Scalable Distributed Data Structures: A Survey*

ADRIANO DI PASQUALE
University of L'Aquila, Italy

ENRICO NARDELLI
University of L'Aquila and Istituto di Analisi dei Sistemi ed Informatica, Italy

Abstract

This paper reviews literature on *scalable* data structures for searching in a distributed computing environment. Starting with a system where one server manages a file of a given size that is accessed by a specific number of clients at a specific rate, a *scalable distributed data structures* (SDDS) can efficiently manage a file that is n times bigger and accessed by n times more clients at the same per-client rate, by adding servers and distributing the file across these servers. We analyze and compare SDDS proposals based on hashing techniques and order preserving techniques. Work regarding the management of multi-dimensional data is also reported. Moreover, issues such as high availability and load control are considered.

Keywords

scalable distributed data structure, message passing environment, high availability

1 Introduction

A network of computers is an attractive environment for many applications, and in particular for the ones with high performance requirements. Among the motivations usually cited for the use of a distributed system there are: ease of expansion, increased reliability, the ability to incorporate heterogeneous resources, and resource sharing among autonomous sites.

When a data structure is managed by more than one of the processors on a network one usually speaks of "distributed data structure". In this area a large number of solutions have been proposed. A main partition among the proposals can be defined on the basis of whether the number of processors managing the data structure is fixed or not. For a fixed number of processors, the proposed data

*Research partially supported by the European Union TMR project "Chorochronos" and by the Italian MURST COFIN project "REACTION: Resource Allocation in Computer Networks".

structures have a focus not only on efficient data access, but on a combination with other relevant issues such as, e.g., load balancing and concurrency control. In this paper we focus instead on the case of a variable number of processors, since this is an essential ingredient to obtain a *scalable distributed data structure* (SDDS). The main objective of an SDDS is in fact to accommodate dynamic file growth with scalable performance, where the key to scalability is the dynamic distribution of a file across multiple servers of a distributed system. Furthermore, an SDDS has a focused view on efficiency considerations, disregarding other aspects completely.

Consider a file that contains a set of records and is managed by a single server at a single node of a computer network, accessed by a number of clients at fixed per-client rate. Ideally, when the file grows by a factor of n and the number of clients also increases by the same factor, we should be able to scale up the system throughput (i.e. to serve n times more clients), without any noticeable degradation of the system performance, by redistributing the file across k servers. This redistribution of data should take place continuously as the file is growing. Servers are created dynamically on demand and fragments of the file are redistributed among them. The widely used technique is to split a server's data into two halves and migrating one half onto a new server. We assume it is always possible to find a new fresh server to involve in the management of the data structure whenever it is necessary.

As perfect scalability (for non-trivial workloads) is achievable only theoretically, we usually speak of a scalable approach already if response time of the system is nearly a constant for reasonably large values of n and increases only very slowly for very large values of n.

Let us formalize the basic environment we are considering. The distributed system is composed by a collection of processing sites, interconnected by a communication network. The data is taken from a domain D. Each data item $d \in D$ consists of two main fields, $d = (Key_d, Record_d)$, where Key_d is a *key* taken from a mono-dimensional or a multi-dimensional domain, and $Record_d$ is a *record* field containing the relevant data. A distributed data structure is composed of a *data organization scheme*, specifying a collection of local data structures storing copies of data items at various sites in the system, coupled with a set of distributed *access protocols* that enable processors to issue modification and query instructions to the system and get appropriate responses. Data are organized in buckets. We assume that each site manages exactly one bucket.

Communication among sites happens by sending and receiving messages. A message can be of the following types:

- point-to-point message. Such messages have one sender site and a unique receiver site.
- multicast message. Such messages have one sender site and many receiver sites. In general, the set of receiver sites of a multicast message corresponds to a subset of all the sites of the structure. In our case, the set of receiver

sites of a multicast message corresponds to the set of server sites of the structure.

We concentrate mainly on *dictionary* structures, which support exact searches, inserts and deletes, and typical extensions supporting range queries, partial queries, nearest neighbor queries and so on.

