Scalable and Hierarchical Distributed Data Structures for Efficient Big Data Management

Spyros Sioutas[1], Gerasimos Vonitsanos[1], Nikolaos Zacharatos[1], and Christos Zaroliagis[1,2](✉)

[1] Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece
{sioutas,mvonitsanos,zacharato,zaro}@ceid.upatras.gr
[2] Computer Technology Institute and Press "Diophantus", Patras University Campus, 26504 Patras, Greece

Abstract. In this work, we survey state of the art hierarchical distributed data structures for the efficient handling of big data, in scenarios where the dominant operation is range queries which have to be answered in real-time. Our main focus is on structures that exhibit stable scalability.

1 Introduction

A great challenge faced by most organizations nowadays concerns their data management. Due to the data endlessly flowing in from sources such as social media activities, Internet of Things (IoT) [6] devices, online streaming services, location based web information, mobile phone usage and consumer preferences expressed on the web, a data-driven revolution is taking place. Analyzing all that information fast can lead to:

– Better decision making based on data-driven insights
– Increased productivity
– Reduced production cost
– Quick fraud detection
– Better customer service

In order to achieve efficient big data management, several infrastructures have been developed. The most popular ones are decentralized systems and MapReduce [5] models.

Decentralized systems, although existed for many years, they have become very popular nowadays and are promoted as the future of Internet networking. They are widely used for sharing resources and store very large data sets, using systems of small computers instead of large costly servers. Typical examples include cloud computing environments, peer-to-peer (P2P) systems and the Internet.

In decentralized systems, data are stored at the network nodes and the most crucial operations are data search and data updates. A decentralized network is represented by a graph, a logical *overlay network*, where its nodes correspond to the network nodes, while its arcs may not correspond to existing communication links, but to communication paths. The complexity (cost) of an operation is measured in terms of the number of messages issued during its execution (internal computations at nodes are considered insignificant). A typical assumption is that messages between nodes are of constant size, they are sent through the communication links, and that communication is asynchronous. Moreover, there is an upper bound on the time needed for a node to send a message and receive an acknowledgement. This facilitates the identification of communication problems (e.g., when communication links or nodes are down).

With respect to its *structure*, the overlay supports the operations *Join* (of a new node v; v communicates with an existing node u in order to be inserted into the overlay) and *Departure* (of an existing node u; u leaves the overlay announcing its intent to other nodes of the overlay). Moreover, the overlay implements an *indexing scheme* for the stored data, supporting the operations *Insert* (a new element), *Delete* (an existing element), *Search* (for an element), and *Range Query* (for elements in a specific range). Throughout this paper, we shall denote by N the number of network nodes and by n the size of data ($N \ll n$).

In terms of efficiency, an overlay network should address the following issues:

- Fast queries and updates: updates and queries must be executed in a minimal number of communication rounds, using a minimal number of messages.
- Ordered data: keeping the data in order facilitates the implementation of various enumeration queries when compared to a simple dictionary that can only answer membership queries.
- Size of nodes: the size of a node is the routing information (links and related data) maintained by this node and it is not related to the number of data elements stored in it. Keeping the size of a node small allows for more efficient update operations, but in general reduces the efficiency of access operations while aggravating fault tolerance.
- Fault tolerance: the structure should be able to discover and heal failures at nodes or links.
- Load balancing: it refers to the distribution of data elements on the nodes. The goal of load balancing is to distribute equally the n elements stored in the N nodes of the network (typically $N \ll n$). That is, if there are N nodes and n data elements, ideally each node should carry approximately k elements, where $\lfloor n/N \rfloor \leq k \leq \lfloor n/N \rfloor + 1$.

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a decentralized algorithm on a cluster (collection of compute servers or nodes), with a designated node as master and the other nodes designated as workers. A MapReduce task consists usually of the following five-step computation.

1. Partition: input is being split and assigned to each worker.
2. Map: each worker node applies the map function to its local data, and writes the output to a temporary storage.
3. Shuffle: worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node.
4. Reduce: worker nodes now process each group of output data, per key, in parallel.
5. Join Results: workers combine their local output data to create the final output result.

The reason why both decentralized architecture networks and MapReduce models became so popular is that in order to increase the computing power of the network/cluster, you can simply add more nodes, so that tasks are divided to more nodes and therefore executed faster, compared with the client-server model, where a brand new server machine is required.

Range query processing in decentralized network environments is a notoriously difficult problem to solve both efficiently and scalably. In cloud infrastructures, a most significant and apparent requirement is the monitoring of thousands of computer nodes, which often requires support for range queries: consider range queries issued in order to identify under-utilized nodes so as to assign them more tasks, or to identify overloaded nodes so as to avoid bottlenecks in the cloud.

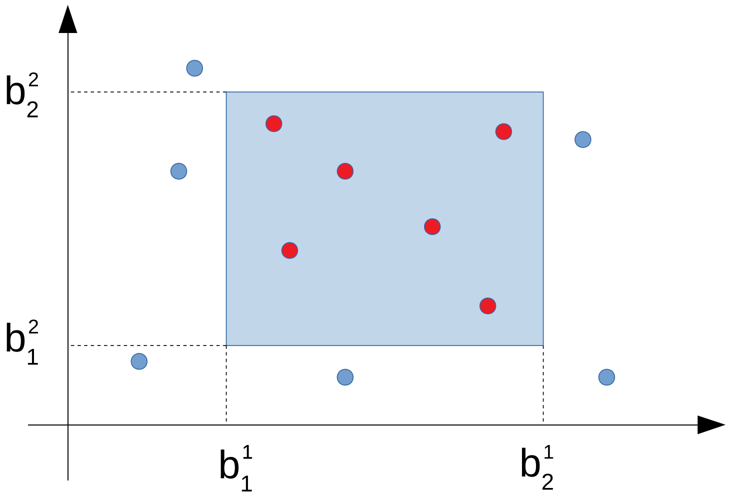


Fig. 1. Two dimensional range query (Color figure online)

The most fundamental, one-dimensional query (also known as the *interval query*) involves retrieving all records x where their value is between an upper bound b_1 and a lower bound b_2, that is, $b_1 \leq value(x) \leq b_2$. Evidently, generalizations to higher dimensions are derived easily. A multi-dimensional query involves retrieving all records $x = (x_1, \ldots, x_d)$ for which $b_1^i \leq value(x_i) \leq b_2^i$, $\forall$ $1 \leq i \leq d$, where d denotes the number of dimensions, b_1^i and b_2^i denote the lower and upper bounds on each dimension of the query respectively, and $value(x_i)$ denotes the value of the elements in the i-th dimension. Each query forms a hyper-rectangle of d dimensions which contains all the elements that satisfy it. For instance (cf. Fig. 1), the two-dimensional range query forms a rectangle on the plane. Red points represent those elements that satisfy the query.

The two dimensional query is very popular because it can answer geographical as well as trajectory queries. Multi-dimensional queries are typically used when browsing online for products, where every filter that you apply is one more dimension on the range query. Many applications require the management and the analysis of massive multi-dimensional datasets.

Overlay structures for decentralized systems can be divided in two big categories: hash-based structures and hierarchical-based structures. Both have their pros and cons, therefore choosing one highly depends on the needs of the users and the applications considered. Hash-based structures (e.g., CAN [15], Chord [19]) use probabilistic methods to distribute the workload among nodes equally, have good exact match query times, but slow range query times [22] since hashing destroys the ordering. On the other hand, hierarchical-based structures support range queries more naturally and efficiently as well as a wider range of operations, since they maintain the ordering of data, but lack the simplicity of hash-based systems.

Due to the importance of the range query problem, we focus in this work on hierarchical overlay structures that support directly range and more complex queries. The main goal of this work is to *present and review* state-of-the-art structures for efficient big data management that exhibit stable scalability.

Over the last years, many data structures have been implemented to address the range query problem on decentralized systems as well as to address the aforementioned efficiency issues of overlay structures. In this framework, we review four important hierarchical structures with their variations, and a ring based one. In particular, we review the following structures.

1. Hierarchical Structures
   (a) BATON [9] and BATON* [8]
   (b) D^2-Tree [3] and D^3-Tree [16]
   (c) ART [18] and ART$^+$ [17]
   (d) SPIS (SPark-based Interpolation Search Tree) [14]
2. Ring-based structures
   (a) P-Ring [4]

The rest of the paper is organized as follows. In Sect. 2 we survey the most popular hierarchical data structures, while in Sect. 3 we survey the ring-based structures. In Sect. 4 we provide a comparison of all structures. We conclude in Sect. 5.

2 Hierarchical Tree-Based Structures

In this section, we present the hierarchical tree-based structures BATON [9], BATON* [8], D^2-Tree [3], D^3-Tree [16], ART [18], ART^+ [17], and SPIS [14].

2.1 BATON

2.1.1 Structure

The Balance Tree Structure for P2P Networks (BATON) [9] is the first overlay network based on a balanced tree structure that can support both exact match and range queries. It is based on a binary balanced tree structure in which each node of the tree is maintained by a node. Each node in the network (cf. Fig. 2) stores a link to its parent, a link to its left child, a link to its right child, a link to its left adjacent node, a link to its right adjacent node, a left routing table to selected nodes on its left hand side at the same level, and a right routing table to selected nodes on its right hand side at the same level. While the tree structure is binary, it has scalability and robustness similar to that of the B-tree.

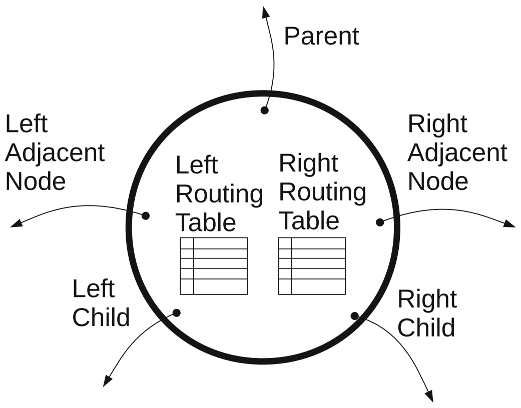


Fig. 2. A node of the BATON structure

Each node of the BATON structure is associated with a *level* and a *number*. The level of the root node is 0, its immediate children are at level 1 and so on. The level of any node is one greater than the level of its parent. Hence the maximum level number in the tree is one less than the height of the tree (which can not be greater than $1.44 \log N$ [11]).

At level L there are at most 2^L nodes in a binary tree. The nodes are numbered from 1 to 2^L (from left to right) within each level, regardless of whether there is a node currently instantiated at that position. The pair of level and number precisely determine the location of a node in the binary tree.

Every physical compute node has an IP address or some other network ID, which can be used to locate the node and communicate with it. Thus every node has a logical ID, which consists of number and level, and a physical ID which consists of its IP address.

The links that each node maintains are the physical IDs of each node. Links to selected neighbors are maintained by means of two special sideway routing tables: a *left routing table* and a *right routing table*. Each of these routing tables contains links to nodes at the same level with numbers that are less (respectively greater) than the number of the source node by a power of 2. The j-th element in the left (right) routing table at node numbered m contains a link to the node at number $m - 2^{j-1}$ (respectively $m + 2^{j-1}$) at the same level in the tree. If there is no such node, an entry is still made in the routing table, but marked as null. A routing table is considered full if all valid links are not null.

Adjacency links are based on an in-order traversal of the tree. Given a node x, the node immediately prior to it in the traversal is *left adjacent* to it, and the node immediately after x is *right adjacent* to it. Note that adjacent nodes may be at different levels of the tree.

2.1.2 Node Join/Departure

A new node that wants to join the network, must know at least one node inside the network. The former sends to the latter a JOIN request which is being carried out in two phases. The first phase is to determine where the new node should join. The second phase is to insert the new node at a specific place and update all the necessary links of the network. The complexity of this action is $\log N$ steps for finding a place for the joining node and $\mathcal{O}(\log N)$ cost for updating the routing tables (which is more efficient than other P2P systems, which usually require $\mathcal{O}(\log^2 N)$ for updating the routing tables).

