ADST: An Order Preserving Scalable Distributed Data Structure with Constant Access Costs

Adriano Di Pasquale[1] and Enrico Nardelli[1,2]

[1] Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila,
Via Vetoio, Coppito, I-67010 L'Aquila, Italia.
{dipasqua,nardelli}@univaq.it

[2] Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche,
Viale Manzoni 30, I-00185 Roma, Italia.

Abstract. Scalable Distributed Data Structures (SDDS) are access methods specifically designed to satisfy the high performance requirements of a distributed computing environment made up by a collection of computers connected through a high speed network. In this paper we propose an order preserving SDDS with a worst-case constant cost for exact-search queries and a worst-case logarithmic cost for update queries. Since our technique preserves the ordering between keys, it is also able to answer to range search queries with an optimal worst-case cost of $O(k)$ messages, where k is the number of servers covering the query range. Moreover, our structure has an amortized almost constant cost for any single-key query.

Hence, our proposal is the first solution combining the advantages of the constant worst-case access cost featured by hashing techniques (e.g. LH*) and of the optimal worst-case cost for range queries featured by order preserving techniques (e.g., RP* and DRT). Furthermore, recent proposals for ensuring high-availability to an SDDS can be easily combined with our basic technique. Therefore our solution is a theoretical achievement potentially attractive for network servers requiring both a fast response time and a high reliability.

Finally, our scheme can be easily generalized to manage k-dimensional points, while maintaining the same costs of the 1-dimensional case.

Keywords: scalable distributed data structure, message passing environment, multi-dimensional search.

1 Introduction

With the striking advances of communication technology, distributed computing environments become more and more relevant. This is particularly true for the technological framework known as *network computing*: a fast network interconnecting many powerful and low-priced workstations, creating a pool of perhaps terabytes of RAM and even more of disc space. This is a computing environment very apt to manage large amount of data and to provide high performances. In fact, the large amount of RAM collectively available combined with the speed of the network allow the so-called *RAM data management*, which can deliver performances not reachable using standard secondary memory.

A general paradigm to develop access methods in such distributed environments was proposed by Litwin, Neimat and Schneider [9]: *Scalable Distributed Data Structures* (SDDSs). The main goal of an access method based on the SDDS paradigm is the management of very large amount of data implementing efficiently standard operations (i.e. inserts, deletions, exact searches, range searches, etc.) and aiming at *scalability*, i.e. the capacity of the structure to keep the same level of performances while the number of managed objects changes and to avoid any form of bottleneck. In particular, a typical distributed structure made up by a set of data server and a unique server directory cannot be considered an SDDS.

The main measure of performance for a given operation in the SDDS paradigm is the number of point-to-point messages exchanged by the sites of the network to perform the operation. Neither the length of the path followed in the network by a message nor its size are relevant in the SDDS context. Note that, some variants of SDDS admit the use of multicast to perform range query.

There are several SDDS proposals in the literature: defining structures based on hashing techniques [3,9,12,15,16], on order preserving techniques [1,2,4,8,10], or for multi-dimensional data management techniques [11,14], and many others.

LH* [9] is the first SDDS that achieves worst-case constant cost for exact searches and insertions, namely 4 messages. It is based on the popular linear hashing technique. However, like other hashing schemes, while it achieves good performance for single-key operations, range searches are not performed efficiently. The same is true for any operation executed by means of a scan involving all the servers in the network.

On the contrary, order preserving structures achieve good performances for range searches and a reasonably low (i.e. logarithmic), but not constant, worst-case cost for single key operations. Among order preserving SDDSs, we recall RP*s [10], based on the B$^+$-tree technique and BDST [4], based on balanced binary search tree. Both these structures achieves logarithmic costs for single key operations in the worst-case. Structures in the DRT family [5,8] can guarantee only a linear bound in the worst-case, but provide very good performances in the amortized case [5]. Finally, Distributed B$^+$-tree [2] is the first order preserving structures with constant exact search worst-case cost, but at the price of a linear worst-case cost for insertion.

