LDT: a Logarithmic Distributed Search Tree

PANAYIOTIS BOZANIS
University of Thessaly, Greece

YANNIS MANOLOPOULOS
Aristotle University, Greece

Abstract

We propose LDT, a new Scalable Distributed Search Tree for the dictionary problem, as an alternative to both random trees and deterministic height balanced trees. Our scheme exhibits logarithmic update time, either constant or logarithmic search time for single key queries and output sensitive query time for range search query, depending whether one affords linear or $O(n \log n)$ space overhead.

Keywords

Distributed Data Structures, Scalability, Network Computing, Binary Search Trees, 1-*d* Range Query, Node Aggregation, Topology Trees.

1 Introduction

As network technology evolves, it provides more prevalent the technological framework known as *network computing*: fast networks interconnect many powerful and low-priced workstations, creating a pool of perhaps terabytes of RAM and even more of disc space [12]. Every site in such a network is either a *server* managing data or a *client* requesting access to data. Every server provides a storage space of *b* data elements, termed *bucket*, to accommodate a part of the file under maintenance. Sites communicate by sending and receiving *point-to-point* messages. The underlying network is assumed error-free; this way we can devote our concern to efficiency aspects only, i.e., on the number of the messages exchanged between the sites of the networks, irrespectively of the message length or the network topology.

The distributed algorithms and data structures in such an environment must be designed and implemented so that (a) they should expand to new servers gracefully, and only when servers already used are efficiently loaded; and (b) their access and maintenance operations never require atomic updates to multiple clients, while there is no centralized "access" site. A data structure that meets these constrains is named *Scalable Distributed Data Structure* (SDDS).

Since the seminal paper by Litwin et al. [12] introduced the model with the LH* data structure, there have been various kinds of SDDS proposals, namely RP* [13, 14], DRT [10], lazy k-d-Tree [16, 17], DEH [2, 18], BDST [3, 4, 5], DSL [1], and ADST [6, 7]. On the other hand, Kröll and Widmayer [11] showed that if all the hypotheses used to efficiently manage search structures in the single processor case are carried over to a distributed environment, then a tight lower bound of $\Omega(\sqrt{n})$ holds for the height of balanced search trees.

In this paper we introduce LDT, a new SDDS for the dictionary problem, based on carefully distributed random tree structure information across servers. This way we achieve the performance expected from a logarithmic tree height, without dependence on data distribution nor node rebalancing operations. Additionally, one can decide whether he affords auxiliary logarithmic or linear to the number of servers space per server according to performance criteria. In sections 2, 3 we briefly introduce SDDS and give the motivation of this work presenting previous solutions. Section 4 introduces LDT and discusses its performance, while Section 5 concludes our work.

2 Motivation

While hashing schemes have been examined first in the SDDS model, search tree structures are soon proposed, since they support nearest neighbor and range queries [1, 3, 4, 5, 10, 13]. Kröll and Widmayer [10] introduced random trees in a distributed environment as they adapt naturally to it. Their primary disadvantages (i.e., their dependence on data distribution and their unbounded height) lead to balanced distributed binary search trees [3, 4, 5], which guarantee worst case logarithmic height. However, their maintenance through rotations seems more complex and more centralized in a broader sense. This led to DSL [1], a SDDS for the dictionary problem, based on a version of Skip Lists, as an alternative to random search trees, exhibiting logarithmic height, kept with highly local criteria without dependence on data distribution.

In [6, 7] ADST were introduced as deterministic alternative binary search trees to random search trees, based on (a) node aggregation, so that logarithmic height is enforced; (b) tree node replication across servers, so that the total auxiliary information requires space $O(n \log n)$, as opposed to $O(n)$ space complexity of the underlying binary search tree; and (c) aggregated super-nodes v with worst-case out-degree$(v) \in [\sqrt{n}, n]$. In our opinion, ADST pointed out the need to seek for a distributed version of a data structure that combines the simplicity of a random tree with the bounded height of a balanced tree, and motivated us to introduce LDT. Table 1 summarizes previous works on one-dimensional tree-like dictionaries and our results.

Table 1: Summary of results. All update bounds are position given. $O(\cdot)$, $\overline{O}(\cdot)$ and $\widetilde{O}(\cdot)$ denote worst case, average and high probability bounds respectively, n is the number of servers, and d is the distance between guessed and actual position.