In particular, many of the proposed SDDSs only consider insertions and search operations. Papers explicitly discussing and analyzing deletions are [6, 20, 1]. In the rest of the paper whenever we consider a request for a single key, we mean an insert or an exact search.

1.1 Performance measures

The main measure of performance for an operation is its *communication complexity*. In the context of SDDS this is defined as the number of messages exchanged between clients and servers to perform a given operation. For this complexity measure, the following assumptions hold:

- The real network topology is not relevant. The graph associated with the communication network is a complete graph. This allows to measure the communication complexity in terms of the number of exchanged messages.
- Each message costs one unit and the size of a message is not relevant.
- The network is free of errors. Both nodes and links never crash. This hypothesis is relaxed in the high availability schemes.

Another relevant complexity measure is the *global load factor*, formally defined as $\alpha = \frac{x}{bn}$, where x is the number of data items stored in the overall structure, b is the capacity of a bucket (equal for all the buckets) and n is the number of servers (we recall that each server manages exactly one bucket).

In addition, the *global* and *local overhead* (introduced in [30]) give a measure on how large is the waste of computational resources deriving from the fact that servers are distributed over a communication network. They are defined as follows:

- *local overhead* (*local_ovh*) measuring the average fraction of *useless* messages that each server has to process; this is expressed by the average, over all servers, of the ratio between the number of useless messages and the number of messages received by a server. A message is useless for a server if it is received by it but it is not pertinent to it.
- *global overhead* (*global_ovh*) measuring the average fraction of *overhead* messages traveling over the network; this is expressed by the ratio between

the overall number of overhead messages and the overall number of requests. A message is considered to be overhead if it is not a query message issued by a client.

The mathematical definition of these two parameters is now provided. Let us denote with n the overall number of servers, and with $rec\_msg(i)$ and $pert\_msg(i)$, the number of messages received by server i and the number of pertinent messages received by server i, respectively. Then, we have:

$$local\_ovh = \frac{\sum_{i=1}^{n} \frac{rec\_msg(i) - pert\_msg(i)}{rec\_msg(i)}}{n}$$

$$global\_ovh = \frac{\sum_{i=1}^{n} rec\_msg(i) - \sum_{i=1}^{n} pert\_msg(i)}{\sum_{i=1}^{n} rec\_msg(i)}$$

The paper is organized as it follows: section 2 discuss SDDS proposal based on hashing, while in section 3 those based on order preserving data organization technique, are presented. In section 4 multi-dimensional data management in SDDS framework is analyzed. Section 5 shows how to take into account in SDDS of high availability requirements. Finally, last section contains a partial survey of *non scalable* distributed data structures.

2 Hash based schemes

2.1 LH*

LH* [17, 20] introduced the concept of a Scalable Distributed Data Structure (SDDS). LH* is a generalization of Linear Hashing to distributed sites. A file F is stored on server sites and it is accessed by client sites. Each server site manages one bucket, which stores some of the file records. Each client site has its own view of the overall file, which may be out-of-date. Client's view is updated through requests, which may require at most two additional messages. The status of overall file is described by two parameters, namely the *hashing level* (i) and the *split pointer* (n). The hashing level defines the couple of hashing functions to be used to assign keys to buckets and the split pointer identifies the next bucket to be split whenever a collision (that is an insertion in a full bucket) happens. Each server knows the hashing level only of the bucket it manages. Each client uses its own view of the overall file status (hashing level and split pointer) to send request messages to servers. If addressing is wrong the receiving server is able to forward key to another server, which either is the correct one or is able to identify the correct one. Moreover, the receiving server communicates back to the client its hashing

level, so that client may bring its own view closer to the file overall status. A designated site act as *split coordinator*, which forces the server site designated by the split pointer to split its bucket and serializes all the splits. The communication network is assumed without delay (if there is a delay then an arbitrarily fast sequence of insertions may cause an arbitrarily long sequence of forwarding messages).

