A Very Efficient Order Preserving Scalable Distributed Data Structure

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. SDDSs (Scalable Distributed Data Structures) 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 present and discuss performances of ADST, a new order preserving SDDS with a worst-case constant cost for exact-search queries, a worst-case logarithmic cost for update queries, and an optimal worst-case cost for range search queries 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. Finally, our scheme can be easily generalized to manage k-dimensional points, while maintaining the same costs of the 1-dimensional case.
We report experimental comparisons between ADST and its direct competitors (i.e., LH*, DRT, and RP*) where it is shown that ADST behaves clearly better. Furthermore we show how our basic technique can be combined with recent proposals for ensuring high-availability to an SDDS. Therefore our solution is very attractive for network servers requiring both a fast response time and a high reliability.

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

1 Introduction

The paradigm of SDDS (*Scalable Distributed Data Structures*) [9] is used to develop access methods in 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 disk space.

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.

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 [3,9,12,16,17], on order preserving techniques [1,2,4,7,8, 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 (e.g., RP* [10] and DRT* [5]) achieve good performances for range searches and a reasonably low (i.e. logarithmic), but not constant, worst-case cost for single key operations.

Here we present and discuss experimental results for ADST, 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 (e.g., RP* and DRT*). For a more detailed presentation of the data structure see [6].

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 Distributed Search Trees

In this section we review the main concepts relative to distributed search trees, in order to prepare the way for the presentation of our proposal and to allow its better comparison with previous solutions.

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. This operation basically consists of transfer half of its keys to a new fresh server s_{new}. Consequently the interval of keys I managed is partitioned in I_1 and I_2. After the split, s reduces its interval I to I_1. When s_{new} receives the keys, it initializes its interval to I_2. This is the first interval managed by s_{new} and we refer to such an interval as the *basic interval* of a server.

From a conceptual point of view, the splits of servers build up a virtual distributed tree, where each leaf is associated to a server, and a split creates a new leaf, associated to the new fresh server, and a new internal node.

Please note that the lower end of the interval managed by a server never changes. A split operation is performed by locking the involved servers. Its cost is a constant number of messages, typically 4 messages. We recall that since in the SDDS paradigm the length of a message is not accounted in the complexity, then it is assumed that all keys are sent to the new server using one message.

After a split, s manages $\frac{b}{2}$ keys and s_{new} $\frac{b}{2}+1$ keys. It is easy to prove that for a sequence of m intermixed insertions and exact searches we may have at most $\lfloor \frac{m}{A} \rfloor$ splits, where $A = \frac{b}{2}$.

The splits of a server is a local operation. Clients and the other servers are not, in general, informed about the split. As a consequence, clients and servers can make an *address error*, that is they send the request to a wrong server.

Therefore, clients and servers have a local indexing structure, called *local tree*. Whenever a client or a server performs a request and makes an *address error*, it receives information to correct its local tree. This prevents a client or a server to commit the same address error twice.

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 $I(s)$. A local tree can be seen as a tree describing the partition of the domain of keys produced by the splits of servers. A local tree can be wrong, in the sense that in the reality a server s is managing an interval smaller than what the client currently knows, due to a *split* performed by s and yet unknown to the client.

Note that for each request of a key k received by a server s, k is within the *basic interval* I of s, that is the interval s managed before its first division. This is due to the fact that if a client has information on s, then certainly s manages an interval $I' \subseteq I$, due to the way overflow is managed through *splits*. In our proposal, like other SDDS proposals, we do not consider deletions, hence intervals always shrinks. Therefore if s is chosen as the server to which to send the request of a key k, it means that $k \in I' \Rightarrow k \in I$.

Given the local tree $lt(s)$ associated to server s, we denote as $I(lt(s))$ the interval of $lt(s)$, defined as $I(lt(s)) = [m, M)$, where m is the minimum of lower ends of intervals in the associations stored in $lt(s)$, and M is the maximum of upper ends of intervals in the associations stored in $lt(s)$.

From now on we define a server s *pertinent* for a key k if $k \in I(s)$, and *logically pertinent* if $k \in I(lt(s))$.

3 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].