Only leaf nodes can voluntarily leave the network, and only if their departure will not affect the tree balance. In any other case, any node that wishes to leave the network must find a replacement for itself, which can only be a leaf whose absence does not affect the tree balance. If the leaf node can leave without disrupting the tree balance, it sends a *LEAVE* message to is neighbor nodes, so they can update their routing tables. If a leaf node (that can not leave because it will disrupt the tree balance) or a inner node wishes to leave, then it sends a *FIND REPLACEMENT* message to the network starting from the same level and moving down in order to find a leaf node that will take their place. This process takes at most as many steps as the height of the tree which is $\mathcal{O}(\log N)$. The BATON needs $\log N$ steps to find a replacement node and $\mathcal{O}(\log N)$ cost for updating the routing tables.

2.1.3 Fault Tolerance

In case of a node failure, the node's IP will become unreachable. The first node to discover an unreachable IP must report it to its father which takes care to manage the node failure and update the routing tables.

It has already been described how a node failure is handled. Fault tolerance denotes the ability of the BATON structure to continue its operation, by routing the messages around the missing node. There are two axes in which messages can travel in BATON, vertical and horizontal. The horizontal axis uses the left and right routing tables links of the nodes, while the vertical axis uses the parent/children and adjacency links. The horizontal axis is naturally fault tolerant, since there is a logarithmic expansion of links and therefore a larger number of paths. The vertical axis is rendered fault tolerant because it can create paths that include different levels of the tree. These additional links that BATON stores suffice to provide efficient recovery even with a large number of node failures.

2.1.4 Load Balancing

The BATON is also equipped with a load balancing mechanism. Its goal is to adjust the range of values in the nodes in order to achieve an equal computational load across the network. This is accomplished through data migrations between adjacent nodes (for internal and leaf nodes), and node re-position in the tree (only for leaf nodes).

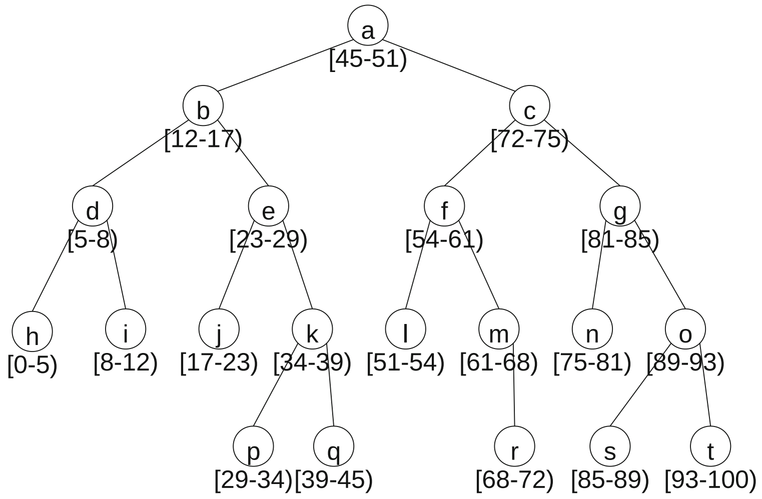


Fig. 3. Range of values for each node of BATON

2.1.5 Queries

In order to answer exact and range queries, a range (or interval) of values is assigned to each node (both leaf and internal). The range of values managed by a node is required to be to the right of the range managed by its left subtree and less than the range managed by its right subtree. Figure 3 shows an instance of BATON for range values $0 - 100$. It is similar to the B^+-tree indexing, but internal nodes also manage a range of data values directly. The queries do not start from the root, but from a random node of the structure.

The search operation in BATON first checks the horizontal axis (using right and left routing tables' links) then the vertical axis (using parent/children & adjacency links) to locate the correct node. A search operation for key k issued at node x works as follows. First, node x checks its own range. If k is within its range, the local index is searched and the search is stopped. Otherwise, it checks its right (left) routing table, if k is greater (less) than its own range to find the rightmost (leftmost) node y that its lower (upper) bound is less (greater) than k. Node x forwards the request to y. This step is performed until no node in the routing tables satisfies the condition. The last node that completed the process now forwards the request to its right (left) child if it exists or its right (left) adjacent node until the correct node is found. The complexity of this process is $\mathcal{O}(\log N)$.

Range query works in the same manner. It follows the same steps, until it finds an intersection of the node range value (interval) with the searched range. Once the intersection is found, partial answers for the range query have always been answered. It then continues to the left and/or to the right, following the adjacency links to cover the remainder search range. Its complexity is $\mathcal{O}(\log N)$ to find the intersection and $\mathcal{O}(1)$ to cover each of the remainder nodes. An answer to a range query that its range is spread across M nodes, requires $\mathcal{O}(\log N + M)$ steps.

2.1.6 Experimental Evaluation

In [9], an experimental evaluation was carried out in a network containing 1000 to 10000 nodes. For a network of size N, $1000 \cdot N$ values were inserted in batches in the domain of $[1, 100000000)$. For each test, 1000 exact queries and 1000 range queries are executed and the average cost is taken. For comparison purposes, CHORD [19] and a Multiway-tree proposed in [12] (which is a simplified version of BATON) were used.

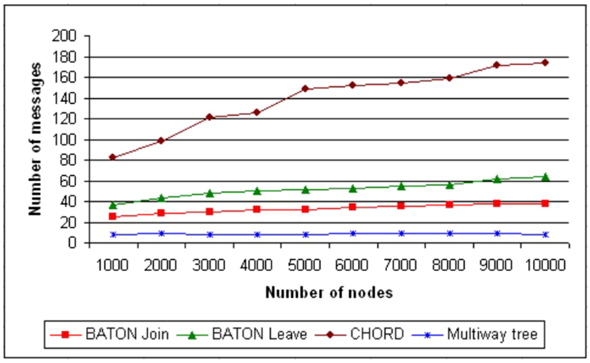


Fig. 4. Updating routing tables in BATON, CHORD and Multiway trees

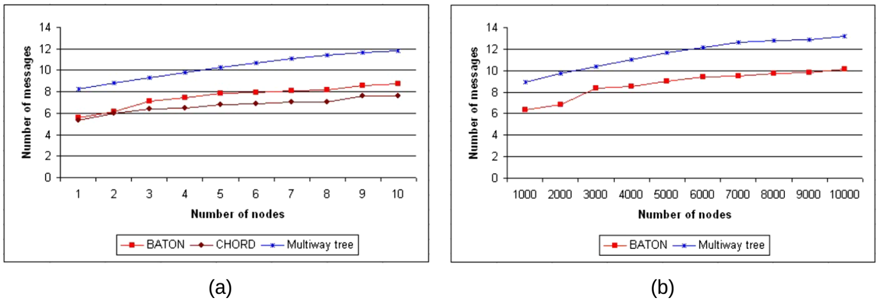


Fig. 5. BATON, CHORD and multiway trees: (a) exact match query; (b) range query

Figures 4 and 5 compare the number of nodes (x-axis) with the number of messages exchanged (y-axis) for routing table updating, exact match and range queries on BATON, CHORD and Multiway-tree structures. We observe that the exchange of messages in BATON remain almost stable, regardless of the number of the network size.

2.2 BATON*

2.2.1 Structure

BATON* [8] is very similar to the BATON structure, but with some core differences.

Each node in BATON* can have up to m children (also called *fanout*) instead of two as in the original structure. In addition to maintaining links to children, the parent node also has to keep track of the ranges of values managed by their children.

Neighbor routing tables at a node maintain links to selected neighbor nodes at the same level which have a distance equal to $d \cdot m^i$, where $d = 1, .., m - 1$ and $i \geq 0$, from the node itself. For example, in Fig. 6, the left routing table of node o maintains links to n, m, l, k, g (shown as purple), which have a distance equal to $1 \cdot 4^0, 2 \cdot 4^0, 3 \cdot 4^0, 1 \cdot 4^1$ and $2 \cdot 4^1$, respectively. Similarly, the right routing table of o maintains links to nodes p, q, r, s. The maximum number of links in routing tables of a node at level L is bounded by $(m - 1) \cdot L$.

For a BATON* structure of fanout m, a range of values managed by a node is greater than the ranges of values managed by the first $\lfloor m/2 \rfloor$ children nodes while less than ranges of values managed by the last $\lceil m/2 \rceil$ children nodes. For instance, in Fig. 6, the range of values managed by o is greater than those of y, x, n, but smaller than those of z, d, p, q.

The cost of search in BATON* becomes $\mathcal{O}(\log_m N)$, as expected. Moreover, the cost of updating routing tables becomes $\mathcal{O}(m \cdot \log_m N)$. It is clear that by increasing the fanout of a node to reduce the cost of the search, the size of the routing tables is increased, and hence the cost of table updating.

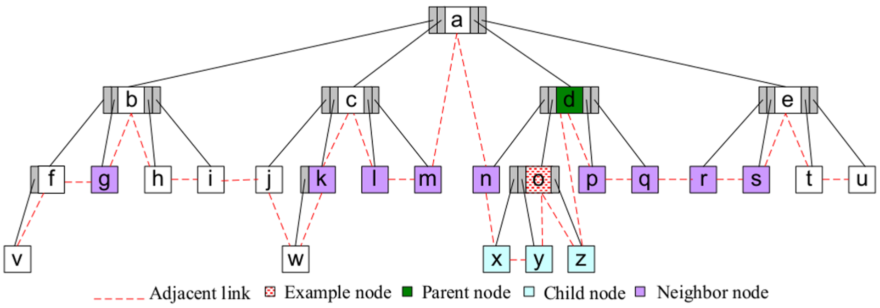


Fig. 6. BATON* structure

2.2.2 Node Join/Departure

A node of the BATON* can only accept a new joining node as a child if it has full neighbor routing tables but does not have m children. Otherwise, it has to forward the join request to either its parent, its lower level adjacent node, or a neighbor node that does not have enough children. In a similar manner, a node can only leave its current position if it does not cause the tree to become unbalanced. Otherwise, it has to find a replacement node by sending a leave request to its lower level adjacent node.

The cost of finding a place for a new joining node or finding a replacement node is $\mathcal{O}(\log_m N)$ since the height of the tree is $\mathcal{O}(\log_m N)$. The cost of updating the routing table is $\mathcal{O}(m \cdot \log_m N)$ for their neighbor routing tables since the maximum number of neighbor nodes a node can have is $\mathcal{O}(m \cdot \log_m N)$, and each of these has to add or remove an entry. Also a newly inserted node has to construct its own routing tables, with up to $\mathcal{O}(m \cdot \log_m N)$ entries, each of which can be obtained in constant time through its parent. In addition, there is a parent link and two adjacency links to create/delete. There can be no children links for a node being joined or departed. Summing these up, the total cost of node join or departure is $\mathcal{O}(m \cdot \log_m N)$.

2.2.3 Fault Tolerance

Fault tolerance in BATON* is very similar to that of its predecessor (BATON). Having even more links on the vertical axis, routing around missing nodes becomes much easier and cheaper, thus making BATON* highly fault tolerant.

2.2.4 Load Balancing

Load balancing in BATON* has two forms. Exchanging data with adjacent nodes, or remove underloaded nodes and place them in overloaded regions of the tree. The former is the easiest and cheapest form of load balancing, however it will not suffice when there are global imbalances. In occasions like these, the latter form of load balancing is used.

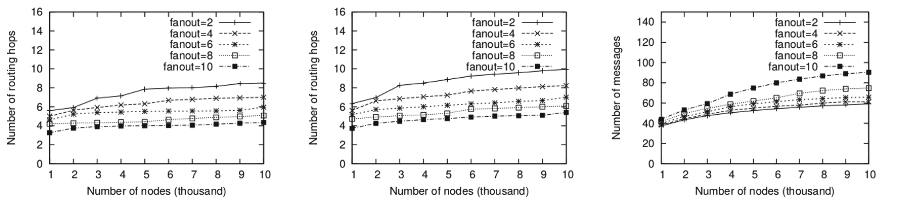
Since BATON* employs the tree structure, internal nodes cannot be easily removed, and hence the latter form of load balancing is only possible for leaf nodes. In general, if a node is overloaded, it first tries to do load balancing with its adjacent nodes. If there is no lightly loaded adjacent nodes, it then tries to find a lightly loaded leaf node to do load balancing. Once such a node is found, that node has to perform a forced leave from its current position and a forced join to the new position to share the workload of the overloaded node.

