DSL: Accommodating Skip Lists in the SDDS Model

PANAYIOTIS BOZANIS
University of Thessaly & DELab, Greece

YIANNIS MANOLOPOULOS
Aristotle University, Greece

Abstract

We propose DSL, a new Scalable Distributed Data Structure for the dictionary problem, based on a version of Skip Lists, as an alternative to both random trees and deterministic height balanced trees. Our scheme exhibits, with high probability, logarithmic search time, constant reconstruction time, and linear space overhead. Additionally, at the expense of two additional pointers per internal node, the search operation could cost $O(\log d)$ expected messages, where d is the distance between guessed and actual position.

Keywords

Distributed Data Structures, Scalability, Network Computing, Skip Lists.

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 [7]. Every site in such a network manages data—so is a *server*—or requests access to data and it is specified as a *client*. Every server provides a storage space of b data elements, termed *bucket* to accommodate a part of the file under maintenance. Each client ignores the presence of other clients. Sites communicate by sending and receiving *point-to-point* messages. The underlying network is assumed error-free; in that way we devote our concern to efficiency aspects only, as it is widely advocated. More specifically, we concentrate 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) should expand to new servers gracefully, and only when servers already used are efficiently loaded; and (b) their access and

Table 1: Summary of results. The data structures are one-dimensional tree-like dictionaries and the 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
RP* [8]	$O(height)$	$O(height)$	$O(n)$
DRT [5]	$O(n)$ $\overline{O}(1)$	$O(n)$ $\overline{O}(1)$	$O(n)$
RBST [12]	$O(\log^2 n)$	$O(\log n)$	$O(n)$
BDST [2]	$O(\log n)$	$O(\log n)$	$O(n)$
DSL	$\widetilde{O}(\log n)$ $\overline{O}(\log d)$	$\widetilde{O}(1)$	$\widetilde{O}(n)$

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. [7] introducing the model with the LH* data structure, there have been various kinds of SDDS proposals, namely R-P* [8], DRT [5], lazy k-d-Tree [11, 13], DEH [1, 15], RBST [12], BDST [2, 3, 4]. On the other hand, Kröll and Widmayer [6] 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 DSL, a new SDDS for the dictionary problem, based on a version of Skip Lists, as an alternative to random search trees. Its logarithmic height, kept with highly local criteria without dependence on data distribution, makes this effort quite appealing. Table 1 summarizes our results. In section 2 we briefly describe Skip Lists and an alternative way of viewing them. Section 3 introduces DSL and discusses its performance, while section 4 concludes our work.

2 Skip Lists

In this section we briefly discuss Skip Lists [14]. More details can be found in the cited reference. Let S be a set of n elements $s_1 < s_2 < \cdots < s_n$ drawn from a totally ordered universe U. Let also $\mp\infty$ denote the smallest and biggest elements of U, respectively. We would like to build a dynamic search structure $\mathcal{S}$ on S with operations `search(`x`)`, `range_search(`$[x,x']$`)`, `insert(`x`)` and `delete(`x`)`. Assume we are given a biased coin with probability of success—the bias—$p \in (0,1)$. Starting with S, we form a sequence σ of subsets of S, of

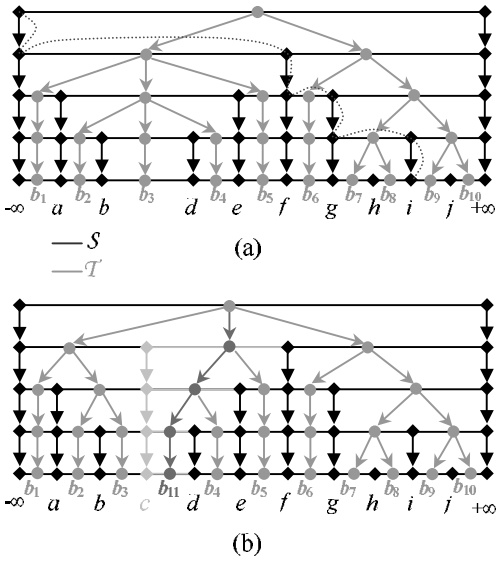

Figure 1: An instance of a Skip List $\mathcal{S}$ and its equivalent tree $\mathcal{T}$. The dotted line denotes a search path. Scheme (b) shows the effect of an insertion.

