Range Trees in the SDDS Model

Panayiotis Bozanis

Department of Computer & Communication Engineering
School of Engineering, University of Thessaly
38221 Volos, Greece
Email: pbozanis@inf.uth.gr

Abstract. 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. In such a network, every server provides a storage space to accommodate a part of the file under maintenance. In order to describe and develop efficiently algorithms and data structures in such a distributed environment, Litwin introduced the SDDS model, a scalable distributed paradigm. In this paper we examine the accommodation of range trees —well-known elegant multi-dimensional data structures for answering range queries— in the SDDS model. Our scheme demands $O(n \log n)$ expected number of servers, $O(n \log^2 n)$ expected auxiliary space, $O(\log^2 n)$ expected insertion time and $O(\log n + k)$ expected query time, where $n = \Theta(N/b)$, N the number of stored 2-dimensional data elements, k the number of buckets holding query results, and b the bucket capacity. Furthermore, we can trade-off increased expected query complexity of $O(\log^2 n + k)$ with lower expected auxiliary space of $O(n \log n)$. Finally, we discuss how our scheme can expand to d dimensions and respond to deletion requests.

1 Introduction

The last years, due to rapid evolution of network technology, the technological framework known as *network computing* emerged; that is, many powerful and low-priced workstations, interconnected over fast networks, create a pool of terabytes of RAM and even more of disc space [13]. This distributed storage, composed of RAM and disk devices, is provided for data file accommodation and maintenance. Every site in such a network is either a *server* managing data or a *client* requesting access to data. Additionally, a site can be either client or server or both. 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; in that 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 from the length of a message or the network topology.

This context calls out for the design and implementation of distributed algorithms and data structures 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 termed *Scalable Distributed Data Structure (SDDS)*. Here we must note that SDDS applies irrespectively of the storage medium; every site can become server as long as it provides storage locations either in RAM or in secondary storage.

Since Litwin *et al.* introduced the SDDS model with the LH* hashing data structure, in the seminal paper [13], there have been various kinds of SDDS proposals: (a) hashing schemes: DDH [5, 21], LH*$_G$ [16]; (b) one-dimensional search trees: DRT [11], RP* [14, 15], BDST [6–8], ADST [9, 10], LDT [3]; (c) multi-dimensional indexes: lazy k-d-Tree [17, 18], k-d Tree [4]; and (d) skip lists DSL [2]. On the other hand, Kröll and Widmayer [12] 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 propose a new scheme for accommodating distributed range trees, which are dynamic random versions of the well-known and thoroughly examined multi-dimensional data structure, in the SDDS model. In section 2 we briefly introduce to range-trees and basic SDDS concepts. Section 3 describes the scheme and discusses performance issues, coping, additionally, with its extension to d dimensions and deletion requests, while section 4 concludes our work.

2 Basic Structures

2.1 Range Tree

Range tree has been devised independently by a number of researchers [19, 1] as a data structure for storing d-dimensional point sets and answering range queries, i.e., report or count all points $p \in \mathbb{R}^d$ lying inside a multi-dimensional (hyper)rectangle $q = [x_1, y_1] \times [x_2, y_2] \times \cdots \times [x_d, y_d]$. In the following, we will consider the reporting case. Their definition is recursive, based on the decomposition data structuring paradigm:

The 1-dimensional case refers to a point set S lying on the line $\mathbb{R}$. Then the 1-dimensional range tree $\mathcal{T}^1$ on S is simply a leaf-oriented balanced binary search tree; that is, the points of S are stored in the leaves, from left to right, in order of occurrence on $\mathbb{R}$. Additionally, the leaves are linked to form a left-to-right singly linked node list. This permits answering 1-dimensional range queries $q = [x_1, y_1]$ in $O(\log n + k)$ output sensitive time, with n denoting the number of stored points and k being the size of output, in a very simple way: one locates, in $O(\log n)$ time, the left-most leaf accommodating a point $p \in S$ greater than or equal to x_1, and then, in $O(k)$ time, follows list pointers as long as leaves storing points less than or equal to y_1 are encountered.