Here we further develop the technique presented in [2] with the major objective to keep logarithmic the worst-case cost of insertions. This allows to obtain the following results: (i) worst-case constant cost for exact searches and insertions that do not causes splits, namely 4 messages; (ii) worst-case logarithmic cost for insertions that causes splits; (iii) amortized almost constant cost for any single-key operations.

Therefore, this is the first order preserving SDDS proposal achieving single-key performances comparable with the LH*, while continuing to provide the good worst-case complexity for range searches typical of order preserving access methods, like RP* and DRT.

Our structure is also able to support deletions: these are not explicitly considered in previous proposals in the literature, but for BDST [4] and, to some degree, for LH* [12]. Moreover, the technique used in our access method can be applied to the distributed k-d tree [14], an SDDS for managing k-dimensional data, with similar results.

2 ADST

We now introduce our proposal for a distributed search tree, that can be seen as a variant of the systematic correction technique presented in [2]. We first present the basic technique and then discuss our variation.

Each server manages a unique *bucket* of keys. The bucket has a fixed capacity b. We define a server "to be in overflow" or "to go in overflow" when it manages b keys and one more key is assigned to it. When a server s goes in overflow it starts the *split* operation. After a split, s manages $\frac{b}{2}$ keys and $\frac{b}{2}+1$ keys are sent to a new server s_{new}. It is easy to prove the following property:

Lemma 1. *Let σ be a sequence of m intermixed insertions and exact searches. Then we may have at most $\left\lfloor \frac{m}{A} \right\rfloor$ splits, where $A = \frac{b}{2}$.*

Moreover, clients and servers have a local indexing structure, called *local tree*. This is needed to avoid clients and servers to make address errors. From a logical point of view the local tree is an incomplete collection of associations $\langle$*server*, *interval of keys*$\rangle$: for example, an association $\langle s, I(s)\rangle$ identifies a server s and the managed interval of keys $I(s)$.

For further details on buckets and local trees management see [2,8].

Let us consider a split of a server s with a new server s'. Given the leaf f associated to s, a split conceptually creates a new leaf f' and a new internal node v, father of the two leaves. This virtual node is associated to s or to s'. Which one is chosen is not important: we assume to associate it always with the new server, in this case s'. s stores s' in the list l of servers in the path from the leaf associated to itself and the root. s' initializes its corresponding list l' with a copy of the s' one (s' included).

Moreover if this was the first split of s, then s identifies s' as its *basic server* and stores it in a specific field. Please note that the interval $I(v)$ now corresponds to the *basic interval* of s.

After the split s sends a correction message containing the information about the split to s' and to the other servers in l. Each server receiving the message corrects its local tree. Each list l of a server s corresponds to the path from the leaf associated with s to the root.

This technique ensures that a server s_v associated to a node v knows the exact partition of the interval $I(v)$ of v and the exact associations of elements of the partition and servers managing them. In other words the local tree of s_v contains all the associations $\langle s', I(s')\rangle$ identifying the partition of $I(v)$. Please note that in this case $I(v)$ corresponds to $I(lt(s_v))$.

This allows s_v to forward a request for a key belonging to $I(v)$ (i.e. a request for which s_v is logically pertinent) directly to the right server, without following the tree structure. In this distributed tree, rotations are not applied, then the association between a server and its basic server never changes.

Suppose a server s receives a requests for a key k. If it is pertinent for the requests ($k \in I(s)$) then it performs the request and answers to the client. Otherwise if it is logically pertinent for the requests ($k \in I(lt(s))$) then it finds in its local tree $lt(s)$ the pertinent server and forwards it the requests. Otherwise it forwards the requests to its basic server s'. We recall that $I(lt(s'))$ corresponds to the basic interval of s, then, as stated before, if the request for k is arrived to s, k has to belong to this interval. Then s' is certainly logically pertinent.

Therefore a request can be managed with at most 2 address errors and 4 messages.

The main idea of our proposal is to keep the path between any leaf and the root short, in order to reduce the cost of correction messages after a split. To obtain this we aggregate internal nodes of the distributed search tree obtained with the above described techniques in compound nodes, and apply the technique of the Distributed B$^+$-tree to the tree made up by compound nodes. For this reason we call our structure ADST (Aggregation in Distributed Search Tree).

