log^{2c} N^{1/2}}} + \log\log N^{1/2})$ and Theorem 4 follows:

Theorem 4 *The maximum overhead of routing information in ART structure is $O(N^{1/4}/\log^c N)$ in the worst case.*

Remark 4 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.

Remark 5 The bigger the constant c is the lower the routing overhead becomes.

The total space of routing tables remains linear, since:

$$\begin{aligned}
\mathit{TotalRoutingSpace} &= \sum_{i=0}^{\log\log_b N} \left(\sqrt{\frac{N^{1/2^i}}{\log^{2c} N^{1/2^i}}} + \log\log N^{1/2^i}\right) \\
&\quad + \sum_{i=0}^{\log_b \log N} \prod_{i=0}^{\log_b \log N} (N^{1/2^i})\left(\sqrt{\frac{N^{1/2^i}}{\log^{2c} N^{1/2^i}}} + \log\log N^{1/2^i}\right) \\
&= O(N).
\end{aligned}$$

The first sum is with respect to the 2-level LRTs at the basic ART structure (without nesting levels) of $O(\log\log_b N)$ height and the second sum with respect to the 2-level LRTs at the $O(\log_b \log N)$ nesting levels. Thus, Theorem 5 follows:

Theorem 5 *The total routing space in ART structure remains linear $O(N)$ in the worst case.*

Lookup algorithms Let us explain the lookup operations in ART. For example, in Fig. 5 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

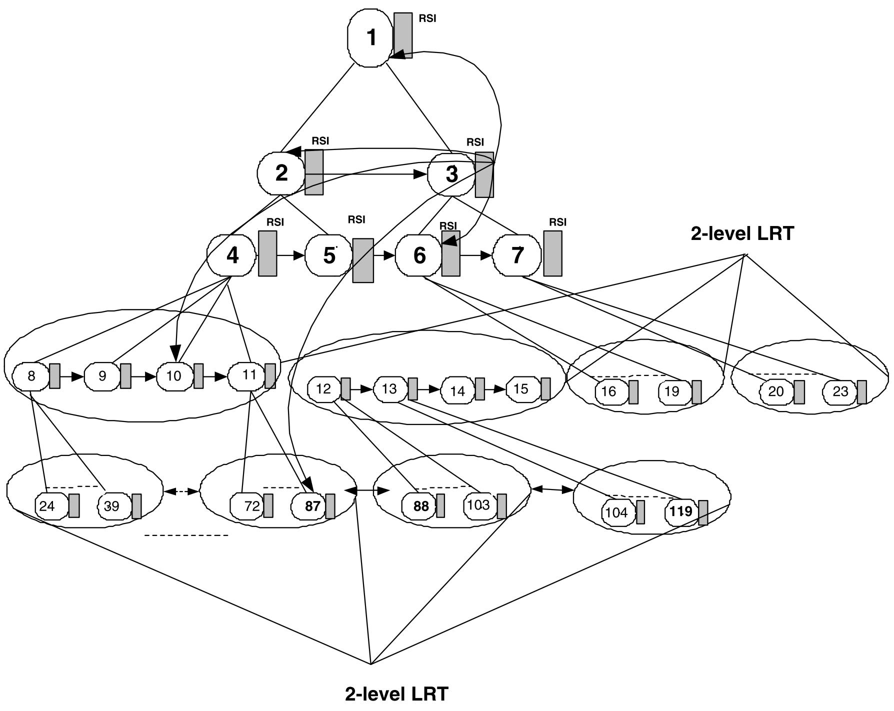


Fig. 5 An example of Lookup Steps via *RSI*[ ] tables and 2-level LRT structures

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:

$$T_1(N) = \sum_{i=0}^{\log_b \log N} \log_b \log\bigl(N^{1/2^i}\bigr) = \log_b \left(\prod_{i=0}^{\log_b \log N} \log\bigl(N^{1/2^i}\bigr) \right) \tag{2}$$

where

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

from which we get:

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

Then, we have to locate the target peer by searching the respective decentralized structure, requiring $T_2(N)$ hops. According to Lemma 4 the load of each cluster_peer never exceeds $\Theta(\log N)$ size in expected w.h.p. case. Since each of the known decentralized architectures requires a logarithmic number of hops, $T_2(N) = O(\log(\Theta(\log N))) = O(\log\log N)$. As a consequence, 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 6 *Exact-match queries in the ART structure require $O(\log_b^2 \log N)$ hops or lookup messages in expected with high probability case.*

Having located the target peer for key k_ℓ 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_ℓ, k_r respectively. The theorem follows.

Theorem 7 *Range queries of the form $[k_\ell, k_r]$ in the ART structure require an $O(\log_b^2 \log N + |A|)$ complexity in expected with high probability case*, *where $|A|$ is the answer size.*

According to Lemma 5, for a skew sequence of join/leave peer operations, the load of each cluster_peer may become $\Theta(N)$ in the worst-case. In this case, $T_2(N) = O(\log N)$. As a consequence, the total process requires $T(N) = T_1(N) + T_2(N) = O(\log N)$ hops or lookup messages in the worst-case and Theorems 8 and 9 follow.

Theorem 8 *Exact-match queries in the ART structure require $O(\log N)$ hops or lookup messages in the worst-case.*

Theorem 9 *Range queries of the form $[k_\ell, k_r]$ in the ART structure require an $O(\log N + |A|)$ complexity in the worst-case*, *where $|A|$ is the answer size.*

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_ℓ, 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_ℓ 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_ℓ, and then, exploiting the order of keys on each peer performs 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_ℓ 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 6. 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.

Algorithm 1 *Range_Search*(s, k_ℓ, k_r, A)

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

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

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

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

```
1: Input: We are at peer $s$ and we want to insert the key $k$
2: Output: The peer $w$ in which $k$ must be inserted
3: BEGIN
4: We compute $idS$: the identifier of Cluster_peer $S$, which contains peer $s$;
5: We compute $idW$: let $j$ be the identifier of target Cluster_ peer $W$, which stores the $k$ key;
6: $ART\_Lookup(T, S, idS, W, idW)$;
7: Let $W$ the target cluster_peer;
8: Search $W$ for peer $w$ containing $k$;
9: If $k$ does not exist into $w$, then insert $k$ into it;
10: END
```

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

```
1: Input: We are at peer $s$ and we want to delete the key $k$
2: Output: The peer $w$ in which $k$ must be deleted
3: BEGIN
4: We compute $idS$: the identifier of Cluster_peer $S$, which contains peer $s$;
5: We compute $idW$: let $j$ be the identifier of target Cluster_ peer $W$, which stores the $k$ key;
6: $ART\_Lookup(T, S, idS, W, idW)$;
7: Let $W$ the target cluster_peer;
8: Search $W$ for peer $w$ containing $k$;
9: If $k$ exists into $w$, then delete it;
10: END
```

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

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

For join (depart) peer operations (for details see Algorithm 5), we need $O(\log_b^2 \log N) + T_{\text{join}}(N)$ $(O(\log_b^2 \log N) + T_{\text{depart}}(N))$ lookup messages, where $T_{\text{join}}(N)$ $(T_{\text{depart}}(N))$ 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 $\Theta(\log N)$ peers in expected w.h.p. case, the following theorem is established:

Theorem 10 *The peer join/departure can be carried out in $O(\log\log N)$ 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 [14], each cluster_peer of an ART structure (according to Lemma 4) never exceeds a polylogarithmic number 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 with high probability case 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. In particular, after peer failures or departs, we have to update appropriately the *RSI* routing tables. Let W be the cluster_peer pointed by $RSI_S[h]$ link, $1 \leq h \leq O(\log\log N)$, of the RSI routing table located at node S. In particular, $RSI_S[h]$ points the root of the decentralized structure of W ($RSI_S[h] = Root(W)$). If the $Root(W)$ peer (node) fails or departs, then we have to execute the respective *failure_departs_Repairing*(W) routine, which is according to the decentralized structure we use and when the recovering has been occurred we update the $RSI_S[h]$ link with the new root of W. The Algorithm 6 follows:

Algorithm 6 *ART_NodeFails_Departs_Repairing*(T, w)

```
1: Input: The peer $w$, which fails or departs
2: Output: The repaired cluster_peer $W$ in which the peer $w$ belonged to.
3: BEGIN
4: Let $W$ be the Cluster_peer in which peer $w$ fails or departs;
5: if ($w = Root(W)$ AND $RSI_s[h] = w$) {there is a link from the RSI table of $s$ to $w$} then
6: Begin
7: $RSI_s[h] = NULL$; {free pointer and disconnect}
8: $failure\_departs\_Repairing(W)$; {f.e. if $W = BATON^*$ then we call the respective routine
   of $BATON^*$}
9: $w' = Root(W)$; {let $w'$ be the new root after the recovering of $W$}
10: $RSI_s[h] = w'$; {allocate pointer and connect}
11: End
12: else $failure\_departs\_Repairing(W)$;
13: END
```

Multi-attribute queries As in [12], 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: $a_1, a_2, \ldots, a_{12}$ 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.

However, the actual solution is to index attributes and some combinations of attributes separately. Thus, this approach has limitations on number of attributes that can be supported, and may encounter another problem of join processing while evaluating queries over attributes that are not indexed together.

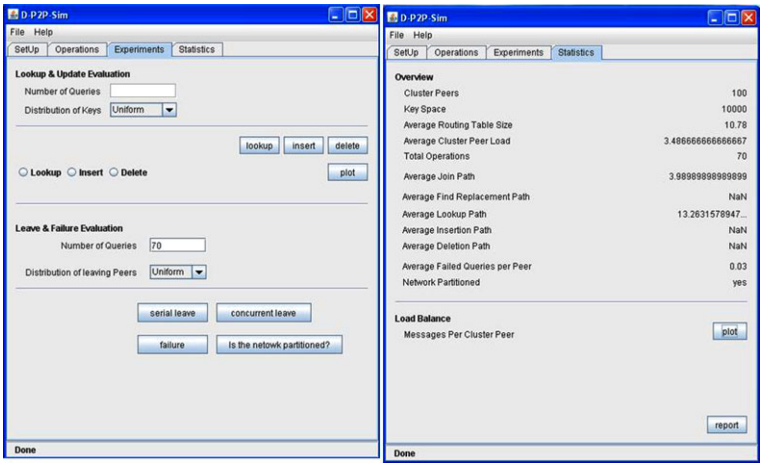
The main decision to design a join evaluation strategy in a P2P context is related to the algorithm to use. This way we can find Sorting-based, Hash-based, Index-based and Nested-Loop Join algorithms that could be implemented in one or several phases, depending upon the available main memory in the system and the memory needed to load the relations implied in the join operation. Sort-based and Hash-based algorithms will be too expensive in terms of processing, number of messages required, and disk space capacity (for details see [9]). These algorithms realize an extra step for sorting and for rehashing of the tuples using the attribute or attributes involved in the join condition. This step is critical in a large-scale distributed environment like P2P systems. As a consequence, we consider only the Index-based join algorithm, because ART maintains indexes to resolve join queries with EqTerms (Equality Terms), IneTerms (Inequality Terms) and LikeTerms. The join evaluation strategy selects first a "query coordinator peer". Such coordinator peer isolates the terms composing the query and initiates their evaluation in a parallel manner. EqTerms, IneTerms and LikeTerms are evaluated and the process to evaluate joins (CondJoin terms) uses the Index-based strategies introduced in [19]. Partial answers obtained by the evaluation of each term of the query are integrated by the coordinator peer. It evaluates the "and/or" operators and returns the final answer.

Low-dimensionality is considered in this work (e.g., $d \leq 6$), where the number of messages required for join operations is negligible in comparison to the number of lookup messages required for querying the six (6) separate ART indexes. This assumption is sufficient for many practical applications. For example, location-aware services in mobile computing systems may require only a few join attributes (e.g., the longitude and latitude corresponding to locations). For higher dimensionality (e.g., tens or hundreds of features as in images and video data), a feasible solution may be to use dimension-reduction techniques [18].

6 Evaluation

For evaluation purposes we used the Distributed Java D-P2P-Sim simulator presented in [25]. 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.

The Admin tools of D-P2P-Sim GUI (see Fig. 6) have specifically been designed to support *reports* on a collection of wide variety of metrics including, protocol operation metrics, network balancing metrics, and even server metrics. Such metrics include frequency, maximum, minimum and average of: number of hops for all basic operations (lookup-insertion-deletion path length), number of messages per node peer (hotpoint-bottleneck detection), routing table length (routing size per node-peer) and additionally detection of network isolation (graph separation). All metrics can tested using a number of different distributions (e.g. normal, weibull, beta, uniform etc.). Additionally, at a system level memory can also be managed in order to execute at low or larger volumes and furthermore execution time can also be logged.


D-P2P-Sim
File Help
SetUp Operations Experiments Statistics
lookup insert delete
Lookup Insert Delete
plot
serial leave concurrent leave
failure Is the netowk partitioned?
Done
D-P2P-Sim
File Help
SetUp Operations Experiments Statistics
plot
report
Done

Lookup & Update Evaluation
Number of Queries
Distribution of Keys Uniform

Leave & Failure Evaluation
Number of Queries 70
Distribution of leaving Peers Uniform

```
Overview
  Cluster Peers                                100
  Key Space                                  10000
  Average Routing Table Size                 10.78
  Average Cluster Peer Load      3.486666666666667
  Total Operations                              70
  Average Join Path              3.989898989898899
  Average Find Replacement Path                NaN
  Average Lookup Path              13.2631578947...
  Average Insertion Path                       NaN
  Average Deletion Path                        NaN
  Average Failed Queries per Peer             0.03
  Network Partitioned                          yes

Load Balance
  Messages Per Cluster Peer
```

Fig. 6 D-P2P-Sim GUI

The framework is open for the protocol designer to introduce additional metrics if needed. Futhermore, XML rule based *configuration* is supported in order to form a large number of different protocol testing scenarios. It is possible to configure and schedule at once a single or multiple experimental scenarios with different number of protocol networks (number of nodes) at a single PC or multiple PCs and servers distributedly. In particular, when D-P2P-Sim simulator acts in a distributed environment (see Fig. 7) 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* [12], 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,[1] uniform[2] and power-law.[3] The distribution parameters can be easily defined in configuration file.[4] Also, the predefined values of these parameters are depicted in Fig. 8.