level l:

$$S = S_1 \supseteq S_2 \supseteq \cdots \supseteq S_{l-1} \supset S_l = \emptyset$$

Each S_i is derived from S_{i-1} by flipping the coin independently for each $s \in S_{i-1}$ and collecting all those elements with a success outcome. Then $\mathcal{S}$ is built as following (see fig. 1). We store each S_i in a sorted linked list $\mathcal{L}_i$. Each s in $\mathcal{L}_i$ stores a descent pointer to its occurrence in $\mathcal{L}_{i-1}$. `Search(`x`)` proceeds like this: Commencing with level l, in each $\mathcal{L}_i$ we locate the maximum element $s \leq x$. We follow the associated descent pointer to level i–1 and so on, until we reach level 1. In case of `range_search(`$[x,x']$`)` after we `search(`x`)`, we move to the right until we find some element bigger than x'.

When we want to `insert(`x`)`, we toss the biased coin until we fail. Let k be the number of successes. After we `search(`x`)` (see fig. 1), we know the element $s_{\mathcal{L}_i}$ that precedes x in every ordered list $\mathcal{L}_i$. So we add x to $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{k+1}$ after $s_{\mathcal{L}_1}, s_{\mathcal{L}_2}, \ldots, s_{\mathcal{L}_{k+1}}$ respectively. The case of `delete(`x`)` is quite simple; we just remove x from every list $\mathcal{L}_i$ containing it.

Besides their simplicity, Skip Lists are popular since they exhibit very good performance. That is, the number of levels l and the search cost is $O(\log_{1/p} n)$ with high probability, the occupied space is $O(n)$ also with high probability, whereas, if s_j, s_{j+1} are two consecutive elements in $\mathcal{L}_i$, then the expected number of elements

between them in $\mathcal{L}_{i-1}$ is $O(1)$. Finally, each element is stored in expected $O(1)$ levels.

There is another view of Skip Lists which we will use in this paper (see for example [10]). Sets S_i partition U in $|S_i| + 1$ intervals (consult fig. 1). Every such interval I belonging to a level i is divided into a number of child subintervals in level i–1. If we associate with each interval a node and connect with an arc each node to its child nodes, then a tree $\mathcal{T}$ results. $\mathcal{T}$ has the following properties: (a) All leaves are at the same depth; (b) Height and storage are $O(\log_{1/p} n)$ and $O(n)$ respectively with high probability; (c) Every node has expected $O(1)$ number of children; (d) An insertion/deletion causes expected $O(1)$ number of reconstructions (splits/merges of nodes).

3 Distributed Skip Lists (DSL)

3.1 Motivation

Search trees are proposed in the SDDS model, since they support nearest neighbor and range queries [2, 3, 4, 5, 8, 11]. Kröll and Widmayer [5] 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 binary search trees [2, 3, 4] which guarantee worst case logarithmic height. However, their maintenance through rotations seems more complex and more centralized in a broader sense. This, in our opinion, leads to seeking for a distributed version of a data structure that combines the simplicity of a random tree with the bounded height of a balanced tree. We choose Skip Lists because (a) they are random structures which respond to input data distribution with their "own" randomness; and (b) their balance criteria are "local", namely split/merges which change the extent of a node only with respect to its initial value.

3.2 Distribution

We follow the standard approach in SDDS model: a portion of the global structure is stored at each server. A client maintains its own view of the structure, initially coarse and partial, becoming better as it issues more and more requests.