A d-dimensional range tree $\mathcal{T}^d$ for a set of d-dimensional points S is built upon a "base" leaf-oriented balanced binary search tree T. T accommodates the members of S in its leaves according to the increasing order of their first $\mathbf{x_1}$ coordinate values. Every internal node v of T is associated to a $(d-1)$-dimensional range tree $\mathcal{T}_v^{d-1}$ which stores S_v^{d-1}; that is, all points of the subtree T_v rooted at v, after they are stripped of their first $\mathbf{x_1}$ coordinate. Now, a range reporting query $q = [x_1, y_1] \times [x_2, y_2] \times \cdots \times [x_d, y_d]$ can

be answered as follows: Let π_{x_1}, π_{y_1} be, respectively, the search paths for x_1 and y_1 in T, and w be the lowest node belonging to both π_{x_1}, π_{y_1}. Let also $\mu_1, \mu_2, \ldots$ $(\nu_1, \nu_2, \ldots)$ be the right (left) sons of the nodes on π_{x_1} (π_{y_1}), from w down to leaf node. Then $S^{d-1}_{\mu_i}$'s and $S^{d-1}_{\nu_j}$'s contain exactly those points of S which have their first coordinate lying inside $[x_1, y_1]$. So, in that way, the initial query q on $\mathcal{T}^d$ reduces to posing the query $q' = [x_2, y_2] \times \cdots \times [x_d, y_d]$ to $\mathcal{T}^{d-1}_{\mu_i}$'s and $\mathcal{T}^{d-1}_{\nu_j}$'s. In Figure 1(a) a range query on a 2-dimensional set of 8 points is presented, while the vertical lines denote the space decomposition resulting from the base tree. Figure 1(b) depicts the respective range tree and the involved μ_i and ν_j nodes. For shake of clarity, we only present the "secondary" structure $\mathcal{T}^1_x$ associated with node x.

The above definition can be —in a bit complicated manner— maintained under insertions and deletions of points. In order to insert a new point $p = (x_1, x_2, \ldots, x_d)$ in $\mathcal{T}^d$, one, firstly, inserts a new leaf l_p in T using x_1, and, secondly, on every $\mathcal{T}^{d-1}_v$ of each node v on the insertion path π_p, from the root of T to l_p, inserts p, recursively, using the $(d-1)$-dimensional point $p' = (x_2, \ldots, x_d)$. The deletion operation is defined symmetrically to insertion. Employing periodical local rebuilding of parts of the tree to keep it balanced [22], it can be proved that:

Theorem 1. *Let S be a set of n d-dimensional points. Using a dynamic range tree, with space complexity $O(n \log^{d-1} n)$, range queries on S can be answered in $O(\log^d n + k)$ time, k the output size, while an update operation can be performed in $O(\log^d n)$ either amortized or worst-case time.*

2.2 Distributed Random Trees

One of the base "building block" structures used in this work is the random, leaf-oriented, binary search tree. That is, a search tree such that 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 [11] as a short introduction to SDDS concepts: Initially, the structure, called DRT, consists of bucket 1 belonging to server 1 (see fig. 2). 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 the global tree T. Actually, they are unaware of the changes occurring to T until they issue a new request. Whenever a client c wants to insert or to search for[1] a key x, it searches it in his local view of T —which, during the first

[1] We follow the standard approach to SDDS context and we do not discuss deletions since (a) they are symmetrical to insertions and (b) they are scarce operations.

