A Lower Bound for the DRT* with Complete Correction Technique [*]

ADRIANO DI PASQUALE
Dipartimento di Informatica, Università di L'Aquila, Italy and Istituto Zooprofilattico Sperimentale dell'Abruzzo e del Molise "G. Caporale"

ENRICO NARDELLI
Dipartimento di Matematica, Università di Roma "Tor Vergata", and Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR, Italy

GUIDO PROIETTI
Dipartimento di Informatica, Università di L'Aquila, Italy and Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR, Italy

Abstract

The DRT* is an order preserving Scalable Distributed Data Structures with an almost constant amortized upper bound costs for exact searches and insertions. The result is based on the correction techinque the DRT* uses when a given request produces an address error. This technique mainly consists in exchanging information among servers about the distribution of data by means of messages. Servers can exchange the maximum (*complete* correction technique) or the minimum (*restricted* correction technique) information they know.

Here, we investigate the amortized lower bound of such distributed searching technique, proving that there is not any advantage in using complete correction technique, because the lower bound result of the complete correction technique is the same of the restricted one.

[*]This work has been partially supported by the CNR-Agenzia 2000 Program, under Grants No. CNRC00CAB8 and CNRG003EF8, and by the Research Project REAL-WINE, partially funded by the Italian Ministry of Education, University and Research.

1 Introduction

In this paper we investigate amortized lower bounds for the distributed searching technique used in DRT* and its variants. The DRT* is an order preserving Scalable Distributed Data Structures (SDDS) [5] able to deal with insertions and exact searches both for the 1-dimensional and the k-dimensional case with an almost constant amortized upper bound costs.

In [2] we analyzed the amortized behavior of DRT* and we proved that a sequence σ of m requests of intermixed exact-searches and insertions over a DRT* starting with one empty server and ending with n servers has a cost of $C(m,n) = O\left(m \log_{(1+m/n)} n\right)$ messages. Such a result is obtained by adapting some of the techniques developed for the solution of the Set Union Problem [7].

The same structural analogy between DRT* and Set Union Problem allows to prove a lower bound for the cost of the sequence σ [1],i.e.,

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

Note that the latter result was based on a DRT* version using a restricted correction rule in the distributed searching process, and it was an open problem whether it holds for other correction rules (details about correction technique are presented in Section 2.3).

Both the upper bound and the lower bound results are based on the use of an analytical tool called *Split Tree*, used to keep track of the cost in term of messages of the requests of σ.

Here, we define a new analytical tool called *Reverse Split Tree* (*RST*). Using the RST we analyze the cost of σ on a DRT* version using the complete correction rule of the distributed searching process. In particular, we are able to prove that the lower bound for such DRT* version is the same of the restricted one.

In Section 2 and in Section 3 we review basic concepts on DRT* and results from Set Union Problem, respectively. In Section 4 we present the algorithm and the complexity analysis. Last Section concludes the paper.

2 Distributed Search Trees

Here we review the main concepts relative to DRT*, in order to prepare the way for the presentation of our result. Servers manage data in buckets, clients manipulate data performing insertions and exact searches. Other operations, like deletions, range searches and so on, are not considered in the paper.

2.1 Bucket management

The protocol of a server managing a bucket is common to all the proposals on distributed search trees. Each server s manages a unique *bucket* of keys. The bucket has a fixed capacity b. We define $I(s)$ to be the interval of keys managed by s, meaning that a key $k \in I(s)$ should be searched or should be inserted in s. We define s "to be in overflow" or "to go in overflow" when it manages b keys and one more key is assigned to it. When s goes in overflow it starts the *split* operation. It requests the address of a new fresh server s_{new} to a special site called *split coordinator*. Whenever s receives the address of s_{new}, it sends to s_{new} half of its keys.

After a split, s and s_new manage $\frac{b}{2}$ and $\frac{b}{2}+1$ keys, respectively. Moreover, $I(s)$ is divided into two sub-intervals: I and I_{new}, with $I(s) = I$ and $I(s_{new}) = I_{new}$.

2.2 Local tree

Each client c and server s has a local indexing structure, called *local tree* ($lt(c)$ and $lt(s)$, respectively) to avoid them to make *address errors* (i.e., they send a request to a wrong server). Whenever a client performs a request and makes an address error, it receives, together with the answer, information to correct its local tree. This prevents a client to commit the same address error twice.