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 α,

[1] http://acs.lbl.gov/software/colt/api/cern/jet/random/Beta.html

[2] http://docs.oracle.com/javase/1.4.2/docs/api/java/util/Random.html

[3] http://acs.lbl.gov/software/colt/api/cern/jet/random/Distributions.html#nextPowLaw(double,double,cern.jet.random.engine.RandomEngine)

[4] http://code.google.com/p/d-p2p-sim/downloads/detail?name=Art-config.xml&can=2&q=

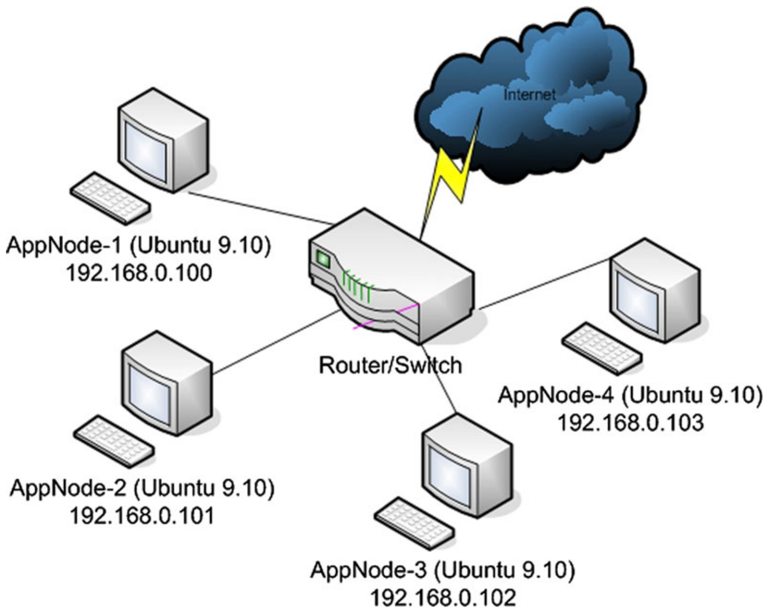


Fig. 7 The distributed environment

Fig. 8 Snippet from config.xml with the pre-defined distribution's parameters setup

```
<distribution>
  <random>
    <seed>1</seed>
  </random>
  <beta>
    <alpha>2.0</alpha>
    <beta>4.0</beta>
  </beta>
  <powerLaw>
    <alpha>0.5</alpha>
    <beta>1.0</beta>
  </powerLaw>
</distribution>
```

where $\alpha \in [1..10]$. The source code of the whole evaluation process is publicly available.[5]

6.1 Single- and multi-attribute query performance

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 "good" or "smooth" distributions like normal, beta and uniform there is

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

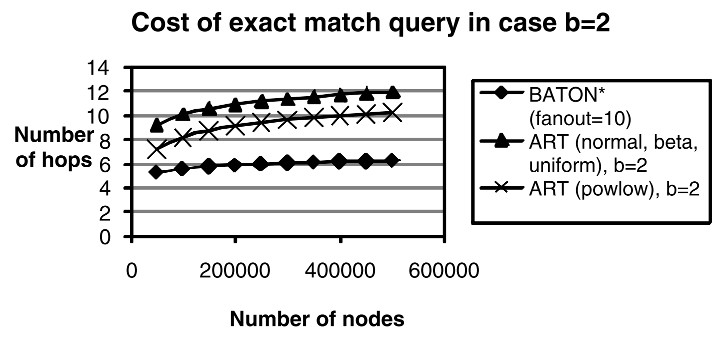


Fig. 9 Cost of 1-dimensional exact match query in case $b = 2$

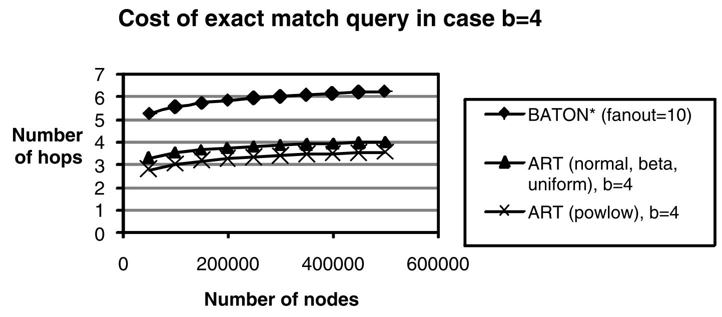


Fig. 10 Cost of 1-dimensional exact match query in case $b = 4$

no cluster_peer storing more than $0.75\log^2 N$ peers and for "bad" or "non-smooth" distributions like pow-law (zipfian like) there is no cluster_peer with more than $2.5\log^2 N$ peers. 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. With other words for "good" distributions the load is distributed well and each cluster-peer is *LIGHT* (its load is a fraction of polylogarithmic size). However, for "bad" distributions there exist *HEAVY* cluster-peers (with load more than 2 times the polylogarithmic size). This is the reason why we have grouped our experiments in that way. Of course the data-sets for normal, beta and uniform distributions may have differences. However the queries executed were exactly the same even if we searched for a specific key that existed in one dataset but didn't exist in the other.

Except for the case where $b = 2$ (Fig. 9), ART outperforms BATON* by a wide margin (Fig. 10, Fig. 11). As depicted in Fig. 9, for $b = 2$ the BATON* structure is almost 2 times faster but for $b > 2$ (in particular for $b = 4$, $b = 16$, etc.) our method is almost 2 (see Fig. 10) up to 3 times (see Fig. 11) faster and as a consequence we have a 50 % up to 66 % improvement. The results are analogous with respect to the cost of range queries as depicted in Figs. 12, 13 and 14.

Figures 15, 16 and 17 depict the cost of updating routing tables. Since BATON* updates its routing tables consuming $m \log_m N$ messages and each cluster_peer struc-

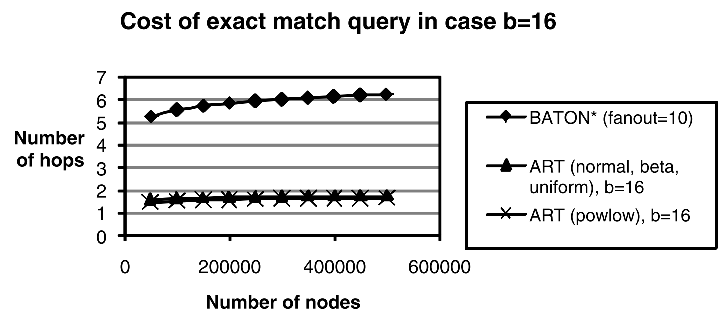


Fig. 11 Cost of 1-dimensional exact match query in case $b = 16$

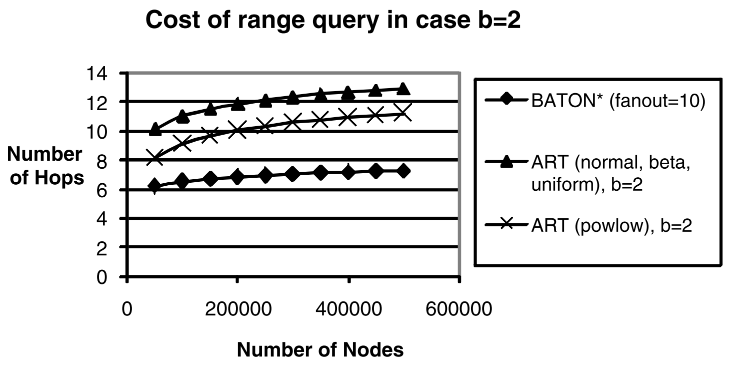


Fig. 12 Cost of 1-dimensional range query in case $b = 2$

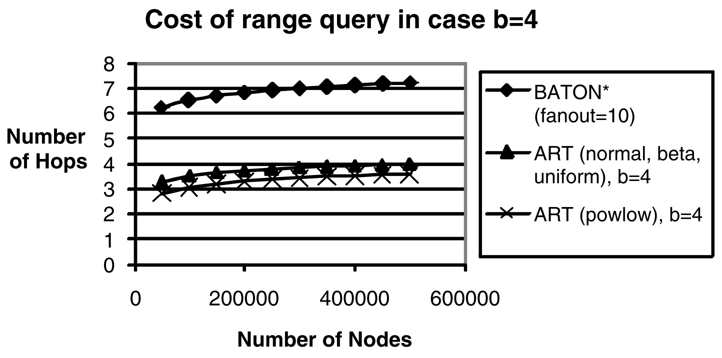


Fig. 13 Cost of 1-dimensional range query in case $b = 4$

tures $\Theta(\log N)$ (and not $O(N)$) peers according to BATON* architecture, ART requires $m \log_m(\Theta(\log N))$ messages only and the results are as expected. As depicted in Figs. 15, 16 and 17, our method updates the routing tables 2.5 up to 4 times faster. In particular for $b = 2, 4$ and 16 our solution requires no more than 40, 32 and 30