Servers. 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 associated to only one server. In contrast, a server can hold several nodes, but only one leaf. Each node stores its extent (subinterval of values), parent and child pointers.

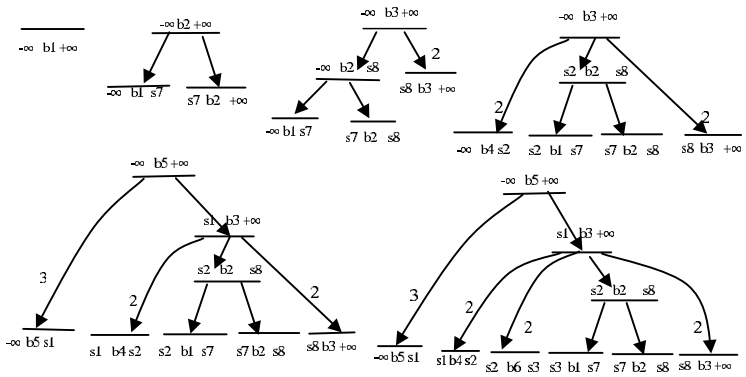

Figure 2: Evolution of the DSL. Each node is denoted with its extent (interval) and bucket/server it belongs. Arc numbers indicate # of similar nodes linearly linked.

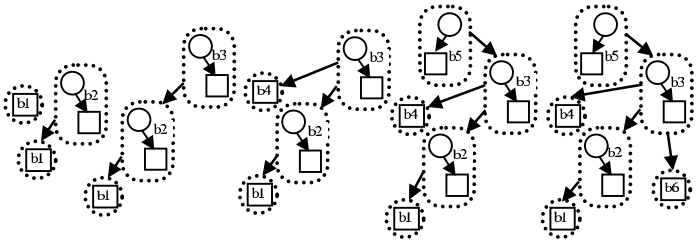

Figure 3: Bucket-server interconnection.

Initially, DSL consists of a bucket b_1 belonging to server s_1 (see fig. 2). Whenever b_1 overflows, it splits in two; the median value is inserted using the Skip List algorithm. The new bucket b_2 and the new root node are assigned to a new server s_2. As long as insertions keep coming, we have bucket splits and correspondence of the new leaf/bucket along with the nodes it introduces to a new server in such a way that unnecessary pointer (re)assignments are avoided. For example, consult figure 2, where the new bucket-server b_5 is associated to the left subinterval so that the path $(s_2, s_8) \rightarrow (s_2, s_7)$ remains unchanged.

Deletions are treated analogously: we just merge the underflow buckets and those nodes on the path towards the root that the underlying Skip List algorithm dictates. So, in the rest of the paper we will no longer refer to deletions. In figure 3 we present figure 2 in another equivalent yet more illuminating way. The dotted ovals denote servers. Inside every oval the internal DSL nodes assigned to the corresponding server are shown. Arcs between ovals correspond to server interconnection.

We close this subsection with four remarks: (a) Balancing after an insertion or a deletion leading to overflow or underflow is a local random decision due to the rationale underlying Skip Lists; (b) Each server is "charged" only expected $O(1)$ internal nodes, each one having expected $O(1)$ fan out. This means that, besides the bucket capacity, each server occupies expected $O(1)$ additional space and it is connected to expected $O(1)$ other servers; (c) The path from the root of DSL to a bucket involves $O(\log_{1/p} n)$ servers with high probability; (d) The extent of a node decreases or increases due to splits or merges, but never exceeds the value it had at the time of its birth.

Clients. As the SDDS 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, DSL 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 standard–in the SDDS context–approaches (for example, cf. [2, 5]).