In LH* the splitting policy obeys to a global rule: whenever a split occurs in the structure, the splitting server is always the n-th one. After that n is increased by one. A server is split even if it is not the overflowing server. This may lead in some cases to a poor global and local load factor. In fact a server could have to manage a great number of keys, waiting to be the n-th server, and in the meantime, servers with few keys are split.

Moreover a server does not know when it has to split. It has to be notified. In [17] a special split coordinator, participating in the bucket split operations, is proposed. Such a coordinator knows the exact configuration of the structure (i.e. the current parameters n and i). It notifies the n-th server to split, and then updates the configuration. The definition of such a special entity is not completely compliant with scalability goals, because it can likely become a bottleneck.

However, in [20] some variants of the base technique is proposed, in particular a version of LH* without split coordinator. The technique is based on a token, which is stored by the next server that has to split (i.e., the n-th server). After the split, the token is sent to the next server (it is assumed that a server always knows the address of the next server). Other variants discussed in [20] regard the control of the load factor, that basically consists in allowing the split of the server n only whenever an estimation of the global load factor reaches a given threshold. The latter technique gives good results if it is possible to assume that the used hash functions actually hash uniformly. In LH*, where the hashing functions are defined as $h_i(c) = c \bmod 2^i$, for each i, this means that the probability of hashing a key to a given address is $1/2^i$.

2.2 DDH

Distributed Dynamic Hashing (DDH) is introduced in [5]. It is a distributed version of Dynamic Hashing technique. A trie based on the rightmost i digits of the key is used as hash function at level i. Clients have local images of the overall logical trie. In general, a logarithmic number (trie-height) of forwarding messages is required for a new client to identify the correct server. Each bucket splits as soon as it overflows. Each client needs to store a trie.

In DDH each server manages many small buckets. In this way the cost of a split is reduced, and a fine-grained load sharing across servers is achieved. Determining if its own bucket has to be split is an autonomous decision that can be made by the affected server. Moreover the splitting server is also the overflowing server, and then there is a more accurate distribution of keys among the servers.

The resulting load factor is better than LH*'s one. Moreover, a special split

coordinator is not needed. Like other SDDS proposal, in DDH a site (that it is ambiguously called split coordinator too) knowing the available servers in the network, is necessary, but it has not to manage special decisions or to correct global informations about the structure.

The basic drawback of DDH in respect to LH* is the communication complexity, i.e. the number of messages needed to satisfy a client request. LH*'s communication protocol ensures a constant number of address errors, in the worst-case, namely 2, hence a worst-case of 4 messages for any request. The number of address errors in DDH can be the height of the virtual trie. If the trie is unbalanced we can have a linear number of address errors for a request, and the same holds for the communication complexity.

In [5] some experiments are reported, but they do not give a complete idea of performances of DDH. Moreover, they regard only a constant number of servers, while the basic property of an SDDS is to always have the possibility to get a new server to scale-up performances.

2.3 Load balancing

Distributed linear hashing with explicit control of cost to performance ratio is presented in [35]. Multiple buckets may be assigned to each server and an *address table* maps logical buckets to server number. Buckets can be redistributed through splitting or migration to keep the overall load at an acceptable level. Each server has a *feasible capacity* (expressed in number of keys), above which it becomes overloaded, and a *panic capacity*, above which it cannot accept any more keys. Servers and clients act like in the LH* schema. Each of them has its own view of the *hashing level* of the overall file and its own copy of the address table. A client uses its own hashing level to access its own address table and to find the server where to send the key. This server may be the wrong one, and in this case it uses its own address table to forward the message to another server, which either is the right one or is able to forward the key to the right one. Moreover, the first server communicates back to the client its own address table and hashing level, so that the client may come closer to the current view of the file.

While the overall load is below a specified threshold and no server has reached its panic capacity, overloading is managed, if possible, through migration of buckets from more loaded servers to less loaded ones. If no such a migration is possible than a new server is acquired, but only when the overall load threshold (defined according to an heuristic estimates) is reached. Whenever a server reaches its panic capacity, the possibility of alleviate its load through migration of buckets to less loaded servers is evaluated. If it is not possible, a new server is acquired. A designated site, named *file advisor*, is in charge of managing and coordinating migrations and splittings. The file advisor uses a probabilistic function to estimate each server load (since each server reports to file advisor only every k insertions in an overloaded status) and the overall load.