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 associated to nodes 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 above technique 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.

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 (see figure 1).

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 $l(va) = 2l(va') + 1$.

In figure 2 an example of ADST is presented.

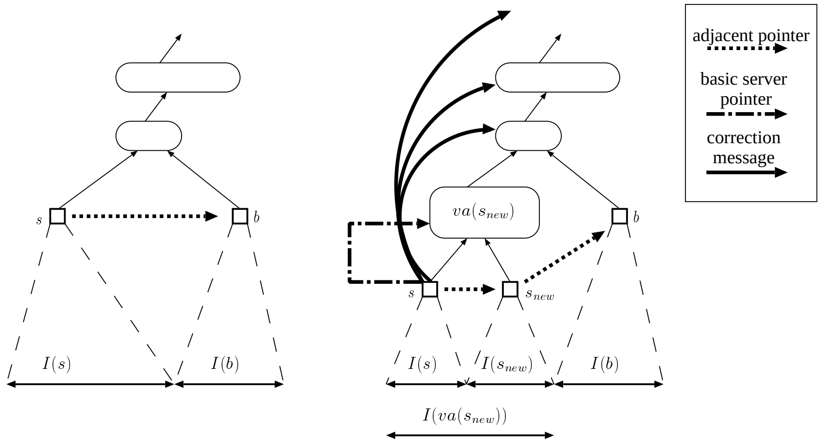

Fig. 1. Before (left) and after (right) the split of server s with s_{new} as new server. Intervals are modified accordingly. Correction messages are sent to server managing compound nodes stored in the list $s.l$ and *adjacent* pointers are modified. Since the aggregation policy decided to create a 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*.

We now show how a client c looks for a key in ADST: 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.

For an 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

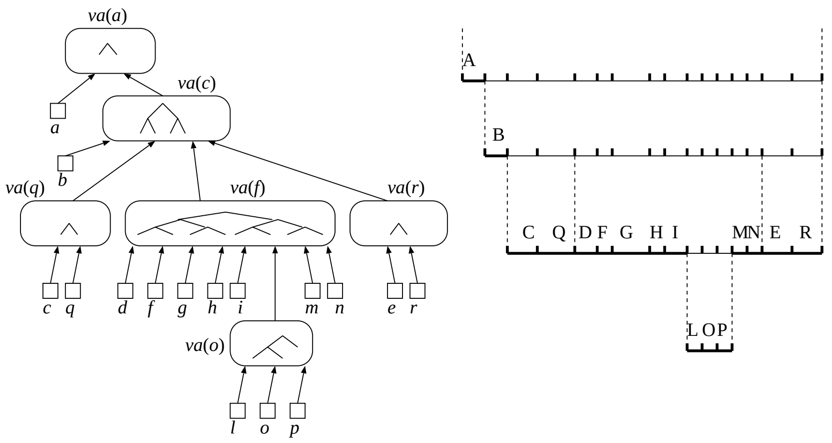

Fig. 2. An example of ADST with policy AP. Lower-case letters denote servers and associated leaves, upper-case letters denote intervals of data domain. The sequence of splits producing the structure is $a \to b \to c \to d \to e$, then $d \to f \to g \to h \to i \to l \to m \to n$, then $l \to o \to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

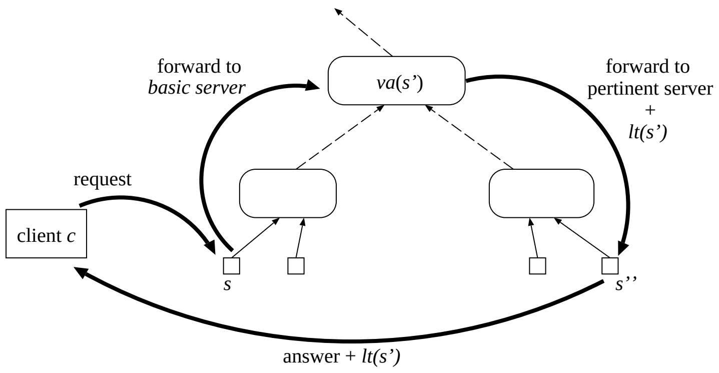

Fig. 3. Worst-case of the access protocol

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.