3.3 Search Operation

The client searches in his local data for the most appropriate server, i.e., the server owing the bucket whose extent contains the data element. When a bucket-server receives a search request that can be served, it answers to the client appropriately. Otherwise, it checks the portion of DSL it owns to forward the demand down to a child server or up to a father server along with its local information. Sooner or later the client receives the answer, and the traversed portion of DSL in the form of piggybacked correction information. In this way, a client can adjust the private index it maintains. After the discussion in previous sections it is easy to see that

Lemma 1 *The search operation costs $O(\log_{1/p} n)$ messages with high probability.* □

In case that level pointers between internal neighboring nodes of same height are afforded, then

Lemma 2 *The search operation costs $O(\log_{1/p} d)$ expected messages, where d is the number of buckets between the initial guess and the bucket actually containing the data element.*

Proof. Every server has the additional ability to forward the request to one of its sibling servers whenever that is possible due to the assigned extent. So, during the searching process, the highest node in our structure we reach is the lowest

common ancestor $lca(b_i, b_a)$ of the initially guessed bucket b_i and the bucket b_a actually containing the item. The lemma follows since $lca(b_i, b_a)$ has $O(\log_{1/p} d)$ expected height (cf. [10]). □

3.4 Insertion, Deletion

After the appropriate search, operation continues in the way described in section 3.2. It follows promptly that

Lemma 3 *Given the position (bucket) of the involved item, an insertion or a deletion costs expected $O(1)$ reconstruction operations and therefore messages.* □

The above lemma substantially states that the logarithmic cost is paid only once during the search operation, as opposed to previous approaches (consult table 1 on page 2).

4 Conclusions

In this paper we proposed DSL, a new SDDS for the dictionary problem, based on a version of Skip Lists, as an alternative to random search trees. We feel that its logarithmic height, kept with highly local criteria without dependence on data distribution, makes our treatment quite appealing. Our future plans include experimental evaluation of our approach and extension to d dimensions employing the decomposition paradigm (see, for example, [9]).

References

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

[2] 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.

[3] A. di Pasquale, E. Nardelli. Fully Dynamic Balanced and Distributed Search Trees With Logarithmic Costs. *Technical Report, Dipartmento di Matematica Pura ed Applicata, Università di L'Aquila*, Via Vetoio, Coppito, I-67010 L'Aquila, Italy, August 1999.

[4] A. di Pasquale, E. Nardelli. Improving Search Time in Balanced and Distributed Search Trees. *Technical Report, Dipartmento di Matematica Pura ed Applicata, Università di L'Aquila*, Via Vetoio, Coppito, I-67010 L'Aquila, Italy, August 1999.

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

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

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

[8] 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.

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

[10] K. Mulmuley. *Computational Geometry: An introduction Through Randomized Algorithms*. Prentice-Hall, Eanglewood Cliffs, NJ, 1994.

[11] E. Nardelli. Distributed k-d Trees. In *Proceedings of the XVI Intl. Conference of the Chilean Computer Science Society (SCCC'96)*, (M. V. Zelkowitz et al., Eds.), Valdivia, Chile, pp. 142–154, November 1996.

[12] E. Nardelli, F. Barillari, M. Pepe. Fully Dynamic Distributed Search Trees can be Balanced in $O(\log^2 n)$ Time. *Technical Report, Dipartmento di Matematica Pura ed Applicata, Università di L'Aquila*, Via Vetoio, Coppito, I-67010 L'Aquila, Italy, July 1997. Accepted for publication *Journal of Parallel and Distributed Computation (JPDC)*

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

[14] W. Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM (CACM)*, 33(6):668–676, June 1990.

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

Panayiotis Bozanis is with the Department of Computer Engineering, Telecommunications & Networks, University of Thessaly, Argonafton & Filellinon, Volos 382 21, Greece, and with the Data Engineering Lab (DELab), Department of Informatics, Aristotle University, 540 06 Thessaloniki, Greece. E-mail: pbozanis@inf.uth.gr, bop@delab.csd.auth.gr

Yiannis Manolopoulos is with the Data Engineering Lab (DELab), Department of Informatics, Aristotle University, 540 06 Thessaloniki, Greece. E-mail: manolopo@delab.csd.auth.gr