Distributed Searching of k-Dimensional Data with Almost Constant 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. In this paper we consider the dictionary problem in the scalable distributed data structure paradigm introduced by Litwin, Neimat and Schneider and analyze costs for insert and exact searches in an amortized framework. We show that both for the 1-dimensional and the k-dimensional case insert and exact searches have an amortized almost constant costs, namely $O\left(\log_{(1+A)} n\right)$ messages, where n is the total number of servers of the structure, b is the capacity of each server, and $A = \frac{b}{2}$. Considering that A is a large value in real applications, in the order of thousands, we can assume to have a constant cost in real distributed structures.
Only worst case analysis has been previously considered and the almost constant cost for the amortized analysis of the general k-dimensional case appears to be very promising in the light of the well known difficulties in proving optimal worst case bounds for k-dimensions.

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

1 Introduction

The constant increase of PCs and workstations connected by a network and the need to manage greater and greater amount of data motivates the research focusing on the design and analysis of distributed databases. The technological framework we make reference to is the so called *network computing*: fast communication networks and many powerful and cheap workstations. There are several aspects making this environment attractive. The most important one is that a set of sites has more power and resources with respect to a single site, independently from the equipment of a site. Moreover the network offers a transfer speed that is not comparable with the magnetic or optical disks one. Therefore this framework is a suitable environment for the newer applications with high performance requirements, like, for example, spatio-temporal databases [14,3].

In this work we consider the dictionary problem in a message passing distributed environment and we follow the paradigm of the SDDS (*Scalable Distributed Data Structure*) defined by Litwin, Neimat e Schneider [8]. The main properties of SDDS paradigm are:

1. Keep a good performance level while the number of managed objects changes.
2. Perform operations locally.

We assume that data are distributed among a variable number of servers and accessed by a set of clients. Both servers and clients are distributed among the nodes of the network. Clients and servers communicate by sending and receiving *point-to-point* messages. We assume network communication is free of errors. Servers store objects uniquely identified by a key. Every server stores a single block (called *bucket*) of at most b data items, for a fixed number b. New servers are brought in as the volume of data increases to maintain the performance level.

The fundamental measure of the efficiency of an operation in this distributed context is the number of messages exchanged between the sites of the network. The internal work of a site is neglected. In order to minimize the number of messages, in a search operation it is possible to use some index locally to a site to better address the search towards another site. The search process in the local index performed by a site is not accounted in the complexity analysis.

The clients are not, in general, up-to-date with the evolution of the structure, in the sense they have some local indexing structure, but do not know, in general, the overall status of the data structure. Different clients may therefore have different and incomplete views of the data structure.

In an extreme case we can design the following distributed structure: there is a server *root* knowing all the other servers. When a split occurs, the new server which is brought in sends a messages to *root* to communicate its presence. When a server is not pertinent for a request, it sends the request to *root*, that looks for the correct server in its local index and sends it the request. Each access has thus a cost of at most 2 messages. But with this solution *root* is a bottleneck, because it has to manage each address error, and this violates the basic scalability requirement of the SDDS paradigm.

However, the above example shows that we can have, within this distributed computing framework, a worst case constant cost for the search process, while in the centralized case the lower bound is well known to be logarithmic.

There are various proposal in the literature addressing the dictionary problem within the paradigm of the SDDS: LH* [8], RP* [9], DRT [7], lazy k-d-tree [10], RBST [1], BDST [4] distributed B+-trees [2].

In this work we propose a variant of the management technique for distributed data used in the DRT [7]. We conduct an amortized analysis of the proposed strategy showing it has an almost constant cost for insert and search and we show how to adapt the strategy to the multi-dimensional case.

2 Description of the Structure

2.1 Split Management

Servers manage their bucket in the usual way. We say a server goes in overflow when it is managing b keys and a new one is sent to it, where b is the capacity of a server. For the sake of simplicity, we assume b is even. When a server goes in overflow it has to split: it finds a new server to bring in (for example asking to a special site, called Split Coordinator), and sends it half of its keys.