From a logical point of view the local tree is an incomplete collection of associations $\langle s, I(s)\rangle$ identifying a server s and the managed interval of keys $I(s)$. The local tree of a client can be wrong, in the sense that in the reality 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.

In a DRT*, 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 it performs it. 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 sequence of address errors 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.

2.3 Correction technique

In order to minimize the number of address errors, the following correction technique is applied during the search process: whenever a server s is not pertinent for a request and has to forward it to another server, it adds to the message to send some information taken from its local tree and then it sends the message. The pertinent server extracts from the request the information added by the servers which have been traversed by the request. It builds up a correction tree C aggregating the received information and its own one. Finally, it sends a *Local Tree Correction* (LTC) message containing C to the client and to all the servers involved in the request, so to allow them to correct their local trees.

In particular, we define a correction technique to be *restricted* if the information added by each server s is $\langle s, I(s) \rangle$. Otherwise, we define a correction technique to be *complete* if the information added by each server s is $lt(s)$.

2.4 Split tree

Let T be a DRT*. 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 with each server s of T a rooted tree $ST(s)$, called the *split tree* of s (Figure 1.a shows a split tree). 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_2 is in the local tree of s_1. When a split occurr in T, the structure of split trees changes (for example, in Figure 1.b, the split of server e adds the node s' and the arc (e, s') in $ST(s)$).

In the same way, if we consider the correction of local trees, the structure of the split tree of s changes. Indeed, due to the correction, after a

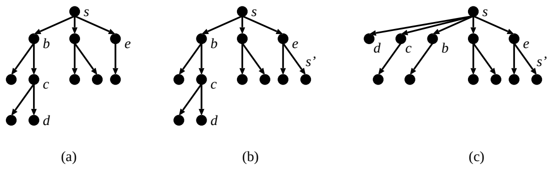

Figure 1: (a) The split tree $ST(s)$. (b) Server *e* splits, with s' as new server. (c) The effect of a compression after a request pertinent for *d* and arrived to *s*.

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 1.c).

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

3 Results from Set Union Problem

The set union is a classical problem that has been deeply analyzed [7]. It is the problem of maintaining a collection of disjoint sets of elements under the operations of find and 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.

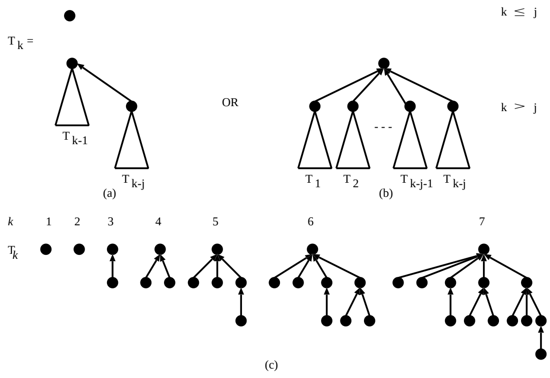

Figure 2: (a),(b) Recursive definitions of T_k. (c) Examples of T_k trees.

To make *finds* possible, each set is represented 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. These trees are named *compressed trees*.

To carry out *find*(e) we follow parent pointers from e until the root, which is then returned. While traversing parent pointers, one can apply some techniques for compressing the path from the elements to the root: compression, splitting, and halving. To carry out union various techniques can be applied: naive linking, linking by rank and linking by size (see [7] for details).

For the sake of the presentation, we provide a number of definitions.

Consider a Tree T. Let $r(T)$ be the root node of T. We denote with $a \rightarrow b$ or $b \leftarrow a$ that b is the father node of a. Let T, T', T'' be trees. We denote $T = T' \leftarrow T''$, if $r(T) = r(T')$ and $r(T') \leftarrow r(T'')$. Moreover, we denote $T = T' \uparrow T''$, if $r(T) \leftarrow r(T')$ and $r(T) \leftarrow r(T'')$.

In [7], Tarjan and Van Leeuwen have conducted a worst-case analysis on the Set Union Problem. In particular, our proof is based on their lower bound results for the instances of Set Union Problem where *finds* are carried out with any compression technique and *unions* are carried out with naive linking.