SCHEME	SEARCH TIME	UPDATE TIME	SPACE OVERHEAD PER SERVER
RP* [13]	$O(height)$	$O(height)$	$O(height)$
DRT [10]	$O(n)$ $\overline{O}(1)$	$O(n)$ $\overline{O}(1)$	$O(n)$
BDST [3]	$O(\log n)$	$O(\log n)$	$O(1)$
DSL [1]	$\widetilde{O}(\log n)$ $\overline{O}(\log d)$	$\widetilde{O}(1)$	$\widetilde{O}(1)$
ADST [6, 7]	$O(1)$	$O(\log n)$	$\Omega(\sqrt{n})$, $O(n)$
LDT	$O(1)$ $O(\log n)$	$O(\log n)$ $O(\log n)$	$O(n)$ $O(\log n)$

3 Random Search Trees

The main, underlying structure aimed for distribution in this work is a random, binary, leaf-oriented search tree. That is, a search tree where all data elements are stored in the leaves. Each leaf represents a block of data elements, while internal nodes store auxiliary information, necessary for guiding the basic tree operations, i.e., search, insert and delete. According to the standard approach in the SDDS model, one must distribute a portion of the main structure at each server, along with whatever auxiliary information dictates the distribution algorithm. On the other hand, a client maintains its own view of the search structure, initially coarse and partial, becoming better as it issues more and more requests.

We briefly overview the approach of [10]: Initially, their structure, named DRT, consists of bucket 1 belonging to server 1 (see fig. 1). Whenever 1 overflows, it splits in two; the new father node is assigned to bucket 1, which also knows the extent of the new bucket node 2. The latter is assigned to a new server with name 2. As long as insertions keep coming, bucket splits and new internal nodes are being generated.

Clients store a part of global tree. Actually, they are unaware of the changes occurring to the search tree T until they issue a new request. Whenever a client c wants to insert or to search[1] for a key k, it searches it in his local view of T—which, during the first access is confined to the address of server 1—in order to find the server s possibly accommodating the bucket holding k. Then it sends the request to server s. If server s is pertinent for the operation, i.e., its bucket contains k, then it performs the requested operation. In opposite case, s searches

[1] We follow the standard approach to SDDS context and we do not discuss deletions since they are symmetrical to insertions.

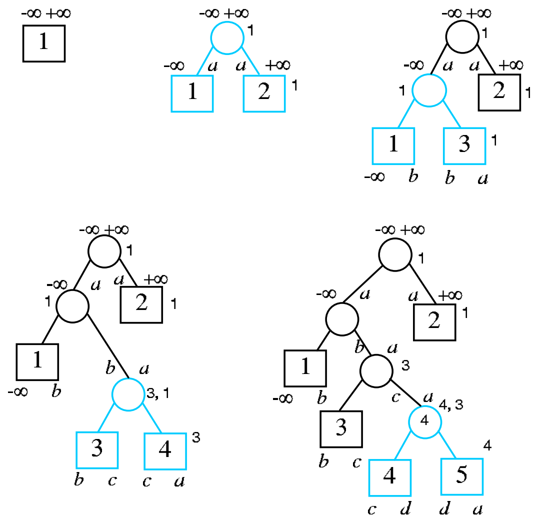

Figure 1: Instance of DRT evolution. Each node (leaf or internal) has the extent of values associated with it. Numbers beside nodes denote, which servers know their extent.

its own view of T to figure out the possibly pertinent server s' to whom it forwards the request. When s' receives the request, it either performs it or forwards it to another server s'' and so on, until the pertinent server s_p is found, which actually serves the requested operation. This forwarding request forms a chain of servers $C \equiv s \rightarrow s' \rightarrow \cdots \rightarrow s_{p-1} \rightarrow s_p$. The pertinent server s_p sends its local information to s_{p-1}. After updating its own view, the latter forwards correcting information to its predecessor in C and so on until s is updated which, additionally, sends the gathered information to client c in order to "refresh" its local view of T.