The interval of the keys managed by s is divided by the *split* in two sub-intervals. From now on, the server s manages one of this sub-intervals (the one that contains the keys remaining in s), while s' manages the other one. We assume that after a *split* the splitting server s always manages the lower half of the two intervals resulting from the *split* and the new server s' manages the upper half. Also, after this *split*, s knows that s' is the server brought in by itself.

After a *split*, one of the two resulting servers manages $\frac{b}{2}$ keys and the other one $\frac{b}{2}+1$ keys. Let $A = \frac{b}{2}$. Whit m requests, it follows directly that we can have at most $\lfloor \frac{m}{A} \rfloor$ *splits*.

2.2 Local Tree

The clients and the servers have a local indexing structure, called *local tree*. From a logical point of view this is a tree composed by an incomplete collection of servers. For each server s the managed interval of keys $I(s)$ is also stored. The local tree of a client 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 unknown to the client. In particular, given the split management policy above described, if $I_r = [a, b)$ is the real interval of s, and $I_{lt} = [c, d)$ is the interval of s in some local tree, then $a = c$ and $b \leq d$. For example in reality $I_r(s) = [100, 200)$, while in a local tree we could have $I_{lt}(s) = [100, 250)$. The local tree can be managed internally with any data structure: list, tree,etc.

Note that for each request of a key k received by a server s, k is within the interval I that 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*. Therefore if s is chosen as server to which to send the request of a key k, it means that $k \in I' \Rightarrow k \in I$.

The local tree of a client c is set up and updated using the answers of servers to request of c. The local tree of a server s is composed at least by the servers generated by s through a *split*. In particular, since a server always knows the next ones brought in by itself through its *splits*, this always guarantees the existence of a path between the initial server and any other server. A server always adds its *local tree* in every message to update clients with information about its view of the overall structure.

2.3 Requests Management

A client c that wants to perform a request chooses in its local tree the server s that should manage the request and sends it a *request message*.

If s is pertinent for the request then performs it (see figure 1-a). In general, if the request is a search operation then an answer is always sent back to the client; if it is an insert no answer is sent.

If s is not pertinent we have an *address error*. In this case s looks for the pertinent server s' in its *local tree* and forwards it the request.

Since also s' can be not pertinent, thus forwarding the request to still another server, in general we can have a series of *address error* that causes a chain of messages between the servers s_1,s_2,..,s_k. Finally, server s_k is pertinent and can satisfy the request. Moreover, s_k receives the local trees of the server s_1,s_2,..,s_{k-1} which have been traversed by the request. It first builds a correction tree C aggregating the local trees received and its own one, and then sends Local Tree Correction (LTC) messages with C to the client (even if it was an insert operation) and to all servers s_1,s_2,..,s_{k-1}, so to allow them to correct their local trees (see figure 1-b).

In figure 1 the possible cases of search process are shown. We have that each request has a cost, without counting the initial request and the final answer messages, either 0 (case a) or $2(k-1)$(case b).

This strategy to manage the distributed structure, is very similar to the one defined by Kröll and Widmayer for DRT [7] and therefore we call it DRT*.

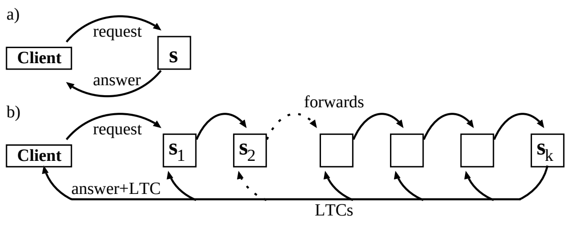

Fig. 1. Possible cases of the search process.

2.4 Split Tree

From the description above of the local trees and how they change due to the distribution of information about the overall structure through LTC messages, it is clear that the number of messages needed to answer a request changes with the increase of the number of requests. To analyze how changes in the content and structure of local trees affect the cost of answering to requests we associate to each server s of DRT* a rooted tree $ST(s)$, called the *split tree* of s. The nodes of $ST(s)$ are the servers pertinent for a request arriving to s. The tree has an arbitrary structure except that the root is s. An arc (s_1, s_2) in $ST(s)$ means that s_1 is in the local tree of s_2. When a server updates its local tree using LTC messages the structure of $ST(s)$ changes.