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.

To perform a range search 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).

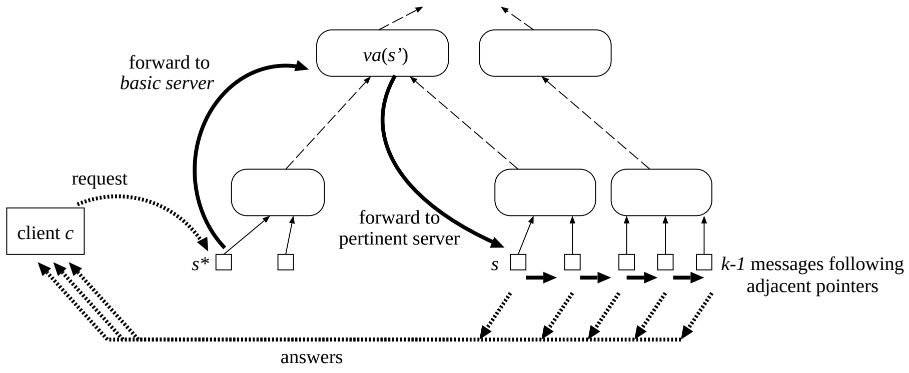

Fig. 4. Worst-case of the range search. s is the server pertinent for the leftmost value of the range.

In the following we give the main results of ADST. Detailed descriptions can be found in the extended version of the paper [6].

An exact search and an insertion that does not cause a split have in an ADST a worst-case cost of 4 messages. A split and the following corrections of local trees have in an ADST a worst-case cost of $\log n + 5$.

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.

Moreover, under realistic assumptions, a sequence of intermixed exact searches and insertions on ADST has an amortized cost of $O(1)$ messages. The basic assumption is that $\frac{\log n}{b} < 1$. For real values of b, e.g. hundreds, thousands or more, the assumption is valid for SDDSs made by up to billions of servers.

ADST has a good load factor, under any key distribution, like all the other order preserving structures, that is 0.5 in worst case and about 0.7 ($= \ln 2$) as expected value.

Another important performance parameter for an SDDS is the convergence of new client's index. This is the number of requests that a new client starting with an initial index has to perform in order to have an index reflecting the exact structure of the distributed file: this means that the client does not make address errors until new splits occur in the structure. The faster is the convergence, the lower is the number of address errors made by clients.

In ADST a new client is initialized with a local tree containing the server associated to the compound root node, or the unique server, in the case ADST is made up just by one server. Due to the correction technique used after a split, this server knows the exact partition of data domain among servers. Then it is easy to show that a new client obtains a completely up-to-date local tree after just one request.

Also in this case ADST notably improves previous results. In particular we recall that, for an n-servers SDDS, the convergence of a new client's index requires in the worst-case:

– n messages in any structure of DRT family.
– $O\left(\frac{n}{0.7f}\right)$ messages in RP*s, where f is the fanout of servers in the kernel.
– $O(\log n)$ messages in LH*.

4 Experimental Comparison

In this section we discuss results of experimental comparisons between ADST performances and RP*, DRT* and LH* ones with respect to sequences of intermixed exact-searches and insertions. The outcome is that ADST behaves clearly better than all its competitors.

As discussed previously ADST presents worst-case constant costs for exact searches, worst-case logarithmic costs for insertions whenever a split occurs, and amortized constant costs for sequences of intermixed exact-searches and insertions. On the other hand, LH* is the best SDDS for single key requests, since it has worst-case constant costs for both exact searches and insertions, and constant costs in the amortized case as well.

The objective of our experimental comparison is to show which is the difference between ADST and its direct competitors. We have not considered in our experimental comparison the case of deletions since this case is not explicitly analyzed in LH*, RP* and DRT*.

In our experiments we perform a simulation of SDDS using the CSIM package [15], which is the standard approach in the SDDS literature for this kind of performance evaluation. We analyze structures with a capacity of buckets fixed at $b = 100$ records, that is a small, but reasonable, value for b. Later we describe the behavior of the structures with respect to different values of b. We consider two situations: a first one with 50 clients manipulating the structures and a second one with 500 clients. Finally, we consider three random sequences of intermixed insertions and exact searches: one with 25% of insertions, one with 50% of insertions and one with 75%.