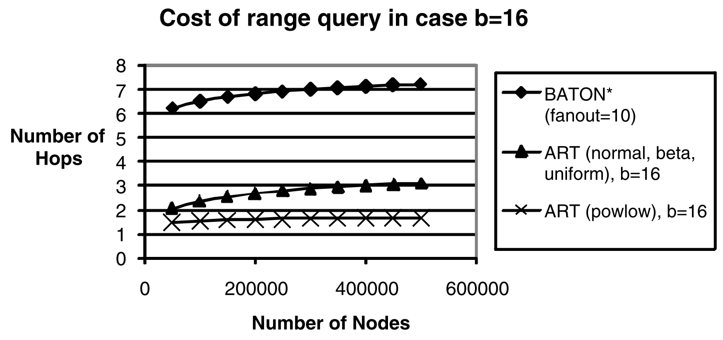


Fig. 14 Cost of 1-dimensional range query in case $b = 16$

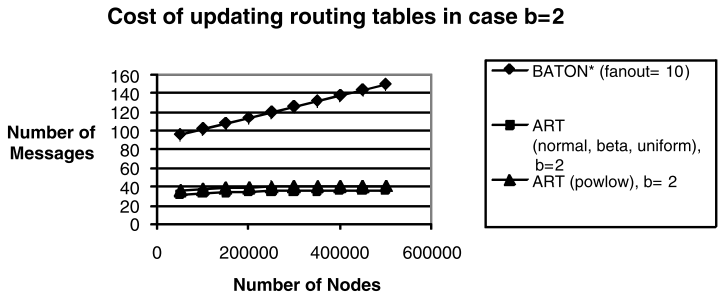


Fig. 15 Cost of updating routing tables in case $b = 2$

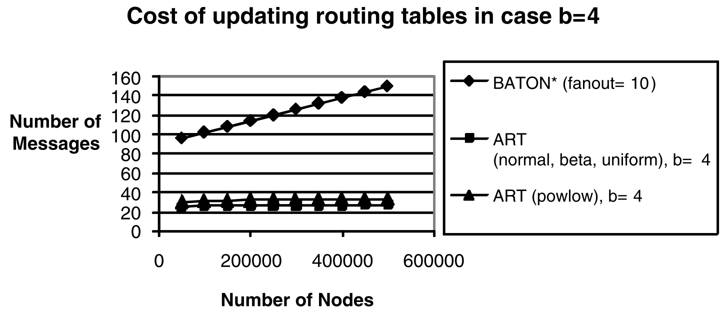


Fig. 16 Cost of updating routing tables in case $b = 4$

messages respectively. On the contrary, BATON* requires from 96 up to 150 messages.

Figures from 18 up to 32 depict the insertion cost in multi-attribute case, where we have k ($2 \leq k \leq 6$) separate indexes. BATON* requires $k \log N$ hops and ART requires $k \log_b^2 \log(N/\text{polylog } N) + k \log(\text{polylog } N)$ hops. We observe that the insertion cost of ART is the lowest for any distribution and in cases where $b > 2$. In particular, for arbitrary k ($2 \leq k \leq 6$) and $b = 2$ BATON* is always 2 times faster

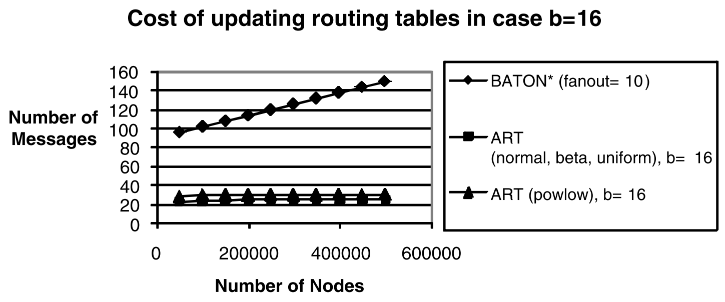


Fig. 17 Cost of updating routing tables in case $b = 16$

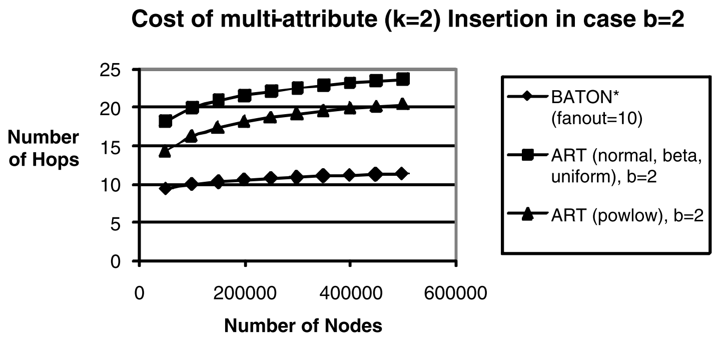


Fig. 18 Cost of multi-attribute $(k = 2)$ insertion in case $b = 2$

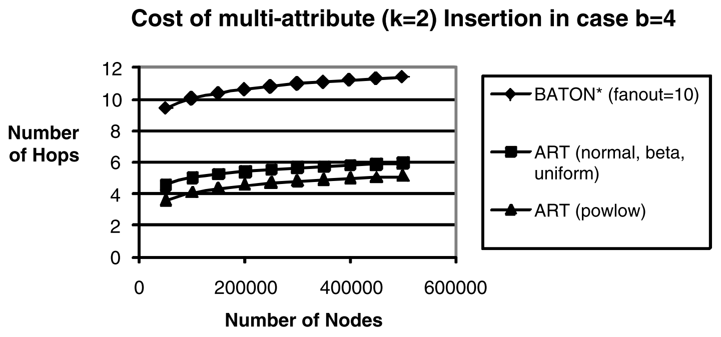


Fig. 19 Cost of multi-attribute $(k = 2)$ insertion in case $b = 4$

(see Figs. 18, 21, 24, 27 and 30). But, for $b > 2$, we observe the following cases: If $k = 2$ our method is from 2 (see Fig. 19) up to 5 (see Fig. 20) times faster and as a consequence we have a 50 % up to 80 % improvement. For $k = 3$, ART is from 2 (see Fig. 22) up to 8 (see Fig. 23) times faster and we have a 50 % up to 87.5 % improvement. For $k = 4$, our method is from 2 (see Fig. 25) up to 6 (see Fig. 26) times faster, achieving a range of 50 % up to 83.33 % improvement. For $k = 5$, our method is from 2 (see Fig. 28) up to 5.5 (see Fig. 29) times faster and as a consequence we have