We call $ST_0(s)$ the split tree of server s obtained from a sequence of requests over a DRT* without applying the correction of the local trees of the servers using LTC messages, i.e. $ST_0(s)$ is shaped only by *splits* of the servers. Initially $ST_0(s)$ is made up only by s. Whenever s splits, with s' as new server, the node s' and a new arc (s', s) are added to $ST_0(s)$. The same holds for the splits of servers which are nodes in $ST_0(s)$ (for example, in figure 2-center, the split of server e adds the node s' and the arc (s', e) in $ST_0(a)$).

Since each server s' in $ST_0(s)$ was created by a chain of splits emanating from s, then s' manages a sub-interval of the initial interval managed by s.

If we consider the correction of local trees, the structure of the split tree of s changes. Infact, due to the correction, after a request to a server d, s adds all the servers in the path between s and d in its local tree. The consequence is that now s can address directly these servers in the future. In order to describe this new situation in the split tree of s, we delete the arcs of the traversed path and add to s the arcs between s and the traversed servers. The result is a compression of the path between s and d (see figure 2-right).

We denote with $ST(s)$ the split tree of s whose structure has been determined by the use of LTC messages. We denote with $T_s(s')$ the sub-tree of $ST(s)$ rooted at server s'. We give some immediate properties of split trees:

Lemma 1. *Each request arriving to s is pertinent for a server in $ST(s)$.*

Lemma 2. *Let s' be a server in $ST(s)$. Let $Q_s(s')$ be the set of servers in the sub-tree of $ST_0(s)$ rooted at s', but for s' itself. Let $p(s', s)$ be the set of servers belonging to the path in $ST_0(s)$ from s' (excluded) to s (included).*

As long as no request pertinent for a server $x \in Q_s(s')$ arrives to a server $y \in p(s', s)$, it is $ST(s') = T_s(s')$.

For example, by comparing figure 2-left and figure 2-right, you can check that $ST(c)$ does not correspond anymore with the sub-tree $T_a(c)$ of $ST(a)$ after the request pertinent for d arrives to a and is forwarded to d.

We use the split trees to takes into account in the amortized analysis the use of LTC messages to reduce the cost of satisfying the request.

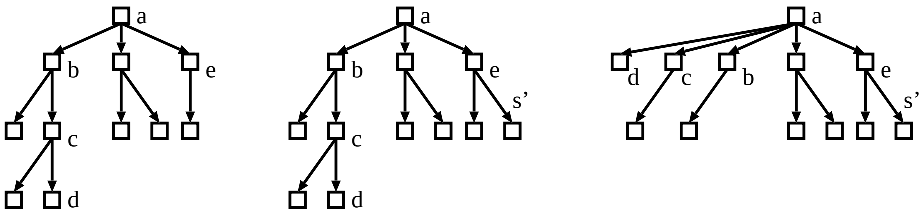

Fig. 2. The splits build up $ST_0(a)$ (e splits, with s' as new server)(left and center). The effect of a compression after a request pertinent for d and arrived to a (right). $ST(c)$ does not correspond anymore to the sub-tree $T_a(c)$ of $ST(a)$. The same for b.

3 Amortized Analysis

Since the way local trees change during the evolution of the overall structure is similar to the structural changes happening in the *set union problem* we now first briefly recall it and then analyze amortized complexity of operations in DRT*.

3.1 The Set Union Problem

The set union is a classical problem that has been deeply analyzed [13,15]. It is the problem of maintaining a collection of disjoint sets of elements under the operation of union. All algorithms for the set union problem appearing in the literature use an approach based on the *canonical element*. Within each set, we distinguish an arbitrary but unique element called the *canonical element*, used to represent the set. Operations defined in the set union problem are:

– *make-set(e)*: create a new set containing the single element e, which at the time of the operation does not belong to any set. The canonical element of the new set is e.
– *find(e)*: return the canonical element of the set containing element e.
– *union(e,f)*: combine the sets whose canonical elements are e and f into a single set, and make either e or f the canonical element of the new set. This operation requires that $e \neq f$.

We represent each set by a rooted tree whose nodes are the elements of the set and the root is the canonical element. Each node x contains a pointer $p(x)$ to its parent in the tree; the root points to itself. This is a *compressed tree* representation [6].