3 Order preserving SDDS

3.1 DRT

DRT (Distributed Random Tree) [15] proposes a distributed search tree for searching both single items and ranges of values in a totally ordered set of keys (allowing insertion of keys). It is basically a search structure, based on key comparisons, managed as a generic tree. The *overall tree* is distributed among the different server sites. Each leaf node is allocated to a different server, together with a partial copy of the overall search structure (called *local tree*). When a leaf node overflows, its bucket is split in two and a new server is brought in. The overflown leaf node is transformed in an internal node with two sons (leaf nodes). One son contains keys remaining with the current node, and the new server takes care of the remaining keys. The leaf node corresponding to new server becomes the root of a new local tree (which is the part of the overall tree allocated to the new server). This node therefore appears twice in the overall tree (once as a leaf in the old, overflown, node and once as a root in a local tree). Internal nodes are therefore distributed to the different servers according to the way the tree has grown. Client sites query the structure, each using its own view of the overall structure.

A *client view* is a portion of the overall tree, and may be out-of-date since a leaf node may has subsequently been split due to an overflow. A client uses its view to identify to which server the search request has to be sent. If this server evolved and has no more the key, then it forwards the request to the server it identifies using its local trees. This forwarding chain ends at the server having the key.

This last server sends a backward chain of ICMs (Index Correction Messages), containing the information about local trees of servers traversed during the forwarding phase, follows the same path followed by the forwarding chain. Information about local trees in an ICM are used by each server receiving it to update its local tree and to build up, in combination with the search path for the requested key in its local tree, the *view adjustment* to send back, figure 1-a (from [15]). The client finally receives, together with the message related to the key, the overall view adjustment, see figure 1-b.

Since there is no explicit mechanism to keep the overall tree balanced, in the worst-case the height of the overall tree is linear in the number of servers. However, for random insertions from a domain with uniform distribution, the average height of the overall tree is logarithmic. This is not surprising since for a uniform distribution the expected height of a search tree under random insertions is logarithmic in the number of insertions [14].

The total number of messages in the worst-case is $O(n)$, where n is the number of servers.

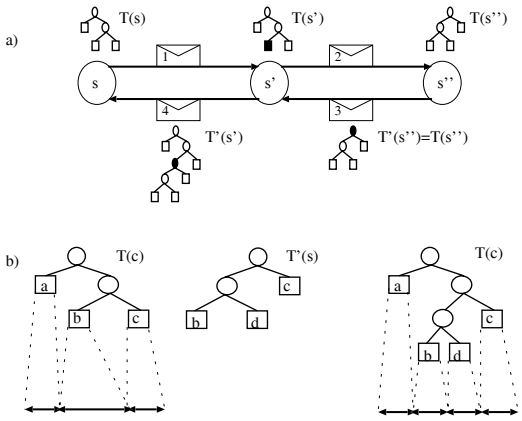

Figure 1: A forwarding chain and ICM messages (a). Update of a local tree $T(c)$ of a client c with a correction tree $T'(s)$ (b).

3.2 RP*

A family of distributed data structures (Range Partitioning - RP*) supporting range queries on totally ordered keys is introduced in [18]. The basic structure of the family, RP^*_n, is basically a B^+-tree distributed among servers and without any index. Each bucket is allocated to a different server and search is done through *multicasting*. Each bucket covers a contiguous portion of the whole key range disjoint from portions covered by other buckets. Each server processes the query and the one (or the ones for a range search query) whose range contains key's value answers with a point-to-point message. Insertions are managed similarly. When a bucket overflows its server brings in a new one and assigns half of its keys to it.

