An Amortized Lower Bound for Distributed Searching of k-dimensional Data [*]

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

1. Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia. E-mail: nardelli@univaq.it
2. Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italia. – **CONTACT AUTHOR**

Printed on:June 21, 2000

Abstract

In this paper we consider the scalable distributed data structure paradigm introduced by Litwin, Neimat and Schneider and analyze costs for insert, exact and range 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 $O\left(\log_{(1+A)} n\right)$, 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 the order of thousands, we can assume to have a constant cost in the 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, order preserving, message passing environment, multi-dimensional environment, range queries.

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 [15, 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 [9]. The main properties of SDDS paradigm are:

1. Keep a good performance level while the number of managed objects changes.

[*]Research partially supported by the Italian MURST 40% project "Algoritmi, Modelli di Calcolo e Strutture Informative" and by the the European Union TMR project "Chorochronos".

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* [9], RP* [10], DRT [8], lazy k-d-tree [11], 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 [8]. 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

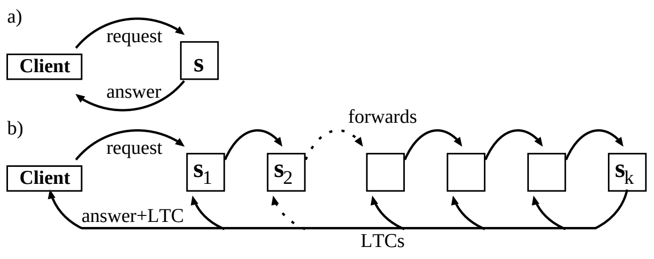


Figure 1: Possible cases of the search process.

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 [8] and therefore we call it DRT*.

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

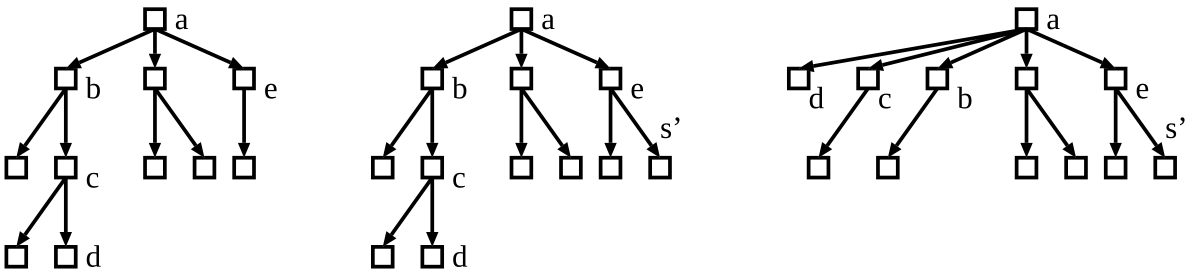


Figure 2: The *splits* build up $ST_0(a)$ (e splits, with s' as new server)(left and center). The effect of a compression after a request pertinent for d and arrived to a (right). $ST(c)$ does not correspond anymore to the sub-tree $T_a(c)$ of $ST(a)$. The same for b.

s of DRT* a distinct 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 (i) the root is s and (ii) 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. In fact, 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 2.1 *Each request arriving to s is pertinent for a server in $ST(s)$.*

Lemma 2.2 *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 how the use of LTC messages reduces the cost of satisfying the request.

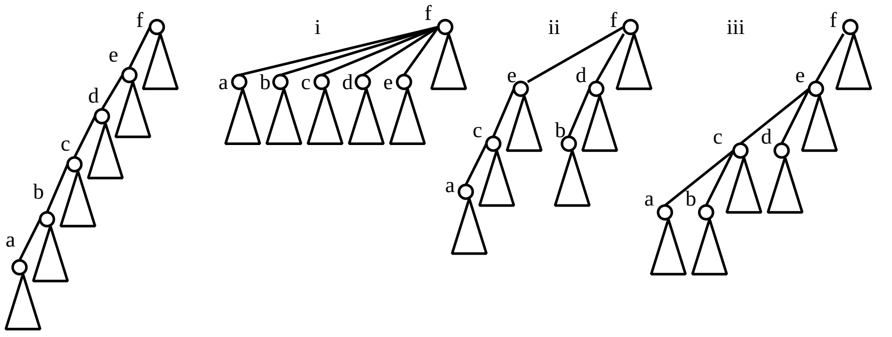


Figure 3: The compression (i), the splitting (ii), and the halving (iii) of a path a,b,c,d,e,f .