To carry out *find(e)*, we follow parent pointers from e until the root, which is then returned. While traversing parent pointer, one can apply some techniques for compressing the path from the elements to the root: *compression*, *splitting*, and *halving* (see figure 3).

To carry out *union* various techniques can be applied: *naive linking*, *linking by rank* and *linking by size.*

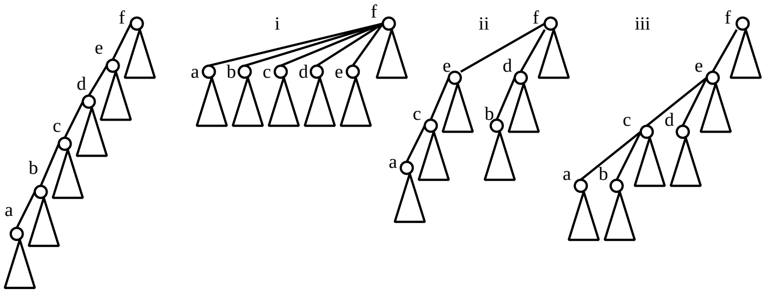

Fig. 3. The compression (i), the splitting (ii), and the halving (iii) of a path a,b,c,d,e,f
.

In [15], Tarjan and Van Leeuwen have conducted a worst-case analysis on the set union problem. In particular, they have shown that *naive linking* coupled with any of the three above described path compression techniques gives a worst-case running time of the set union problem of $\Theta\left(m \log_{(1+m/n)} n\right)$, where m is the number of finds and n is the number of elements, and it is assumed that $m \geq n$.

3.2 Upper Bound

For shortness of space, in this paper we suppose to operate in an environment where the clients work slowly. More precisely, we suppose that between two requests the involved servers have the time to complete all updates of their local tree. This restriction can be easily overcome through the introduction of a suitable lock mechanism [5] providing similar complexity result.

The cost of a sequence of operations is the sum of the cost of each operation. For the complexity analysis we can view insert and search operations as composed by two parts: the request part, possibly with forwarding and LTC messages, and the split part. Then the cost of an operation is the sum of the cost for the request part and of the cost for the split part. The cost of the request part is the number of messages needed to answer it. We do not count the two messages of request from the client and of answer to it, since these are always present in any operation and add only a constant term to the analysis. For the cost of the split part we assume that a *split* takes 4 messages (like in the DRT [7]).

In order to give an upper bound on the complexity of queries on DRT*, we show an equivalence between split trees and the compressed trees used for the set union problem solved by means of *naive linking* coupled with the *compression* technique. In particular we show that there is an equivalence between:

– A server and an element.
– A compressed tree $CT(s)$ with canonical element s and the split tree $ST(s)$ of the server s.
– A $find(s')$, where s' is in the compressed tree $CT(s)$ with canonical element s, and a request (insert or search) pertinent for a server s', where s' is in the split tree $ST(s)$, and arriving to a server s.
– A *make-set*(s') and a *split* of a server s, where s' is the new server.

Please note that in *union*(e, f) with the *naive linking* technique we always make e point to f. We now show that for each sequence of requests in a DRT* there is a specific sequence of *finds*, *make-sets* and *unions* in the set union problem whose complexity bounds the DRT* one.

Let us consider a request arrived at server s and pertinent for s'. This can be a search or an insert of a key in a server s'. We can view this request as the search of the server s' in $ST(s)$ and we call this view *server search*(s', s). Please note that a request and its view as a *server search* in the split tree have the same cost. Therefore, in order to calculate the cost of a sequence of requests in a DRT* we can consider a corresponding sequence of operations in split trees, made up by *server searches* and *splits*, and calculate the cost of this sequence.