We have considered a more realistic situation of fast working clients, in the sense that it is possible that a new request arrives to a server before it has terminated with updating operations. This happens more frequently when considering 500 clients with respect to the case of 50 clients.

The protocol of operations is the usual one. In case of exact searches an answer message arrives to the client which issued the request, with the information to correct the client index. In case of insertions, the answer message is not sent back. This motivates the slightly higher cost for the structures in case of low percentage of inserts, even if more exact searches means more possibility to correct an index of a client.

Note that, although all costs in LH* are constant, while ADST has a logarithmic split cost, in practice ADST behaves clearly better with respect to LH* in this environment, as shown in figure 5, 6, 7, 8, 9 and 10. This is fundamentally motivated by the better capacity of ADST to update client indexes with respect to LH*, and then to allow clients to commit a lower number of address errors. This difference of capability is shown by the fact that while LH* slightly increases its access cost passing from 50 to 500 clients, for ADST the trend is opposite: with 500 clients access cost is slightly decreased with respect to 50.

The logarithmic cost of splits for ADST become apparent for lower values of b, where the weight of term $\frac{\log n}{b}$ increases its relative weight. However lower values for b, e.g. 10, are not realistic for an SDDS involving a large number of servers. On the contrary the situation shown in figures is even more favourable to ADST for larger values of b, like for example 1000 or more (in this case the term $\frac{\log n}{b}$ decreases its relative weight). This also happens for larger number of clients querying the structure, due to the relevant role played by the correction of client indexes.

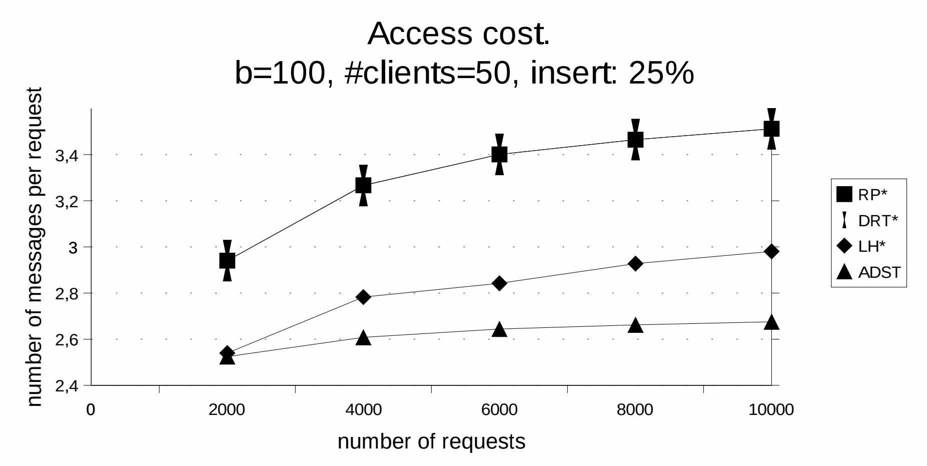

Fig. 5. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 50$ and a sequence of requests of intermixed exact searches and inserts, with 25% of inserts.

Other possible experiments could have involved the load factor of the structures, but there we fundamentally achieve the results of other order preserving SDDSs. For a series of experimental comparisons see [7].

For range searches, ADST is clearly better than LH*, where a range search can require to visit all the servers of the structure, even for small ranges. This is a direct consequence of the fact that ADST preserves the order of data.