Since the evolving search tree T is a random one, it exhibits poor worst-case performance due to linear worst case height, and linear storage overhead since each node is stored in exactly two places. ADST [6, 7] deals with the problem of linear worst-case behavior by carefully distributing auxiliary local information to servers along paths leading to the root (fig. 2): each server knows the extents of the subtree T_v rooted at its first allocated node v. This rule guarantees the constant query cost of ADST, since the forwarding request chain consists of only 3 nodes in the worst case. Whenever a bucket split occurs, the respective server informs all the servers "owing" the nodes on the path towards the root about the undergoing changes to the node extents. However, since the randomness of T does not guarantee bounded height, ADST restricts the auxiliary information copying by *node aggregation*: nodes of T are grouped in supernodes or compound nodes and each group is assigned to a server. This process resembles to node aggregation taking

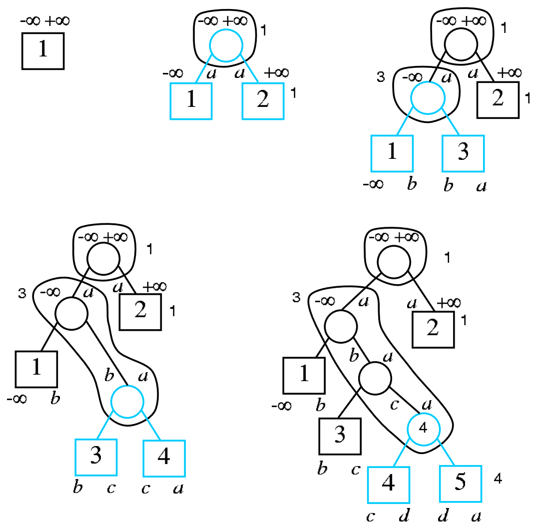

Figure 2: An instance of ADST. Servers 1 and 3 own the two supernodes of the aggregation procedure.

place in order to store binary search trees to secondary memory. ADST employs the following aggregation rule: each supernode v with father f_v consists of up to $2 \cdot l + 1$ simple tree nodes, where l is the number of simple nodes composing f_v. This simple rule guarantees that the number of supernodes along a leaf-to-root path is kept logarithmic to the number of servers ([7]), so that the update cost is logarithmic as well. ADST needs total $O(n \log n)$ space overhead and between $\sqrt{n}$ and n worst case space overhead per server.

4 Logarithmic Dictionary Tree (LDT)

4.1 Distribution

An alternative way to manipulate a random tree T with good operation performance is to build an auxiliary, logarithmic height search tree T' on top of it; Frederickson introduced the topology trees [8, 9] in the context of link-cut trees and dynamic expression trees: Given a binary tree $T = (V, E)$, a topology tree T' is a balanced tree constructed on top of the nodes of T by *node aggregation* or *clustering*. T' consists of multiple levels, with each level i containing a tree structure T_i on the nodes V_i of level i, which actually form a partition of original set V. To be specific, the nodes of V consist the level 0 nodes V_0 and also the leaves of T'. Further, we have that level 0 tree T_0 equals T. The nodes V_0 of level 0 are

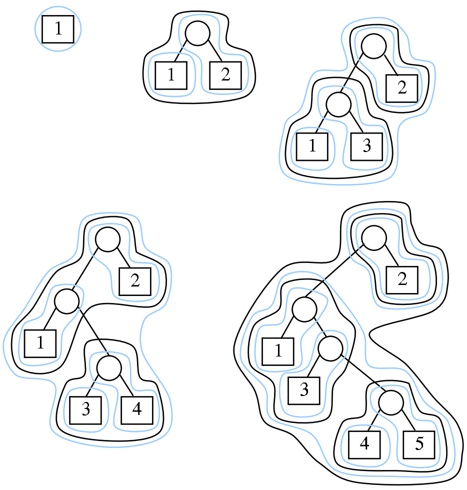

Figure 3: Instances of the clustering procedure.

clustered to form the compound nodes or clusters V_1 of level 1 and so a new level tree T_1 results if we maintain all edges of T that connect two clusters. These edges are termed *level edges*. The degree of a cluster of nodes is defined as the number of edges with exactly one endpoint in the cluster. From each node v of T_1 there are edges (pointers) to the nodes of T_0, which were aggregated to form v. We call these edges *cross edges*. Next, clustering is performed on the nodes of T_1 to form V_2 and T_2 of level 2 and so on until we come up with a level consisting of a single cluster node. The cluster nodes V_i and the cross edges between cluster nodes of consecutive levels form the topology tree T'. [8, 9] proposed the following clustering rules so that the resulting topology tree T' is of logarithmic height: (a) each cluster of degree 3 contains only one node; (b) each cluster of degree less than 3 contains at most two nodes; and (c) no two adjacent clusters—i.e., there is an edge with endpoints in both of them—can be combined and still satisfy rules (a) and (b). This clustering procedure is depicted in fig. 3. In the sequel, we will adapt this clustering policy in the context of SDDS model, so that we can efficiently distribute a random search tree T among a number of servers avoiding the linear worst case performance of the DRT proposal.

