An RP* Extension with Almost Constant Amortized Costs *

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 RP^*_s is an order preserving Scalable Distributed Data Structure (SDDS) ables to manage exact searches and insertions with a cost of $O(\log_{\lfloor f/2 \rfloor} n)$ messages in the worst case, where n is the final number of servers and f is a large value. Unfortunately, the RP^*_s presents the same logarithmic costs for both the operations in the amortized case.

On the contrary, in the DRT*, another order preserving SDDS, exact searches and insertions have linear cost in the worst case, but in the amortized case they have a cost of $O(\alpha(m,n))$ messages, where m it the number of intermixed exact-searches and insertions, and $\alpha(m,n)$ is the classic inverse of the Ackermann function.

In this paper, we propose an extension of the RP^*_s, named RP^+_s, coupling the B^+-tree based technique of the RP^*_s with a variant of the DRT* correction technique. The result is that an exact-search or insertion in the RP^+_s has a worst-case cost of $O(\log_{\lfloor f/2 \rfloor} n)$ messages, and an amortized cost of $O(\alpha(m,n))$ messages.

1 Introduction

The Scalable Distributed Data Structures (SDDS) paradigm [9] is used to define access methods specifically designed to satisfy the high performance requirements of a Multi-computers environment made up by a large number of computers connected through a high speed network.

*This work has been partially supported by the Research Project GRID.IT, funded by the Italian Ministry of Education, University and Research, and by the Research Project REAL-WINE, partially funded by the Italian Ministry of Education, University and Research.

An access method based on the SDDS paradigm has to be *dynamic*: it has to expand to new servers, but only when already used servers are efficiently loaded. Moreover, it has to be *scalable*: it has to keep the same level of performances while the number of managed objects increases.

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.

Hashing based SDDSs (e.g., LH* [9]), while allow to achieve worst-case constant cost for exact searches and insertions, namely 4 messages, they do not support efficiently operations like range search, nearest neighbor search, the search of the minimum or the maximum and so on. For such operations they have a worst case cost of $O(n)$ messages, where n is the number of servers in the structure.

This motivates the study of order preserving structures [1, 2, 3, 4, 5, 8, 10]. One of the prominent order preserving SDDSs is the *Range-Partitioning** (RP_s^*) [10]. It uses a B^+-tree based technique to organize data. More precisely, in the RP_s^*, there are two kind of servers: (i) the *data servers*, where data are stored, corresponding to the leaves of the virtual B^+-tree created by the RP_s^* technique, and (ii) the *index servers*, where routing information are stored, corresponding to the internal nodes of the virtual B^+-tree. An exact search and an insertion in the RP_s^* has a cost of $O(\log_{\lfloor f/2 \rfloor} n)$ messages in the worst case, where n is the number of data servers and f is the fan-out of the index servers. However, the drawback of the RP_s^* is its behaviour in the amortized case. Indeed, a sequence σ of m requests of intermixed exact-searches and insertions over a RP_s^* starting with one empty server and ending with n servers has a cost of $O\left(m \cdot \log_{\lfloor f/2 \rfloor} n\right)$ messages.

On the contrary, in the *Distributed Random Tree** (DRT*) [5], an efficient variant of the DRT presented in [8], the cost of σ is $O(m \cdot \alpha(m,n))$ [6], where $\alpha(m,n)$ is the classic inverse of the Ackermann function, while in the worst case the cost for exact-searches and insertions is linear. Moreover, another drawback of such a structure is that it requires heavy lock mechanisms after a split. In fact, a logarithmic number of servers has to be locked in the worst case.

In this paper we propose an extension of RP_s^*, named RP_s^+. The extension is based on the B^+-tree based RP_s^* technique coupled with a variant of the DRT* correction technique. Using an RP_s^+, a sequence of m requests of intermixed exact-searches and insertions starting with one empty server and ending with n servers has a cost of $C(m,n) = O(m \cdot \alpha(m,n))$ messages, and at the same time any operation has a cost of $O\left(\log_{\lfloor f/2 \rfloor} n\right)$ messages in the worst case. Due to the well known slow growth of the function $\alpha(m,n)$, we can assume to have $C(m,n) \simeq O(m)$ in realistic scenarios of SDDS made up by thousands or even millions of servers. Moreover, the lock mechanisms after a split are basically the ones defined for the RP_s^*, where, at each moment, always a constant (low) number of servers has to be locked. Hence, this structure presents a very good amortized behaviour and it is well suited for high concurrency systems.