To get the lower bound result, the class of compressed trees T_k is defined. Figure 2-a and Figure 2-b show the recursive definitions of a T_k

tree for $j \geq 2$. Figure 2-c shows examples of T_k trees with $j = 2$.

Main properties of the class of T_k trees are the following:

Definition 1 $T_k = T_{k-1} \leftarrow T_{k-j}$ *(see Figure 2.a).*

Definition 2 $T_k = T_1 \uparrow T_2 \uparrow ... \uparrow T_{k-j-1} \uparrow T_{k-j}$ *(see Figure 2.b).*

Lemma 1 *If we link a tree containing a single node with a T_k tree and then perform j compressions, we obtain a new T_k tree with an extra node.*

In the following, we report the main lower bound result for the Set Union Problem (from [7]).

Theorem 1 *Let S be a generic instance of Set Union Problem where* finds *are carried out with any compression technique and* unions *are carried out with naive linking.*

The time of a sequence ρ *of m* finds *and n* unions *on S is*

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

Proof. We obtain bad examples as follows: Suppose $m \geq n \geq 2$. Let $j = \lfloor m/n \rfloor$, $i = \lfloor log_{j+l}(n/2) \rfloor + 1$, and $k = ij$. Build a T_k tree. Note that $|T_k| \leq (j+1)^{i-1} \leq n/2$. Repeat the following operations $\lfloor n/2 \rfloor$ times: Link a single-node tree with the existing tree, which consists of T_k with some extra nodes. Then perform j finds, each traversing a path of $i+1$ nodes, to reproduce T_k with some extra nodes. There are at most m finds, and the total number of nodes on find paths is at least $j\lfloor n/2 \rfloor (i+1) = \Omega\left(m \log_{1+m/n} n\right)$. □

In [1, 2], we proved that a sequence ρ of m *finds* and n *unions* in the Set Union Problem can be associated to a sequence σ of m requests of intermixed exact-searches and insertions over a DRT*, and vice-versa. This has been possible showing the relation between split trees on DRT* and compressed trees on Set Union Problem.

In particular, this relation has been used to prove that a sequence σ of m requests of intermixed exact-searches and insertions over a DRT* starting with one empty server and ending with n servers has a cost of $C(m,n) = \Omega\left(m \log_{(1+m/n)} n\right)$ messages. But, the result holds only in case the restricted correction technique is used in DRT*.

4 The Reverse Split Tree

The *Reverse Split Tree* (*RST*) of a server s, say $RST(s)$, is a virtual structure similar to $ST(s)$. While $ST(s)$ shows the path of a request starting from s, on the contrary, a path from a node t to s in $RST(s)$ describes a request arriving to t and pertinent for s, $\forall t \in RST(s)$.

We define such a request *server search* and we denote that with $srch(t,s)$. The definitions of nodes and arcs in a RST are the same as of split trees.

The following algorithm in pseudo code describes how to build up $RST(s)$ from a given configuration of a DRT* T of n nodes.

```
ALGORITHM: RST_BUILD(s)
$RST(s) = \{s\}$;
$N^* = \{s_0, \ldots, s_{n-1}\}$;
extract $s$ from $N^*$.
while ( $N^* \neq \emptyset$ )
    {
    extract $t \in N^*$ from $N^*$.
    foreach( $r \in RST(s)$ )
        {
        if( $r \in lt(t)$ and $I(r) = \min\{I(r')|r' \in lt(t)\ \&\&\ I(s) \subseteq I(r')\}$ )
            {
            add node $r$ in $RST(s)$;
            add arc $(t,r)$ in $RST(s)$;
            }
        }
    }
```

Figure 3 shows a possible configuration of a DRT* T, $ST(s_0)$ and $RST(s_4)$.

In the following, we show how it is possible to build up a RST isomorph to a T_k tree for a given k. We calculate the cost in terms of messages of such procedure. Finally, we use the result in [7] and [1] to prove the main result.