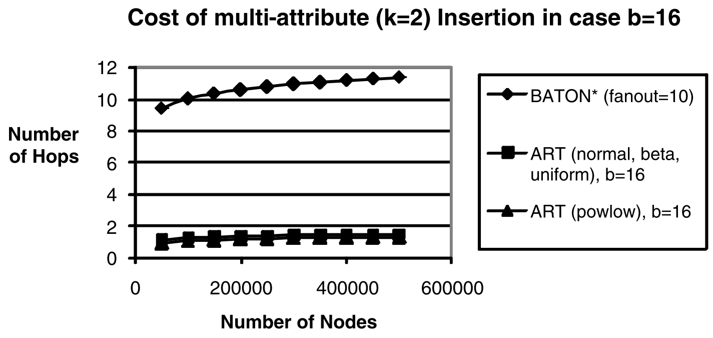


Fig. 20 Cost of multi-attribute ($k = 2$) insertion in case $b = 16$

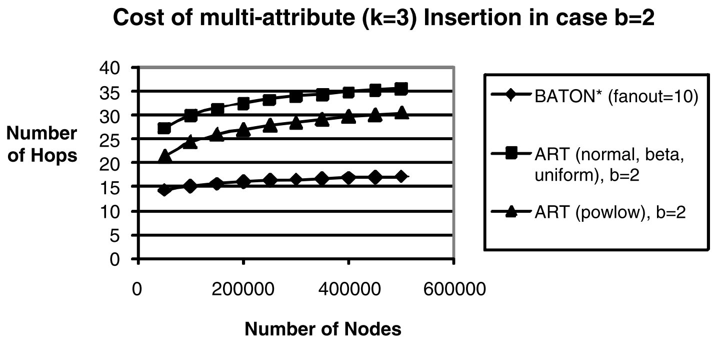


Fig. 21 Cost of multi-attribute ($k = 3$) insertion in case $b = 2$

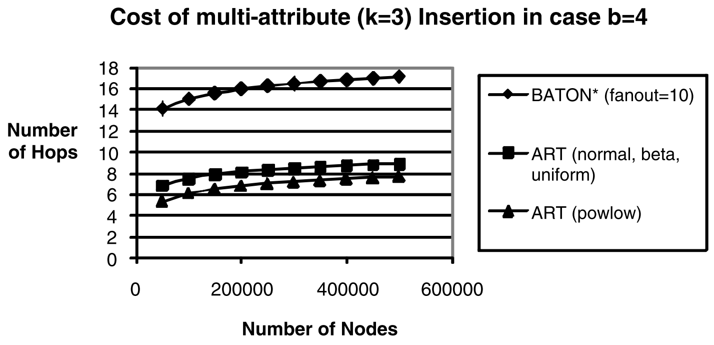


Fig. 22 Cost of multi-attribute ($k = 3$) insertion in case $b = 4$

a 50 % up to 81.81 % improvement. Finally, if $k = 6$, ART is from 2 (see Fig. 31) up to 7 (see Fig. 32) times faster, meaning a 50 % up to 85.72 % improvement.

Finally, the results are analogous for multi-attribute exact-match (see Figs. 33, 34 and 35) and range queries (see Figs. 36, 37 and 38) respectively. We used k sepa-

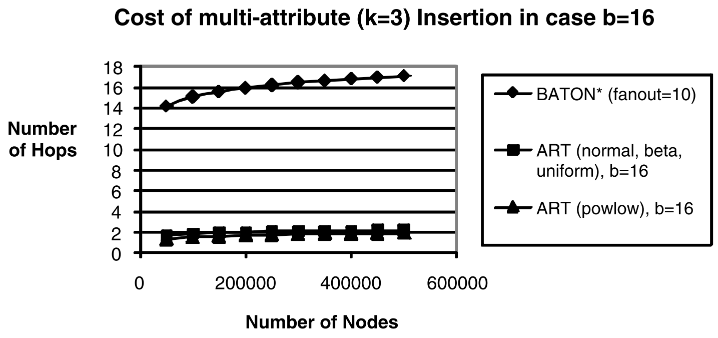


Fig. 23 Cost of multi-attribute $(k = 3)$ insertion in case $b = 16$

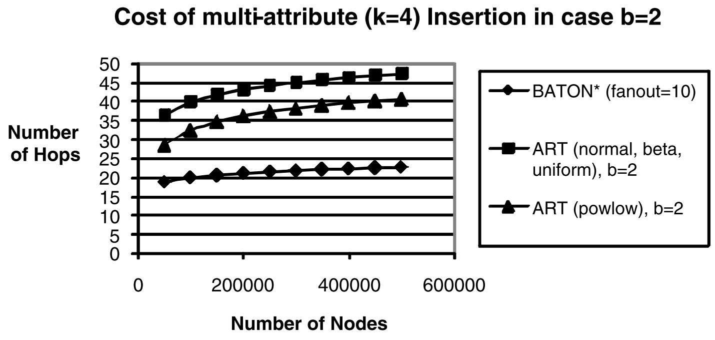


Fig. 24 Cost of multi-attribute $(k = 4)$ insertion in case $b = 2$

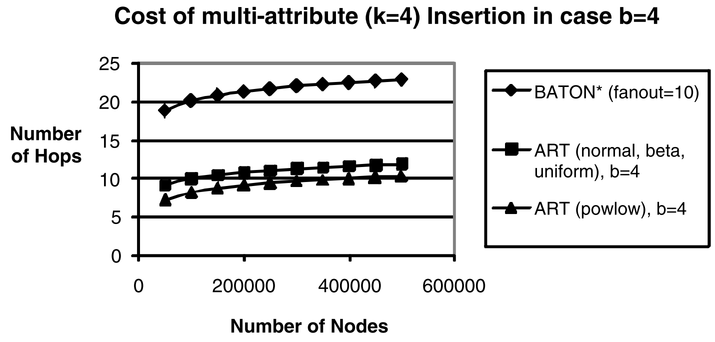


Fig. 25 Cost of multi-attribute $(k = 4)$ insertion in case $b = 4$

rate indexes for running the k separate queries in parallel where for $k \leq 6$ the join processing overhead is negligible. The later does not hold for $k > 6$.

Thus, for $k \leq 6$ and $b = 2$, BATON* is almost 2 times faster for both multi-attribute exact-match and range queries respectively (see Figs. 33 and 36). But for $b > 2$ the things are different. In particular, for multi-attribute exact-match queries and $b = 4$ and 16, our method is from 2 (see Fig. 34) up to 4 (see Fig. 35) times faster

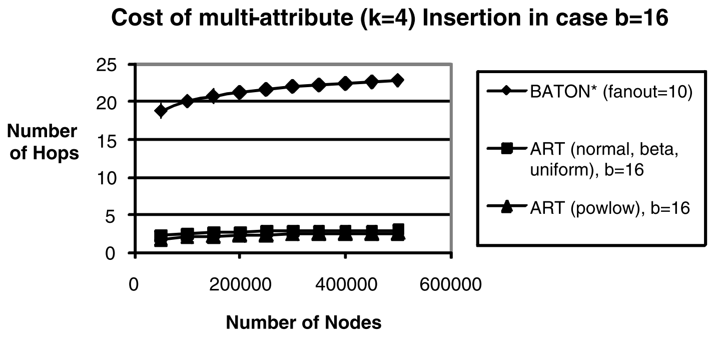


Fig. 26 Cost of multi-attribute ($k = 4$) insertion in case $b = 16$

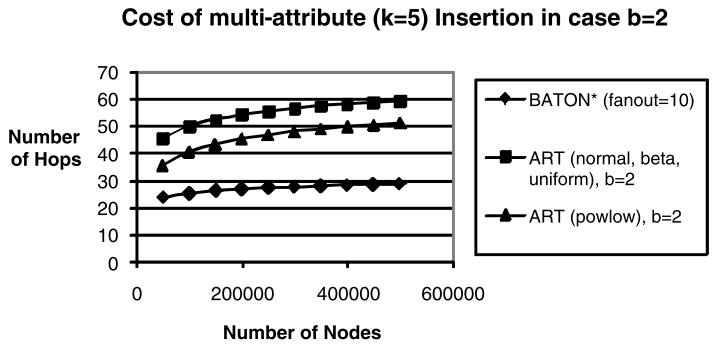


Fig. 27 Cost of multi-attribute ($k = 5$) insertion in case $b = 2$

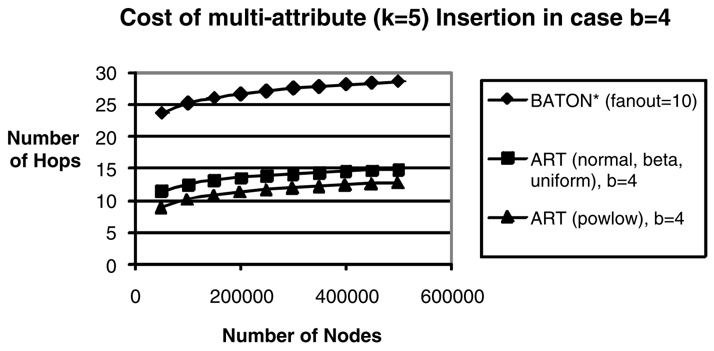


Fig. 28 Cost of multi-attribute ($k = 5$) insertion in case $b = 4$

respectively and as a consequence we have a 50 % up to 75 % improvement. Exactly the same holds for multi-attribute range queries (see Figs. 37 and 38 respectively).