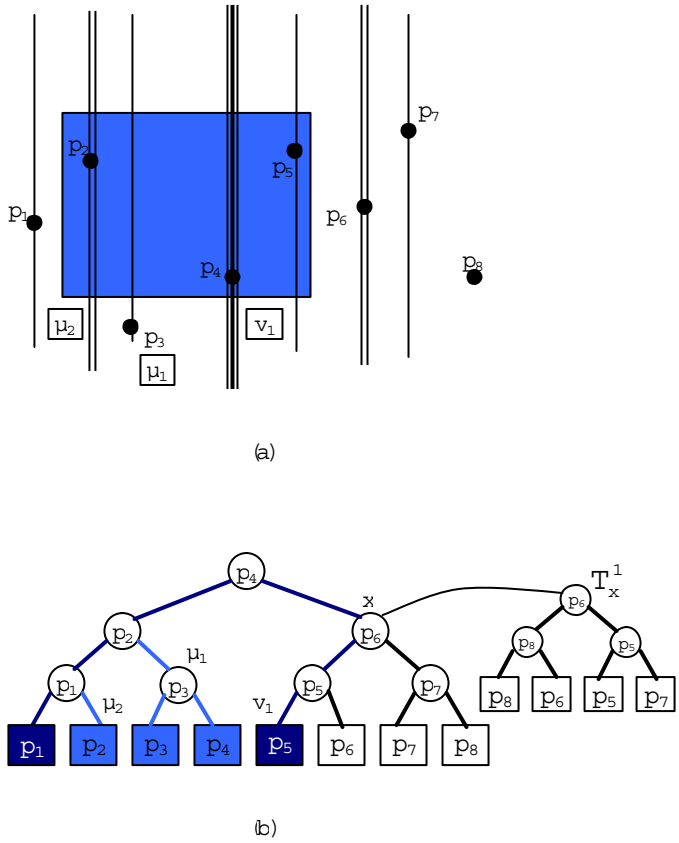

Fig. 1. An instance of pseudo 2-dimensional range tree: (a) range query, (b) data structure

access, is confined to the address of server 1— in order to find the server s possibly accommodating the bucket holding x. Then it sends the request to server s. If s is pertinent, i.e., its bucket contains x, then it performs the requested operation. In opposite case, s searches 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}, which, after updating its own view, 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.

Logarithmic height can be enforced by periodical reorganization balancing [19]. However, if one adjusts this technique to a distributed environment, the bounds he achieves are only amortized. As a matter of fact, there would be time periods during which the underlying network would experience linear to the number of servers

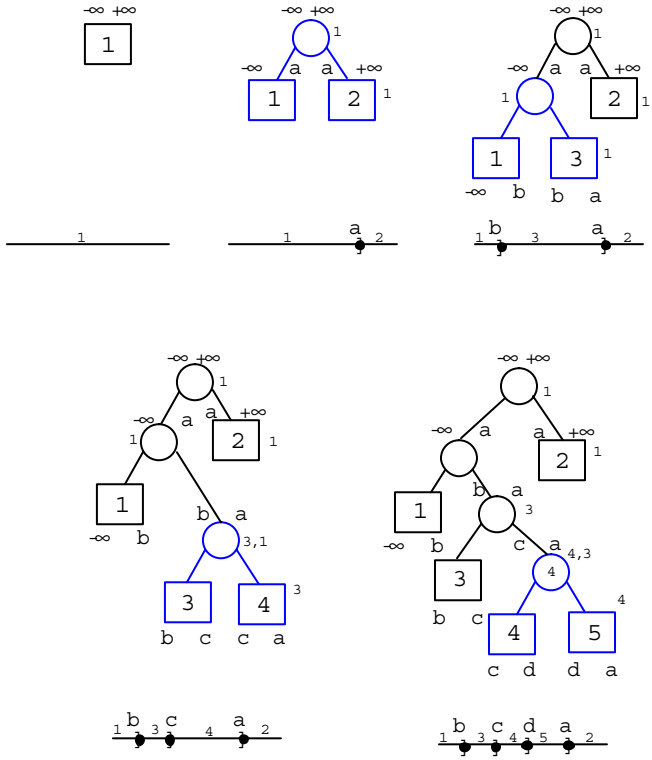

Fig. 2. Instance of DRT evolution. Each node (leaf or internal) has the extent of values associated with it. Numbers beside nodes denote which servers know their extent.

reorganization messages. There are two alternative ways to cope with balancing: The first approach [6], capitalizing on extra father pointers, adjusts the rotation balancing technique in the SDDS context. The second treatment is based on the observation that one can manipulate a random tree T by carefully restricting which servers receive auxiliary routing information; this can be achieved by either introducing super-nodes with exponential to the super-level fan-out size [9,10] or building a logarithmic height auxiliary search tree T' on top of it [3]. In this work, we use the latter structure, named LDT, which exhibits the following performance:

Theorem 2 ([3]). *Let S be a set of N 1-dimensional points. S can be stored in a LDT structure, using $n = \Theta(N/b)$ servers (b the bucket capacity), so that the search operation costs $O(1)$ messages, the range search operation costs $O(k)$ messages, and an update operation has $O(\log n)$ time, in case that one can afford $O(n \log n)$ replication information. If, however, only $O(n)$ replication information is accepted, the search operation costs $O(\log n)$ messages, the range search operation costs $O(\log n + k)$ messages, and an update operation has $O(\log n)$ time. All bounds are worst-case, and k denotes the number of buckets holding query results.*