The paper is organized as follows: In Section 2 we review basic concepts of distributed search trees, in Section 3 we present the technique and the complexity analysis. Finally, Section 4 concludes the paper.

2 The Correction Technique of the DRT*

In this section we review the main concepts relative to distributed search trees and to the correction technique of *local trees* of servers used in DRT*.

2.1 Bucket Management

The protocol of a server managing a bucket is common to all the proposals on distributed search trees. Each server manages a unique *bucket* of keys. The bucket has a fixed capacity b. We define a server "to be in overflow" when it manages b keys and one more key is assigned to it. When a server s is 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 manages $\frac{b}{2}$ keys and s_{new} manages $\frac{b}{2}+1$ keys. It is easy to prove the following property:

Lemma 1 *Let σ be a sequence of m intermixed insertions and exact searches. Then we can have at most $\lfloor \frac{2m}{b} \rfloor$ splits.*

2.2 The Local Tree

Clients have a local indexing structure, called *local tree*. The local tree $LT(c)$ of a client c is needed to avoid clients to make address errors. Whenever a client performs a request which results in an *address error*, (i.e., it sends the request to a wrong server), 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 $\langle$*server, interval of keys*$\rangle$ managed internally with any data structure: list, tree, etc. For example, an association $\langle s, I(s) \rangle$ identifies 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.

Now, we sketch the DRT* correction technique. 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 this is satisfied. 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 to it the request.

Since s' can be not pertinent as well, it might forward the request to still another server. In general, we can have a series 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. Moreover, s_k receives the local trees of the servers s_1,s_2,..,s_{k-1} which have been traversed by the request. It first builds up a correction tree C aggregating the local trees received and its own one, and then sends an *Index Correction Message* (ICM) containing C to the client and to all servers s_1,s_2,..,s_{k-1}, so to allow them to correct their local trees.

2.3 The Split Tree

Here, we sketch the analytical tool used in DRT* to analyze the amortized cost of exact searches and insertions. A variant of this tool will be used to analyze the RP$_s^+$.

Let T be a DRT*. From the above description of the local trees and how they change due to the distribution of information about the overall structure through ICM 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 in T occurs, 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. 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 1.c).

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

3 The RP$_s^+$ Technique

3.1 The Data Structure

Let T be an RP$_s^*$ made up by *n data servers*. Let Σ be the set of index servers with data servers as children (see Figure 2.a), and let $n' = |\Sigma|$. Clearly, $n \geq \lfloor f/2 \rfloor \cdot n'$, where f is the fan-out of index servers.

Following the RP$_s^*$ technique, each data server d stores a pointer to its parent

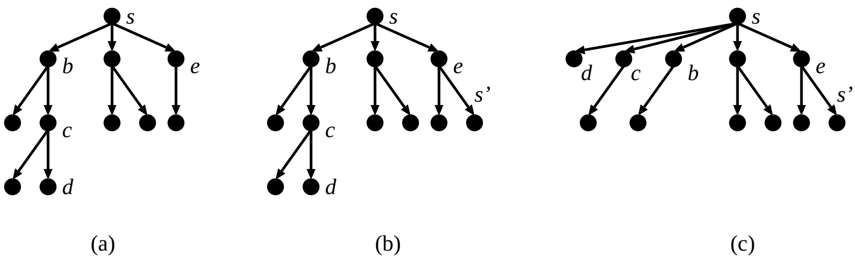

Figure 1: The split tree $ST(s)$ (a). Server e splits, with s' as new server (b). The effect of a compression after a request pertinent for d and arrived to s (c).

index server, say $pp = pp(d)$. Each index server s stores a pointer to its parent index server, say $pp = pp(s)$, and $\lfloor f/2 \rfloor \leq n_s < f$ child pointers $cp_1, ..., cp_{n_s}$ ($cp_i = cp_i(s)$, for each $1 \leq i \leq n_s$). The set of child pointers of s is indicated with $CP(s)$. If $s \in \Sigma$ then cp_i is a data server for each $1 \leq i \leq n_s$, otherwise it is an index server. With each child pointer cp_i, the associated Interval of data domain $I(cp_i)$ is also stored. If s is the root index server, then $pp(s)$ is a null pointer. With each data server d, the interval $I(d)$ of the managed data domain is associated. For each index server s, the interval $I(s)$ is defined as the union of the intervals $I(sp), \forall cp \in CP(s)$.