Lemma 3. *Let σ be a sequence of requests in a DRT* and σ' the corresponding sequence of splits and server searches in split trees. Let σ'' be a permutation of σ', keeping the relative order between the splits and between the server searches, and such that all the splits are at the beginning of the sequence. Then σ' and σ'' have the same cost.*

Proof. Note that since split is a local operation its cost does not depend on its position in the sequence of operations on the split tree. Moreover its advance in the sequence of operations on the split tree does not affect the cost of any *server searches.*

Lemma 4. *Let σ'' be a sequence of operations in split trees, where all splits are at beginning. The cost of the sequence does not change if the order of two consecutive server searches is inverted.*

Proof. Let $p(x, y)$ be the set of nodes in the path from node x to its ancestor y in a split tree. Let $|p(x, y)|$ be the number of arcs in $p(x, y)$. Let $search(s', s)$ and $search(t', t)$ be two consecutive server searches for server s' (resp. t') in $ST(s)$ (resp. $ST(t)$). If $p(t', t) \bigcap p(s', s) = \emptyset$, then the two server searches do not affect each other and their position can be exchanged. Let us assume $p(t', t) \bigcap p(s', s) = p \neq \emptyset$ and without loss of generality let $t \in p(s', s)$ and $search(s', s)$ precedes $search(t', t)$. Then $search(s', s)$ costs $|p(s', s)|$ and $search(t', t)$ costs $1+|p(t', t)-p|$, because of the compression of path from s' to s in $ST(s)$ and of path p in $ST(t)$. Let us now exchange the position of the two server searches. Now $search(t', t)$ costs $|p(t', t)|$ and $search(s', s)$ costs $1+|p(s', s)-p|$, because of the compression of path from t' to t in $ST(t)$ and of path p in $ST(s)$. In both cases the total cost of the two server searches is the same.

Lemma 5. *Let σ'' be a sequence of operations in split trees, where all splits are at beginning. We say a server $search(s', s)$ in σ'' is of height $h(s)$, where $h(s)$ is the height of s in $ST_0(s_0)$, assigning height 0 to s_0. Let h be the height of the highest server in $ST_0(s_0)$. Let σ''' be a permutation of σ'', obtained through exchanges of adjacent server searches, where all server searches of height k precedes all server searches of height $k-1$, for $k = h, h-1, ..., 1$. Then σ'' and σ''' have the same cost.*

Proof. We reorder σ'' exchanging consecutive server searches. For lemma 4 each exchange does not change the total cost.

We now show how to build a sequence ρ of operations for the set union problem that is equivalent to σ'''. We can write $\sigma''' = \sigma*, \sigma_h, \sigma_{h-1}, ..., \sigma_0$, where $\sigma*$ is the initial sub-sequence of *splits* and σ_k denotes the sub-sequences of *server searches* of height k, for $k = h, h-1, ..., 0$. We start ρ with a sequence of *make set*, each corresponding to a *split* in $\sigma*$. Then we add to ρ a *find* for each *server search* in σ_h. Then for each server s' at height $h-1$ and satisfying the condition that s' is the parent of s'' in $ST(s_0)$ we add to ρ operation $union(s'', s')$. Now, for each $k = h-1, h-2, ..., 1$ we repeat the above process of adding to ρ a *find* operation for each *server search* in σ_k and a *union* for each server at height $k-1$ satisfying the above condition. Finally we add to ρ a *find* for each *server search* in σ_0.

Note that since the sequence σ''' has been reordered according to server heights, each *server search*(s', s) is executed when s satisfies the hypothesis of lemma 2. Hence it is $ST(s) = T_{s_0}(s)$ and the cost of *server search*(s', s) can be analyzed in $ST(s_0)$.

Let $p_{CT}(s', s) = \langle s' = x_1, x_2, \ldots, x_r = s\rangle$ be the path connecting s' to its ancestor s in $CT(s)$. Let $p_{ST}(t', t) = \langle t' = y_1, y_2, \ldots, y_r = t\rangle$ be the path connecting t to its descendant t' in $ST(s_0)$. We say $p_{CT}(s', s)$ and $p_{ST}(t', t)$ are isomorphic if elements x_i corresponds to server y_i for $i = 1, 2, ..., r$.

Lemma 6. *Let* server-search(s', s) *belong to* σ_k *(*$k = h, h-1, ..., 0$*) and* $find(s')$ *be the corresponding operation in* ρ*. Then* $find(s')$ *is executed in a compressed tree* $CT(s)$*, which has the same structure of* $T_{s_0}(s)$*, and after its execution* $CT(s)$ *has the same structure of* $T_{s_0}(s)$ *after the execution of* server-search(s', s).