Please note that the aggregation only happens at a logical level, in the sense that no additional structure has to be introduced. What happens in reality is simply that a server associated to a compound node maintains the same information maintained by the one associated to an internal node in the Distributed B$^+$-tree.

Each server s in ADST is conceptually associated to a leaf f. Then, as a leaf, s stores the list l of servers managing compound nodes in the path from f and the (compound) root of the ADST. If s has already split at least one time, then it stores also its *basic server* s'. In this case s' is a server that manages a compound node and such that $I(lt(s'))$ contains the *basic interval* of s.

Any server records in a field called *adjacent* the server managing the adjacent interval on its right. Moreover, if s manages also a compound node $va(s)$, then it also maintains a local tree, in addition to the other information.

2.1 Split Management

Let s be a server conceptually associated to a leaf f, and let the father compound node va^* of f be managed by a server s^*. Let us now consider a split of s with s_{new} as new server. The first operation performed by s is to send correction messages to each server in l.

Then, exactly like in the technique described for distributed B$^+$-tree [2], a new leaf f' and a new internal node v father of the two leaves are conceptually created. v is associated to s_{new}.

In ADST two situations are possible:

– The node v has to be *aggregated* with the compound node va^*. Then v is released, s does not change anything in its list l and s_{new} initializes its list l_{new} with a copy of l. If this was the first split of s, then s identifies s^* as its *basic server* and stores it in a specific field. Please note that the interval $I(lt(s^*))$ contains the *basic interval* of s.
– The node v has not to be *aggregated* with the compound node va^*. Then a new compound node va is created as a son of va^* aggregating the single internal node v. s_{new} is called to manage va. s changes its list l adding s_{new}. s_{new} initializes its list l_{new} with a copy of l. If this was the first split of s, then s identifies s_{new} as its *basic server* and stores it in a specific field. Please note that the interval $I(lt(s_{new}))$ is now exactly the *basic interval* of s.

The field *adjacent* of s_{new} is set with the value stored in the field of s. The field *adjacent* of s is set with s_{new} (see figure 1).

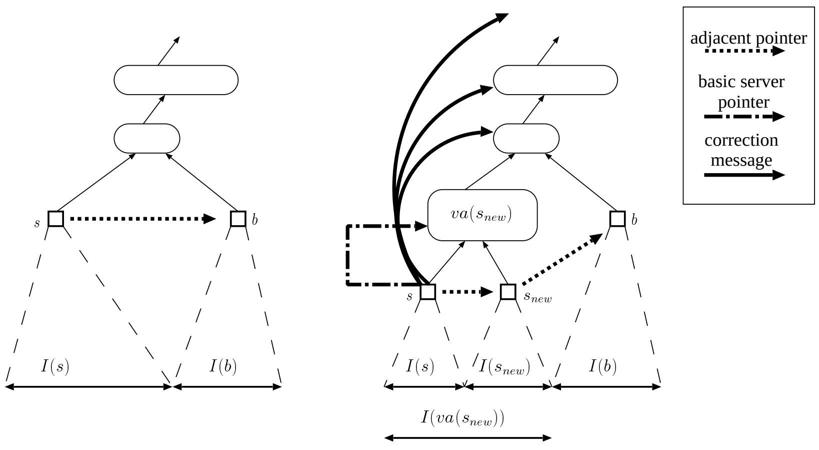

Fig. 1. Before (left) and after (right) the split of server s with s_{new} as new server. Intervals are modified accordingly. Correction messages are sent to server managing compound nodes stored in the list $s.l$ and *adjacent* pointers are modified. Since the aggregation policy decided to create a new compound node and s_{new} has to manage it, then s_{new} is added to the list $s.l$ of servers between the leaf s and the compound root nodes, s_{new} sets $s_{new}.l = s.l$. If this is the first split of s, then s sets s_{new} as its *basic server*

2.2 Aggregation Policy

The way to create compound nodes in the structure is called *aggregation policy.* We require that an aggregation policy creates compound nodes so that the height of the tree made up by the compound nodes is logarithmic in the number of servers of the ADST. In such a way the cost of correcting the local trees after a split is logarithmic as well.