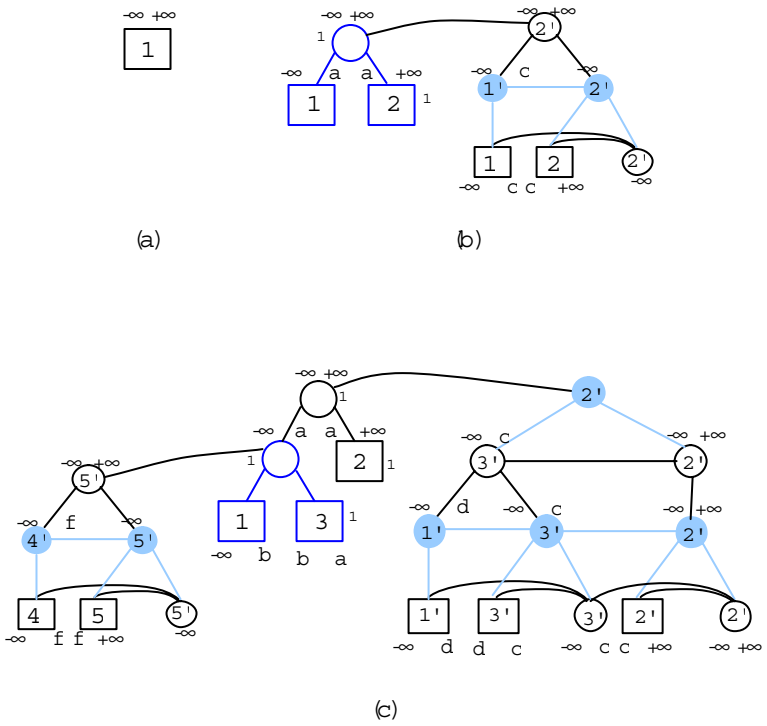

Fig. 3. (a)-(c) Snapshots of distributed range tree evolution under random insertions. Primed numbers refer to servers allocated to $\mathcal{T}_2$ structures

3 Distributing Range Trees

In the paragraphs that follow, we will restrict the discussion to the 2-dimensional case since d-dimensional structures are built recursively on top of it. The main obstacle of distributing dynamic range trees in a SDDS environment is the complexity of the scheme in [22]; even the amortized bounds are based on local rebuilding of parts of the tree, a fact which means that there will be time periods with linear to the number of servers rebuilding messages across the network, resulting in clients experiencing high delay times. This observation indicates the need to maintain the "base" binary tree unaltered as it evolves due to insertions of new points.

3.1 Servers

For the data accommodation we will employ a version of the random tree of [11] and the logarithmic distributed tree of [3]. The first structure will serve as the first level structure $\mathcal{T}_1$ —defined on **x**-axis— while the second one will be based on **y**-axis and considered as the second-level structure $\mathcal{T}_2$.

Following the approach of the SDDS tree structures, each leaf node of $\mathcal{T}_1$ represents a bucket capable of holding up to b data items. Every such item lies in the **x**-extent,

i.e., a continuous subinterval of **x**-axis values, associated with the leaf node. 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 hold several nodes, but only one leaf. Each node v stores its extent, that is, the subinterval of **x**-axis defined by the corresponding set S_v^2, parent, child pointers and, additionally, a pointer to a $\mathcal{T}_2$ root node.

Initially, the range tree consists of bucket 1 belonging to server 1 (please, consult fig. 3). Whenever 1 overflows, it splits in two; the new father node f 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. Also, a $\mathcal{T}_2^f$ structure is initialized and related to f — and, thus, server 1 is aware of it. From this moment, every time a new insertion of a data element $\delta = (x, y)$ is taking place either to bucket 1 or 2, it will trigger a second insertion, w.r.t. **y**-axis, to the $\mathcal{T}_2^f$ structure. As long as insertions keep coming, bucket splits and new internal nodes (and thus new $\mathcal{T}_2$ structures) are being generated. Please note that, during range tree deployment, the associations of $\mathcal{T}_1$ nodes —and thus $\mathcal{T}_1$ servers— with $\mathcal{T}_2$'s structures are stable, i.e., remain the same, all the time.