For other order preserving proposals (e.g. RP*s, BDST), the worst-case range search cost is $O(k + \log n)$ messages, when using exclusively the point-to-point protocol, without accounting for the request message and the k response mes-

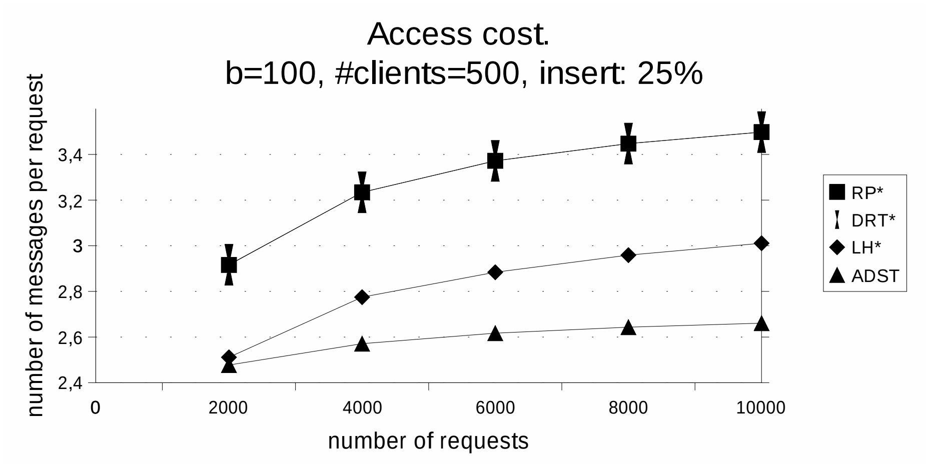

Fig. 6. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 500$ and a sequence of requests of intermixed exact searches and inserts, with 25% of inserts.

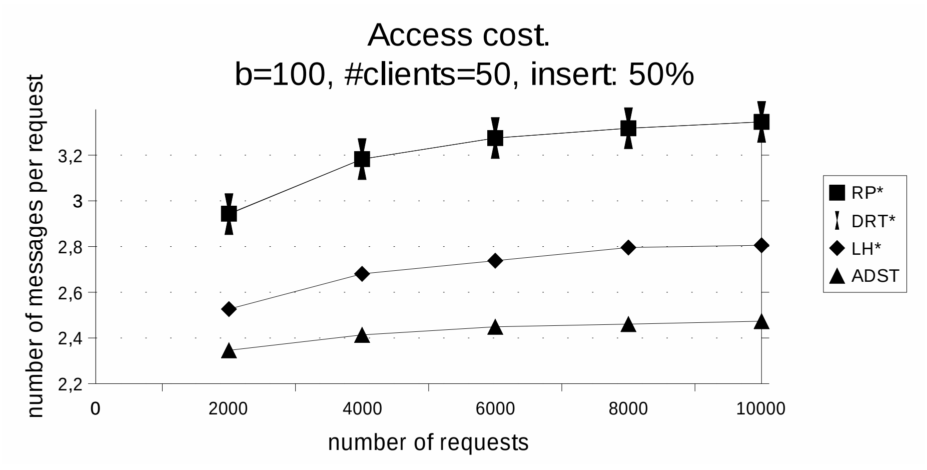

Fig. 7. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 50$ and a sequence of requests of intermixed exact searches and inserts, with 50% of inserts.

sages. The logarithmic term is due to the possibility that the request arrives to a wrong server and then has to go up in the tree to find the server associated to the node covering the entire range of the query. The base of the logarithm is a fixed number (it is equal to 2 for BDST and to the fanout of servers in the kernel for RP*s), while n is assumed unbounded.

Hence, in the case of use of point-to-point protocol, our algorithm clearly improves the cost for range search with respect to other order preserving proposals