Lemma 2 *Let us consider $RST(s_0)$ with a generic shape T. There exists a sequence of requests which creates an RST isomorph to a tree $T_k \leftarrow T$. (see Figure 4.a).*

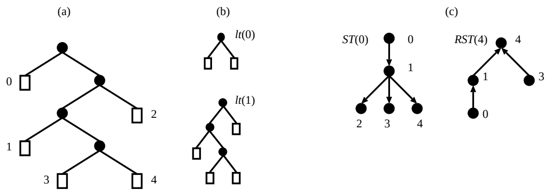

Figure 3: (a) A configuration of a DRT* T: the distributed virtual tree; (b) local trees of servers s_0 and s_1; (c) $ST(s_0)$ and $RST(s_4)$.

Proof. The result holds for $k \leq j$ with a split of $r(T)$ (Figure 4.b).

For $k = j+1$, we perform 2 splits producing $a \leftarrow b \leftarrow r(T)$. After that, a $srch(r(T), a)$ produces $RST(a) = T_{j+1} \leftarrow T$.

Suppose now by induction to build a RST $R = T_{k-j} \leftarrow T$. By the inductive hypothesis we can build up a new RST $R' = T_{k-j-1} \leftarrow R$. For the same reason, we can build up a new RST $R'' = T_{k-j-2} \leftarrow R'$.

Continuing with this procedure, we can build up a $RST = T_1 \leftarrow T_2 \leftarrow ... \leftarrow T_{k-j} \leftarrow T$.

After that, we perform a further split of $r(T_1)$, with a as new server, producing $RST(a) = a \leftarrow T_1 \leftarrow T_2 \leftarrow ... \leftarrow T_{k-j} \leftarrow T$ and a $srch(r(T_{k-j}), a)$.

For the definition 2, $RST(a) = T_k \leftarrow T$. (see Figure 4.e and 4.f). □

Lemma 3 *Consider a compressed tree belonging to the class of T_k trees, with $j \geq 2$. There exists a sequence of requests which creates an RST isomorph to T_k.*

Proof. Consider a $RST(s_0)$ made up by the unique server s_0. The result holds for $k \leq j$ without any request.

In the case $j+1 \leq k \leq 2j$, a chain of $k-j$ splits creating servers $s_1, ..., s_{k-j}$ is performed, such that $s_0 \leftarrow s_1 \leftarrow ... \leftarrow s_{k-j}$. After that, a $srch(s_0, s_{k-j})$ creates $RST(s_{k-j})$ isomorph to T_k.

Suppose by induction that a sequence of requests builds up a RST made up by a tree T_{k-1}. Hence, the same applies for a $T_i k-j$ tree.

For the lemma 2, it is possible to build up a RST isomorph to $T_{k-1} \leftarrow T_{k-j}$, and hence, isomorph to T_k. □

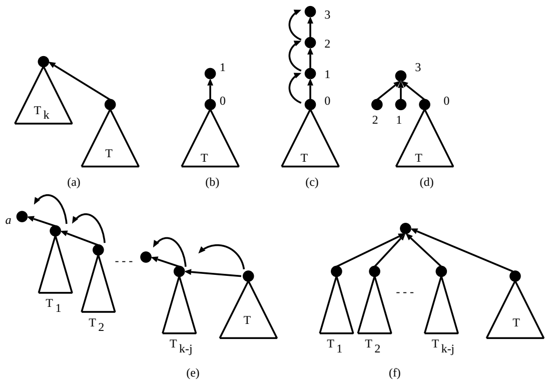

Figure 4: Evolution of lemma 2.

Lemma 4 *The number of messages needed to create an RST isomorph to T_k is $O(k)$.*

Proof. Let $C(k)$ be the number of messages need to create an *RST* isomorph to T_k. For the definition 1, $T_k = T_{k-1} \leftarrow T_{k-j}$, and, hence, $T_{k-1} = T_{k-2} \leftarrow T_{k-1-j}$.

Following the procedures to build up T_k of lemmas 3 and 2, we build up the tree $T_{k-2} \leftarrow T_{k-1-j} \leftarrow T_{k-j}$, and we perform the last $srch(r(T_{k-j}), r(T_{k-2}))$, that costs 2 messages.

Hence, we can state that $C(k) = C(k-2) + C(k-1-j) + C(k-j) + 2$ The solution of this recursive function is just $C(k) = O(k)$. □

A simpler formula for $C(k)$ can be derived thinking that one message of the last *server_search* has been used to build up T_{k-1} and one message to build up T_k. Hence, $C(k) = C(k-1) + C(k-j) + 1$.