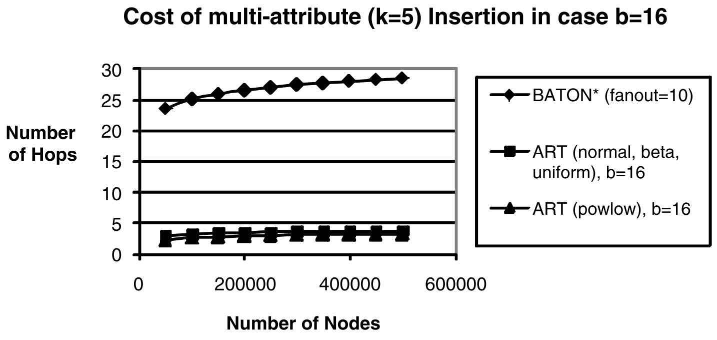


Fig. 29 Cost of multi-attribute ($k = 5$) insertion in case $b = 16$

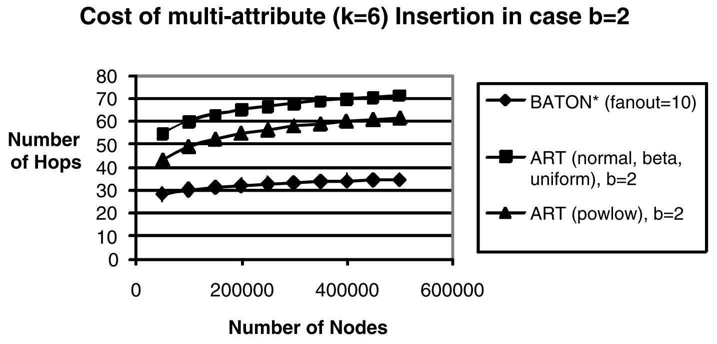


Fig. 30 Cost of multi-attribute ($k = 6$) insertion in case $b = 2$

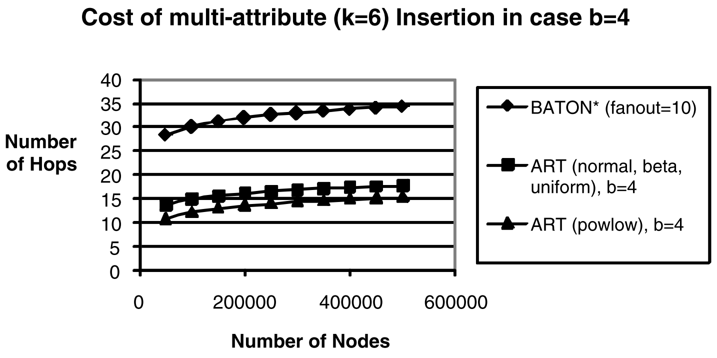


Fig. 31 Cost of multi-attribute ($k = 6$) insertion in case $b = 4$

6.2 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

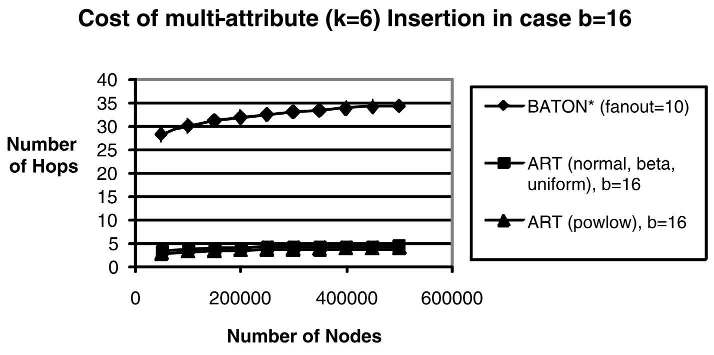


Fig. 32 Cost of multi-attribute ($k = 6$) insertion in case $b = 16$

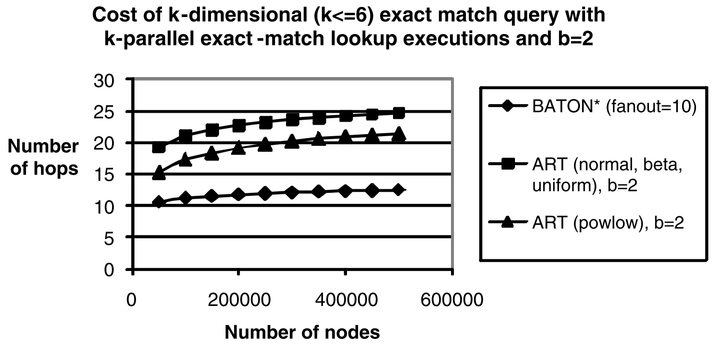


Fig. 33 Cost of k-dimensional ($k \leq 6$) exact match query in case $b = 2$

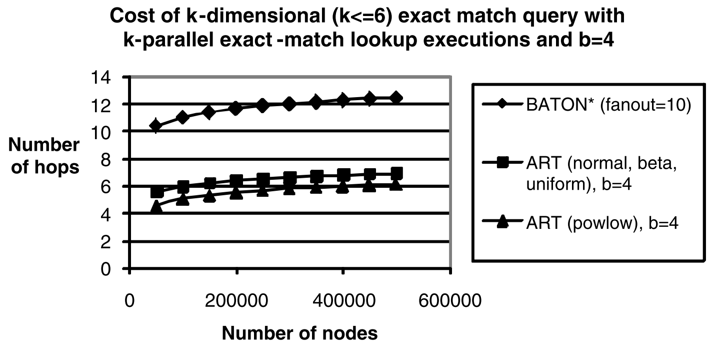


Fig. 34 Cost of k-dimensional ($k \leq 6$) exact match query in case $b = 4$

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

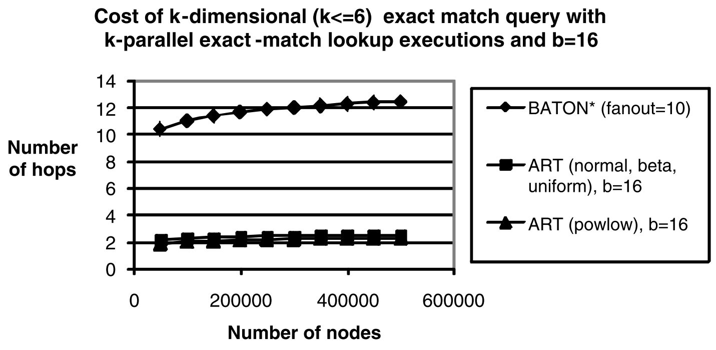


Fig. 35 Cost of k-dimensional ($k \leq 6$) exact match query in case $b = 16$

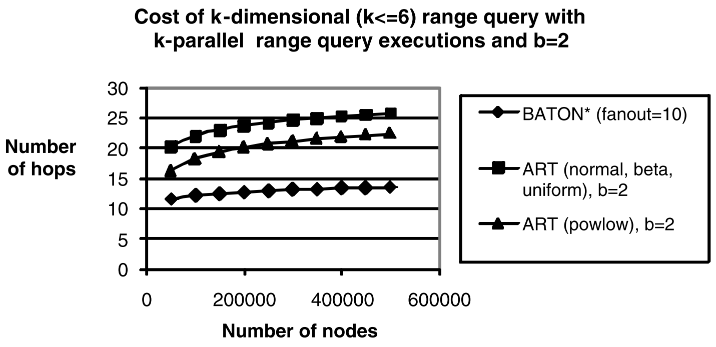


Fig. 36 Cost of k-dimensional ($k \leq 6$) range query in case $b = 2$

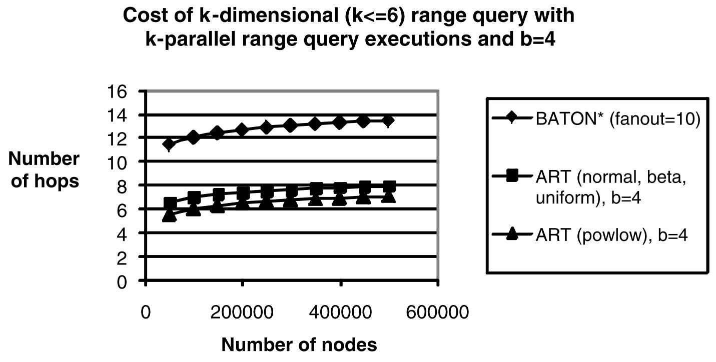


Fig. 37 Cost of k-dimensional ($k \leq 6$) range query in case $b = 4$

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

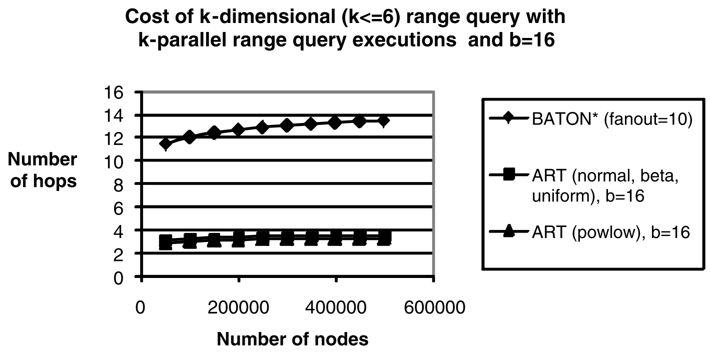


Fig. 38 Cost of k-dimensional ($k \leq 6$) range query in case $b = 16$

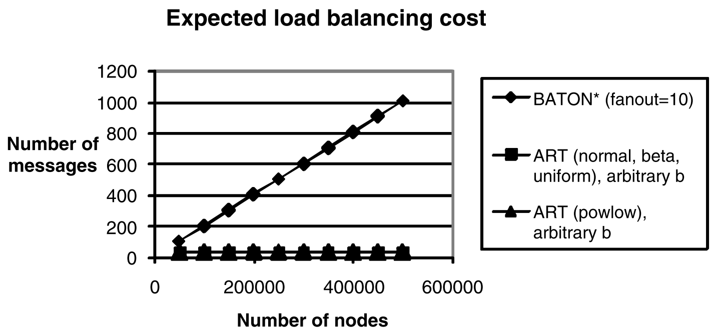


Fig. 39 Cost of expected load balancing

peers join or leave the network. Thus, the load-balancing performance is restricted inside a cluster_peer (which is a new BATON* structure), meaning that the b parameter does not affect the expected load balancing cost 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 Fig. 39.

6.3 Expected search cost in case of massive failures

The problem that massive failures cause is the invalidation of links among them. As the search procedures have to overcome non-reachable connections, it is hard to choose a path that does not include failed nodes. Queries oscillate inside the overlay until the alternative path is determined. Thus an increase in node failures is expected to result in an increase to search costs. To evaluate the system's search cost 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 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 in-

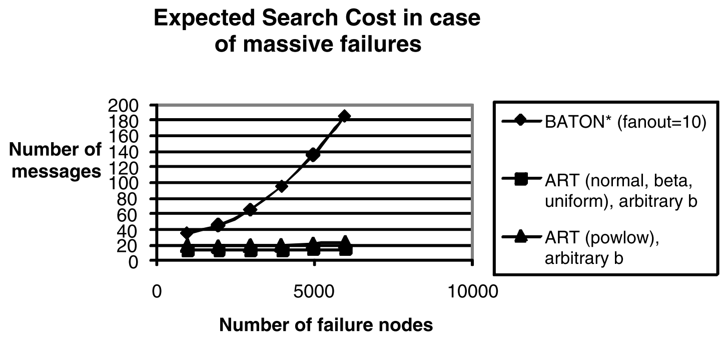


Fig. 40 Search cost in case of massive failures