The RP_s^+ needs the following further structures. Each index server s stores a local tree $LT(s)$ as described in section 2.2.

3.2 Evolution of the File

The evolution of the file through splits of buckets follows the basic RP_s^* technique. As in the RP_s^*, we do not consider deletions. We recall that the technique creates a virtual B^+-tree, where data servers are the leaves and index servers are internal nodes (for further details, see [10]). In particular, index servers in Σ are internal nodes directly connected to the leaves (see Figure 2.a).

When a data (index) server s splits, a new data (index) server t is created. With respect to the virtual B^+-tree, t is a sibling of s. This split will be notified to $pp(s)$. The latter adds t to $CP(pp(s))$. Possibly, $pp(s)$ may also split.

If s was a data server, the new data server t receives half of the keys stored at s, while if s was an index server, the new index server t receives half of the pointers in $CP(s)$, and sets $LT(s)$ equal to the set of child pointers and associated intervals, i.e. $\langle cp, I(cp)\rangle \in LT(t), \forall cp \in CP(t)$.

When t adds a new child pointer to $CP(t)$, due to a split in the system, the related information is added to $LT(t)$. After that, if the set of child pointers has to be split with another index server, $LT(t)$ remains unchanged.

Notice that, due to the split of index servers technique, the following lemma holds:

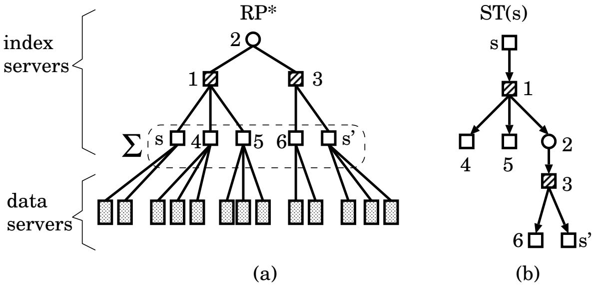

Figure 2: The virtual B$^+$-tree built up by the RP$^*_s$ technique (a). An initial configuration of split tree $ST(s)$ (b). In the example of figure, the fan-out of index servers is $f = 4$.

Lemma 2 *Let s be an index server and $s \in \Sigma$. After a split of s, it will still be $s \in \Sigma$.*

The initial configuration of the RP^+_s is made up by an index server r and a data server d_0. Clearly, $r = pp(d_0)$, $d_0 = cp_1(r)$ and $LT(r) = \{\langle d_0, [-\infty, \infty]\rangle\}$.

Each client c manages a local tree $LT(c)$ in the standard fashion. A new client c sets $LT(c) = \{\langle d_0, [-\infty, \infty]\rangle\}$.

3.3 The Search Process

A client c issues a request for a key k sending a *request R*-message m_r. As usual, the request may arrive to a wrong data server d. The server d simply forwards the request to $s = pp(d)$, sending a copy of m_r.

When a server $s \in \Sigma$ receives a forward from a client, it has to reach the correct server $s' \in \Sigma$, which has the pertinent server d for the request as a child pointer. If $s = s'$, s' simply send the request to d. Suppose $s \neq s'$. The idea is to apply a correction technique similar to the one of DRT* to the tree made up by internal nodes of virtual B$^+$-tree. Hence, what s tries to do is to use $LT(s)$ to find s'. Server s sends a *local tree LT*-message to the index server s'' corresponding to the correct server from the $LT(s)$ point of view. If $s'' = s'$, then s'' sends the request to d.