4.1.1 Servers

Following the approach of previous random structures, the leaf nodes represent buckets capable of holding up to b data items lying into the corresponding extent of the leaves. Internal nodes and leaves are associated with the server managing them. A node or a leaf is managed by only one server. In contrast, a server can

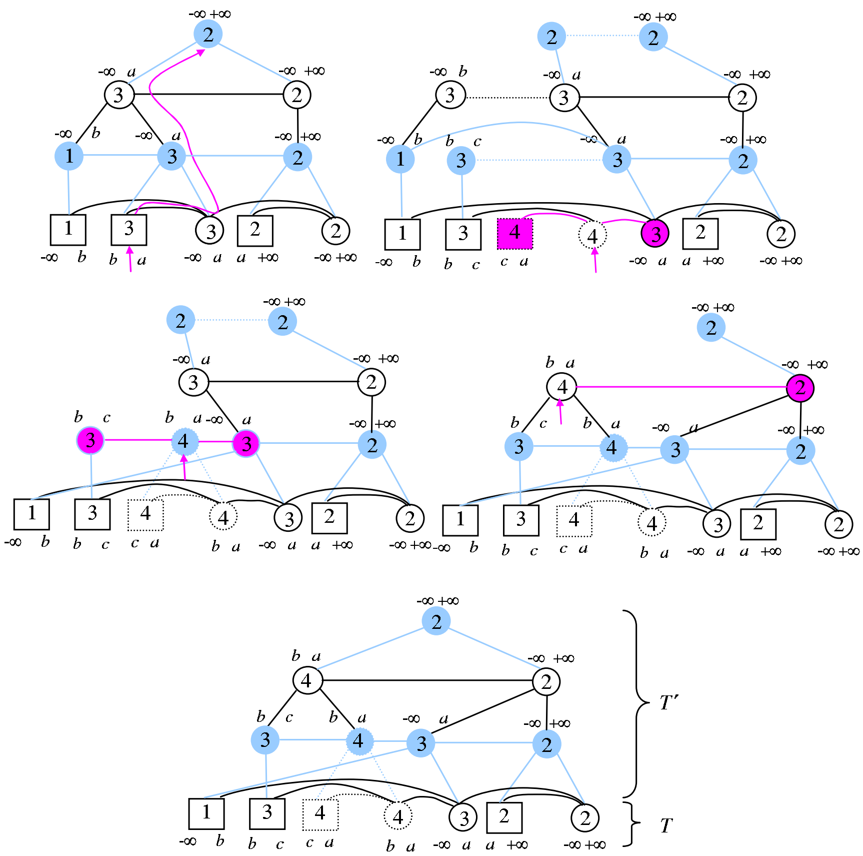

Figure 4: Bucket split in LDT and the evolution of the auxiliary tree T' it causes. Each node is denoted with its extent and the server to which is assigned.

hold several nodes, but only one leaf. Each node stores its extent (subinterval of values), parent and child pointers.

The main idea behind building the auxiliary structure of LDT is the following: Let (a,b) and (c,d) be two intervals on the line. We consider $(a,b) \succeq (c,d)$ if $(a,b) \supseteq (c,d)$ or $a \geq d$. Let S_t^T be the reverse topological sort of the nodes V of the original tree T according to relation $\succeq$ on their associated extents. We scan S_t^T and merge each unclustered internal node with the biggest—according to $\succeq$— of its unclustered children. This procedure is applied level by level, until we come up with just one node. Please, examine the bottommost tree of fig. 4, ignoring the numbers inside the nodes for the moment. Here, we must note that this scheme extends naturally the aforementioned procedure of constructing topology trees to fit in the case of binary search trees since, in the latter case, one has to respect the node extents and the dependencies stemming from the $\succeq$ relation and the need for efficient distribution among a set of servers.