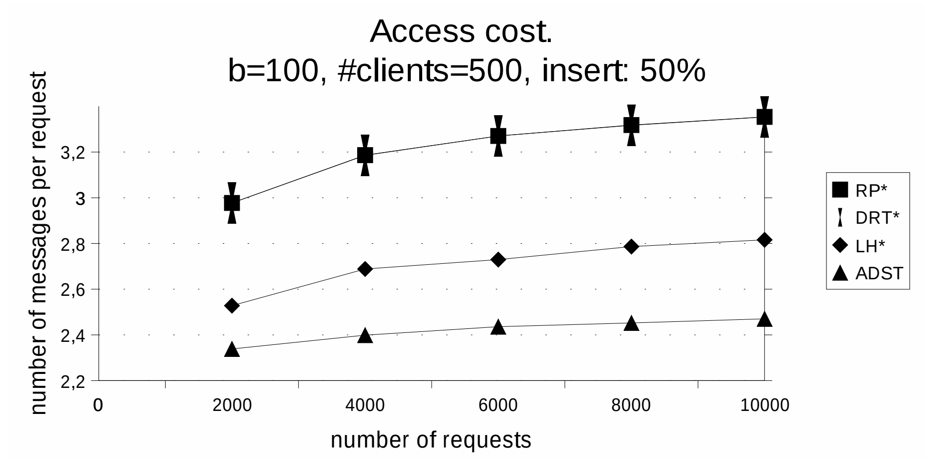

Fig. 8. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 500$ and a sequence of requests of intermixed exact searches and inserts, with 50% of inserts.

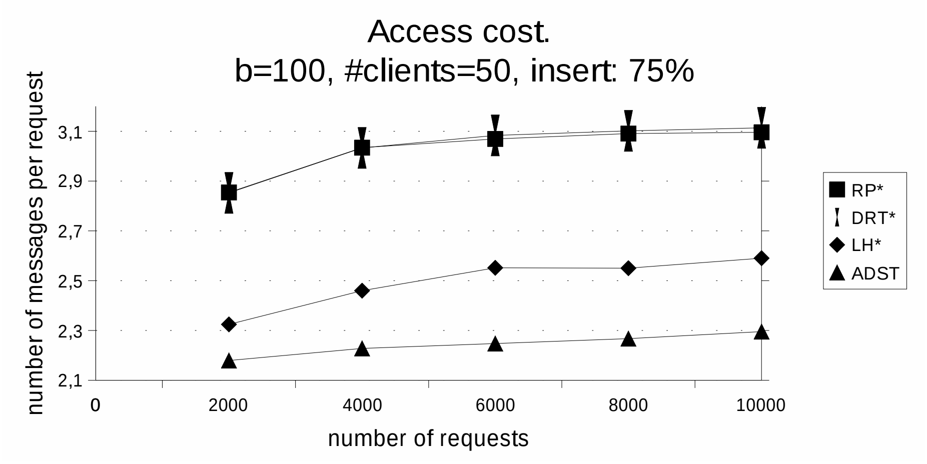

Fig. 9. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 50$ and a sequence of requests of intermixed exact searches and inserts, with 75% of inserts.

and reaches the optimality. Whenever multicast is used, all proposals have the same cost since in this case the nature of access method does not affect the cost.

5 Extensions

The basic ADST technique can be extended to:

- manage k-dimensional data. This is obtained considering the distributed k-d tree with index at client and server sites [14];
- manage deletions.

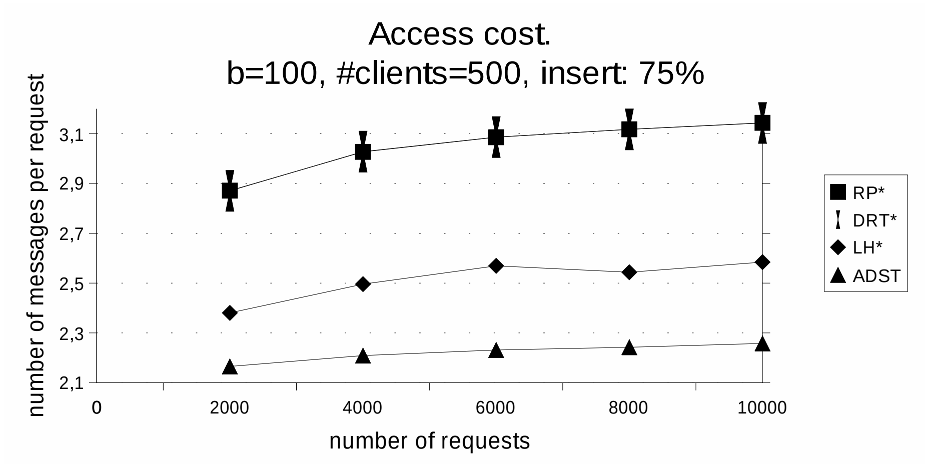

Fig. 10. Access cost for a bucket capacity $b = 100$ and for a number of clients $c = 500$ and a sequence of requests of intermixed exact searches and inserts, with 75% of inserts.