Proof. Lemma is trivially true for server searches in σ_h, since $CT(s)$ and $T_{S_0}(s)$ are made up only by s.

Let us now assume, by induction, that lemma is true for all $k = h-1, h-2, ..., j$. Let us consider *unions* in ρ following the *finds* corresponding to *server searches* in σ_j. The execution of each of these *union* links an element of the set union problem corresponding to a server s of height $j-1$ to the sets of the set union problem corresponding to servers s' children of s in $ST(s_0)$. The new $CT(s)$ is made up by the compressed trees $CT(s')$ and the arcs (s', s). Then $CT(s)$ has the same structure of $T_{s_0}(s)$ (see figure 4).

We now have to show that after the execution of *server-search*(s', s) belonging to σ_{j-1} and the corresponding execution of $find(s')$ $CT(s)$ has the same structure of $T_{s_0}(s)$.

Since the execution of *server-search*(s', s) in $T_{s_0}(s)$ follows a path $p_{ST}(s', s)$ and the corresponding execution of $find(s')$ follows a path $p_{CT}(s', s)$ in $CT(s)$ and $p_{ST}(s', s)$ and $p_{CT}(s', s)$ are isomorphic, due to the fact that $CT(s)$ and $T_{s_0}(s)$ have the same structure before the execution of the operation, then their executions compresses the two paths in an isomorphic way and we have the thesis.

Theorem 1. *Let* $C(m, n)$ *be the cost in terms of number of messages of a sequence of* m *requests over a DRT* starting with one empty server and with* n *servers at the end. We have:*

$$C(m, n) = O\left(m \log_{(1+m/n)} n\right).$$

Proof. We always have $m > n$, because of the result in section 2.1.

From lemma 6 we have that *server-search*(s', s) and the corresponding $find(s')$ have the same cost. Let C_s be the cost of the $n-1$ *splits* that have produced the n servers. Let C_m be the cost of the n *make-sets*, and C_l be the cost of all the *unions*, which are at most $n-1$. It is $C_s = O(C_m + C_l)$, hence:

$$C(m, n) = O\left(m \log_{(1+m/n)} n\right).$$

Since in DRT* there is a relation between m and n (see section 2.1), namely $n \leq \frac{m}{A}$, then we have:

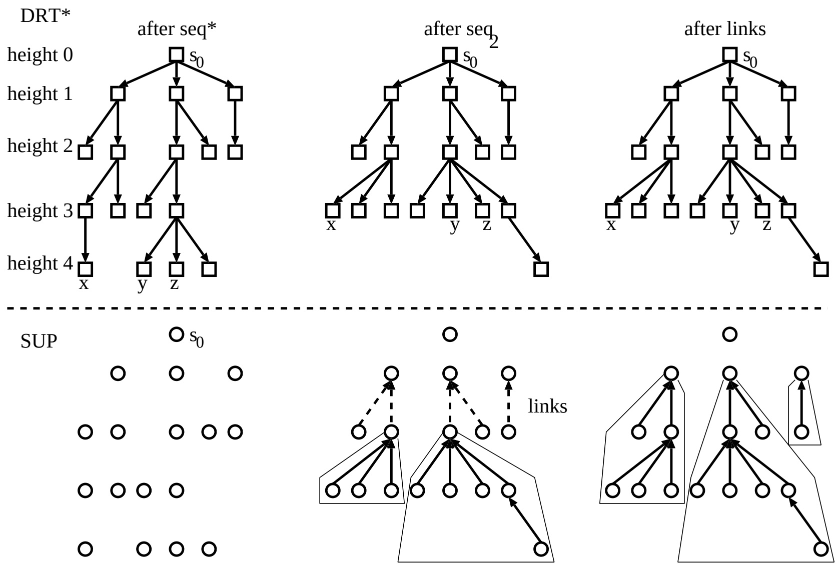

Fig. 4. The evolution of $ST(s_0)$ in the DRT* and the set of compressed trees in the set union problem (SUP) during the server searches in theorem 1.