2.2.5 Queries

Searching in BATON* is very similar to BATON. The sole difference is that when the search request has to be forwarded to a suitable child node, there are $m/2$ options instead of two.

A node u receiving a search request checks to see if there is a neighbor node it knows about which is more appropriate to handle the search. If the searched value is greater than u's upper bound, while there is no right hand side neighbor node of u whose lower bound is less than the searched value, then u checks to find the most suitable child to forward the request. That is the rightmost child whose lower bound is less than the searched value. Similarly, if the searched value is less than the node's lower bound while there is no left hand side neighbor node whose upper bound is greater than the searched value, then the node has to try to find the leftmost child whose upper bound is greater than the searched value, to forward the search request.

The range query algorithm is modified in an analogous way.



(a) Cost of exact match query (b) Cost of range query (c) Cost of updating routing table

Fig. 7. Effect of varying fanout values in BATON*

2.2.6 Experimental Evaluation

In [8], an experimental evaluation was carried out in networks consisted of 1000 to 10000 nodes and the fanout m used was from 2 to 10. For a network of size N, $1000 \cdot N$ values were inserted in batches in the domain of $[1, 1000000000)$. For each test, 1000 exact queries and 1000 range queries are executed and the average cost is taken.

Figure 7 shows the effect of the different fanouts on an exact match query, on a range query and on the cost of updating routing tables. It is clear that by increasing the fanout, exact match queries and range queries become faster at the expense of slower routing table updating. In order for BATON* to become really efficient, one has to tune the fanout to his own needs to get the best results.

Figure 8 shows how much node failure BATON and BATON* can withstand before they are unable to complete a lookup operation. It is obvious that the larger the fanout, the more fault tolerant the structure is.

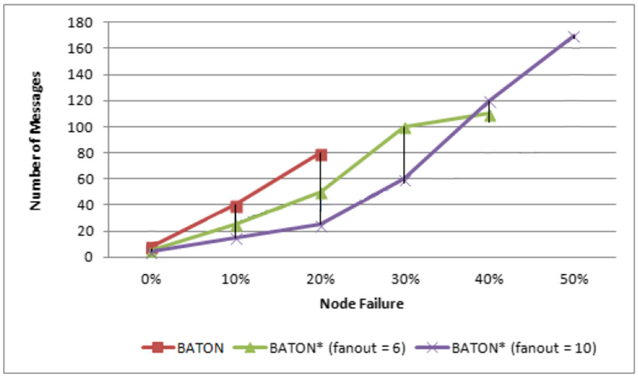


Fig. 8. Lookup operations with node failures in BATON and BATON*

2.3 D^2-Tree

2.3.1 Structure

The Deterministic Decentralized tree (D^2-Tree) [3] is a hierarchical overlay consisting of two levels as shown in Fig. 9. The upper level of the overlay is a perfect binary tree (PBT). The leaves of the tree are representatives of the buckets that constitute the lower level of the overlay. Each bucket is a set of $\mathcal{O}(\log N)$ nodes and it is structured as a doubly linked list. Each node of the bucket points to the node which is a leaf of the PBT and is called the *representative* of the bucket. Additionally it maintains its routing table w.r.t the nodes of all buckets.

Each node in the upper binary tree, maintains an additional set of links to other nodes apart from the standard links which form the tree. More specifically each node v in the tree maintains the following links (cf. Fig. 10):

- Links to its father (if there is one) and its children.
- Links to its adjacent nodes based on an in-order traversal of the tree.
- Links to its leftmost and rightmost leaves of its subtree.
- Links to nodes at the same level as v. These links facilitate an exponential search on the nodes of the same level. Assume that node v lies at level l. In a binary tree, the maximum number of nodes at level l is equal to 2^l. Node v maintains at most $2l$ links: l links to nodes to the right and l links to nodes to the left. The links are distributed in exponential steps, that is the first link

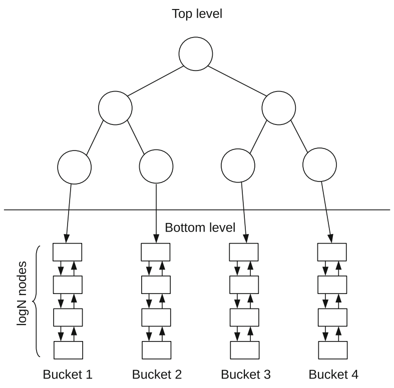


Fig. 9. The D^2-Tree structure

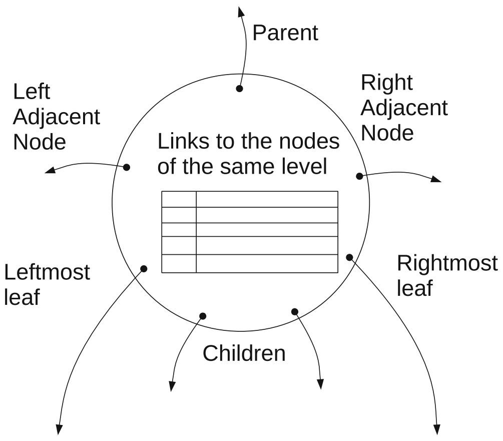


Fig. 10. A D^2-Tree node

points to a node (if there is one) 2^0 positions to the left (right), the second 2^1 positions to the left (right), and the i-th link 2^{i-1} positions to the left (right). These links constitute the routing table of v.

Regarding the complexity bounds, the D^2-Tree:

– uses $\mathcal{O}(\log N)$ space per node;
– achieves a deterministic $\mathcal{O}(\log N)$ query bound;
– achieves a deterministic (amortized) $\mathcal{O}(\log N)$ update bound for elements as well as for node joins and departures;
– exhibits a deterministic (amortized) $\mathcal{O}(\log N)$ bound for load-balancing;
– supports ordered data queries optimally, and tolerates node failures.

2.3.2 Node Join/Departure

When a node z makes a join request to v, then this node is forwarded to its left adjacent leaf u. Then, node z is added to the doubly linked list representing the bucket of u by manipulating a constant number of links. The routing table of z is updated.

When a node v leaves (departs from) the network, then it is replaced by its left adjacent node u (if there is no left adjacent node, then the right one is chosen), which in turn is replaced by its first node z in its bucket as shown in Fig. 11. Link and data information are copied from v to u and from u to z.

When a node v is discovered to be unreachable, its left adjacent node u is first located. This is accomplished by traversing the path to the rightmost leaf starting from the left child of v. Node u fills the gap of v and the first child z in the bucket of u fills the gap left by u. The data contents of u are not moved to another node, but the navigation data (routing tables and other links) are moved to node z that takes its place. Node u has its routing tables recomputed, its links to adjacent nodes set, and the links to the rightmost and leftmost leaves of its subtree are copied from its left and right child respectively.

The join and departure of nodes may cause the size of the buckets to be uneven, which in the long run renders the structure unbalanced. To control the size of the buckets, a weight-based approach is used.

2.3.3 Fault Tolerance

If a node v discovers that node u is unreachable, then it contacts a sibling of u through the routing tables of the siblings of v. This sibling of u is able to reconstruct all links of node u and a node departure of u is initiated, which resolves this failure.

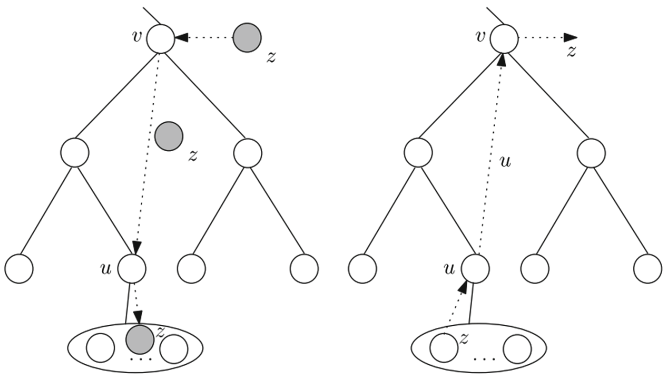


Fig. 11. D^2-tree: To the left (right), the join of z (departure of u) is depicted. The *dotted* labeled *arrows* represent the movement of the nodes denoted by the label.

Due to the way the search operation is implemented, near to root nodes are not crucial, and their failure will not cause more problems that the failure of any other node.

2.3.4 Load Balancing

In the D^2-Tree, the index from the overlay structure is separated using the load balancing mechanism. The number of elements per node is dynamic w.r.t. node joins and departures and it is controlled by the load-balancing mechanism. Moreover, the number of nodes of the perfect binary tree is not connected by any means to the number of elements stored in the structure. The overlay structure supports the operations of node join and node departure, while at the same time it tackles failures of nodes whenever these are discovered.

The load balancing technique of D^2-Tree distributes almost equally the elements among nodes by making use of weights. Weights are used to define a metric of load balance, which shows how uneven the load is between nodes. When the load is uneven, then a data migration process is initiated to equally distribute elements.

The load balancing technique can be described in two steps. The first step is a mechanism that allows efficient local updates of weight information when elements are added or removed at the leaves, which is necessary to avoid hotspots, and the next step is the load-balancing scheme in the tree overlay.

Assume that the overlay structure is denoted by T. When an element is added/removed to/from a leaf u in T, the weights on the path from u to the root must be updated. Assume that node v lies at height h and its children $v_1, v_2, ..., v_s$ are at height $h-1$. The variable *virtual weight* $b(v)$ of v is defined as the weight stored in node v. In particular, for a node v the algorithm maintains the *virtual weight invariant* that $b(v)$ is approximately equal to $e(v)+\sum_{i=1}^{s} b(v_i)$, where $e(v)$ denotes the number of elements residing in a node v.

Assume that an update takes place at leaf u. The path from u to the root is traversed until a node z is found, for the virtual weight invariant holds. Let v be the child of z, for which the virtual weight invariant does not hold. The weights are then recomputed in the path from u to v. Node's z weight information is updated by taking the sum of the weights written in its children plus the number of elements residing at z.

The load balancing mechanism redistributes the elements among nodes when the load between nodes is not distributed equally enough, but it does not tamper with the structure of T. For ease of exposition, assume that T is binary (the algorithm generalizes easily for trees whose nodes have a O(1) number of children).

Let node v at height h have two children p and q at height $h-1$. The density $d(v)$ of v denotes the mean number of elements per node in the subtree of v. Let $c(p,q) = \frac{d(p)}{d(q)}$ denotes the *criticality* of the two brother nodes p and q, representing their difference in densities. The algorithm maintains also the *criticality invariant*, namely that $\frac{1}{c} \leq c(p,q) \leq c$, for some $1 < c \leq 2$. That is, there are no large differences between densities. For instance, choosing $c = 2$ implies that the density of any node can be at most half of that of its brother.

Combining the two steps, each time an update takes place at leaf u, weights in the path from u to the root are updated until a node z is found for which the virtual weight invariant holds. Weights from u to z's child are recomputed. Then, the highest ancestor w of u is located where the criticality invariant is violated, and a node redistribution between w and his brother takes place.

2.3.5 Queries

The search for an element a in the overlay may be initiated from any node v at level l that has range of values $[x_v, x'_v]$. Let z be the node with range of values containing a. Assume without loss of generality that $x'_v < a$. Then, by using the routing tables of v, level ℓ is searched for a node u with right sibling w (if there is such a sibling) such that $x'_u < a$ and $x_w > a$ unless a is in the range of u and the search terminates. This step has $\mathcal{O}(\ell)$ cost, since it simulates a binary search.