But, as usual, $LT(s)$ can be not up-to-date. Therefore, if $s'' \neq s'$, then s'', differently from s, does not use $LT(s'')$ to find the correct server. Instead, it decides to follow the structure of the virtual B$^+$-tree to find s'. Hence, it sends a *forward F*-message either to the parent index server or to a child index server, according to the intervals of data domain of $pp(s')$ and to the servers in $CP(s')$. Eventually, after a chain of F-messages, s' is reached and the request is sent to d. After that, s' sends a *correction ICM*-message to all index servers involved in the search process and to the client. Each *ICM*-messages contains a collection of servers involved in the search process and the corresponding set of child pointers. Each index server receiving a *ICM*-message, aggregates the contained information in its local tree in a DRT* fashion.

Here, we provide a formal description of all possible cases of the search process. The search process always starts with a m_r *R*-message from a client c. The message m_r contains a key k to be searched or to be inserted.

Let us consider the search process cases from a data server s point of view. A data server s can only receive a request m_r. If it is pertinent for the request (i.e., the key k of m_r is such that $k \in I(s)$), s executes the request and answer to the client c. Otherwise, it sends a copy of m_r to the index server $s' = pp(s)$. Notice that $s' \in \Sigma$.

Let us consider the search process cases from a client point of view.

1. A client c may decide to perform an exact search or an insert for a key k. The client looks for the pertinent server in $LT(c)$. Let s be the resulting index or data server, i.e., in $\langle s, I(s) \rangle \in LT(c)$ and $k \in I(s)$. The client s creates a *R*-message m_r including k in it. Finally, c sends m_r to s.

2. A client c can receive a *ICM*-message m_c as an answer to a m_r previously sent by c itself. The message m_c contains the aggregated tree T_a made up by a set of $CP(s)$, where s is an index server involved in the search process. The client c extracts T_a from m_c and uses it to update $LT(c)$.

Let us now consider the search process cases from a server point of view.

1. The server $s \in \Sigma$ receives a *R*-message m_r. The sender can be either a client or a data server. If m_r is pertinent for a $d \in CP(s)$, s sends the m_r to d; otherwise, s looks for the correct server in $LT(s)$. Let s' be the resulting index server. The server s creates a LT-message with m_r included and sends it to s'.

2. The server $s \notin \Sigma$ receives a *R*-message m_r. The sender has to be a client. The server s looks for the correct server in $LT(s)$ and sends a LT-message as in the previous case.

3. The server $s \in \Sigma$ receives a LT-message m_l. If m_r contained in m_l is pertinent for a $d \in CP(s)$, s sends m_r to d; otherwise s creates a F-message m_f and includes m_l (hence, m_r is also included) and $CP(s)$ in m_f. Finally, s sends m_f to $pp(s)$.

4. The server $s \notin \Sigma$ receives a LT-message m_l. The server s creates a F-message m_f and includes m_l (hence, m_r is also included) and $CP(s)$ in m_f. After that, s evaluates the key k contained in m_r. If $k \in I(cp)$ for a given $cp \in CP(s)$, then it sends m_f to sp, otherwise it sends m_f to $pp(s)$.

5. The server $s \notin \Sigma$ receives a F-message m_f. The server s creates a new F-message m'_f and includes m_f (hence, m_r is also included) and $CP(s)$ in m'_f. After that, s evaluates the key k contained in m_r. If $k \in I(cp)$ for a given $cp \in CP(s)$, then it sends m'_f to sp, otherwise it sends m'_f to $pp(s)$.

6. The server $s \in \Sigma$ receives a F-message m_f. Due to the correctness of the RP^*_s search process, s has to be the correct servers. Hence, s evaluates the key k contained in m_r, finding an $cp \in CP(s)$ such that $k \in I(cp)$. Then s sends m_r to sp. After that, it extract the addresses of all the servers $s_1, ..., s_r$ involved in the search process from m_f. Notice that they are all senders of F-messages $m_1, ..., m_{r-1}$ and of LT-message included in m_f. Moreover it extract all the corresponding $CP(s_1), ..., CP(s_r)$. It creates an aggregated tree T_a with $CP(s_1), ..., CP(s_r)$ and $CP(s)$. Finally, s creates a ICM-message m_c. In this message it includes T_a and sends it to $s_1, ..., s_r$ and to the client c.