The above configuration has the following allocation cost:

Lemma 1. *Let T be a distributed range tree constructed by N random insertions. Then (a) the expected number of servers ν_T accommodating T is $O(n \log n)$; and (b) the expected auxiliary information α_T, in the form of pointers and node associations stored in the servers, is $O(n \log^2 n)$, $n = \Theta(N/b)$.*

Proof. (a) The number of servers necessary for accommodating $\mathcal{T}_1$ is clearly bounded by n. Since the insertions are random, this means that the root node ρ of $\mathcal{T}_1$, has i servers on its left subtree $\mathcal{T}_1^l$ and $n - i$ servers on its right subtree $\mathcal{T}_1^r$ with probability $1/n$. This gives the following recurrence:

$$\nu_T(n) = \frac{1}{n} \sum_{i=1}^{n-1} \big(\nu_T(i) + \nu_T(n-i) + n\big) \tag{1}$$

which solves to $O(n \log n)$.

(b) The $\mathcal{T}_1$ structure results in $O(n)$ auxiliary information while a $\mathcal{T}_2$ one occupying k servers needs $O(k \log k)$ of such information (theorem 2). These facts mean that:

$$\alpha_T(n) = \frac{1}{n} \sum_{i=1}^{n-1} \big(\alpha_T(i) + \alpha_T(n-i) + cn \log n\big), \quad c \text{ constant} \tag{2}$$

and so the claim follows.

Next, we describe the insertion algorithm: Firstly, we locate the server s owing the bucket b_s that must accommodate the new item $\delta = (x, y)$. This will be the topic of the next paragraph, so let us assume that s is found. If b_s has free space then δ is placed in b_s. In opposite case, b_s overflows and a bucket split must take place. That is, a new server s' must be included in the set of servers storing the tree. s' will provide the additional bucket $b_{s'}$ for storing half the contents of b_s. This means that the leaf node

b_s of $\mathcal{T}_1$ must be replaced by a triple of nodes $(b_s, b_{s'}, v)$ such that v is the common father of $b_s, b_{s'}$. v is assigned to a new participant server s' and associated with a new (elementary) $\mathcal{T}_2$ structure, which, in constant time, is initialized with the point set stored in b_s and $b_{s'}$.

Afterwards, δ must be inserted to all $\mathcal{T}_2$ structures associated with nodes lying on the path π_δ that leads from the leaf bucket storing δ to the root node of $\mathcal{T}_1$. This can be easily accomplished since parent pointers, and, thus, server associations, are maintained. A single insertion to a $\mathcal{T}_2$ structure, distributed across k servers, costs $O(\log k)$ messages in the worst-case (theorem 2.) Since $\mathcal{T}_1$ resulted under a sequence of random insertions, the expected total insertion cost $\iota_T(n)$:

$$\iota_T(n) = \frac{1}{n}\sum_{i=1}^{n-1}\big(\iota_T(i) + \iota_T(n-i) + c'n\log n\big), \quad c' \text{ constant}, \tag{3}$$

summing the respective costs of inserting in the **x**-axis extents of the $\mathcal{T}_1$ buckets, is bounded by $O(n\log^2 n)$. Which means that:

Lemma 2. *An insertion to a distributed range tree storing N points has an expected cost of $O(\log^2 n)$ messages, $n = \Theta(N/b)$.*

3.2 Clients

As the SDDS dictates, each client should maintain local data reflecting its own view of the distributed data structure. So, every client, willing to access the data structure, keeps node-servers associations and their **x**-axis extent for the "base" $\mathcal{T}_1$ structure as they were during the last time it accessed the range tree. Based on its local view, it issues the desired operation to the most appropriate server. In case of valid local information, the request will be served without any correction messages. Otherwise, the scheme forwards the request to the pertinent(s) server(s) and informs back the client about the part of the data structure visited during the routing. The information is piggybacked in the answer, adopting standard —in the SDDS context— approaches (for example, cf. [11, 17, 18].)

Insertion. Let $\delta = (x, y)$ be the data element the client wishes to insert in the range tree. Using its local information, the client determines the pertinent server; that is, the server s owning the bucket b_s with **x**-axis extent e_s such that $x \in e_s$. In case of valid local information, s will accommodate δ in its bucket and then will initiate the insertion of δ in the appropriate $\mathcal{T}_2$ structures as described in Section 3.1. Otherwise, s will trigger a search for the pertinent server s', which will, actually, manage the accommodation of δ. Then the client will receive the acknowledgement and the traversed portion of $\mathcal{T}_1$, during the search for s', in the form of piggybacked correction information. Using the latter, it can refine the private index it maintains, employing the techniques, e.g., of [11].