If the search continues, then node z will either be an ancestor of u or w or in the subtree rooted at the right child $r(u)$ of u or in the subtree rooted at the left child $l(w)$ of w. First, the rightmost leaf r of u and the leftmost leaf l of w are located. If $x'_r \geq a$ then a is in the subtree of $r(u)$ and symmetrically if $x_l \leq a$ then a is in the subtree of $l(w)$. Note that at most one of these cases may hold for a. For instance, if $x'_r \geq a$ then an ordinary top down search from node $r(u)$ suffices to find z in $\mathcal{O}(\log N)$ steps (or in its bucket). Symmetrically, this is true also for $l(u)$. However, if both cases do not hold, then z is an ancestor of u or w. In this case a bottom-up search is initiated from u towards the root. This step can be carried out in $\mathcal{O}(\log N)$ steps as well.

A range query $[a, b]$ initiated at a node v, invokes a search operation for element a. Node u that contains a returns to v all elements in this range. If all elements of u are reported, then the range query is forwarded to the right adjacent node (based on the in-order traversal) and continues until an element larger than b is reached for the first time.

2.4 D^3-Tree

The Dynamic Deterministic Decentralized Tree (D^3-Tree) [16] is an extension of D^2-Tree that adopts all of its strengths and extends it in two respects: it introduces an enhanced fault tolerant mechanism and it is able to answer efficiently search queries when massive node failures occur. D^3-Tree achieves the same deterministic (worst-case or amortized) bounds as D^2-Tree for search, update and load-balancing operations, and answers search queries in $\mathcal{O}(\log N)$ amortized cost under massive node failures.

The D^3-Tree has a significantly small redistribution rate (structure redistributions after node joins or departures), while element load-balancing is rarely necessary. It also achieves a significant success rate in element queries, even under massive node failures.

2.4.1 Structure

Similar to the D^2-Tree, the D^3-Tree consists of two levels. The upper level is a Perfect Binary Tree (PBT) of height $\mathcal{O}(\log N)$. The leaves of this tree are representatives of the buckets that constitute the lower level of the D^3-Tree. Each bucket is a set of $\mathcal{O}(\log N)$ nodes which are structured as a doubly linked list as shown in Fig. 9. Each node v of the D^3-Tree maintains an additional set of links (described below) to other nodes apart from the standard links which form the tree. The first four sets are inherited from the D^2-Tree, while the fifth set is a new one that contributes in establishing a better fault-tolerance mechanism.

- Links to its father and its children.
- Links to its adjacent nodes based on an in-order traversal of the tree.
- Links to nodes at the same level as v. The links are distributed in exponential steps; the first link points to a node (if there is one) 2^0 positions to the left (right), the second 2^1 positions to the left (right), and the i-th link 2^{i-1} positions to the left (right). These links constitute the routing table of v and require $\mathcal{O}(\log N)$ space per node.
- Links to leftmost and rightmost leaf of its subtree. These links accelerate the search process and contribute to the structure's fault tolerance when a considerable number of nodes fail.
- For leaf nodes only, links to the buckets of the nodes in their routing tables. The first link points to a bucket 2^0 positions left (right), the second 2^1 positions to the left (right) and the i-th link 2^{i-1} positions to the left (right). These links require $\mathcal{O}(\log N)$ space per node and keep the structure fault tolerant, since each bucket has multiple links to the PBT.

2.4.2 Node Joins/Departures

When a node z makes a join request to v, v forwards the request to an adjacent leaf u. If u is a PBT node, the request is forwarded to the left adjacent node, w.r.t. the in-order traversal, which is definitely a leaf (unless v is a leaf itself). In case v is a bucket node, the request is forwarded to the bucket representative, which is a leaf. Then, node z is added to the doubly linked list of the bucket represented by u. In node joins, a simplification is made, that the new node is clear of elements and it is placed after the most loaded node of the bucket. Thus the load is shared and the new node stores half of the elements of the most loaded one.

When a node v leaves the network, it is replaced by an existing node, so as to preserve the in-order adjacency. All navigation data are copied from the departing node v to the replacement node, along with the elements of v. If v is an internal PBT node, then it is replaced by the first node z in its bucket. If v is a leaf, then it is directly replaced by z. Then v is free to depart.

2.4.3 Node Redistribution

Node redistribution guarantees that if there are z nodes in total in the y buckets of the subtree of v, then after the redistribution each bucket maintains either $\lfloor z/y \rfloor$ or $\lfloor z/y \rfloor + 1$ nodes. The redistribution in the subtree v works as follows.

Assume that the subtree v at height h has K buckets. A traversal of all the buckets is carried out to determine the exact value $|v|$, which denotes the number of nodes in the buckets of the subtree of v. The redistribution starts from the rightmost bucket b and it is performed in an in-order fashion so that elements in the nodes remain unaffected. Assume that b has q extra nodes that must be transferred to other buckets. Since bucket b maintains a link to the next bucket on the left, b', the extra nodes q are transferred there, while the internal nodes of PBT are also updated (because the in-order traversal must remain untouched). Finally, bucket b informs b' to take over, and the same procedure applies again with b' as the source bucket. The case where q nodes must be transferred to bucket b from b' is symmetric. In the case that b' does not have the q nodes that b needs, b' has to find them on the remaining buckets on the left, so it travels towards the leftmost bucket of the subtree until $q \leq \sum_{i=1}^{s} |b_i|$, where $|b_i|$ is the size of the i-th bucket on the left. Then, nodes of b_s move towards b' one bucket at a time, until it goes to b' and finally into b.

2.4.4 Load Balancing

The load balancing technique in a subtree v (with $|v|$ nodes in the subtree) is carried out as follows. A bottom-up calculation of the weights in all nodes of v is performed, to find $w(v)$ of v. The algorithm starts from the right most node w of the rightmost bucket b and it is performed in an in-order fashion. Assume that w has m extra elements which must be transferred to node w'.

- If w is a bucket node, w' is its left node, unless w is the first node of the bucket and then w' is the bucket representative.
- If w is a leaf node, then w' is the left in-order adjacent of w.
- If w is an internal binary node, then its left in-order adjacent is a leaf and w' is the last node of its bucket.

The first m elements removed from w and are added to end of the element queue of w', in order to preserve the indexing structure of the tree. The ranges of both w and w' nodes are updated respectively. The case where m elements must be transferred from w' to w is symmetric. When w' contains less elements than the m elements that w needs, it travels towards the leftmost node of the subtree following the in-order traversal, until $m \leq \sum_{i=1}^{s} e(u_i)$, where $e(u_i)$ is the number of elements of the i-th node on the left. Then the elements of u_s are transferred to u_{s-1}, from u_{s-1} to u_{s-2} and so on, until the m elements are moved from w' to w.

2.4.5 Fault Tolerance

When a node w discovers that v is unreachable, the network initiates a *node withdrawal* procedure by reconstructing the routing tables of v, in order for v to be removed smoothly, as if v was departing. If v belongs to a bucket, it is removed from the structure and the links of its adjacent nodes are updated. In case v is an internal binary node, its right adjacent node u is first located, in order to replace v.

If v is a leaf, then it should be replaced by the first node u in its bucket. In the D^2-Tree, if a leaf was found unreachable, contacting its bucket would be infeasible, since the only link between v and its bucket would have been lost. This weakness was eliminated in the D^3-Tree, by maintaining multiple links towards each bucket, distributed in exponential steps (in the same way as the horizontal adjacency links). This way, when w is unable to contact v, it contacts directly the first node of its bucket u and u replaces v. Regardless of node's v position in the structure, the elements stored in v are lost.

2.4.6 Queries

The search for an element a may be initiated from any node v at level l. If v is a bucket node, then if its range contains a the search terminates, otherwise the search is forwarded to the bucket representative, which is a binary node. If v is a PBT node, then let z be the node with range of values containing a, $a \in [x_z, x'_z]$ and assume without loss of generality that $x'_v < a$. The opposite case is completely symmetric. A horizontal binary search is performed at level l using the routing tables of v. More specifically, the rightmost links of the routing tables are followed until a node q is found, such that $x_q > a$, or until the rightmost node q_r of level l is reached. If the first case holds, a is between q and the last visited node in the left of q. The search continues to the left, decreasing the travelling step by one. The algorithm continues travelling left and right while gradually decreasing the travelling step until it finds a node u with sibling w (if there is such sibling) such that $x'_u < a$ and $x_w > a$. If the second case holds, then $x'_{q_r} < a$ and according to the in-order traversal, the search continues to the right subtree of q_r. If a is in the range of any of the visited nodes of level l, the search terminates.

Having located nodes u and w, the horizontal search is terminated and a vertical search is initiated. Node z will either be the common ancestor of u and w, or it will be in the right subtree rooted at u, or in the left subtree rooted at w. Node u contacts the rightmost leaf y of its subtree. If $x_y > a$ then an ordinary top down search from node u will suffice to find z. Otherwise node z is in the bucket of y, or in its right in-order adjacent node (this is also the common ancestor of u and w), or in the subtree of w.

Overall, the search for an element a is carried out in $\mathcal{O}(\log N)$ steps.

A range query $[a, b]$ initiated at a node v, invokes a search operation for element a. Node z that contains a returns to v all elements in its range. If all elements of u are reported, then the range query is forwarded to the right adjacent node (based on the in-order traversal) and continues until an element larger than b is reached for the first time.

2.4.7 Queries with Node Failures

In a network with node failures, an unsuccessful search for element a refers to the cases where either z (the node with range of values containing a, i.e., $a \in [x_z, x'_z]$) is unreachable, or there is a path to z but the search algorithm can not follow it to locate z due to failures of intermediate nodes. D^2-Tree provides a preliminary fault-tolerant mechanism that succeeds only in the case of a few node failures. That mechanism cannot deal with massive node failures (also known as churn) i.e., its search algorithm may fail to locate a. The difference in D^3-Tree is that during the horizontal search, if the most distant right adjacent of v located in position 2^j is unreachable, v keeps contacting its right adjacent nodes by checking positions $2^{j-1}, 2^{j-2}, \ldots$ (i.e., by decreasing repeatedly the exponent by 1), until it finds a node q which is reachable.

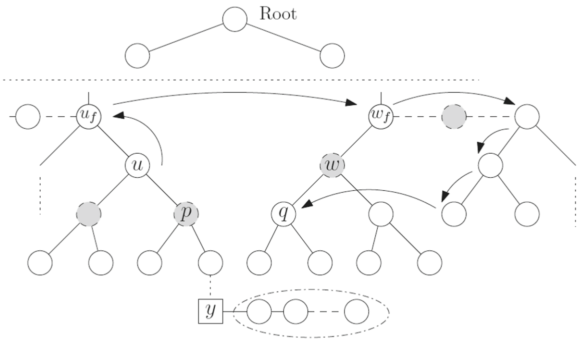


Fig. 12. Example of vertical search between u and unreachable w

In case $x'_q < a$ the search continues to the right using the most distant right adjacent of q. Otherwise, the search continues to the left and q contacts its most distant left adjacent p which is in the right of v. If p is unreachable, q does not decrease the exponent by 1, but contacts directly its nearest left adjacent (at position 2^0) and asks it to search to the left. This improvement reduces the number of messages that are meant to fail, because of the exponential positions of nodes in routing tables and the nature of binary horizontal search.

A vertical search to locate z is always initiated between two siblings u and w, which are either both active, or one of them is unreachable, as shown in Fig. 12 where the left sibling u is active and w, the right one, is unreachable. In both cases, the subtree of the active sibling is searched first, then the common ancestor is contacted and then, if the other sibling is unreachable, the active sibling tries to contact its corresponding child (right child for left sibling and left child for right sibling). When the child is found the search is forwarded to its subtree.

In general, when node u wants to contact the left (right) child of unreachable node w, the contact is accomplished through the routing table of its own left (right) child. If its child is unreachable (Fig. 12), then u contacts its father u_f and u_f contacts the father of w, w_f. Then w_f contacts its grandchild through its left and right adjacents and their grandchildren.