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 [14, 16]. 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 [7].

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.* In the rest of the paper we assume that in *union*(e, f) with the *naive linking* technique we always make e point to f.

In [16], 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

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. The cost of a sequence of operations is the sum of the cost of each operation.

Let us assume 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.

Under the previous assumption, in [6] we give an upper bound on the complexity of queries on DRT*, showing 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 the following we recall the main results of this analysis.

Theorem 3.1 *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:*

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

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

Corollary 3.2 *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.

3.3 Lower bound

We now want to show a correspondence between sequences of *finds*, *make-sets* and *unions* in set union problem, and sequences of requests in a DRT*, in order to give a lower bound for the complexity of operations on DRT*.

Let σ be a sequence of m_s *finds*, n *make-sets* and l *unions*, with $l < n$. For the sake of simplicity we assume σ terminates with a single compressed tree $CT(\widehat{s})$.

The corresponding sequence ρ of DRT* operations is made up by two sub-sequences ρ_1 and ρ_2 as it follows. We assume the DRT* starts with one server associated to *make-set*$(\widehat{s})$. ρ_1 is then a sequence building an n-server DRT* by means of a series of inserts producing $n-1$ *splits* and which do not cause any address error. In this way for each element of the set union problem we have a server in DRT*. For each *make-set*(s'), inserts in ρ_1 are used to create a server s' in DRT*. Now two cases are possible: (i) *union*(s', s) exists after *make-set*(s'), (ii) *union*(s', s) does not exist in σ. In the former case we perform the minimum number of inserts over s required to obtain the server s' as a new server from the split of s (see figure 4). In the latter case there is no specific server on which we have to carry out insertions in order to obtain s' from its split: then we can freely choose any of the existing server.

After having built ρ_1, we have "translated" all *make-sets* and *unions* of σ in terms of inserts and *splits*. To build ρ_2 we now have to associate to each *find*(s') in set union problem, where s' is in $CT(s)$, a search operation in the DRT* pertinent for the server s' and arriving to the server s. We can view this operation as a *server-search*(s', s) over a split tree, without affecting the resulting complexity.

At the end we obtain a sequence ρ_1 with m_i inserts without address errors and a sequence ρ_2 of m_s *server-searches*.

Clearly ρ_1 is a legal sequence, from the point of view of building a DRT*, since it is made up by just inserts without address errors and *splits*. We now discuss the legality of ρ_2.

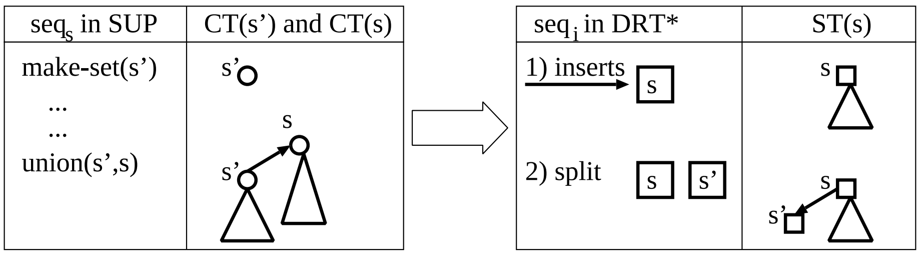


Figure 4: To a *make-set*(s') and a following *union*(s', s) in set union problem (SUP) we make to correspond inserts in DRT* that cause the split of s, with s' as new server.

Lemma 3.3 *Let us consider a* server-search(s', s) *in* ρ *corresponding to a* $find(s')$ *in* σ *performed over* $CT(s)$*.* server-search(s', s) *is legal with respect to the DRT* built by* ρ_1 *in the sense that it is correct that a request pertinent for server* s' *arrives to server* s*.*

Proof.