To reduce communication network load each client may maintain a *local index* (this structure is called RP^*_c) which is updated in consequence of searches. The local index is a collection of couples $\langle \textit{bucket range}, \textit{bucket address} \rangle$. A client sends a point-to-point request to the server identified as the potential owner of the key in its local index. If the client has no information about the server owning a certain range it issues a multicast request. If the receiving server is not the correct one, it issues a multicast request, including the range of keys it manages. Client then uses answers from point-to-point and multicast requests, which may include one or two couples $\langle \textit{bucket range}, \textit{bucket address} \rangle$, to update its local index.

A third variant, called RP^*_s, is a structure which maintains indexes also at server sites and completely avoids the need of multicasting. In this case we have a full-blown B^+-tree distributed among server sites, one node for each server. Leaf nodes are servers managing buckets. Internal nodes are managed by dedicated servers and have a structure and a behavior similar to that of an internal B^+-

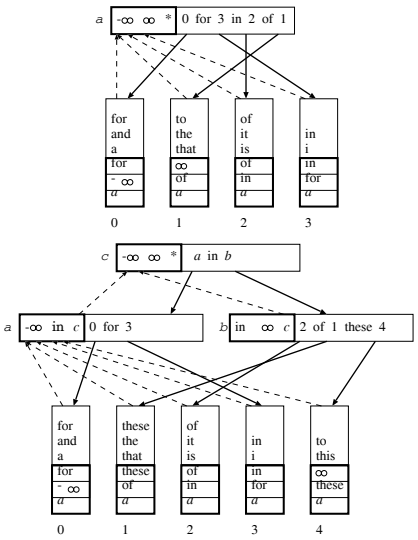

Figure 2: An RP^*_s sample file with a kernel with two level (up) and tree level (down).

tree node, with separating keys and pointers to lower nodes, plus a pointer to the parent node. The set of dedicated servers is called *kernel*. If the receiving server is not the correct one, this may only happen because the client has an out-of-date local index where the server is managing a range of keys larger than what the server is really managing. Therefore the request can either be forwarded up in the overall B^+-tree, so that an ancestor of the receiving server can identify the correct server to manage it, or be forwarded down to the server which is really taking care of the key range containing the requested key (internal nodes in the B^+-tree do not directly manage keys). In both cases the server finally answering the request signals back to client the up-to-date view of the search structure, by sending back the path visited in the tree during the search for the correct serving bucket. A node which splits leaves parent pointers of its sons out-of-date. They are adjusted only in two cases: either when the sons themselves split and need to communicate this fact to their true parent, or when the sons receive a request from their true parent.

In figure 2 (taken from [18]) an example of a RP^*_s file is shown.

3.3 The straight guiding property

With the aim to investigate intrinsic efficiency of SDDSs, Krll and Widmayer analyzed from a theoretical point of view performance bounds of distributed search trees. [16]. The main result is the impossibility to extend to the distributed case

both of two important properties used in managing data structures in the single-processor case:

1. The search process is monotone.
2. Rotations are used to keep the tree balanced .

In the single processor case a search tree is a binary tree such that every node represents an interval of the data domain. The overall data organization satisfies the invariant that the search process visits a child node only if it lies inside the father node's interval. Krll and Widmayer call this behavior the *straight guiding property*.

For satisfying the *straight guiding property* in the distributed case is needed to ensure that a key received by a server belongs to the set of keys the server represents, i.e. to the interval of the node associated to the server. This interval is given by the union of intervals associated to descendants of the node. The *straight guiding property* ensures that the search process goes always down in the tree and never goes up. In this case a bound on the height of the tree directly correspond to a bound on the cost of the search process.

In [16] is proved that if rotations in the distributed tree are used, it is impossible to keep the *straight guiding property*.