7. The index server s receives an ICM-message m_c. It extract T_a from m_c and uses it to update $LT(s)$.

3.4 The Variant of Split Tree Model

Our goal is to calculate the cost of a sequence σ of m requests made up by intermixed inserts and exact searches over the RP^+_s starting with one empty server and ending with n data servers and $n' = |\Sigma|$. Due to the fact that n and n' are related, we first concentrate on the virtual B^+-tree structure made up by index servers.

As in the DRT*, we use *split trees* (see section 2.3) to take into account the cost of σ. However, in this case we have to consider a variant of split trees model. We associate a split tree $ST(s)$ with each index server s. Each node of a split tree is an index server. Data servers are not considered in the analysis. The root of $ST(s)$ is the server s. Moreover:

- a *simple* $pp-$arc (s', s'') in $ST(s)$ means that $s'' = pp(s')$;
- a *simple* $LT-$arc (s', s'') in $ST(s)$ means that $s'' \in LT(s')$;
- a *simple* $CP-$arc (s', s'') in $ST(s)$ means that $s'' \in CP(s')$;
- a *compound* arc (s', s'') in $ST(s)$ connects a server $s' \in LT(s)$ and a server s'' unknown to s (i.e., $s'' \notin LT(s)$, $s'' \notin CP(s)$ and $s'' \neq pp(s)$). This means that s' can reach s'' following a path of virtual B^+-tree $(s' = s_0, s_1, ..., s_p = s'')$, where, at least, $s_1 = pp(s')$ and each server s' in the path but s'' is already known by s (i.e., $s' \in LT(s)$ or $s' \in CP(s)$ or $s' = pp(s)$).

A compound arc (s', s'') is in $ST(s)$ because previously s has reached s'. The path $(s' = s_0, s_1, ..., s_p = s'')$ is the one the search process has to visit in the virtual B^+-tree, through F-messages as described in the case 5 and 6 of the search process.

Any simple arc has cost 1. The cost of the compound arc is defined as the length p of the path.

Here below, given a configuration of the RP_s^+, we show how the corresponding split trees can be built.

Let us consider an index server s. We want to build up $ST(s)$. The node s is the root of $ST(s)$. Moreover:

1. a $pp-$arc (s, s_p) is in $ST(s)$, where $s_p = pp(s)$;
2. a $LT-$arc (s, s_l) is in $ST(s)$, for each $s_l \in LT(s)$;
3. a $CP-$arc (s, s_s) is in $ST(s)$, for each $s_s \in CP(s)$;
4. a compound arc (s_l, s_x) is in $ST(s)$, for each $s_l \in LT(s)$, $s_x \notin LT(s)$, following the definition of compound arcs;
5. let S_x be the set of nodes s_x; from now on we continue to build $ST(s)$ using the algorithm below:

While S_x is not empty, repeat the following steps.

- Extract a node $s_x \in S_x$. Remove s_x from S_x.
- If s_x is not the root of the RP_s^+ and $pp(s_x)$ is not already in $ST(s)$, add the $pp-$arc $(s_x, pp(s_x))$ to $ST(s)$. Insert $pp(s_x)$ to S_x.
- If $s_x \notin \Sigma$, add the $CP-$arc (s_x, s_s) to $ST(s)$, for each $s_s \in CP(s_x)$. Insert s_s into S_x.

Let (s', s) be a $LT-$arc and (s', t) be a compound arc of cost C. Let $s' = s_0, s_2, ..., s_C = t$ be the path associated with (s', t). We define $T_c(t)$ the subtree of $ST(s)$ rooted at t.

An initial split tree $ST(s)$, where s has never sent a LT-message is shown in Figure 2.b. An evolution of $ST(s)$ from this initial configuration is shown in Figure 3. Here, the configurations of the RP_s^+ is shown on the first row, the configurations of $ST(s)$ is shown on the second row and the path associated to the compound arc of $ST(s)$ is shown on the last row.

The evolution of $ST(s)$ we now describe starts from the configuration of $ST(s)$ shown in Figure 2.b. Figure 3.a shows $ST(s)$ after a request pertinent for s' and arrived to s. The server s' splits and t is the new index server. A compound arc (s', t) is added to $ST(s)$, the related path of such an arc is $(s', 3, t)$, and $T_c(t)$ is made up by just the node t, i.e., the height is $h = 0$ (Figure 3.b).