In the case where the initial node v is a bucket node, then if its range contains a the search terminates, otherwise the search is forwarded to the bucket representative. If the bucket representative has failed, the bucket contacts its other representatives right or left, until it finds a representative that is reachable. Then the procedure continues as described above for the case of a binary node.

2.4.8 Experimental Evaluation

In [16], an experimental evaluation was carried out in networks consisting from 1000 to 10000 nodes. For a size of network N, $1000 \times N$ elements were inserted. The number of passing messages between the nodes was used to measure the performance of the system.

For Node Join/Departures. $2 \times N$ nodes were updated. Figure 13 shows that the D^3-Tree update and redistribution mechanism achieves a better amortized redistribution cost, compared to those of BATON, BATON* and P-Ring.

Cost of Queries with/without Node Failures. To measure the network performance for the operation of single queries, experiments were conducted for each N (1000 to 10000), performing $2M$ (M is the number of binary nodes) searches. The search cost is shown in Fig. 14.

To measure the network performance for the operation of element search with node failures, experiments were conducted for different percentages of node failures: 10%, 20%, 30%, 40%, 50%, 75%. For each value of N considered (in the range from 1000 to 10000) and node failure percentage, $2M$ searches were performed. In order to get a better estimation of the search cost, a different set of nodes was forced to fail each time. Figure 15 depicts the increase in search cost when massive node failures take place in D^3-Tree, BATON, different fanouts of BATON* and P-Ring. The graph is irrelevant to N.

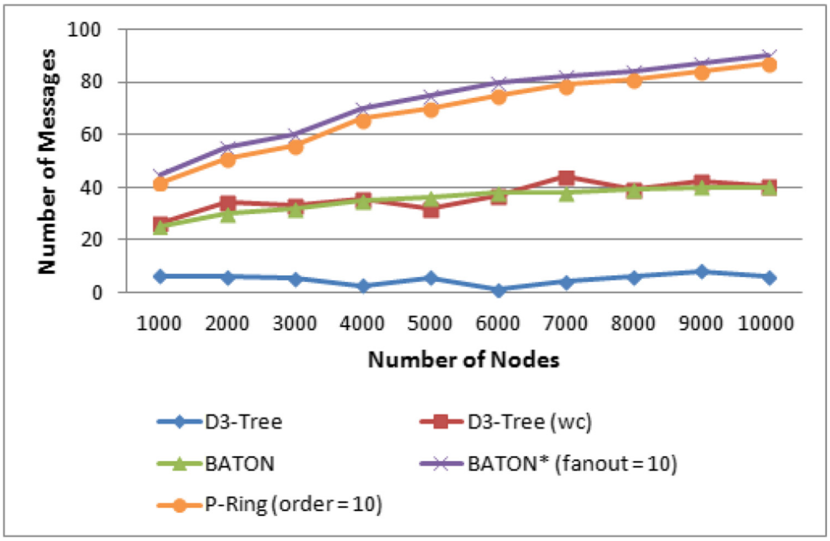


Fig. 13. Average messages for node updates

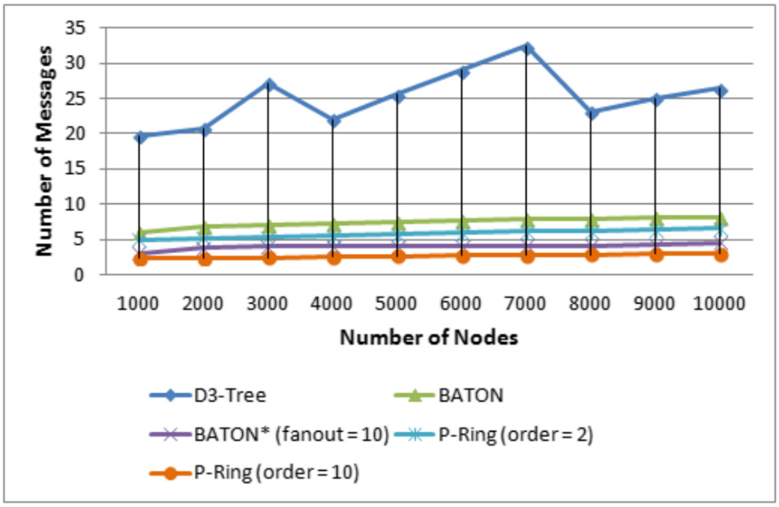


Fig. 14. Cost of queries without node failures

We observe that D^3-Tree can withstand up to 50% node failure while keeping the search cost low. P-Ring and BATON* (both of fanout/order 10) can withstand the same percentage of node failure, but the cost of search operation rises above that of the D^3-Tree after 30%. BATON can not handle the search operations after 20% of node failure, while BATON* (with fanout 6) can withstand up to 40%.

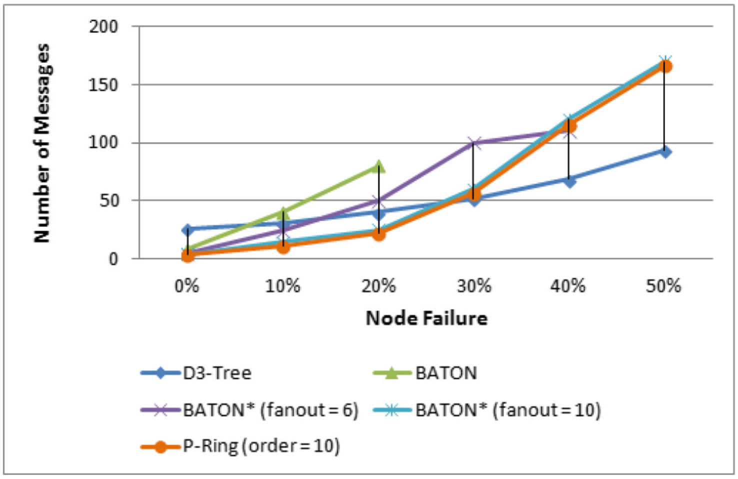


Fig. 15. Cost of queries under node failure

2.5 ART

The Autonomous Range Tree (ART) [18] is an exponential tree structure, which remains unchanged with high probability (w.h.p.), and organizes a number of fully dynamic buckets of nodes. The communication cost of query and update operations is $\mathcal{O}(\log_2 b \log N)$ hops, where $b = 2^{2^i}$, $i = 1, 2, 3...$ Moreover, ART is a fully dynamic and fault-tolerant structure, which supports the join/leave node operations in $\mathcal{O}(\log\log N)$ expected number of hops w.h.p.

2.5.1 Structure

One of the basic components of the ART structure is the Level Range Tree (LRT). LRT will be called upon to organize collections of nodes at each level of ART. LRT is built by grouping nodes 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 nodes share the same direct ancestor, where b is a double-exponentially power of two. Thus, multiple independent trees are imposed on the collection of nodes. Figure 16 shows the LRT structure for $b = 2$.

The degree of the nodes at level $i > 0$ is $d(i) = t(i)$, where $t(i)$ indicates the number of nodes at level i. It holds that $d(0) = b$ and $t(0) = 1$. Let n be w-bit keys. Each node 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 nodes. Each node is also equipped with a table named Left Spine Index (LSI), which stores pointers to the nodes of the left-most spine. Furthermore, each node of the left-most spine is equipped with a table named Collection Index (CI), which stores pointers to the collections of nodes presented at the same level (see pointers directed to collections of last level). Nodes having the same father belong to the same collection.

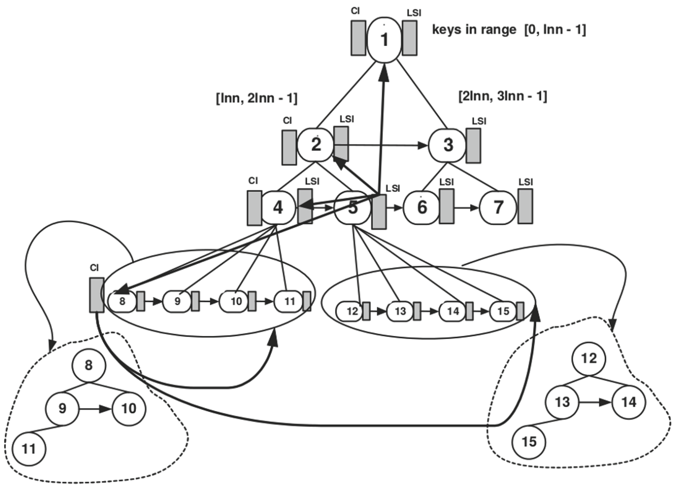


Fig. 16. The LRT structure for $b = 2$

ART stores cluster of nodes only, each of which is structured as an independent decentralized architecture (it can be BATON*, Chord, Skip-Graphs, etc). The backbone-structure of ART is exactly the same with LRT. Moreover instead of LSI, which reduces the robustness of the whole system, a Random Spine Index routing table is introduced, which stores pointers to randomly chosen cluster nodes.

2.5.2 Node Joins/Departures

The operation of join/leave of nodes inside a cluster-node is modelled as the combinatorial game of balls in bins presented in [10]. In this way, for a random sequence of join/leave node operations drawn from a distribution of density $\mu(\cdot)$, the expected load w.h.p. of each cluster-node never exceeds $\Theta(\log N)$ in size and never becomes zero. In skew sequences, though, the load of each cluster-node may become $\Theta(N)$ in the worst case.

When a node wants to join the network, it is assumed that this node is accompanied by a key, and that key designates the exact position in which the new node must be inserted. If an empty node u makes a join request at a particular node v (which is called *entrance node*) then there is no need to get to a different cluster node than the one in which u belongs. Similarly, the algorithm for the departure of a node u assumes that the departure can be made from any node in the ART structure. This may not be desirable, and in many applications it is assumed that the choice for departure of node u can be made only from this node.

2.5.3 Fault Tolerance

In the ART structure, the overlay of cluster nodes remains unchanged in the expected case w.h.p., so in each cluster_node the algorithms for node failure and restructuring are those inherited by the decentralized architecture used.

2.5.4 Queries

The search algorithm gets as input a node in which the query is initiated, and a key to search. The first step of the algorithm is to locate the levels of the ART where the desired cluster nodes are located. This is achieved by using the RSI index. The next step is to locate the correct cluster node in the right level. The first position of RSI (notated as RSI[1]) always points to the next cluster node at the same level. Following RSI[1], the correct cluster node can be found at the right level. The final step is to search inside the decentralized structure that each cluster node holds to locate the key.

The Range search algorithm gets as input a node in which the query is initiated and a range of keys $[k_l, k_r]$. It then calls the search algorithm on the same node with key k_l and by exploiting the order of the keys on each node it performs a right linear scan until it finds a key $K > k_r$.

2.5.5 Experimental Evaluation

In [18] an experimental evaluation was carried out, including a detailed performance comparison with BATON*. In particular, each cluster-node is implemented as a BATON*. The network was tested with different number of nodes ranging up to 500000. The data inserted was 2000 times the size of the network, with numbers in the universe $[1, ..., 1000000000]$ inserted in batches, following beta, uniform and power law distributions. For each test, 1000 exact match queries and 1000 range queries were executed, and the average costs of operations are calculated.

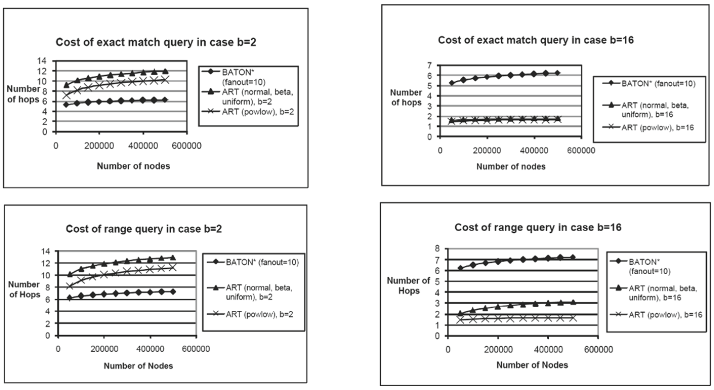


Fig. 17. Exact and range query times of BATON* and ART with $b = 2, 16$ for normal, beta, uniform and power-law input distributions

We observe in Fig. 17 that except for the case where $b = 2$ (right part of Fig. 17), the ART structure outperforms BATON* structure in both exact and range queries by a wide margin.