Next, we describe the insertion algorithm of LDT. During the discussion, it will become apparent how the nodes of LDT are distributed among the servers: Firstly, we locate the server s owing the bucket b_s that must accommodate the new item x. This will be discussed in the next paragraph; thus, let us assume that s is found. If b_s has free space, then x is placed in b_s and the procedure stops. In the opposite case, b_s overflows and a bucket split must take place (fig. 4). That is, a new server s' must be included in the set of servers storing the file. s' will provide the additional bucket $b_{s'}$ for storing half of the contents of b_s. This means that node b_s of T must be replaced by a triple of nodes $(b_s, b_{s'}, v)$ such that $v \succeq b_{s'} \succeq b_s$ and v the common father of $b_s, b_{s'}$. v is also assigned to new participant server s'.

Afterwards, the change occurred to T must be reflected to the upper tree T'. For the sake of simplicity the update is described as a double pass procedure, but, non the less, it can be executed as a single step algorithm. During the first pass, we climb the path π towards the root of T' and break the clusters we meet. All these clusters are related to the overflowed bucket node b_s.

The second pass incorporates $s, b_s, b_{s'}$ into LDT by traversing π and remedying the level trees T_i with local merges according to the $\succeq$ relation: At the lowest level 0 we check whether the predecessor and the successor of v can be aggregated with the smallest, unclustered, reachable by a level edge, node. The extent of a resulting cluster c is defined as the biggest of the participating extents. The 'winning' extent also determines the server to which c is assigned. At the end of the check (i) we possibly recluster old clusters; and (ii) we introduce into level 1 a new cluster $c_{v,b_{s'}}$, which corresponds to v and $b_{s'}$. The procedure is repeated for $c_{v,b_{s'}}$ and level 1, and so on until we reach the top level consisting of a single cluster node. Figure 5 exhibits an example of LDT evolution due to consecutive insertions.

Lemma 1 *The insert operation costs $O(\log n)$ messages in the worst-case.*

Proof. The aggregation rules and the maximality of clustering guarantee the logarithmic height of T', since a bounded fraction of nodes 'survives' the clustering procedure as we move from level to level [8, 9]. At each level we check at most two cluster nodes for reclustering. Since each node is assigned to exactly one server, the number of messages exchanged per level is constant. And so, due to the logarithmic number of levels, the total cost is also logarithmic. □

The space complexity of LDT is considered in the next lemma.

Lemma 2 *LDT structure has linear to the number of servers space complexity. Also, the auxiliary per server space is logarithmic, in the worst case.*

Proof. The base tree T is binary and, thus, it has linear space complexity. T' has logarithmic height and its nodes are of degree at most 2. Therefore, it demands