crease 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), meaning that the *b* parameter does not affect again the overall expected cost 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 40 illustrates this effect.

6.4 Fault tolerance

Node failures and departures are important in order to achieve real life conditions during simulation. D-P2P-Sim give us all the necessary services to implement strategies for

- the self-willing departure of peer nodes (node departure)
- the failure or their sudden retirement for any reason (network failure, application failure etc.)
- monitoring whether the overlay network is parted after successive node failures or departures
- statistics and management in order to monitor all different strategies natively in the simulator.

Statistics include: (i) cost monitoring of importing and retirement of nodes, (ii) the number of failed queries due to node failures and (iii) the number of nodes leading to overlay partition.

Self-willing departure of nodes could be simulated following two approaches. It is possible to simulate scenarios that departure of nodes is initiated in parallel, so that random nodes in the network depart simultaneously. A second approach is to set nodes departing in a sequential mode, where each next node departs after another's complete leave. Though the simulator supports both modes, we have seen that sequential departures are not realistic and as a result they tend to hide problems that might appear (e.g. simultaneous departure of a node and its backup node). On the other hand, multiple concurrent departures make P2P protocol designers to deal with such cases.

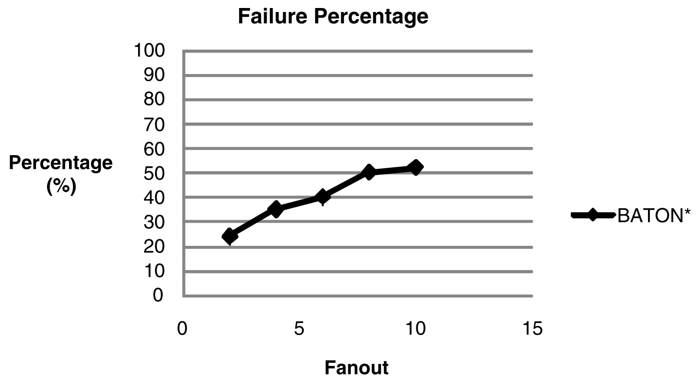


Fig. 41 Failure percentage for BATON*

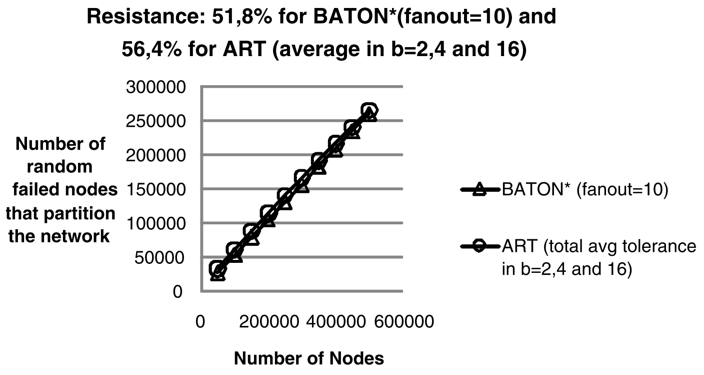


Fig. 42 Failure resistance comparisson between ART and BATON*

In our setup, a network of 0.5M nodes is initialized, with a 10 % of randomly chosen nodes being assigned to leave abruptly (sudden node death), without rebuilding the network. The experiments continue increasing the percentage of nodes failing in the network by 1 % at each round. In each step, the network is checked in order to verify that it is not partitioned into isolated areas that cannot communicate. All the experiments are repeated for fanout 2 up to 10 for Baton*. Figure 41 shows the average number of nodes that is expected to fail before the network is partitioned. The results verify that the network is resistant to failures when a quarter of the nodes fails for fanout = 2. A fanout increase results in higher degrees of tolerance.