2.6 ART$^+$

ART$^+$ [17] is similar to its predecessor ART, regarding the structure's outer level. Their difference, which introduces performance enhancements, lies in the fact that each cluster-node of ART$^+$ is structured as a D^3-Tree.

2.6.1 Structure

The backbone structure of ART$^+$ (cf. Fig. 18) is similar to the Level Range Tree (LRT), in which some interventions have been made to improve its performance and increase the robustness of the whole system. ART$^+$ is built by grouping cluster-nodes having the same ancestor and organizing them in a tree structure recursively. A cluster-node is defined as a bucket of ordered nodes. The innermost level of nesting (recursion) will be characterized by having a tree in which

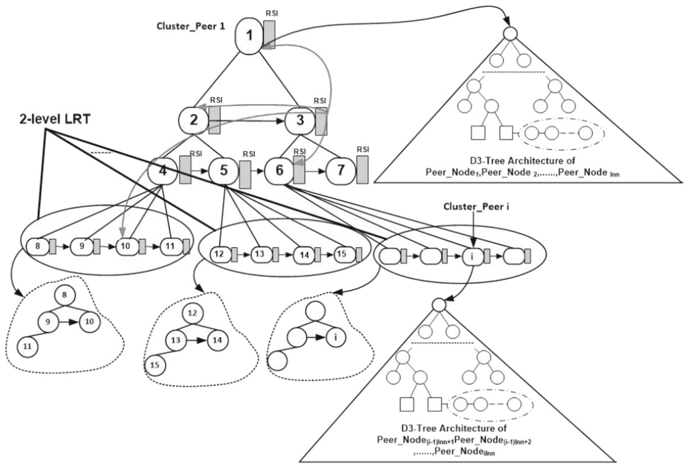

Cluster_Peer 1
RSI
2-level LRT
D3-Tree Architecture of Peer_Node1,Peer_Node 2,......,Peer_Node lnn
Cluster_Peer i
D3-Tree Architecture of Peer_Node(i-1)lnn+1Peer_Node(i-1)lnn+2 ,......,Peer_Nodeilnn

Fig. 18. ART$^+$ structure

no more than b cluster-nodes share the same direct ancestor (where $b = 2^{2^i}$, $i = 1, 2, 3..$). Thus, multiple independent trees are imposed on the collection of cluster-nodes. The height of ART$^+$ is $\mathcal{O}(\log \log_b N)$ in the worst case. The ART$^+$ structure remains unchanged w.h.p.

Similarly to ART, the degree of the cluster-nodes at level $i > 0$ is $d(i) = t(i)$, where $t(i)$ indicates the number of cluster-nodes at level i. It holds that $d(0) = b$ and $t(0) = 1$. At initialization step, the first node, the $(\ln n + 1)$-th node, the $(2 \cdot \ln n + 1)$-th node and so on are chosen as bucket representatives.

Let n be w-bit keys, N be the total number of nodes and N' be the total number of cluster-nodes. Each node with label i (where $1 \leq i' \leq N$) of a random cluster stores ordered keys that belong in the range $[(i' - 1) \ln^2 n, i' \ln^2 n - 1]$, where $N' = n/\ln n$. Each cluster-node with label i' (where $1 \leq i' \leq N'$) stores ordered nodes with sorted keys belonging in the range $[(i' - 1) \ln^2 n, i' \ln^2 n - 1]$, where $N' = n/\ln^2 n$ or $N' = N/\ln n$ is the number of cluster-nodes.

ART$^+$ stores cluster-nodes only, each of which is structured as an independent decentralized architecture, which changes dynamically after node join/leave and element insert/delete operations inside it.

In contrast to its predecessor, ART, whose inner level was structured as a BATON*, each cluster-node of ART$^+$ is structured as a D^3-Tree. Each cluster-node is equipped with a routing table named Random Spine Index (RSI), which stores pointers to cluster-nodes belonging to a random spine of the tree (instead of the LSI of LRT which stores pointers to the nodes of the left-most spine, decreasing this way the robustness of the structure). Moreover, instead of using fat Collection Index (CI) tables, which store pointers to the collections of nodes presented at the same level, the appropriate collection of cluster-nodes is accessed by using a 2-level LRT structure.

2.6.2 Node Joins/Departures

In ART$^+$, the overlay of cluster-nodes remains unaffected in the expected case w.h.p., when nodes join or leave the network.

A node u can make a join/leave request to a node v, which is located at cluster node W. Since the expected size of W is w.h.p. $O(\log^k N)$, for some $k = O(1)$, the node join/leave can be carried out in $\mathcal{O}(\log\log N)$ hops. The outer structure of ART$^+$ remains unchanged w.h.p. as mentioned before, but each D^3-Tree structure changes dynamically after node join/leave operations. According to D^3-Tree performance evaluation, the node join/leave can be carried out in $\mathcal{O}(\log\log N)$ hops.

Similarly to ART, the operation of join/leave of nodes inside a cluster-node is modelled as the combinatorial game of balls in bins presented in [10]. In this way, for a random sequence of join/leave node operations drawn from a distribution of density $\mu(\cdot)$, the expected load w.h.p. of each cluster-node never exceeds $\Theta(\log N)$ in size and never becomes zero. In skew sequences, though, the load of each cluster-node may become $\Theta(N)$ in worst case.

2.6.3 Fault Tolerance

In the ART$^+$ structure, similarly to ART, the overlay of cluster-nodes remains unchanged in the expected case w.h.p., so in each cluster-node the algorithms for node failure and restructuring are those of the decentralized architecture used. D^3-Tree is a highly fault-tolerant structure, since it supports procedures for node withdrawal and handles massive node failures efficiently.

2.6.4 Queries

Since the structure's maximum number of nesting levels is $\mathcal{O}(\log_b \log N)$ and at each nesting level i the standard LRT structure has to be applied in $N^{1/2^i}$ collections, the whole searching process requires $\mathcal{O}(\log_b^2 \log N)$ hops. Then, the target node has to be located by searching the respective decentralized structure. Since there is a polylogarithmic load in each cluster node, the total query complexity of $\mathcal{O}(\log_b^2 \log N)$ follows.

By exploiting the order of the keys on each node, it turns out that a range query requires $\mathcal{O}(\log_b^2 \log N + |A|)$ hops, where $|A|$ is the answer size.

2.6.5 Experimental Evaluation

In [17], the performance of ART$^+$ was evaluated by experiments that ran on different number of nodes N from 50000 to 500000. Each cluster node stores no more than $0.75 \log^2 N$ nodes in smooth distributions (as proved in [18]) and no more than $2.5 \log^2 N$ nodes in non-smooth distributions. Moreover, the elements inserted were $2000 \cdot N$ which are numbers from the universe $[1, .., 1.000.000.000]$. The number of passing messages was used to measure the performance.

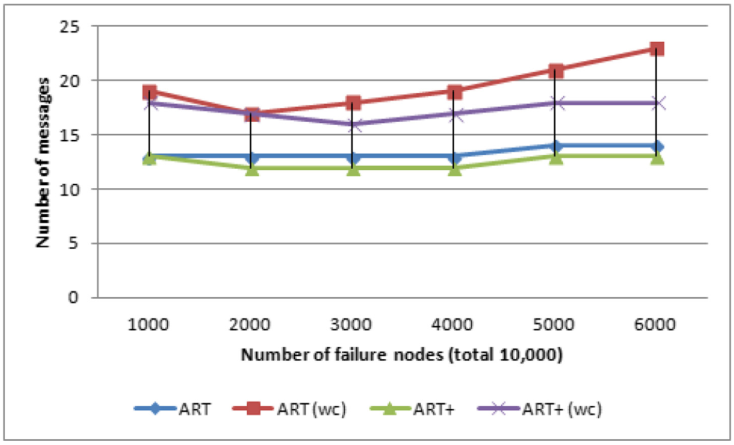
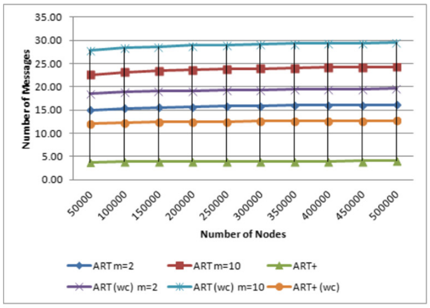


Fig. 19. Lookup Operations with Node Failures in ART and ART$^+$

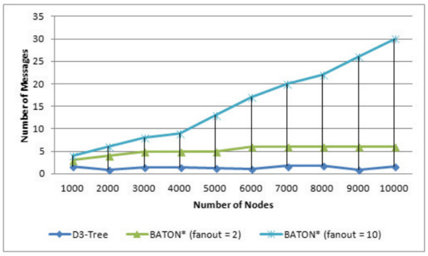
Cost of Queries under massive node failures. In case of massive node failures, the search algorithm has to find alternative paths to overcome the unreachable nodes. Thus, an increase in node failures results in an increase in search costs. To evaluate the system in case of massive failures, the system was initiated with 10000 nodes and they were let to randomly fail without recovering. Since the backbone of ART$^+$ remains unaffected w.h.p., the search cost is restricted inside a cluster-node (D^3-Tree), meaning that parameter b does not affect the overall expected cost. Figure 19 illustrates the effect of massive failures of ART and ART$^+$.

Cost of Load-Balancing Operations. To evaluate the cost of load-balancing, the network was tested with a variety of distributions. For a network of N total nodes, $2N$ node updates were performed. Both ART and ART$^+$ remain unaffected w.h.p., when nodes join or leave the network, thus the load-balancing performance is restricted inside a cluster-node (BATON* for ART, and D^3-Tree for ART$^+$), meaning that parameter b does not affect the overall cost. The load-balancing cost is depicted in Fig. 20a. Both expected and worst case values are depicted in the same graph.

Experiments confirm that ART$^+$ has an $\mathcal{O}(\log \log N)$ load-balancing performance, instead of the ART performance of $\mathcal{O}(m \cdot \log_m \log N)$. Thus, even in the worst case scenario, the ART$^+$ outperforms ART, since D^3-Tree has a more efficient load-balancing mechanism than BATON*; cf. Fig. 20b.



(a) ART and ART^{+}



(b) D^3-Tree and BATON*

Fig. 20. Cost of load-balancing operation.

2.7 SPark-based Interpolation Search Tree (SPIS)

Spark [21] is the successor of Hadoop [20], which is the open source implementation of the MapReduce model. In [14], the classic Interpolation Search Tree [13] was integrated into Spark's [21] distributed environment. Spark uses *Resilient Distributed Datasets* (RDDs) as its fundamental data organization scheme. An RDD is an immutable (i.e., read-only) distributed collection of objects. The datasets are divided into partitions, which are further computed on different nodes of the cluster. However, *Data Frames* (DFs) are also supported that provide more rich semantics and also provide additional optimizations for running SQL queries over distributed data.

The classic Interpolation Search Tree (IST) has the following properties:

– It requires space $\mathcal{O}(n)$ for a data set of cardinality n.
– The amortized insertion and deletion cost is $\mathcal{O}(\log\log n)$
– The expected search time on data sets with smooth probability density is $\mathcal{O}(\log\log n)$
– The worst case search time is $\mathcal{O}((\log n)^2)$.
– The data structure supports sequential access in linear time and operations Predecessor, Successor, and Min in time $\mathcal{O}(1)$. In particular, it can be used as a priority queue.

In [14], the Spark's RDD API was used since the focus was in providing faster search capabilities at the partition level. Since RDDs are immutable, inserting (deleting) elements in (from) the tree (even though the tree supports such actions) were not a concern, and the focus was on search and range queries. Range search queries turned out to be faster than Spark's built-in functions. Figure 21 shows how the sorting and partitioning is done in Spark's distributed environment.