Assume $s = s'$. Then *server-search*(s', s) in ρ correspond to a request for s' without address errors. This is clearly a correct operation.

Assume now $s \neq s'$. *server-search*(s', s) has been generated by a *find*(s') that has followed a path in $CT(s)$. This means that in σ there is *union*(s_0, s_1), *union*(s_1, s_2), .., *union*(s_{k-1}, s_k), where $s' = s_0$ and $s_k = s$. But each of this *unions* corresponds to a *split* in the DRT* built by ρ. By considering all these *splits* together it is easy to check that s' manages a sub-interval of the initial interval of s. Then it is correct that a request for s' arrives to s.

■

To analyze equivalence between *server-searches* and *finds*, we proceed in two steps for clarity of presentation. First we prove the equivalence for the first *find* in σ. Then we generalize the result to a generic occurrence of *find* in σ.

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 3.4 *Let* find(s') *be the first* find *in* σ*. Let* $p_{CT}(s', s)$ *be the path followed in* $CT(s)$ *by* find(s') *and let* $p_{ST}(s', s)$ *be the path followed in* $ST(s)$ *by the corresponding* server-search(s', s)*. Then* $p_{CT}(s', s)$ *and* $p_{ST}(s', s)$ *are isomorphic.*

Proof.

Let k be the position of *find*(s') in σ. Two cases are possible: (i) only *make-sets* precede *find*(s') in σ, (ii) *make-sets* and *unions* precede *find*(s') in σ.

Case (i). Let *make-set*(t) be the operation in position $k - 1$ in σ. If $s' = t$ then *find*(s') is executed in $CT(s')$ and it follows a path of zero arcs. Its corresponding operation in ρ is *server-search*(s', s'), following as well a path of zero arcs in $ST(s')$. If $s' \neq t$ we can neglect the $(k-1)$-th operation since it does not affect the path followed by *find*(s') and we apply again the previous arguments to the $(k-2)$-th operation until we arrive to operation *make-set*(s'), which is at latest the first operation in σ.

Case (ii). The proof is by induction on the number of *unions* preceding *find*(s') in σ. Let us assume only one *union*(t', t'') exists in σ before *find*(s'). If the $(k - 1)$-th operation is a *make-set*, using the same arguments as in case (i) we either prove the thesis or apply again the analysis to the $(k - 2)$-th operation. In going backwards in σ we therefore arrive sooner or later to *union*(t', t''). If $s' \neq t'$ we can neglect *union*(t', t'') since it does not affect the path followed by *find*(s') and by apply again the same arguments as above we prove the thesis. If $s' = t'$, then the path $p_{CT}(t', t'')$ followed by *find*(s') is made up by one arc linking t' to t'' in $CT(t'')$. This is clearly isomorphic to $p_{ST}(t', t'')$ in $ST(t'')$ and the thesis is proved.

Let us assume now that n *unions* precede *find*(s') in σ and that by induction the thesis holds for the first $n-1$ *unions* preceding *find*(s') in σ. Let *union*(t',t'') be the n-th *union*. Moreover let *union*(t',t'') be the operation preceding *find*(s') to which we arrive going backwards in σ on the basis of the same arguments used above (in the other cases we have the above results). If t' is not an ancestor of s' and $s' \neq t'$, then *union*(t',t'') does not affect the path followed by *find*(s') and by the inductive hypothesis the thesis is proved. If t' is not an ancestor of s' and $s' = t'$, then we can apply the same arguments used for the base case of the induction. If t' is an ancestor of s' then $p_{CT}(t',t'')$ is made up by $p_{CT}(s',t')$ and the arc linking t' to t''. By the inductive hypothesis we have $p_{CT}(t',t'')$ is isomorphic to $p_{ST}(t',t'')$.

■

Lemma 3.5 *Let $p_{CT}(s',s)$ be the path followed in $CT(s)$ by a* find(s') *and let $p_{ST}(s',s)$ be the path followed in $ST(s)$ by the corresponding* server-search(s',s). *Then $p_{CT}(s',s)$ and $p_{ST}(s',s)$ are isomorphic.*

Proof.

By lemma 3.4 the thesis is true for the first *find* in σ.