Figure 42 shows the resistance (tolerance) comparison between BATON* (fanout = 10) and ART (we considered the average cost in all alternate b parameters). For 50.000 nodes, BATON* and ART resist till 25.900 and 232.031 nodes have randomly failed respectively. For 500.000 nodes, BATON* and ART resist until 259.000 and 264.109 nodes have randomly failed respectively. Thus, the average resistance is 51,8 % for BATON* (fanout = 10) and almost 56,4 % for ART. The existence of LSI and CI routing tables in ART result in higher degrees of tolerance, and as a result

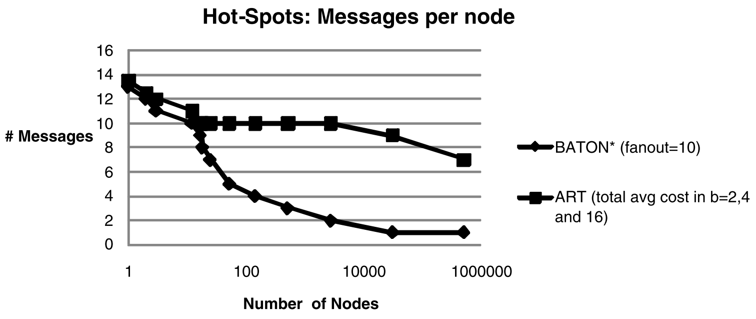


Fig. 43 Messages per node

ART is more Robust than BATON*. However, as we describe in the following, we pay an extra cost of hot-spots existence.

6.5 Hot spots

Figure 43 presents the number of messages that each node receives, for any of the operations (search, update, join, leave), for node populations up to 0.5M nodes and for 3K operations. For ART structure we considered the average cost in all alternate values of b parameter ($b = 2$, $b = 4$ and $b = 16$). Due to the existence of large CI tables in ART, we expect that our method will appear more hot-spots than BATON*. As shown for BATON*, the maximum number of messages in the experiments reached 13 and this only happens for a single node. On the contrary, 30,590 nodes (up to 0.5M) receive a single message. For our method, the maximum number of messages in the experiments reached 14 and this only happens for a single node. On the contrary, from 30,590 nodes (up to 0.5M) receive from 9 down to 7 messages. As a result, we can safely say that even ART retains large CI tables at each level and the probability of hot-spots existence is higher than in BATON*, it still successively balances the visiting rate among nodes.

7 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 ith-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 Fig. 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.

8 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., Xu, Z.: Scalable, efficient range queries for grid information services. In: Proceedings 2nd International Conference on Peer-to-Peer Computing (P2P), Linkoping, Sweden, pp. 33–40 (2002)
2. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Baltimore, MD, pp. 384–393 (2003)
3. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), Portland, OR, pp. 353–366 (2004)
4. Cai, M., Frank, M., Chen, J., Szekely, P.: Maan: a multi-attribute addressable network for grid information services. In: Proceedings 4th International Workshop on Grid Computing (GRID), Phoenix, AZ, pp. 184–191 (2003)

5. Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: Querying peer-to-peer networks using p-trees. In: Proceedings 7th International Workshop on Web and Databases (WebDB), Paris, France, pp. 25–30 (2004)
6. Gupta, A., Agrawal, D., El Abbadi, A.: Approximate range selection queries peer-to-peer systems. In: Proceedings 1st Biennial Conference on Innovative Data Systems Research, Asilomar, CA (2003)
7. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA (2003)
8. Goodrich, M.T., Nelson, M.J., Sun, J.Z.: The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In: Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Miami, FL, pp. 384–393 (2006)
9. Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the internet with PIER. In: Proc. 29th Int. Conf. on Very Large Data Bases, pp. 321–332 (2003)
10. Harvey, N.J.A., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: a scalable overlay network with practical locality properties. In: Proceedings USENIX Symposium on Internet Technologies and Systems, Seattle, WA (2003)
11. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: a balanced tree structure for peer-to-peer networks. In: Proceedings 31st International Conference on Very Large Data Bases (VLDB), Trondheim, Norway, pp. 661–672 (2005)
12. Jagadish, H.V., Ooi, B.C., Tan, K.L., Vu, Q.H., Zhang, R.: Speeding up search in P2P networks with a multi-way tree structure. In: Proceedings ACM International Conference on Management of Data (SIGMOD), Chicago, IL, pp. 1–12 (2006)
13. Karger, D., Kaashoek, F., Stoica, I., Morris, R., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), San Diego, CA, pp. 149–160 (2001)
14. Kaporis, A., Makris, Ch., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, Ch.: Improved bounds for finger search on a RAM. In: Proceedings 11th Annual European Symposium on Algorithms (ESA), Budapest, Hungary, pp. 325–336 (2003)
15. Li, X., Kim, Y.J., Govindan, R., Hong, W.: Multi-dimensional range queries in sensor networks. In: Proceedings 1st International Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, CA, pp. 63–75 (2003)
16. Liau, C.Y., Ng, W.S., Shu, Y., Tan, K.L., Bressan, S.: Efficient range queries and fast lookup services for scalable P2P networks. In: Proceedings 2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing(DBISP2P), Toronto, Canada, pp. 93–106 (2004)
17. Maymounkov, P., Mazieres, D.: Kademlia: a peer-to-peer information system based on the XOR metric. In: Proceedings 1st International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, pp. 53–65 (2002)
18. Perpinan, M.: A review of dimension reduction techniques. Technical report CS-96-09, University of Sheffeld (1997)
19. Prada, C., Villamil, M., Roncancio, C.: Join queries in P2P DHT systems. In: DBISP2P 2008, pp. 93–105 (2008)
20. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), San Diego, CA, pp. 161–172 (2001)
21. Rowstron, A., Druschel, P.: Pastry: a scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (MIDDLEWARE), Heidelberg, Germany, pp. 329–350 (2001)
22. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Prefix hash tree. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing Table of Contents (Brief Announcement), Newfoundland, Canada, p. 368 (2004)
23. Sahin, O.D., Gupta, A., Agrawal, D., El Abbadi, A.: A peer-to-peer framework for caching range queries. In: Proceedings 20th International Conference on Data Engineering (ICDE), Boston, MA, pp. 165–176 (2004)
24. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. **11**(1), 17–32 (2003)
25. Sioutas, S., Papaloukopoulos, G., Sakkopoulos, E., Tsichlas, K., Manolopoulos, Y.: A novel distributed P2P simulator architecture: D-P2P-Sim. In: ACM CIKM 2009, pp. 2069–2070 (2009)

26. Sioutas, S., Papaloukopoulos, G., Sakkopoulos, E., Tsichlas, K., Manolopoulos, Y.: Brief announcement: ART–sub-logarithmic decentralized range query processing with probabilistic guarantees. In: ACM PODC 2010, pp. 118–119 (2010)
27. Triantafillou, P., Pitoura, T.: Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In: VLDB 03 Workshop on Databases, Information Systems, and Peer-to-Peer Computing (2003)
28. Zhang, H., Goel, A., Govindan, R.: Incrementally improving lookup latency in distributed hash table systems. In: SIGMETRICS, San Diego, CA, pp. 114–125 (2003)
29. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: a resilient global-scale overlay for service deployment. IEEE J. Sel. Areas Commun. **22**(1), 41–53 (2004)