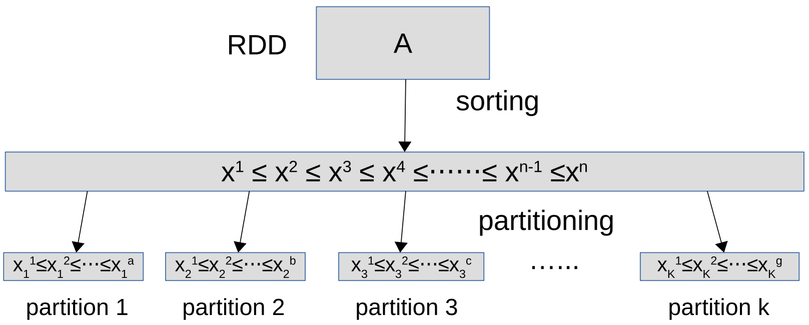


Fig. 21. Spark's sorting procedure

2.7.1 Structure

The Interpolation Search Tree (IST) is a multi-way tree, where the degree of a node depends on the cardinality of the set stored below it. It requires $\mathcal{O}(n)$ space for an element set of cardinality n. More precisely, the degree of the root is $\Theta(n^a)$, for some constant $0 < a < 1$. The root splits the set into $\Theta(n^{1-a})$ subsets. The children of the root have degree equal to $\Theta(n^{(1-a)a})$. An illustration of an IST can be found in Fig. 22.

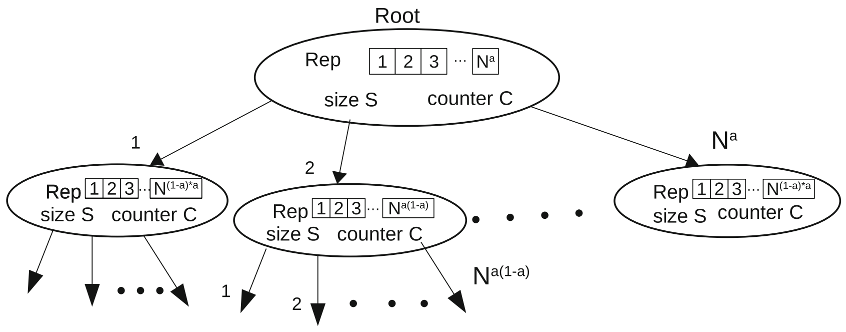


Fig. 22. Interpolation Search Tree

Each node u in the IST is associated with (i) a REP_u array, which contains a sample of the subset of elements that is stored below u; (ii) a variable S_u, which denotes the size of the subset; and (iii) a variable C_u, which counts how many insertions/deletions have been performed since the last rebuilding (of the IST) that involved u; cf. Fig. 22.

The idea is that on every partition of the RDD an IST is created that manages the elements of that same partition. The Spark's sorting function was used to sort and partition the elements to the worker nodes. After sorting is completed, each partition holds roughly the same number of sorted elements.

The next step is to create an IST on each partition. Instead of using the insertion algorithm to add the elements one by one in the tree, a bulk-insertion is used to insert all sorted elements in the tree, and globally rebuild it from the root.

This way, only one rebuilding is needed to create an ideal IST for each partition. Note that the IST object for the specific partition has to fit in the memory of each worker. However, this issue can be resolved since the input dataset can be split in a larger number of partitions if necessary.

2.7.2 Fault Tolerance

Spark operates on top of fault tolerant systems, like Hadoop Distributed File System (HDFS), making all the RDDs fault tolerant. Since RDDs are immutable, Spark keeps the lineage of the deterministic operations that were used on the input dataset to create it. If due to a worker node failure any partition is lost, then that partition can be recomputed to another worker node from the original dataset using the lineage of operations.

2.7.3 Queries

The classic search algorithm performs interpolation search in the REP array of every node of the tree, starting from the root, in order to locate the subset in which the search should be continued. The expected search time is $\mathcal{O}(\log\log(n))$.

In Spark, the search algorithm works as follows. Each partition is queried with they key that has to be searched. If the key is inside the partition's interval, the search algorithm is performed on the IST of the same partition returning true or false depending on whether the key exists in the structure or not. If the key is not inside the partition's interval, nothing is returned. Thus only one partition executes the search algorithm for each key that is queried.

The algorithm for a range query in the interval $[min, max]$ works in a similar manner. Each partition is queried with min and max, and all elements with keys between those values have to be returned.

Let x_i^F denote the first item of the i-th partition and let x_i^L denote the last item of the i-th partition. The following algorithm is concurrently executed in each partition i.

- If min is inside the i-th partition's interval $[x_i^F, x_i^L]$, then the search algorithm is performed and the corresponding element B is found.
    - If max is inside the partition's interval, then the search algorithm is performed again and the corresponding element E is found. All elements in-between B and E are returned.
    - Else if max isn't inside the partition's interval, then E is assigned to the last element of the partition (x_i^L). All elements in-between B and E are returned.
- Else if max is inside the i-th partition's interval, then the search algorithm is performed and the corresponding element E is found. B is assigned to the first element of the partition (x_i^F). All elements in-between B and E are returned.

– Else if $min < x_i^F$ and $max > x_i^L$, the whole partition is returned.
– Else if none of the above happens, zero is returned.

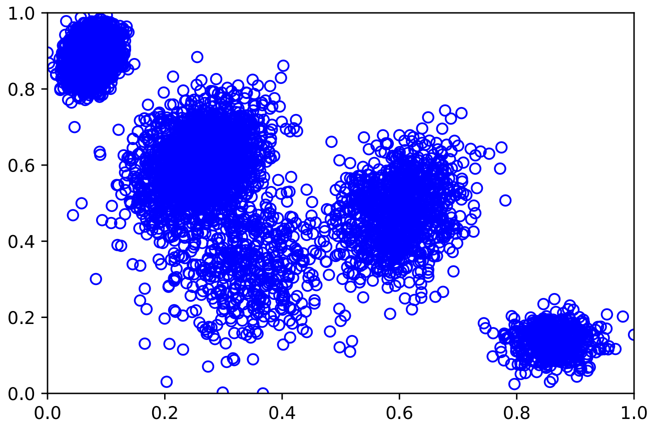


Fig. 23. Dataset distribution for SPIS

2.7.4 Experimental Evaluation

In [14], an experimental evaluation was conducted on a cluster with 32 physical computing machines running Hadoop 2.7 and Spark 2.1.0. Synthetic datasets were used for the experimentation with different cardinalities. The dataset contained one-dimensional values that were produced by a mixture of Gaussian distributions. The selection of this dataset was based in the fact that many real-world datasets contain clusters and are frequently modeled as Gaussian mixtures. Figure 23 presents such a distribution in the two-dimensional space. In the experiments, the projection in the x and y axis were used, in order to construct the one-dimensional dataset for the performance evaluation.

Two kinds of experiments were performed. First a runtime performance test, comparing three different algorithmic techniques, one using IST (A), and two using Spark's built-in features (F and M).

Technique A consists of the following steps.

– Create an RDD by referencing the dataset file. Map its contents to Float numbers. Sort and partition the RDD to the workers.
– Generate an array of 5000 random float pairs in the interval $[0, 1]$ to perform range search queries. The array is created at the Driver node and broadcasted to all workers in the cluster.
– Create an IST on each partition.
– Execute 5000 Range Queries on the IST of each partition using the pairs of the array as input parameters, and monitor the total runtime.

Technique M consists of the following steps.

- Create an RDD by referencing the dataset file. Map its contents to Float numbers. Sort and partition the RDD to the Workers.
- Generate an array of 5000 random float pairs in the interval $[0, 1]$ (to perform range search queries).
- Using *mapPartition* and *find* functions, perform 5000 range queries on the elements of each partition.

Technique F consists of the following steps.

- Create an RDD by referencing the dataset file. Map its content to Float numbers.
- Generate an array of 100 random float pairs in the interval $[0, 1]$ (to perform range search queries).
- Filter the input RDD using the elements of the array as bounds.

Since the number of queries is different, it is only logical to compare the elapsed time per query for all algorithmic techniques. The corresponding results are given in Table 1. The runtime results correspond to 32 Spark Workers.

Table 1. Runtime performance comparison.

Input size $\times 10^6$	Number of partitions	A (ms)	M (ms)	F (s)
10	64	7.88	13.46	2.58
20	64	8.74	19.58	5.22
100	128	12.2	83.12	2.46
200	128	14.5	125.04	4.28
1000	512	56.54	507.14	15.10

We observe that algorithmic technique A is significantly faster than Spark's built-in techniques F and M. The difference is more evident for bigger datasets.

The second set of experiments was carried out in order to test the scalability of the proposed organization scheme. Using an input dataset of ten million float numbers, 5000 range queries were performed on the IST while gradually adding more Workers to the cluster.

The total runtime also includes sorting time. Sorting is performed by three Workers (note that the file is stored in three partitions in the HDFS) before being split across the cluster. This is the reason behind the significant improvement in the first three tests. After that, the runtime is steadily decreases which shows the good scalability of the proposed approach; see Fig. 24.

3 Decentralized Ring-Based Structures

3.1 P-Ring

P-Ring [4] is implemented in the context of a modular framework that identifies and separates the different functional components of an overlay index structure.

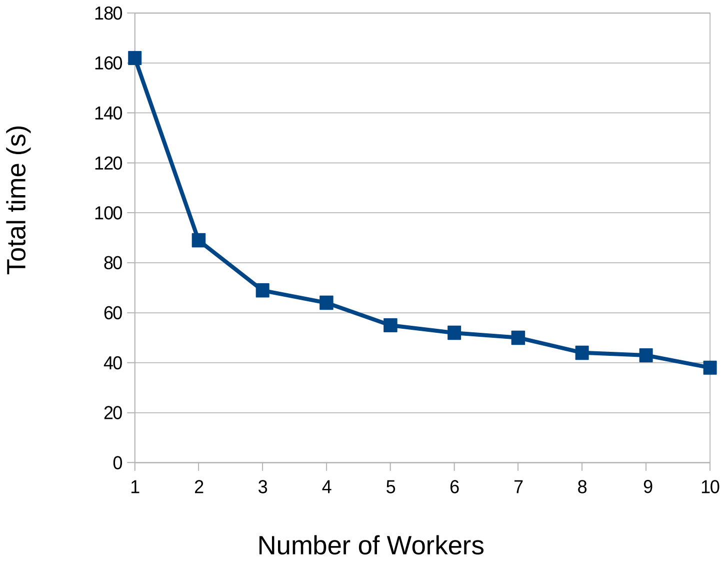


Fig. 24. Scalability performance

3.1.1 Structure

The P-Ring consists of the following four levels.

Fault Tolerant Ring: The Fault Tolerant Ring connects the nodes in the system along a ring, and provides reliable connectivity among these nodes even in the case of failures. For a node p, $succ(p)$ (respectively, $pred(p)$) denotes the node adjacent to p in a clockwise (resp., counter-clockwise) traversal of the ring. The Ring provides methods to get the address of the successor or predecessor, insert a new successor, join the ring or leave the ring (of course, a node can just fail). The Ring also generates events such as *newSuccessor*, and *newPredecessor-Value* that can be caught by higher layers and processed either synchronously or asynchronously.

Data Store: The Data Store, built on top of the Fault Tolerant Ring, is responsible for distributing the items to nodes. Ideally, the distribution should be uniform so that each node stores about the same number of items. The Data Store provides API methods to insert and delete items into and from the system.

Content Router: The Content Router, built on top of the Data Store, is responsible for efficiently routing messages to nodes that have items satisfying a given predicate.

Replication Manager: The Replication Manager, built on top of the Data Store, ensures that items assigned to a node are not lost if that node fails. The Replication Manager algorithms were used, where the items stored at a node are replicated by its successors in the ring.

P-Ring nodes are divided in owner nodes and helper nodes. Helper nodes are not assigned any items. The rest are called owner nodes. The helpers change over time and help with node joins/departures.