To understand why consider figure 3. Assume that in the search tree T the server s_1 manages a node v_1 and the server $s_2 \neq s_1$ manages a node v_2. Assume now that a rotation is needed at v_1 to rebalance the tree. T_{new} is the tree after the rotation, where we assume the assignment of nodes to servers has not changed. Note that the set of keys visiting v_1 in the search tree T (i.e. before the rotation) is a superset of the set of keys visiting v_1 in the search tree T_{new} (i.e. after the rotation). Thus, after the rotation, server s_1 may receive the request for a key whose search path ends in T_0, since v_1 is assigned to s_1 in T. For example the request could be issued by a client with an obsolete view, believing that server s_1 still manages an interval containing T_0. But, after the rotation, server s_1 should not manage any search path for keys in T_0. To reach T_0 from s_1 we have to go up in the tree violating the *straight guiding property*. The same problem exists if we exchange the assignment of nodes to server between v_1 and v_2. In fact in this case it is the server s_2 that may receive the request for a key whose search path ends in T_0. Hence whether we maintain the assignment of servers s_1 and s_2 to nodes v_1 and v_2 in T_{new} or we exchange such an assignment, we fail in any case to guarantee the *straight guiding property*.

Moreover, Krll and Widmayer show that if rotations are not used and the *straight guiding property* is maintained, a lower bound of $\Omega(\sqrt{n})$ holds for the height of balanced search trees, hence for the worst-case of searching.

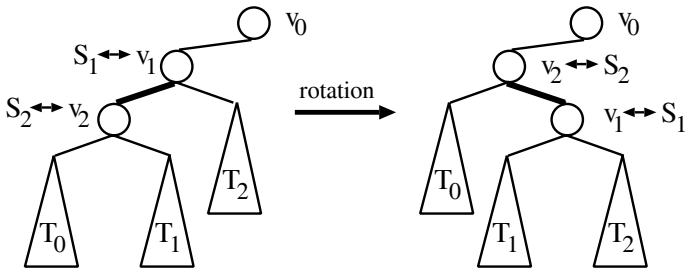

Figure 3: The rotation does not allow to keep the *straight guiding property*.

3.4 RBST

A natural approach to overcome the complexity lower bound of Krll and Widmayer is to violate the *straight guiding property*: this means the search process can go up and down in the distributed tree. Then a bound on the height of the tree does not correspond anymore to a bound on the cost of the search process and it is possible to use rotations to keep the tree balanced. But to obtain a good complexity it is then necessary to bound the number of times a search operation can go up and down in the tree.

The first work in this direction is the RBST (Relaxed Balanced Search Trees) [1]. This is a search tree where nodes have the following structure: each node but the root has a pointer (*father pointer*) to its father (internal) node. Each internal node has a pointer (*left pointer*) to a node in its left subtree and one (*right pointer*) to a node in its right subtree. Note that these pointed nodes may be, in general, different from the direct sons of the internal node. Insertions and deletions are managed with a technique similar to that used for AVL-trees. In [1] it is shown in detail how to maintain these invariant properties of pointers coming out from nodes.

Suppose a request for a key k arrives to a node v (corresponding to a server). If v is a leaf and k belongs to the interval of keys $I(v)$ associated to v, then the search terminates. Otherwise if v is an internal node and $k \in I(v)$, then, according with routing information of v, the left or the right pointer of v is used. Since the *straight guiding property* is not satisfied, it is possible that $k \notin I(v)$. In this case the father pointer is used to forward the request.

The search process is therefore changed, but the number of times a search operation can go up and down in the tree is bounded by the logarithmic of the number of servers [1]. The cost for an exact search, an insertion and a deletion is of $O(\log^2 n)$ messages in the worst-case.

3.5 BDST

Another approach to obtain good worst-case performances in SDDS is discussed in [6], where BDST (Balanced Distributed Search Trees), an SDDS taking an approach similar to those of RBST, is presented. The *straight guiding property* is violated, the tree is kept balanced through rotations and the search process is modified. However, in the case of BDST, the number of times a search operation can go up and down in the tree is bounded by 2. Then BDST improves the result of RBST, giving a cost for an exact search, an insertion and a deletion of $\Theta(\log n)$ messages in the worst-case. In the following we discuss in some detail the data structure.