One can design several aggregation policies, satisfying the previous requirement. The one we use is the following.

(AP): To each compound node va a bound on the number of internal nodes $l(va)$ is associated. The bound of the root compound node ra is $l(ra) = 1$. If the compound node va' father of va has bound $l(va')$, then $l(va) = 2l(va') + 1$.

Suppose a server associated to a leaf son of va splits. If the bound $l(va)$ is not reached, va aggregates the new internal nodes v. Otherwise a new compound node has to be created as a son of va and aggregating v.

It is easy to prove the following:

Invariant 1 *Let va_0 be the compound root node. Let $va_0, va_1, ..., va_k$ be the compound nodes in the path between va_0 and a leaf. Then* $\#\text{internal\_nodes}(va_i) = l(va_i)$*, for each* $0 \leq i \leq k-1$*, and* $\#\text{internal\_nodes}(va_k) \leq l(va_k)$*.*

With reference to figure 2, we have for example: $I(a) = A$, $I(b) = B$, and so on; $a.adjacent = b$, $b.adjacent = c$, $c.adjacent = q$, and so on; $a.l = \{a\}$, $b.l = \{c, a\}$, $c.l = \{q, c, a\}$, and so on; $lt(f) = \{\langle d, D\rangle, \langle f, F\rangle, \langle g, G\rangle, \langle h, H\rangle, \langle i, I\rangle, \langle l, L\rangle, \langle o, O\rangle, \langle p, P\rangle, \langle m, M\rangle, \langle n, N\rangle\}$ and then $I(va(f)) = I(lt(f)) = D \cup F \cup G \cup H \cup I \cup L \cup O \cup P \cup M \cup N$; from the given sequence of splits, $a.basic\_server = a$, $b.basic\_server = c$, $c.basic\_server = c$, and so on.

Theorem 2. *Aggregation policy AP guarantees that the length of any path between a leaf and the compound root node is bounded by $h_a = k \leq \lfloor \log n \rfloor + 1$, where n is the number of internal nodes of the distributed tree.*

Proof. Let us consider a generic leaf f, and let $h_a = k$ be its height in the tree of compound nodes. Then there are k compound nodes $va_0, va_1, .., va_{k-1}$ in the path between f and the root compound node, and va_0 is just the compound root node. It follows directly from the definition of policy AP that: *#internal_nodes*$(va_0) = l(va_0) = 1$, *#internal_nodes*$(va_1) = l(va_1) = 2^1 + 1$, *#internal_nodes*$(va_2) = l(va_2) = 2^2 + 1$, ..., *#internal_nodes*$(va_{k-2}) = l(va_{k-2}) = 2^{k-2} + 1$ and *#internal_nodes*$(va_{k-1}) \geq 1$. Then the number n of internal nodes, in the case $k > 1$, is such that:

$$n \geq 1 + \sum_{i=1}^{k-2} (2^i + 1) + 1 = k - 2 + \frac{2^{k-1} - 1}{2 - 1} + 1 = k - 2 + 2^{k-1}$$

$$\Rightarrow n \geq 2^{k-1}, \forall k \geq 1$$

Then we have $h_a = k \leq \lfloor \log n \rfloor + 1$.