Let us assume by induction the thesis holds for the first n *finds* in σ. Let *find*(s') be the $n+1$-th *find* and let k be its position in σ.

For the proof we have to consider three cases.

(i) The $(k-1)$-th operation is a *make-set*(t'). If $s' = t'$ we can apply the analysis of lemma 3.4. If $s' \neq t'$ we can neglect the $(k-1)$-th operation and apply again the analysis to the $(k-2)$-th operation.

(ii) The $(k-1)$-th operation is a *find*(t') executed in a $CT(t'')$. If $s' = t'$ by the inductive hypothesis the thesis is true. If $s' \neq t'$ then we have two sub-cases:

(a) $s' \notin CT(t'')$; then $p_{CT}(s',s)$ is not changed by the execution of *find*(t') and we can neglect the $(k-1)$-th operation and apply again the analysis to the $(k-2)$-th operation.

(b) $s' \in CT(t'')$; In this case it is $t'' = s$. Let $\widehat{t}$ be the lowest ancestor of s' lying on $p_{CT}(t',t'')$. By inductive hypothesis $p_{CT}(t',t'')$ is isomorphic to $p_{ST}(t',t'')$ and $p_{CT}(s',\widehat{t})$ is isomorphic to $p_{ST}(s',\widehat{t})$. After the compression executed by *find*(t'), $\widehat{t}$ is a direct son of t'' both in $CT(t'')$ and in $ST(t'')$ (see figure 5). Therefore the path followed by the execution of *find*(t') both in $CT(t'')$ and in $ST(t'')$ is made up by one arc linking $\widehat{t}$ to its father t'' plus the path linking s' to $\widehat{t}$. By the inductive hypothesis we therefore have $p_{CT}(s',t'')$ is isomorphic to $p_{ST}(s',t'')$.

(iii) The $k-1$-st operation is *union*(t',t''). If t' is an ancestor of s' then $p_{CT}(s',t'')$ is made up by $p_{CT}(s',t')$ and the arc linking t' to t''. By the inductive hypothesis we have $p_{CT}(s',t'')$ is isomorphic to $p_{ST}(s',t'')$. In the other cases we can apply the analysis of lemma 3.4.

■

Let $C_s(m_s,n)$ the cost of sequence σ of m_s *finds* and n *make-sets* in the set union problem. Let C_i be the cost of the initial m_i inserts in ρ_1 and $C(m,n)$ be the cost of sequence ρ of $m = m_s + m_i$ requests in the DRT*. Then:

Lemma 3.6 *It is: $C(m,n) > C_s(m_s,n) + C_i$*

Proof.

By construction the cost of ρ is made up by the sum of the cost of ρ_1 and ρ_2. The first term is clearly C_i. For the second term note that by the lemma 3.5 each *server-search* corresponding to a *find* has the same cost of the *find*. In fact the cost for both operations is two times the length of the path followed in the compressed tree or in the split tree.

Moreover, for each *make-set* we have a *split*, and each *split* has a cost greater than 2 messages. Therefore the total cost of *splits* cover the cost of all *make-sets* and *unions* (with n *make-set* we have at most $n-1$ *unions*). Then the cost of the sequence of m_s *finds* and n *make-sets* and u *unions* has a cost smaller than the relative cost of the corresponding m_s *server-searches* and $n-1$ *splits* in the DRT*.

■

Please note that to obtain n servers starting with one empty server, we have to carry out a sequence of $b+1$ inserts to split the first server, and at least other $\frac{b}{2}(n-2)$ inserts over the servers with $\frac{b}{2}+1$

	SUP
	CT(s)
	find(s')
		DRT*
		ST(s)
		server-search(s',s)

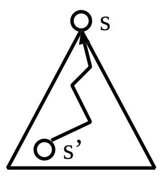
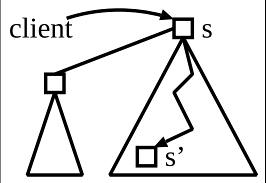
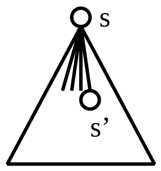
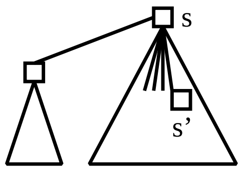
before		
after		