Search Operation. Let $q = [x_1, y_1] \times [x_2, y_2]$ be the range query the client wishes to issue. It searches in its local data for the server s owing the internal node v_s of $\mathcal{T}_1$ with the smallest **x**-axis extent e_{v_s} such that $[x_1, y_1] \subseteq e_{v_s}$. In case of valid local information, v_s will be the lowest level $\mathcal{T}_1$ node belonging to both the search paths π_{x_1}, π_{y_1} for x_1 and y_1, respectively. Then s, using the child pointers of v_s it owns, will initiate a downward traversal of π_{x_1} and π_{y_1}, posing the 1-dimensional query $q' = [x_2, y_2]$ to the right and left child node $\mathcal{T}_2$ structures that are met, respectively. The latter structures can answer such queries purely output-sensitively, i.e., with linear to number of servers accommodating parts of the output set messages (theorem 2.)

In case of invalid local information, because of $\mathcal{T}_1$ deployment, s will have a "successor" server s' owing the descendant node $v_{s'}$ of v_s with the desired property of being the lowest level node such that $[x_1, y_1] \subseteq e_{v_{s'}}$. s' can be easily found with a simple downward search due to child pointers and **x**-axis extent auxiliary information maintained in the participating servers. When s' is reached, the procedure continues as described previously. Finally, the client receives the answer and the traversed portion of $\mathcal{T}_1$ during the seek for s' in the form of piggybacked correction information, in order to refine the private index it maintains.

Using a recurrence equation describing the expected total external path length of $\mathcal{T}_1$, which is the same as eq. (1) of lemma 1(a), it results that $\mathcal{T}_1$ has expected $O(\log n)$ external path length. So, we have that:

Lemma 3. *A range query costs expected $O(\log n + k)$ messages, with k being the number of buckets holding query results.*

As a matter of fact if one cannot spend expected $O(n \log^2 n)$ replication information, he could use as $\mathcal{T}_2$ structures, the "cheap" LDT version [3] which demands linear to the number m of servers auxiliary information and exhibits worst-case $O(\log m + k)$ range query time. Then:

Lemma 4. *Given a set of N 2-dimensional points, there exists a range tree structure, distributed to expected $O(n \log n)$ servers, which demands expected $O(n \log n)$ auxiliary information and exhibits expected $O(\log^2 n)$ insertion time and expected $O(\log^2 n + k)$ range query time. ($n = \Theta(N/b)$ and k is the number of buckets holding query results.)*

3.3 Extensions

The 2-dimensional scheme that we have just described can be used as a "building block" to recursively construct d-dimensional distributed range trees $\mathcal{T}^d$, $d \geq 3$: We employ a "base" $\mathcal{T}_1$ structure to store the point set according to the first coordinate $\mathbf{x_1}$. Every internal node v of $\mathcal{T}_1$ will be associated to a $(d-1)$-dimensional distributed range tree $\mathcal{T}_v^{d-1}$ built on S_v^{d-1}. Then, we have:

Theorem 3. *Let S be a set of N d-dimensional points. S can be stored in a $\mathcal{T}^d$ structure so that range queries on S can be answered in expected $O(\log^{d-1} n + k)$ time, an insertion can be served in expected $O(\log^d n + k)$ time using expected $O(n \log^d n)$ servers and expected $O(n \log^{d+1} n)$ auxiliary information, where $n = \Theta(N/b)$ and b denotes the bucket capacity.*

Closing the section we must note that our scheme can also work in a fully dynamic environment; that is, when not only insertions but also deletions can occur. In this case, which is not consider in the majority of the SDDS proposals, one can show that for the base 2-dimensional case the expected update time is $O(\sqrt{n}\log n)$, the expected query time is $O(\sqrt{n}+k)$ while the expected number of servers is $O(n^{1.5})$. These bounds result from the fact that the expected $\mathcal{T}_1$ height can be proved to be $O(\sqrt{n})$ following the techniques in [20].

4 Conclusions

In this paper we proposed a scheme for distributing range trees in the SDDS context. Our proposal employs as simple operations as possible and exhibits design level tradeoffs. The experimental evaluation of our scheme is left for future consideration. We also left open the investigation whether it is possible one to design a fully dynamic distributed range tree with polylogarithmic update time without using the —rather expensive in a distributed environment— technique of local rebuilding [19, 22].