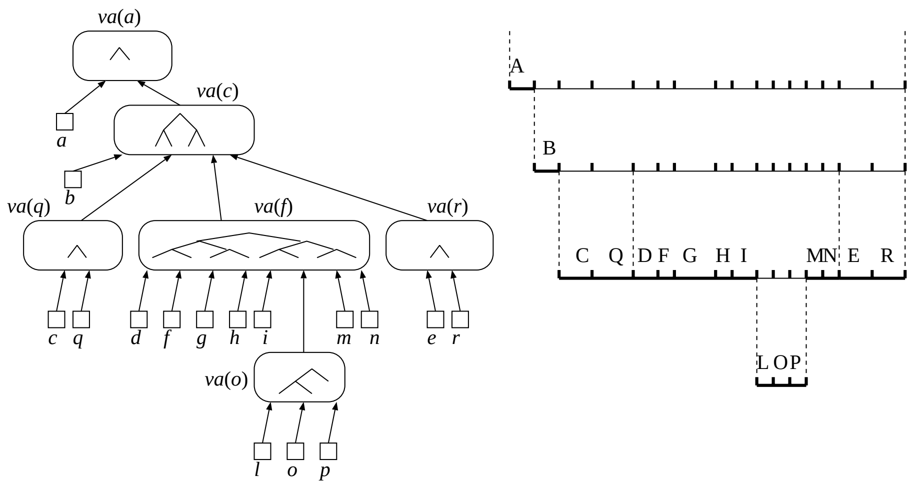

Fig. 2. An example of ADST with policy AP. Lower-case letters denote servers and associated leaves, upper-case letters denote intervals of data domain. The sequence of splits producing the structure is $a \to b \to c \to d \to e$, then $d \to f \to g \to h \to i \to l \to m \to n$, then $l \to o \to p$, then $c \to q$ and finally $e \to r$, meaning with $x \to y$ that the split of x creates the server y

We recall that for the binary tree we are considering, that is a tree where every node has 0 or two sons, the number of leaves, and then of servers, is $n = n' + 1$, where n' is the number of internal nodes. Substantially the previous theorem states that the cost of correcting local trees after a split is of $O(\log n)$ messages, where n is the number of servers in the ADST.

2.3 Access Protocols

We now analyze what happens when a client c has to perform a single-key request for a key k. We describe the case of exact search, insertions and deletion:

Exact Search: c looks for the pertinent server for k in its local tree, finds the server s, and sends it the request. If s is pertinent, it performs the request and sends the result to c.

Suppose s is not pertinent. If s does not manage a compound node, then it forwards the request to its *basic server* s'. We recall that $I(lt(s'))$ includes the basic interval of s, then, as stated before, if the request for k is arrived to s, k has to belong to this interval. Therefore s' is certainly logically pertinent: it looks for the pertinent server for k in its local tree and finds the server s''. Then s' forwards the request to s'', which performs the request and answers to c. In this case c receives the local tree of s' in the answer, so to update its local tree (see figure 3).

Suppose now that s manages a compound node. The way in which compound nodes are created ensures that $I(lt(s))$ includes the basic interval of s itself. Then

s has to be logically pertinent, hence it finds in $lt(s)$ the pertinent server and sends it the request. In this case c receives the local tree of s in the answer.

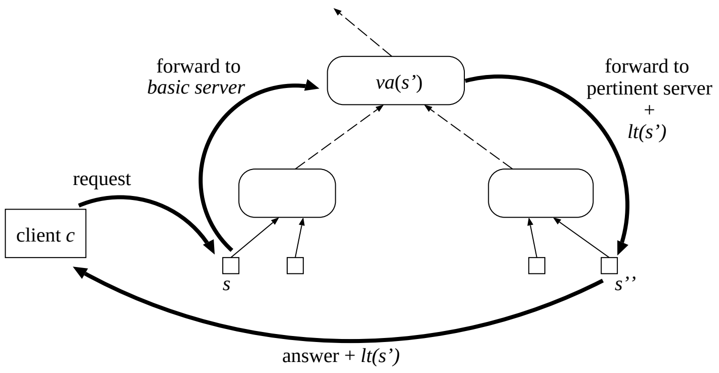

Fig. 3. Worst-case of the access protocol

Insertion: the protocol for exact search is performed in order to find the pertinent server s for k. Then s inserts k in its bucket. If this insertion causes s to go in overflow then a split is performed. After the split, correction messages are sent to the servers in the list l of s.

Deletion: the protocol for exact search is performed in order to find the pertinent server s for k. Then s deletes k from its bucket. If this deletion causes s to go in underflow then a *merge*, which is the opposite operation of a split, is performed.