The server t splits two times. Servers 7 and 8 are the new index servers. After the last split of t, $pp(t) = 3$ has to split. Server t' is the new index server. The

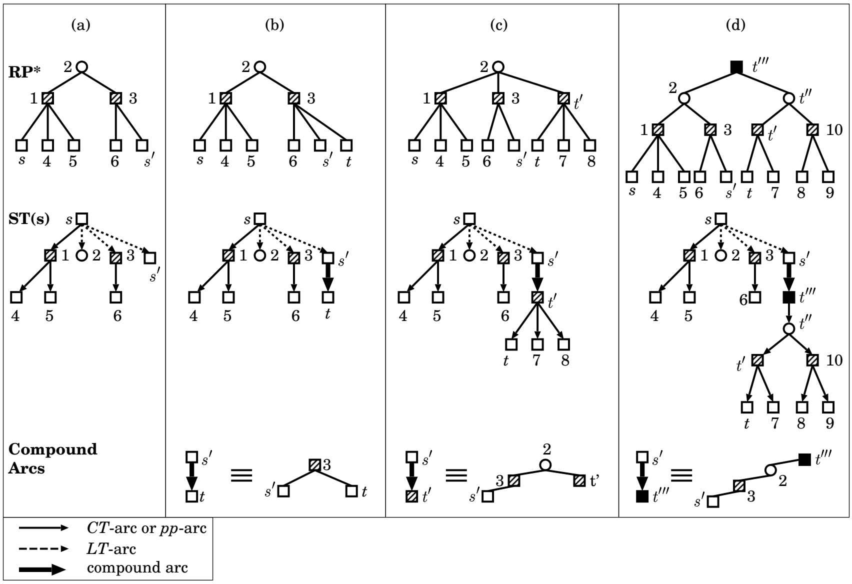

Figure 3: An evolution of $ST(s)$. After a request pertinent for s' and arrived to s (a). The server s' splits and t is the new server (b). The split of t (c). A split causes the creation of a new root (d).

compound arc (s',t) and $T_c(t)$ are removed from $ST(s)$. A compound arc (s',t') is added to $ST(s)$, the related path of such an arc is $(s',3,2,t')$, and $T_c(t')$ is made up by nodes $\{t',t,7,8\}$, hence the height is $h=1$. (Figure 3.c).

Other splits causes the split of t' and the root of the RP_s^+. Consequently, the server $t''=pp(t')$ and a new root $t'''=pp(t'')$ are created. The compound arc $(s',t'$ and $T_c(t')$ are removed from $ST(s)$. A compound arc (s',t''') is added to $ST(s)$, the related path of such an arc is $(s',3,2,t''')$, and $T_c(t''')$ has height $h=3$. (Figure 3.d).

3.5 Amortized Analysis

We first calculate the cost $C(m,n')$ of a sequence σ of m requests made up by intermixed inserts and exact searches over an RP_s^+ starting with one empty server and ending with n' index servers in Σ, considering the virtual B^+-tree made up by index servers. Some preliminary definitions and results are needed.

Let us consider a split tree $ST(s)$. Consider a path p connecting s to a node $s' \in ST(s)$. We define the *length* of p as the sum of the costs of each arc in p. We define the *height* H of $ST(s)$ as the length of the longest path from s to a node $s' \in ST(s)$.

Lemma 3 *Let $ST(s)$ be the split tree of an index server s and let s' be a server in $ST(s)$. In the path connecting s to s' there is at most one compound arc.*

Proof. The result follows directly from the definition of $ST(s)$. □

Lemma 4 *Let $ST(s)$ be the split tree of an index server s. Consider a request arriving to s and pertinent to a server s'. The servers visited in the search process of such a request are the servers in the path between s and s' in $ST(s)$.*

Proof. The result follows directly by definition of the search process, simple arcs and compound arc and from Lemma 3. □

Lemma 5 *Let $ST(s)$ be the split tree of an index server s, (s',t) be a compound arc of cost C and h be the height of $T_c(t)$. It is $C \leq h+2$.*

Proof. Suppose $T_c(t)$ is made up by just t (see Figure 3.b). This means that, in the past and completely unknown to s, s' has split, a new server t has been created in the structure, t has been added to $CP(pp(s'))$ and $pp(s')$ has not split. Moreover, $pp(s')$ is known by s by definition of compound arc and $T_c(t)$.