Let T be a binary search tree with n leaves (and then with $n-1$ internal nodes). $f_1, \ldots, f_n$ are the leaves and $t_1, \ldots, t_{n-1}$ are the internal nodes. $h(T)$ is the height of T, that is the number of internal nodes on a longest path from the root to a leaf. To each leaf a bucket capable of storing b data items is associated. Let $s_1, \ldots, s_n$ be the n servers managing the search tree. We define *leaf association* the pair (f, s), meaning that server s manages leaf f and its associated bucket, *node association* the pair (t, s), meaning that server s manages internal node t. In an equivalent way we define the two functions:

- $t(s_j) = t_i$, where (t_i, s_j) is a node association,
- $f(s_j) = f_i$, where (f_i, s_j) is a leaf association.

To each node x, either leaf or internal one, the interval $I(x)$ of data domain managed by x is associated.

Every server s but one, with leaf node association (t, s) and leaf association (f, s), records at least the following information:

- The internal node $t = t(s)$ and the associated interval of key's domain $I(t)$,
- The server $p(s)$ managing the parent node $pn(t)$ of t, if t is not the root node,
- The server $l(s)$ (resp., $r(s)$) managing the left child $ls(t)$ (resp., right child $rs(t)$) of t, and the associated interval $I_l(t)$ (resp., $I_r(t)$),
- The leaf $f = f(s)$ and the associated interval of key's domain $I(f)$,
- The server $pf(s)$ managing the father node $pn(f)$ of f, if f is not the unique node of global tree (initial situation).

This information constitutes the local tree $lt(s)$ of server s (see figure 4).

Since in a global tree of n nodes there are $n-1$ internal nodes, there is one server s' managing only a leaf association, hence $lt(s')$ is made up by only the two last pieces of information in the above list.

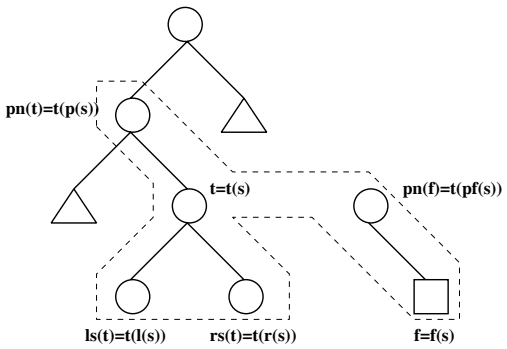

Figure 4: Local tree of the server s.

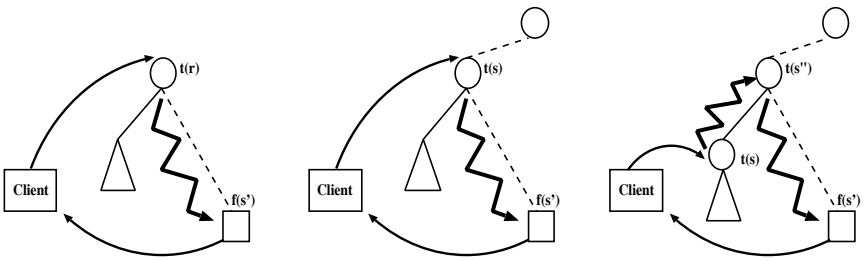

Figure 5: The BDST search process for a request from a new client (left), from a client with addressing error sending its request to: a logically pertinent server (center) and a non logically pertinent server (right).

We say a server s is *pertinent* for a key k, if s manages the bucket to which k belongs. In our case if $k \in I(f(s))$. Moreover we say a server s is *logically pertinent* for a key k, if k is in the key interval of the internal node associated to s, that is if $k \in I(t(s))$. Note that the server managing the root is logically pertinent for each key. Note also that, due to the effect of rotations, it is not necessarily $I(f(s)) \subseteq I(t(s))$.

Suppose that a request for a key k arrives to a server s. If s is the pertinent server for k then s directly manages the request. If it is logically pertinent, then s forwards the request downwards in the tree, using $l(s)$ or $r(s)$. If s is not logically pertinent, then it forwards the request upwards in the tree, using $p(s)$. In general, a request of a client travels the global tree structure until arrives to the leaf corresponding to the pertinent server. In figure 5, the various cases of the search process are described.