We consider a server to be in underflow whenever it manages less than $\frac{b}{d}$ keys in the bucket, for a fixed constant d. The merge operation consists basically in releasing an existing server s which is in underflow. If s is not empty, it sends its remaining keys to another server s'. From now on $I(s')$ is enlarged by uniting it with $I(s)$. After the merge, correction messages are sent to the servers in list l of s. Moreover, an algorithm to preserve the invariant of policy AP is applied after a merge. A detailed presentation of this algorithm is quite long, and is left to the extended version of this paper. Please note that s' is only chosen if it is able to receive all the keys of s without going in overflow. If there is not such a server, a server s^*, whose interval is adjacent to $I(s)$, is chosen and it sends a proper number of keys to s in order to allow s to exit from the underflow state. $I(s)$ and $I(s^*)$ are modified accordingly.

Previous SDDSs, e.g LH*, RP*, DRT, etc., do not explicitly consider deletions. Hence, in order to compare ADST and previous SDDSs performances, we shall not analyze behavior of ADST under deletions.

2.4 Range Search

We now describe how a range search is performed in ADST.

The protocol for exact search is performed in order to find the server s pertinent for the leftmost value of the range. If the range is not completely covered by s, then s sends the request to server s' stored in its field *adjacent*. s' does the same. Following the adjacent pointers all the servers covering the range are reached and answer to the client. The operation stops whenever the server pertinent for the rightmost value of the range is reached, see figure 4.

The above algorithm is very simple and applies to the case when only the point-to-point protocol is available. Other variants can be considered, for example a client can send more than one request messages, if it discovers from its local tree that more than one server intersects the range of the query.

Usually whenever the range of a query is large, the multicast protocol is applied, if it is available in the considered technological framework. The same technique can be applied for ADST as well.

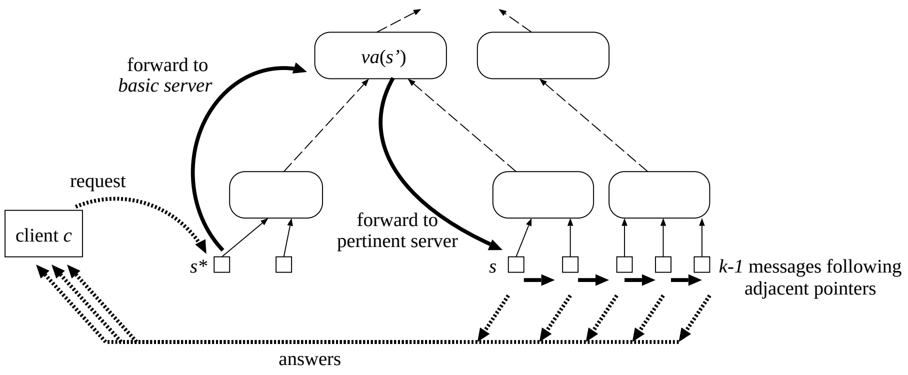
va(s')
forward to basic server
request
client c
s*
forward to pertinent server
s
k-1 messages following adjacent pointers
answers

Fig. 4. Worst-case of the range search. The server s is pertinent for the leftmost value of the range

3 Complexity Analysis

In what follows we suppose to operate in an environment where clients work slowly. More precisely, we suppose that between two consecutive requests the involved servers have the time to complete all updates of their local trees: we call this a low concurrency state. Any communication complexity result in SDDS proposals is based on this assumption. In the extended version of this paper [6] we show how the complexity analysis of SDDSs is influenced by this assumption, and we give some ideas on how to operate in the case of fast working clients (also called high concurrency).

3.1 Communication Complexity

The basic performance parameter for an SDDS is the communication complexity, that is the number of messages needed to perform the requests of clients. We present the main results obtained with ADST for this parameter.

Theorem 3. *An exact search has in an ADST a worst-case cost of 4 messages. An insertion that does not cause a split and a deletion that does not cause a merge has also in an ADST a worst-case cost of 4 messages.*

Proof. Follows directly from the presented algorithms.