In this case the path associated with (s',t) is $(s',pp(s'),t)$, hence, $C = 2 = 0+2 = h+2$.

Suppose by induction the lemma is true for other $n \geq 1$ splits in $T_c(t)$ and a new split occurs in $T_c(t)$. If $pp(s')$ does not split, the lemma holds.

Suppose $pp(s')$ splits. A new server t', sibling of $pp(s')$, is introduced in the structure. First of all, suppose $pp(t')$ does not split. Two cases are possible: $pp(t') = pp(pp(s'))$ is either (i) known or (ii) not known by s.

Case (i). Due to definitions, the compound arc (s',t') and $T_c(t')$ are added to $ST(s)$. Please note, that $T_c(t')$ is made up by servers t' as root and servers in $CP(t')$. If the split is such that $t \in CP(t')$, the arc (s',t) and $T_c(t)$ are removed from $ST(s)$. The path related to (s',t') is $(s',pp(s'),...,pp(t') = t')$, that is, $h+3$, but now, the height of $T_c(t')$ is $h+1$. Hence, in this case the lemma holds (see Figure 3.c).

Case (ii). Consider the path $s' = s_0,...,s_r = t'$ and let $0 \leq i < r$ be the first index such that s_{i-1} is known by s and s_i is not known by s. Due to definitions, the compound arc (s',s_i) and $T_c(s_i)$ are added to $ST(s)$. If the split is such that $t \in CP(t')$, the arc (s',t) and $T_c(t)$ are removed from $ST(s)$. Notice that (s',s_i) has a cost less than (s',t), while the height of $T_c(s_i)$ is greater than the height of $T_c(t)$. Hence, in this case the lemma holds.

In case $pp(t')$ splits, case (i) and (ii) applies to $pp(pp(t'))$.

The same arguments can be easily applied to other possible splits from $pp(pp(t'))$ to the root of virtual B$^+$-tree (for instance, Figure 3.d shows a case (ii) when $pp(t')$ splits). □

Lemma 6 *Let $ST(s)$ be the split tree of an index server s. If the correction technique compresses x servers to s, then the requests costs $O(x)$.*

Proof. The result follows from Lemma 5 and from lemma 4.

In particular, considering the search process, in the worst case the cost of the request is 2 messages from a data server where the request is arrived and to the pertinent data server. Moreover, there can be one LT-message, $2x$ F-messages and $2x$ ICM-messages. Hence the cost is $4(x+1)$ messages. □

Lemma 7 *Let h be the height of virtual B^+-tree made up by index servers and let H be the height of $ST(s)$ of an index server s. It is $H = 2h+1$ in the worst case.*

Proof. In the worst case, a request can cause a LT-message, and an upwards chain of at most h F-messages followed by a downwards chain of at most h F-messages. From Lemma 4, the result holds. □

From previous lemma, we have directly:

Lemma 8 *Let H be the height of $ST(s)$ of an index server s. It is $H = 2\log_{\lfloor f/2 \rfloor}(n') + 1$ in the worst case, where f is the fan-out of index servers and $n' = |\Sigma|$.*

Lemma 9 *Let T be an RP_s^+ starting with one empty server and ending with n' index servers in Σ. The number of messages of a sequence σ of m requests made up by intermixed inserts and exact searches over T is $C(m, n') = O(m \cdot \alpha(m, n'))$.*

Proof. From Lemma 4 and lemma 6, we can apply the technique presented in [5]. In this technique, it is shown that a request arrived to s and pertinent to s' can be seen as the *server search* of s' following the path from s to s' in $ST(s)$, hence the sequence σ can be seen as a sequence of σ' of server searches and splits in the split trees related to server of SDDS. The other important result is that, given σ', it is possible to build a sequence ρ of finds, make-sets and unions for the set union problem, such that the cost of σ', and hence of σ, in terms of number of messages is bounded by the cost of ρ in terms of number of steps of an algorithm for the set union problem. Moreover, from Lemma 8 and from the result in [13] the result holds. □

We are now ready to prove the main results of the paper.