Figure 5: A *find* in set union problem(SUP) and a corresponding *server-search* in the DRT*.

keys or at most other $\left(\frac{b}{2}+1\right)(n-2)$ inserts over the servers with $\frac{b}{2}$ keys, to obtain the other *splits*. In total $\frac{b}{2}n+1 \le m_i \le \left(\frac{b}{2}+1\right)n-1$. We perform each insert without address error, and with a cost of 2 messages for each insert, we have $C_i = 2m_i$.

Theorem 3.7 *If $m > 2m_i$, then:*

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

Proof.
From [16] we have that if it is $m_s > n$, then we have $C_s(m_s,n) = \Omega\left(m_s \log_{(1+m_s/n)} n\right)$. From the hypothesis, since $m = m_s + m_i$, we have $m_s > \frac{m}{2}$. Given the result in lemma 3.6, we have:

$$C(m,n) > C_s(m_s,n) + C_i > C_s(m_s,n) = \Omega\left(m_s \log_{(1+m_s/n)} n\right).$$

Note that $\log_{(1+m_s/n)} n > \log_{(1+m/n)} n$, because $m_s < m$. Then:

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

.

Since $m_s > \frac{m}{2}$, we have:

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

■

From theorem 3.1 and theorem 3.7, we directly obtain:

Corollary 3.8 *If $m > 2m_i$, then:*

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

Note that the hypothesis $m > 2m_i$ means that each key inserted into the DRT* should be searched on the average at least once.

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 [11] and extensively analyzed in [12, 13], 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 Extensions

The set union study suggests other heuristics to manage a DRT* other than the *Compression* commonly used in the DRT and lazy k-d tree. For example the *Splitting* heuristic is easily implementable in the search process of a DRT*. In the search process corresponds to the following protocol:

Let us assume that a search operation has to follow a path from the servers $s_1, .., s_k$. When a server s_2 receives a routing message from a server s_1, it routes to s_3 and sends to s_1 the message with its local tree. The same is performed by the other servers. No local tree LT is built by the final pertinent server s_k.

The splitting heuristic keeps the same complexity of the compression one in the DRT*, but it is more indicated for example in an high concurrent system. In fact in this case our requirement on the time between two consecutive requests arriving at the same server become $T_R \geq 2D$, in fact, the answer to a server that has routed a request arrives just from the server destination of the routing message. Therefore the lock time of a server drastically decreases.

An analogous DRT* extension could be performed for the *Halving* heuristic.

6 Conclusions

We have introduced and analyzed a variant, called DRT*, of the addressing method for SDDSs used in DRT [8]. 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 [11, 13].

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 [14, 16]. 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, submitted for publication.

[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. 00/12, Dept. of Pure and Applied Mathematics, Univ. of L'Aquila, March 2000.

[6] A.Di Pasquale, E. Nardelli: Distributed searching of k-dimensional data with almost constant costs, *ADBIS-DASFA 2000*, Praha, September 2000.

[7] B.A. Galler, M.J. Fisher, An improved equivalence algorithm, *Commun. ACM* 7, 5(1964), 301-303.

[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] E. Nardelli: Distribuited k-d trees, in *XVI Int. Conf. of the Chilean Computer Science Society (SCCC'96)*, Valdivia, Chile, November 1996.

[12] 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.

[13] E. Nardelli, F.Barillari, M. Pepe: Distributed Searching of Multi-Dimensional Data: a Performance Evaluation Study, *Journal of Parallel and Distributed Computation*, 49, 1998.

[14] R.E. Tarjan, Efficiency of a good but non linear set union algorithm, *J. Assoc. Comput. Mach.*, 22(1975), pp. 215-225.

[15] T.Tzouramanis, M.Vassilakopoulos, Y.Manolopoulos: Processing of Spatio-Temporal Queries in Image Databases. ADBIS 1999, pp.85-97.

[16] J. Van Leeuwen, R.E. Tarjan, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, 31(1984), pp. 245-281.