Theorem 4. *A split and the following corrections of local trees have in an ADST a worst-case cost of* $\log n + 5$.

Proof. Follows directly from the fact that a split costs 4 messages and from theorem 2.

Theorem 5. *A range search has in an ADST a worst-case cost of* $k + 1$ *messages, where* k *is the number of servers covering the range of query, without accounting for the single request message and the* k *response messages.*

Proof. Follows directly from the algorithm. Theorem 3 ensures that the search of server covering the leftmost limit of the range adds, without accounting for the requests and answer messages, 2 messages at the cost. Then we add other $k - 1$ messages to reach by following the *adjacent* pointers the remaining $k - 1$ servers covering the range.

As presented in sub-section 2.3, in ADST a merge is followed by the correction of local trees and by a restructuring of the tree made up by compound nodes in order to keep the invariant 1. In the extended version of this paper we prove the following theorem.

Theorem 6. *In ADST a merge costs* $O(\log n)$ *messages, accounting the local trees correction messages and the restructuring algorithms.*

The following theorems show the behavior of ADST in the amortized case.

Theorem 7. *A sequence of intermixed exact searches and insertions on ADST has an amortized cost of* $4 + \frac{2(\log n+5)}{b}$ *messages per operation, where* b *is the capacity of a bucket.*

Proof. Let us consider a sequence of m intermixed exact searches and insertions performed on ADST. From lemma 1 we have at most $\left\lfloor \frac{m}{A} \right\rfloor$ splits, where $A = \frac{b}{2}$. From theorems 3 and 4 the total number of message for the sequence is $C \leq 4m + \left\lfloor \frac{m}{A} \right\rfloor (\log n + 5) \leq m\left(4 + \frac{2(\log n+5)}{b}\right)$, hence the result holds.

Theorem 8. *Let* b *be the capacity of a bucket and* $\frac{b}{d}$ *be the merge threshold, for a fixed constant* $d > 2$. *Then a sequence of intermixed exact searches, insertions and deletions on ADST has an amortized cost of* $4 + \frac{2d}{d-2}\frac{O(\log n)}{b}$ *messages per operation.*

Proof. (Sketch). Let us consider a sequence of m intermixed exact searches, insertions, and deletions performed on ADST. Let $D = \frac{b}{2} - \frac{b}{d} = \frac{d-2}{2d}b$. For ease of presentation, we assume without loss of generality that $\lfloor \frac{m}{D} \rfloor = \frac{m}{D}$. From lemma 1, it is easy to verify that we can have at most $\lfloor \frac{m}{D} \rfloor$ operations among splits and merges. In the extended version we show that the worst-case for such a sequence is when, respect to a server, one split is followed by one merge, and vice-versa. From theorems 3, 4 and 6 the total number of message for the sequence is $C \leq 4m + \lfloor \frac{m}{D} \rfloor O(\log n) \leq m \left(4 + \frac{2d}{d-2} \frac{O(\log n)}{b}\right)$, hence the result holds.

We conclude showing a basic fact that holds in a realistic framework.

Assumption 9. *The number n of servers participating in an ADST is such that $\log n < kb$, where b is the capacity of a bucket and $k > 0$ is a constant.*

In fact, since in a realistic situation b is at least in the order of hundreds or thousands, then, assuming $k = 1$, it is true in practice that $n < 2^b$. Hence the assumption is realistically true. For the rest of the paper we therefore assume that: $\log n < b$.

Under the assumption 9 the results of theorems 7 and 8 show that in practice ADST has an amortized constant cost for any single-key operation.

4 Conclusions

We presented the ADST (Aggregation in Distributed Search Tree). This is the first order preserving SDDS, obtaining a constant single key query cost, like LH*, and at the same time an optimal cost for range queries. More precisely our structure features: (i) a cost of 4 messages for exact-search queries in the worst-case, (ii) a logarithmic cost for insert queries producing a split, (iii) an optimal cost for range searches, that is a range search can be answered with $O(k)$ messages, where k is the number of servers covering the query range, (iv) an amortized almost constant cost for any single-key query.