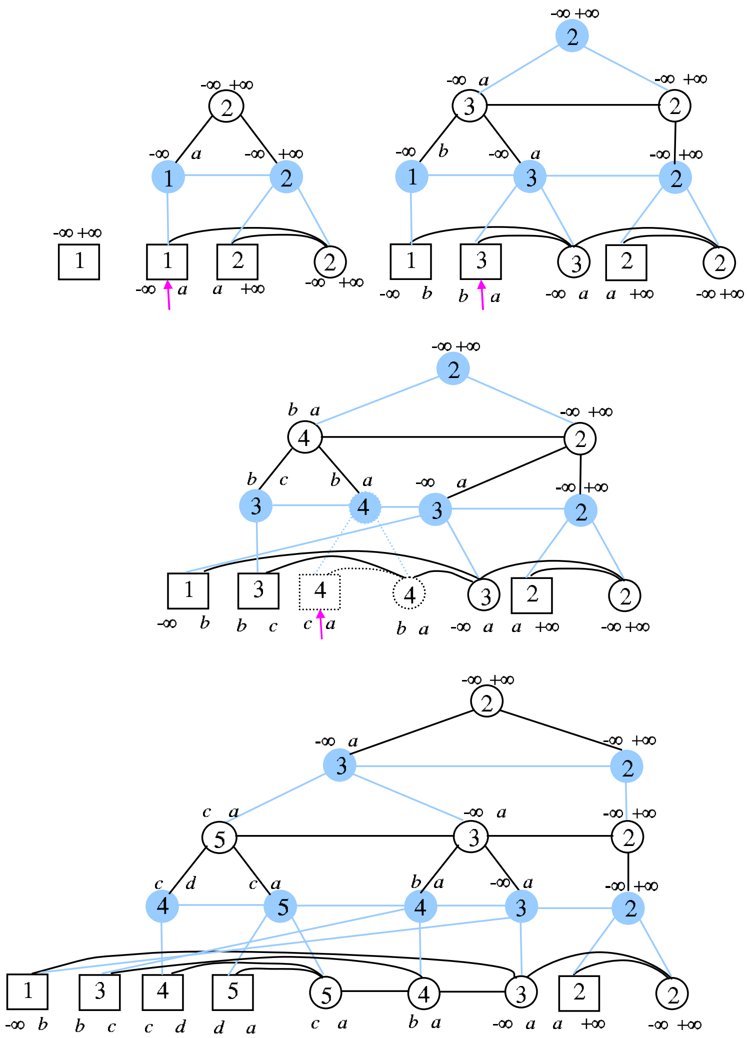

Figure 5: An example of LDT evolution under a sequence of insertions.

linear space. The second part of lemma follows also easily, since each server is assigned at most two nodes per level. □

4.1.2 Clients

As the SDDS model dictates, each client maintains local data reflecting its own view of the distributed data structure. Specifically, it keeps node associations and their extent as they were during the last time it accessed the data structure. Based on local view, it issues the desired operation to the most appropriate server. If it succeeded in its choice, it receives the answer. Otherwise, LDT routes the request to the pertinent server and informs back the client about the part of the data structure visited during the routing. The information is piggybacked in the answer, adopting the standard–in the SDDS context–approach, as it was briefly discussed in Section 3. For further details, please refer to [7, 10].

Next, we discuss the search operation for an item x. Let s be the server that received the client request. If s is pertinent, i.e., x belongs to the extent of bucket b_s, then it serves the query. In the opposite case, the algorithm repeatedly considers two cases until locating the pertinent server. Thus, let i denote the current level under consideration. Initially, $i = 0$. **Case I.** s does not own a cluster node in level $i+1$. Then it forwards the request to the server owning the cluster node at level $i+1$ that covers the node under consideration. This information is available due to cross edges. **Case II.** s owns cluster nodes in level $i+1$. Let c_s^{i+1} be the cluster node of s at level $i+1$ with the biggest extent. s checks the extents of the neighbors of c_s^{i+1}; there are two of them, at most. If a neighbor's extent contains x, then s forwards the query to the server s' owing it. s', based on its local information, can either answer the query or forward the request to a child node at level i. Otherwise, the procedure recurs to level $i+2$, following the cross edge.

Lemma 3 *The search operation costs $O(\log n)$ messages.*

Proof. The search operation accesses a constant number of servers at each level. Since there are $O(\log n)$ levels, the claimed bound follows. □

In case that 1-d range queries $r = [a,b]$ must be served, we can employ 'threaded' buckets. That is, buckets knowing their successor and predecessor bucket according to $\succeq$ relation. This information can be easily maintained during bucket splits. Then, in order to answer range queries, first, we locate the bucket containing the left border a of r, and, second, we access buckets following the threads to the right as long as we meet extents containing the right border b. So, we have:

Lemma 4 *The range search operation costs $O(\log n + k)$ messages, where k is the number of buckets holding query results.* □

If one can afford $O(n \log n)$ replication, or, equivalency, $O(n)$ auxiliary information per server, then the bounds of the above lemmas can be reduced to $O(1)$

and $O(k)$ respectively, employing standard replication procedures along the lines of [6, 7], and as briefly discussed in Section 3. In LDT context this means that, after a bucket b_s splits, the respective server s has to inform all the servers owing cluster nodes with extents covering the old extent of b_s. It can be easily seen that all these serves can be located along the b-to-the-root path and, thus, the update complexity remains the same. It follows that:

Lemma 5 *The search operation costs $O(1)$ messages, whereas the range search operation costs $O(k)$ messages, where k is the number of buckets holding query results.* □

5 Conclusions

In this paper we proposed LDT, a new SDDS for the dictionary problem, as an alternative to both RDTs and ADSTs. It seems that the simple operations it employs and the trade-offs it exhibits make our proposal appealing. Our future plans include experimental evaluation of our approach and extension to d dimensions employing the decomposition paradigm [15].