Theorem 1 *An exact search or an insertions in an RP_s^+ costs $O(\log_{\lfloor f/2 \rfloor} n)$ in the worst case, where f is the fan-out of index servers and n is the number of data servers.*

Proof. Directly from Lemma 8 and considering that $n \geq \lfloor f/2 \rfloor \cdot n'$, where $n' = |\Sigma|$. □

Theorem 2 *Let T be an RP_s^+ starting with one empty server and ending with n servers. The number of messages of a sequence σ of m requests made up by intermixed inserts and exact searches over T is*

$$C(m,n) = O(m \cdot \alpha(m,n)).$$

Proof. From Lemma 9 and considering that $n \geq \lfloor f/2 \rfloor \cdot n'$ the result holds, where f is the fan-out of index servers and $n' = |\Sigma|$ and therefore, $\alpha(m,n') \leq \alpha(m,n)$. □

3.6 Final Comparisons and Extensions

From a practical point of view, an implementation of RP_s^* with index servers having a large fan-out may have a behaviour not worse than the one of RP_s^+. But the real advantages of our proposal lie in two aspects:

1. our structure has a theoretical upper bound for its behaviour in the amortized case which is better than RP_s^* one;
2. for a good practical behaviour of RP_s^* a large fan-out of index servers is required, while for our structure the good behaviour in practice is guaranteed by the theoretical almost constant amortized upper bound.

The technique can be easily extended to consider the correction technique applied to the whole virtual B^+-tree and not only to the tree made up by internal nodes. The only requirement is that data server has to store a local tree. In this case the fixed messages from a data server to the index and vice-versa are avoided.

The technique can be easily applied to any tree structure using a balancing technique based on split of internal nodes, causing the growth of the tree toward the root. If the height of the tree is $O(\log n)$ in the worst case, then, applying the correction technique, the same amortized result holds. For example, the technique can be applied to the trees defined in [7].

Moreover, we can apply the same extension to the k-RP_s^* [11], obtaining a structure able to manage multi-dimensional data with the same amortized results.

4 Conclusions

In this paper, we extended the RP_s^* technique, defining the RP_s^+. The basic RP_s^* has an amortized logarithmic cost for exact-searches and insertions. With our extension, a sequence of m requests of intermixed exact-searches and insertions over a RP_s^+ starting with one empty server and ending with n servers has a cost of $O(m \cdot \alpha(m,n))$ messages, where $\alpha(m,n)$ is the classic inverse of the Ackermann function. Due to the well known slow growth of the function $\alpha(m,n)$, we can assume to have amortized constant costs for inserts and exact searches in realistic scenarios of SDDS made up by thousands or even millions of servers. The same approach can be used to extend k-RP^*_s, obtaining a structure able to manage multi-dimensional data and with good performance for multi-keys requests, typical of order preserving SDDS. Moreover, the lock mechanisms after a split are basically the ones defined for RP^*_s. Hence, this structure is also well suited for high concurrency systems.

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] A.Di Pasquale, E. Nardelli: Fully Dynamic Balanced and Distributed Search Trees with Logarithmic Costs, *Workshop on Distributed Data and Structures (WDAS'99)*, Princeton, NJ, May 1999.

[4] 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, Carleton Scientific, June 2000.

[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, G. Proietti: An Improved Upper Bound for Scalable Distributed Search Trees, *Workshop on Distributed Data and Structures (WDAS 2002)*, Paris, Carleton Scientific, February 2002.

[7] G.N. Frederickson: A Data Structure for Dynamically Maintaining Rooted Trees. *Journal of Algorithms*, 24(1), pp.37-65, July 1997.

[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*, pp. 327-336, 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*, pp. 342-353, 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*, pp. 120-131, December 1996.

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

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

Adriano Di Pasquale is with the Dipartimento di Informatica, Università di L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italy and with the Istituto Zooprofilattico Sperimentale dell'Abruzzo e del Molise "G. Caporale". E-mail: dipasqua@univaq.it

Enrico Nardelli is with the Dipartimento di Matematica, Università di Roma "Tor Vergata", Via della Ricerca Scientifica, 00133 Roma, Italy, and with the Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italy. E-mail: nardelli@univaq.it

Guido Proietti is with the Dipartimento di Informatica, Università di L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italy and with the Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italy. E-mail: proietti@univaq.it