The internal load of a server is the typical one of order preserving SDDSs proposed till now. In particular, servers managing compound nodes may be also the ones managing buckets of the structure, like it is in DRT [8], or they may be dedicated servers, like in RP* [10]. The choice does not influence the correctness of ADST technique.

Moreover ADST is also able to manage deletions, and it is easily extendible to manage k-dimensional data, keeping the same results. Furthermore, ADST is an orthogonal technique with respect to techniques used to guarantee fault tolerance, in particular to the one in [13], that provides a high availability SDDS.

Hence our proposal is a theoretical achievement potentially attractive for distributed applications requiring high performances for single key and range queries, high availability and possibly the management of multi-dimensional data.

In [7] experimental comparisons exploring behavior of ADST in the average case and comparing it with existing structures are considered. The result is that

ADST is the best choice also in the average case. This makes ADST interesting, beyond the theoretical level, also from an application point of view.

Future work will also study the impact of using different aggregation policies. Any aggregation policies ensuring results of theorem 2 can be applied.

References

1. P. Bozanis, Y. Manolopoulos: DSL: Accomodating Skip Lists in the SDDS Model, *Workshop on Distributed Data and Structures (WDAS 2000)*, L'Aquila, June 2000.
2. Y. Breitbart, R. Vingralek: Addressing and Balancing Issues in Distributed B^+-Trees, *1st Workshop on Distributed Data and Structures (WDAS'98)*, 1998.
3. R.Devine: Design and implementation of DDH: a distributed dynamic hashing algorithm, *4th Int. Conf. on Foundations of Data Organization and Algorithms (FODO)*, Chicago, 1993.
4. A.Di Pasquale, E. Nardelli: Fully Dynamic Balanced and Distributed Search Trees with Logarithmic Costs, *Workshop on Distributed Data and Structures (WDAS'99)*, Princeton, NJ, May 1999.
5. A.Di Pasquale, E. Nardelli: Distributed searching of k-dimensional data with almost constant costs, *ADBIS 2000*, Prague, September 2000.
6. A.Di Pasquale, E. Nardelli: ADST: Aggregation in Distributed Search Trees, Technical Report 1/2001, *University of L'Aquila*, February 2001.
7. A.Di Pasquale, E. Nardelli: A Very Efficient Order Preserving Scalable Distributed Data Structure, accepted for pubblication at *DEXA 2001 Conference*.
8. B. Kröll, P. Widmayer: Distributing a search tree among a growing number of processor, in *ACM SIGMOD Int. Conf. on Management of Data*, pp 265-276 Minneapolis, MN, 1994.
9. W. Litwin, M.A. Neimat, D.A. Schneider: LH* - Linear hashing for distributed files, *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D. C., 1993.
10. W. Litwin, M.A. Neimat, D.A. Schneider: RP* - A family of order-preserving scalable distributed data structure, in *20th Conf. on Very Large Data Bases*, Santiago, Chile, 1994.
11. W. Litwin, M.A. Neimat, D.A. Schneider: k-RP_s^* - A High Performance Multi-Attribute Scalable Distributed Data Structure, in *4th International Conference on Parallel and Distributed Information System*, December 1996.
12. W. Litwin, M.A. Neimat, D.A. Schneider: LH* - A Scalable Distributed Data Structure, *ACM Trans. on Database Systems*, 21(4), 1996.
13. W. Litwin, T.J.E. Schwarz, S.J.: LH^*_{RS}: a High-availability Scalable Distributed Data Structure using Reed Solomon Codes, *ACM SIGMOD Int. Conf. on Management of Data*, 1999.
14. E. Nardelli, F.Barillari, M. Pepe: Distributed Searching of Multi-Dimensional Data: a Performance Evaluation Study, *Journal of Parallel and Distributed Computation (JPDC)*, 49, 1998.
15. R.Vingralek, Y.Breitbart, G.Weikum: Distributed file organization with scalable cost/performance, *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, 1994.
16. R.Vingralek, Y.Breitbart, G.Weikum: SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load, *Distr. and Par. Databases*, 6, 2, 1998.