We now provide an example of the described procedure. Suppose we want to build up a *RST* with the shape of T_6, with $j = 2$. Consider the following sequence of requests on a DRT* T starting with the unique server s_0: a set of inserts creates s_1 from the split of s_0, and s_2 from the split of s_1. Now, $RST(s_2) = s_2 \leftarrow s_1 \leftarrow s_0$ (see Figure 5.a).

A $srch(s_0, s_2)$ modifies the shape of $RST(s_2)$ to the one of Figure 5.b.

Servers s_3 and s_4 are created by a chain of splits starting from s_2. A

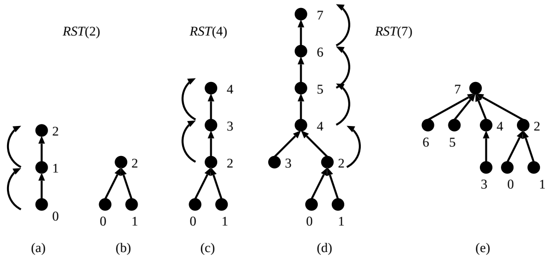

Figure 5: A sequence of requests on DRT* builds up $RST(s_7)$ with the shape of T_6.

$srch(s_2, s_4)$ is performed (see Figure 5.c).

Servers s_5, s_6, s_7 are created by a chain of splits starting from s_4. Please note that now $RST(s_7) = T_1 \leftarrow T_2 \leftarrow T_3 \leftarrow T_4$ (see Figure 5.d). A $srch(s_2, s_7)$ is performed, producing $RST(s_7)$ with the shape of T_6 (see Figure 5.e).

From the previous lemma and from the results in [1] and [7], the following theorem holds.

Theorem 2 *Let T be a DRT* with a complete correction technique, starting with one empty server and ending with n servers. Then, the number of messages of a sequence σ of m requests made up by intermixed inserts and exact searches over T is*

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

5 Conclusions

We have analyzed amortized lower bound for distributed searching technique used in DRT and its variants, where the *complete* correction technique is used. Previously, lower bound results were presented only for the DRT* version with the *restricted* correction technique [1].

From an implementation point of view, we recall that the complete technique is very expensive in terms of size of messages, with respect to the restricted one, since, a server always adds its entire local tree to a forwarding message. Our result suggests us that using such an expensive technique does not help to improve the overall performance.

A similar analysis can be performed on the variant of DRT*, like the ones defined in [3, 4], showing that also in those cases the bounds are tight.

Our proof is based on the structural analogy between DRT* and compressed trees used in the set union problem [6, 7]. A deeper analysis of this analogy suggests other protocols, in some cases more efficient, for the management of distributed data.

References

[1] A. Di Pasquale, E. Nardelli: An Amortized Lower Bound for Distributed Searching of k-dimensional data, *Workshop on Distributed Data and Structures (WDAS 2000)*, L'Aquila, Proceedings in Informatic 9, pp. 71-85, Carleton Scientific, June 2000.

[2] 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.

[3] A. Di Pasquale, E. Nardelli, G. Proietti: An Improved Upper Bound for Scalable Distributed Search Trees, *Workshop on Distributed Data and Structures (WDAS 2002)*, Paris, Proceedings in Informatic 14, pp. 15-28, Carleton Scientific, February 2002.

[4] A. Di Pasquale, E. Nardelli, G. Proietti: An RP* Extension with Amortized Almost Constant Costs, *Workshop on Distributed Data and Structures (WDAS 2003)*, Thessaloniky, to be published by Carleton Scientific, June 2003.

[5] W. Litwin, M.A. Neimat, D.A. Schneider: LH* - Linear hashing for distributed files, *ACM SIGMOD Int. Conf. on Management of Data*, pp. 324-336, Washington, D. C., 1993.

[6] R.E. Tarjan, Efficiency of a good but nonlinear set union algorithm, *J. Assoc. Comput. Mach.*, 22,2 (1975), pp. 215-225.

[7] J. Van Leeuwen, R.E. Tarjan, Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.*, 31,2 (1984), pp. 245-281.