3.1.2 Load Balancing

The search key space is ordered on a ring, wrapping around the highest value. The Data Store partitions this ring space into ranges and assigns each of these ranges to a different node. The system is initiated with one owner node that owns the entire indexing domain. All other nodes join the system as helper nodes, and become owner nodes during load balancing.

Whenever the number of items in a node's p Data Store becomes larger than a bound u, an overflow occurs. Then, node p tries to *split* its assigned range and its items with a helper node.

Whenever the number of items in p's Data Store becomes smaller than a bound l, an underflow occurs. Then, p tries to acquire a larger range and more items from its successor in the ring. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a helper node.

Let now discuss in detail the basic operations when an overflow or an underflow occurs.

A node p that overflows executes a *split* operation. During a split, node p tries to find a helper p' and transfer half of its items, and the corresponding range to p', After p' is found, half of the items are removed from p and its range is split accordingly. Then, p invites p' to join the ring as its successor. Using the information received from p, p' initializes its index components and joins the ring.

If there is an underflow at node p, then a *merge and redistribution* is executed. Node p invokes the merge function on its successor in the ring. The successor sends back the action decided, *merge* or *redistribute*, a new range, and the list of items that are to be re-assigned to p. Then, p appends the new range and the new items to its own. The invoked node $p' = succ(p)$, checks whether a redistribution of items is possible between the two "siblings". If indeed, then it sends some of its items and the corresponding range to p. If a redistribution is not possible, then p' gives up all its items and its range to p, and becomes a helper node.

3.1.3 Fault Tolerance

Node failures and insertions as well as splits and merges at the Data Store level, disrupt the consistency of the Content Router. A simple *Stabilization Process* is executed on each node periodically that repairs the inconsistencies of Content Router. This process guarantees that the Content Router structure eventually becomes fully consistent as long as the nodes remain connected at the ring level.

3.1.4 Experimental Evaluation

To evaluate the load balancing of the system and show that the P-Ring achieves good load balance at low cost, a simulated environment and a real implementation were tested in [4].

Initially, 256 nodes were inserted, and no items. Then, items were randomly inserted/deleted in three phases: insert only, insert and delete, and delete only. In each phase 50000 operations are executed at the rate of 1 operation/second. Three different distributions for the items inserted were tested: uniform, Zipf 0.5, and Zipf 1. The domain is $[1, 65536]$. The items to be deleted are chosen uniformly at random from the existing items.

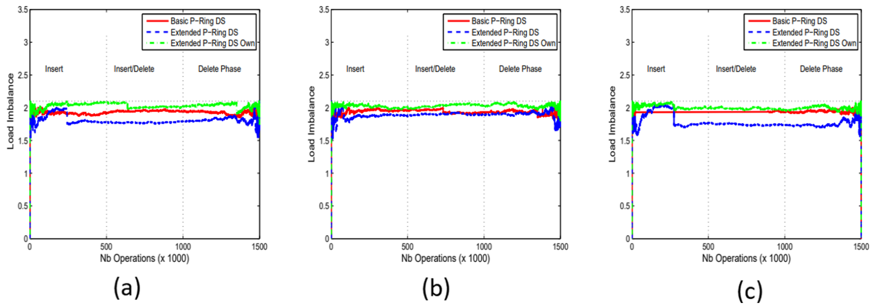


Fig. 25. P-Ring Imbalance (a) uniform, (b) Zipf 0.5, (c) Zipf 1

Figure 25 shows the imbalance measured every 60 simulated operations. The three subfigures are very similar, showing that regardless of the data skew, the system maintains its load balance. Next, the performance of the P-Ring Content Router is investigated, where the search cost (number of messages required to evaluate a range query, averaged over 100 random searches) is measured. The main variable component in the cost of range queries is finding the item with the smallest qualifying value, so only that cost is reported. P-Ring is compared to BATON*, Chord and Skip Graphs [1].

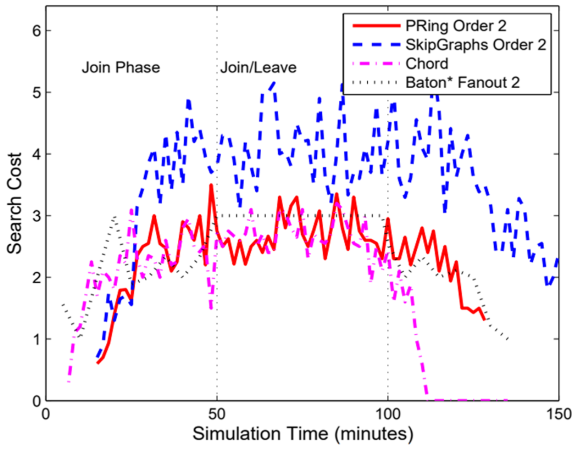


Fig. 26. Search performance

Figure 26 illustrates the search cost of P-Ring's Content Router, Skip Graphs, BATON* and Chord. It is clear that P-Ring's cost is lower than the cost of Skip Graphs and approximately equal to the cost of BATON* and Chord.

4 Comparison of Hierarchical Structures

In this Section we provide a comparison of the overlay structures presented in the previous sections. Table 2 demonstrates the complexities of the overlay structures for the operations of: Range Search, Insert/Delete Key, Maximum size of routing tables, and Join/Depart Node.

Table 2. Time complexities of structures' actions.

Structures	Range search	Insert/Delete key	Max size of routing table	Join/Depart node
BATON [9]	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\overline{\mathcal{O}}(\log N)$
BATON* [8]	$\mathcal{O}(\log_m N)$	$\overline{\mathcal{O}}(m \cdot \log_m N)$	$\mathcal{O}(m \cdot \log_m N)$	$\overline{\mathcal{O}}(m \cdot \log_m N)$
D^2-Tree [3]	$\mathcal{O}(\log N)$	$\tilde{\mathcal{O}}(\log N)$	$\mathcal{O}(\log N)$	$\tilde{\mathcal{O}}(\log N)$
D^3-Tree [16]	$\mathcal{O}(\log N)$	$\tilde{\mathcal{O}}(\log N)$	$\mathcal{O}(\log_m N)$	$\tilde{\mathcal{O}}(\log N)$
ART [18]	$\hat{\mathcal{O}}(\log_b^2 \log N)$	$\overline{\mathcal{O}}(m \cdot \log_m \log N)$	$\mathcal{O}(N^{1/4}/\log^c N)$	$\overline{\mathcal{O}}(m \cdot \log_m \log N)$
ART$^+$ [17]	$\hat{\mathcal{O}}(\log_b^2 \log N)$	$\tilde{\mathcal{O}}(\log \log N)$	$\mathcal{O}(N^{1/4}/\log^c N)$	$\tilde{\mathcal{O}}(\log \log N)$
P-Ring [4]	$\mathcal{O}(\log_d N)$	$\tilde{\mathcal{O}}(d \cdot \log_d N)$	$\mathcal{O}(\log N)$	$\tilde{\mathcal{O}}(d \cdot \log_d N)$
SPIS [14]	$\mathcal{O}(\log \log(n/N))$	$\tilde{\mathcal{O}}(1)$	$\mathcal{O}((n/N)^a)$	$\tilde{\mathcal{O}}(1)$

N: number of nodes; n: number of elements with ($N << n$); m: fanout;
d: order of the ring; a: constant $0 < a < 1$; $\widetilde{O}$: amortized bound; $\overline{O}$: expected amortized bound.

In the case of node failure in the SPIS structure, a replica of the respective partition is ready to be assigned to another worker, while in case of node join, a simple repartition of the data is performed.

We notice that the SPIS solution is the fastest when it comes to Insert/Delete Key and Join/Depart Node, since the actions on the RDDs (or Dataframes) partitions of Spark Cluster, occur mostly in memory and in bulk processing fashion. For this reason (bulk processing), the complexities of insert/delete key and join/departure node operations are amortized.

As Table 2 shows, all the structures have different complexities on every operation. This means that there is no clear answer on which structure is the best to use. It highly depends on the nature of the problem, the type of network, and the application at hand that determines which operations uses more than others.

5 Conclusions

In this work we focused on range query processing for big data. We presented and reviewed state-of-the-art hierarchical (and not DHT-based) distributed overlay structures for efficient big data management that exhibit stable scalability.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Baltimore, MD, pp. 384–393 (2003)
2. Barkai, D.: Technologies for sharing and collaborating on the net. In: 1st International Conference on Peer-to-Peer Computing (P2P 2001), 27–29 August 2001, Linköping, Sweden, pp. 13–28 (2001)
3. Brodal, G.S., Sioutas, S., Tsichlas, K., Zaroliagis, C.: D^2-tree: a new overlay with deterministic bounds. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010. LNCS, vol. 6507, pp. 1–12. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17514-5_1
4. Crainiceanu, A., Linga, P., Machanavajjhala, A., Gehrke, J., Shanmugasundaram, J.: Load balancing and range queries in P2P systems using P-Ring. ACM Trans. Internet Technol. **10**(4), 1–30 (2011)
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
6. Ding, G., Wang, L., Wu, Q.: Big data analytics in future internet of things. CoRR, abs/1311.4112 (2013)
7. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, 31 August–3 September 2004, pp. 444–455 (2004)
8. Jagadish, H.V., Ooi, B.C., Tan, K.-L., Vu, Q.H., Zhang, R.: Speeding up search in peer-to-peer networks with a multi-way tree structure. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, 27–29 June 2006, pp. 1–12 (2006)
9. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: BATON: a balanced tree structure for peer-to-peer networks. In: Proceedings of the 31st Conference on Very Large Databases (VLDB 2005), Trondheim, Norway, pp. 661–672 (2005)
10. Kaporis, A.C., Makris, C., Sioutas, S., Tsakalidis, A.K., Tsichlas, K., Zaroliagis, C.D.: Improved bounds for finger search on a RAM. Algorithmica **66**(2), 249–286 (2013)
11. Knuth, D.E.: The Art of Computer Programming, vol. III, 2nd edn. Addison-Wesley, Redwood City (1998)
12. 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: Ng, W.S., Ooi, B.-C., Ouksel, A.M., Sartori, C. (eds.) DBISP2P 2004. LNCS, vol. 3367, pp. 93–106. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31838-5_7
13. Mehlhorn, K., Tsakalidis, A.K.: Dynamic interpolation search. In: Automata, Languages and Programming, 12th Colloquium, Nafplion, Greece, 15–19 July 1985, Proceedings, pp. 424–434 (1985)
14. Papadopoulos, A.N., Sioutas, S., Zacharatos, S., Zaroliagis, C.: Efficient distributed range query processing in apache spark. In: Proceedings of 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing - CCGRID 2019, pp. 569–575. IEEE Computer Society (2019)
15. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: SIGCOMM, pp. 161–172 (2001)
16. Sioutas, S., Sourla, E., Tsichlas, K., Zaroliagis, C.: D^3-tree: a dynamic deterministic decentralized structure. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 989–1000. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_82

17. Sioutas, S., Sourla, E., Tsichlas, K., Zaroliagis, C.: ART$^+$: a fault-tolerant decentralized tree structure with ultimate sub-logarithmic efficiency. In: Karydis, I., Sioutas, S., Triantafillou, P., Tsoumakos, D. (eds.) ALGOCLOUD 2015. LNCS, vol. 9511, pp. 126–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29919-8_10
18. Sioutas, S., Triantafillou, P., Papaloukopoulos, G., Sakkopoulos, E., Tsichlas, K.: Art: Sub-logarithmic decentralized range query processing with probabilistic guarantees. J. Distrib. Parallel Databases (DAPD) **31**(1), 71–109 (2012)
19. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. **31**(4), 149–160 (2001)
20. White, T.: Hadoop: The Definitive Guide. O'Reilly (2015)
21. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)
22. Zhang, Y., Liu, L., Li, D., Liu, F., Lu, X.: DHT-based range query processing for web service discovery. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2009), Los Angeles, CA, pp. 477–484, IEEE, July 2009