The detailed presentation of the extensions would exceed the limit of the paper and can be found in extended version of this paper [6]. In the following we just consider the fault tolerance extension of ADST.

5.1 High Availability

In this section we want to focus on the fact that our scheme is not restrictive with respect to techniques for fault tolerance in SDDSs, and we can consider ADST as an access method completely orthogonal to such techniques.

In particular we focus on the techniques for high availability using *Reed Solomon codes* and in general based on *record grouping* [13] and on the very interesting scalable availability provided by this scheme. One of the important aspect of this work is that with full availability of buckets, the normal access method can be used, while recovery algorithms have to be applied whenever a client cannot access a record in normal mode. In such a case we say an operation enters in *degraded* mode.

A more detailed description of record grouping and of techniques based on Reed Solomon codes would exceed the limits of this paper, hence we give only a brief sketch in the following.

In [13] LH* is used as access method in normal mode. The operations in *degraded* mode are handled by the coordinator. From the managed state of file, it locates the bucket to recover, and then proceeds with the recovery using the parity bucket. But for the search of the bucket to recover, the access method does not influence the correctness of the recovery algorithm, based on parity buckets. Moreover the coordinator is, in some sense, considered always available.

As already stated in [13], the same technique may be applied to other SDDS than LH*. For example we consider ADST. Buckets can be grouped and the parity bucket can be added in the same way as in [13]. We just have to associate a rank to a record in a bucket. This can be the position of the record at the moment of insertions.

ADST technique is used in normal mode. Whenever an operation enters in *degraded* mode, it is handled by the coordinator. We assume that the coordinator is the server s managing the compound root node (or a server with a copy of the local tree of s that behaves like the coordinator in LH*. This is not important here). From the request entered in degraded mode, the coordinator finds the server and then the bucket pertinent for the request. Then s can proceed with the recovery, following algorithms of [13].

The cost of operations in degraded mode are just increased with the cost of the recovery. In this case we want to emphasize that ADST, extended for achieving high availability, still keeps better worst-case and amortized case performances than the extensions of other proposals, e.g. RP*s or DRT, to a high availability schema.

6 Conclusions

We presented an evaluation of performances of 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, like RP* and DRT*. 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 in the worst-case, (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 experimental analysis compares ADST and its direct competitors, namely LH*, RP* and DRT*. The outcome is that ADST has better performances than LH* (hence better than RP* and DRT*) in the average case for single-key requests. Moreover ADST is clearly better with respect to other order preserving SDDSs, like RP* and DRT*, for range searches (hence better than LH*).

ADST can be easily extended to manage deletions and to manage k-dimensional data. Moreover, we have shown that 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 very attractive for distributed applications requiring high performances for single key and range queries, high availability and possibly the management of multi-dimensional data.

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, Carleton Scientific, May 1999.
5. A.Di Pasquale, E. Nardelli: Distributed searching of k-dimensional data with almost constant costs, *ADBIS 2000*, Prague, Lecture Notes in Computer Science, Vol. 1884, pp. 239-250, Springer-Verlag, September 2000.
6. A.Di Pasquale, E. Nardelli: ADST: Aggregation in Distributed Search Trees, Technical Report 1/2001, *University of L'Aquila*, February 2001, submitted for publication.
7. 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.
8. B. Kröll, P. Widmayer. Balanced distributed search trees do not exists, in *4th Int. Workshop on Algorithms and Data Structures(WADS'95)*, Kingston, Canada, (S. Akl et al., Eds.), Lecture Notes in Computer Science, Vol. 955, pp. 50-61, Springer-Verlag, Berlin/New York, August 1995.
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. H. Schwetman: Csim reference manual. Tech. report ACT-ST-252-87, Rev. 14, MCC, March 1990
16. R.Vingralek, Y.Breitbart, G.Weikum: Distributed file organization with scalable cost/performance, *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, 1994.
17. R.Vingralek, Y.Breitbart, G.Weikum: SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load, *Distr. and Par. Databases*, 6, 2, 1998.