References

1. Bentley J.R.: Decomposable Searching Problems. *Information Processing Letters*, 8:244-251 (1979)
2. Bozanis P., Manolopoulos Y.: DSL: Accommodating Skip Lists in the SDDS Model. *Proceedings Workshop on Distributed Data and Structures (WDAS)*, LAquila, Italy (2000) 1-9
3. Bozanis P., Manolopoulos Y.: LDT: A Logarithmic Distributed Search Tree. *Proceedings Workshop on Distributed Data and Structures (WDAS)*, Paris, France (2002)
4. Bozanis P.: Accommodating k-d Trees in the SDDS Model. *Proceedings Workshop on Distributed Data and Structures (WDAS)*, Thessalonki, Greece (2003)
5. Devine R.: Design and Implementation of DDH: Distributed Dynamic Hashing. *Proceedings Foundations of Data Organization on Algorithms (FODO'93)*, Springer LNCS, Vol.730, Chicago, IL (1993) 101-114
6. di Pasquale A., Nardelli E.: Fully Dynamic Balanced and Distributed Search Trees with Logarithmic Costs. *Proceedings Workshop on Distributed Data and Structures (WDAS)*, (1999)
7. di Pasquale A., Nardelli E.: Distributed Searching of k-dimensional Data With almost Constant Costs. *Proceedings ADBIS Conference (ADBIS-DASFAA'00)*, Springer LNCS Vol.1884, Prague, Czech Republic, (2000) 239-250
8. di Pasquale A., Nardelli E.: An Amortized Lower Bound for Distributed Searching of k-dimensional Data. *Proceedings Workshop on Distributed Data and Structures (WDAS)*, LAquila, Italy (2000)
9. di Pasquale A., Nardelli E.: A Very Efficient Order Preserving Scalable Distributed Data Structure. *Proceedings DEXA Conference*, Springer LNCS, Vol.2113. Munich, Germany (2001) 186-199
10. di Pasquale A., Nardelli E.: ADST: An Order Preserving Scalable Distributed Data Structure with Constant Access Costs. *Proceedings SOFSEM Conference*, Springer LNCS, Vol.2234. Piestany, Slovak Republic (2001) 211-222

11. Kröll B., Widmayer P.: Distributing a Search Tree Among a Growing Number of Processors. *Proceedings ACM SIGMOD Conference*, Minneapolis, MN (1994) 265-276
12. Kröll B., Widmayer P.: Balanced Distributed Search Trees do not Exist. *Proceedings 4th Workshop on Algorithms and Data Structures*, Springer LNCS, Vol.955. Kingston, Ontario, (1995) 50-61
13. Litwin W., Neitmat M.A., Schneider D.A.: LH*-Linear Hashing for Distributed Files. *ACM Transactions on Database Systems*, 21:480-525, (1996)
14. Litwin W., Neitmat M.A., Schneider D.A.: RP*-A Family of Ordered-Preserving Scalable Distributed Data Structures. *Proceedings 20th VLDB Conference*, Santiago, Chile, (1994) 342-353
15. Litwin W., Schneider D.A.: RP*$_{RS}$— A High Availability Scalable Distributed Data Structure using Reed Solomon Codes. *Proceedings ACM SIGMOD Conference*, Dallas, TX, (2000) 237-248
16. Litwin W., Risch T.: LH*$_G$— A High-Availability Scalable Distributed Data Structure by Record Grouping. *IEEE Transaction on Knowledge Data Engineering*, 14:923-927 (2002)
17. Nardelli E.: Distributed k-d Trees. *Proceedings XVI Conference Chilean Computer Science Society (SCCC)*, Valdivia, Chile (1996) 142-154
18. Nardelli E., Barillari F., Pepe M.: Distributed Searching of Multidimensional Data: a Performance Evaluation Study. *Journal Parallel Distributed Communications*, 49:111-134 (1998)
19. Overmars M.H.: *The Design of Dynamic Data Structures*, Springer, (1987)
20. Sedgewick R., Flajolet Ph.: *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading, MA (1996)
21. Vingralek R., Breitbart Y., Weikum G.: Distributed File Organization with Scalable Cost/Performance. *Proceedings ACM SIGMOD Conference*, Minneapolis, MN, (1994) 253-264
22. Willard D.E., Lueker G.S.: Adding Range Restriction Capability to Dynamic Data Structures. *Journal of the ACM*, 32:597-617 (1985)