Corollary 1. *Let $C(m, n)$ the cost in terms of number of messages of a sequence of m requests over a DRT* starting with one empty server and with n servers at the end. We have:*

$$C(m, n) = O\left(m \log_{(1+A)} n\right) .$$

Please note that for $A = 10^3$ we have $\log_{(1+A)} n \leq 4$ for $n \leq 10^{12}$servers. We therefore can assume to have an amortized constant cost in real SDDSs.

4 Extension to the Multi-dimensional Case

In the multi-dimensional case we use as indexing structure a distributed version of k-d tree called *lazy k-d tree*, introduced in [10] and extensively analyzed in [11, 12], with index on clients and servers. The local tree is also a lazy k-d tree.

Therefore for the multi-dimensional case we modify the search process of lazy k-d trees as in the case of DRT*. More precisely, with reference to the figure 1, when a request generates a chain of address error, the pertinent server builds up the correction tree C and sends it within the LTC messages to each server in the chain. In this case C is a connected portion of the overall k-d tree. It contains the whole path from the node associated to s_0 to the one associated to s_k. A server simply adjusts its local tree adding the unknown portion of the tree. The analysis of previous section exactly applies to the multi-dimensional case.

5 Conclusions

We have introduced and analyzed a variant, called DRT*, of the addressing method for SDDSs used in DRT [7]. Our variant, DRT*, has a very good behavior in the amortized case, close to the optimality.

The method is also extendible to the multi-dimensional case, applying the same variation to the lazy k-d tree [10,12].

In particular for a real SDDS (made up by hundreds or thousands of servers) we can assume to have an almost constant amortized cost for the insert and search operations.

To prove the result we used a structural analogy between DRT* and compressed trees used in the set union problem [13,15]. A deeper analysis of this analogy might suggest other protocols, possibly more efficient, for the management of distributed data.

In the k-dimensional case only worst case analysis was previously considered and the almost constant cost for the general k-dimensional case appears to be very promising in the light of well known difficulties in proving optimal worst case bounds for such a case.

References

1. F. Barillari, E. Nardelli, M. Pepe: Fully Dinamic Distribuited Search Trees Can Be Balanced in $O(\log^2 N)$ Time, Technical Report 146, Dipartimento di Matematica Pura ed Applicata, Universita' di L'Aquila, July 1997, accepted for publication on the *Journal of Parallel and Distributed Computation.*
2. Y. Breitbart, R. Vingralek: Addressing and Balancing Issues in Distributed B+-Trees, *1st Workshop on Distributed Data and Structures (WDAS'98)*, 1998.
3. Chorochronos: A Research Network for Spatiotemporal Database Systems. *SIGMOD Record* 28(3): 12-21 (1999).
4. A.Di Pasquale, E. Nardelli: Balanced and Distributed Search Trees, *Workshop on Distributed Data and Structures (WDAS'99)*, Princeton, NJ, May 1999.
5. A.Di Pasquale, E. Nardelli: Design and analysis of distributed searching of k-dimensional data with almost constant costs, Tech.Rep. 00/14, Dept. of Pure and Applied Mathematics, Univ. of L'Aquila, May 2000.
6. B.A. Galler, M.J. Fisher, An improved equivalence algorithm, *Commun. ACM* 7, 5(1964), 301-303.
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. 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.
9. 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.
10. E. Nardelli: Distribuited k-d trees, in *XVI Int. Conf. of the Chilean Computer Science Society (SCCC'96)*, Valdivia, Chile, November 1996.
11. E.Nardelli, F.Barillari and M.Pepe, Design issues in distributed searching of multi-dimensional data, *3rd International Symposium on Programming and Systems (ISPS'97)*, Algiers, Algeria, April 1997.

12. E. Nardelli, F.Barillari, M. Pepe: Distributed Searching of Multi-Dimensional Data: a Performance Evaluation Study, *Journal of Parallel and Distributed Computation*, 49, 1998.
13. R.E. Tarjan, Efficiency of a good but non linear set union algorithm, *J. Assoc. Comput. Mach.*, 22(1975), pp. 215-225.
14. T.Tzouramanis, M.Vassilakopoulos, Y.Manolopoulos: Processing of Spatio-Temporal Queries in Image Databases. ADBIS 1999, pp.85-97.
15. J. Van Leeuwen, R.E. Tarjan, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, 31(1984), pp. 245-281.