References

[1] P. Bozanis, Y. Manolopoulos. DSL: Accommodating Skip Lists in the SDDS Model. In *Proceedings of the 3rd Workshop on Distributed Data and Structures (WDAS'00)*, L'Aquila, Italy, Proceedings in Informatics, Vol. 9, pp. 1–9, Carleton Scientific, July 2000.

[2] R. Devine. Design and Implementation of DDH: Distributed Dynamic Hashing. In *Proceedings of the 4th International Conference on Foundations of Data Organization on Algorithms (FODO'93)*, Chicago, Illinois, LNCS, Vol. 730, pp. 101–114, Springer-Verlag, October 1993.

[3] A. di Pasquale, E. Nardelli. Balanced and Distributed Search Trees. In *Proceedings of the DIMACS Workshop on Distributed Data and Structures (WDAS'99)*, Princeton, NJ, Proceedings in Informatics, Carleton Scientific, May 1999.

[4] A. di Pasquale, E. Nardelli. Distributed Searching of k-dimensional Data With almost Constant Costs. In *Proceedings of the Conference on Advances in Databases and Information Systems (ADBIS'00)*, LNCS, Vol. 1884, pp. 239–250, Springer-Verlag, September 2000.

[5] A. di Pasquale, E. Nardelli. An Amortized Lower Bound for Distributed Searching of k-dimensional Data. In *Proceedings of the 3rd Workshop on Distributed Data and Structures (WDAS'00)*, L'Aquila, Italy, Proceedings in Informatics, Vol. 9, pp. 71–86, Carleton Scientific, July 2000.

[6] A. di Pasquale, E. Nardelli. A Very Efficient Order Preserving Scalable Distributed Data Structure. In *Proceedings of the 12th International Conference on Database and Expert Systems Applications (DEXA'01)*, Munich, Germany, LNCS, Vol. 2113, pp. 186–199, Springer-Verlag, September 2001.

[7] A. di Pasquale, E. Nardelli. ADST: An Order Preserving Scalable Distributed Data Structure with Constant Access Costs. In *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'01)*, Piestany, Slovak Republic, LNCS, Vol. 2234, pp. 211–222, Springer-Verlag, November-December 2001.

[8] G.N. Frederickson. Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees. *SIAM J. Comput.*, 26(2):484–538, April 1997.

[9] G.N. Frederickson. A Data Structure for Dynamically Maintaining Rooted Trees. *Journal of Algorithms*, 24(1):37–65, July 1997.

[10] B. Kröll, P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'94)*, Minneapolis, MN, pp. 265–276, May 1994.

[11] B. Kröll, P. Widmayer. Balanced Distributed Search Trees do not Exist. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS'95)*, Kingston, Ontario, Canada, LNCS, Vol. 955, pp. 50–61, Springer-Verlag, August 1995.

[12] W. Litwin, M. A. Neitmat, D. A. Schneider. LH*-Linear Hashing for Distributed Files. *ACM Transactions on Database Systems*, 21(4):480–525, December 1996.

[13] W. Litwin, M. A. Neitmat, D. A. Schneider. RP*-A Family of Ordered-Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago, Chile, pp. 342–353, September 1994.

[14] W. Litwin, D. A. Schneider. LH*$_{RS}$- A High Availability Scalable Distributed Data Structure using Reed Solomon Codes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'00)*, Dallas, TX, USA, pp. 237–248, May 2000.

[15] K. Mehlhorn. *Data Structures & Algorithms, Vol. 3: Multidimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, Heidelberg, 1984.

[16] E. Nardelli. Distributed k-d Trees. In *Proceedings of the XVI International Conference of the Chilean Computer Science Society (SCCC'96)*, Valdivia, Chile, pp. 142–154, November 1996.

[17] E. Nardelli, F. Barillari, M. Pepe. Distributed Searching of Multidimensional Data: a Performance Evaluation Study. *Journal of Parallel and Distributed Computation*, 49(1):111–134, March 1998.

[18] R. Vingralek, Y. Breitbart, G. Weikum. Distributed File Organization with Scalable Cost/-Performance. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'94)*, Minneapolis, MN, pp. 253–264, May 1994.

Panayiotis Bozanis is with the Department of Computer & Communication Engineering, University of Thessaly, Volos 382 21, Greece. E-mail: `pbozanis@inf.uth.gr`

Yannis Manolopoulos is with the Department of Informatics, Aristotle University, Thessaloniki 540 06, Greece. E-mail: `manolopo@csd.auth.gr`