It is clear that the number of messages for a request, accounting for the request and the answer messages, is $2h(T)+2$ in the worst-case. Rotations do not influ-

ence the search process. In this case any balancing strategy based on rotations can be used, and the worst-case cost for exact search, insertion or deletion is shown to be $\Theta(\log n)$ messages.

3.6 DRT*

The main drawback of BDST is represented by the fact that the use of rotations requires locking servers involved in the rotation process. In fact locks reduce the concurrency in the system.

On the contrary DRT, i.e. the implementation of a distributed tree without using balancing strategies, suffers of the well known worst-case cost of $O(n)$ messages for a request. Although the worst-case result is very bad, the amortized analysis shows a very nice behavior. DRT* [8, 9], extends the DRT technique by means of a different use of correction technique based on ICMs.

These modifications improve the response time of a request with respect to DRT, but do not influence the amortized communication cost, in the sense that the communication complexity results of DRT* hold also for the DRT. More details on DRT* can be found in another paper [8] of these proceedings.

3.7 Distributed B$^+$-tree

In DRT and DRT* the goal of reducing communication complexity is reached by means of a "continuous" improvement of the system knowledge of local trees of servers. This is obtained in a lazy manner, in the sense that the local tree of a server s is corrected only whenever s makes an address error. It is clear that the best possible situation is the one where each local tree is always correct.

An alternative strategy is to try to ensure servers have correct local trees by sending corrections exactly when local trees become obsolete, that is whenever a split occurs in the structure. A structure designed to this aim is Distributed B$^+$-tree [4]. For the sake of simplicity we describe Distributed B$^+$-tree in a slightly different way with respect to the original paper.

If server s was created through the split of server s' then server s stores a pointer to the server s'. With respect to the node v associated to s, the pointer to s' is a pointer to the father node v' of v.

Whenever a split occurs, the splitting server s sends a correction message containing the information about the split to s'. s' corrects its local tree and sends the message upwards in the tree so to arrive to the root. This technique ensures that a server s associated to a node v has a completely up-to-date view of the subtree rooted at v. This means that it knows the exact partition of the interval $I(v)$ of v and the exact associations between elements of the partitions and servers. This allows s to forward a request for a key belonging to $I(v)$ directly to the right server, without sending a chain of messages along the tree. This distributed tree does not use rotations, hence each request arriving to s belongs to $I(v)$ because of

the *straight guiding property*.

The main result is a worst-case constant number of messages for a request, which is a very good result for an order preserving data structure. The drawback is the cost of a split. In fact, like in DRT, if keys are not uniformly distributed over the data domain, the height of the distributed tree may be linear, and then the cost of correcting local trees after a split can be linear as well. In [7] a comparison between Distributed B^+-tree and BDST is performed, showing that BDST behaves better in the amortized case, for the case of a sequence of intermixed exact searches and inserts. In fact Distributed B^+-tree has a linear amortized cost, while for BDST we clearly have a logarithmic cost. The amortized cost of a sequence of intermixed exact searches and inserts of DRT* is better than the BDST one, hence it is better than the Distributed B^+-tree one as well.

3.8 DSL

Another SDDS dealing with order preserving data management is DSL (Distributed Skip Lists). It is fundamentally based on the extension of the skip list technique to the distributed environment. The main results, based on a probabilistic analysis - which is the standard approach to measure skip list performances, are an access cost of $O(\log n)$ messages and an $O(1)$ number of reconstruction operations after merges or splits of nodes with high probability. A complete presentation of DSL can be found in paper [3] of these proceedings.

4 Multidimensional search structures

Many of the newest application areas, like CAD, GIS, Multimedia and others, deal with very large amounts of complex (i.e. multi-attribute) data and require high performance. A distributed environment offers a good answer to these requirements. Some SDDSs present efficient solutions for searching multi-dimensional data.

The main proposals for multi-dimensional SDDSs are based on the extension to the distributed context of the *k*-d tree [2].

4.1 Distributed *k*-d tree

In [28, 29, 30] an SDDS version of the *k*-d tree [2] is proposed. Different variants are discussed, and their use depends on the availability of the multicast protocol in the communication network.

The base technique used to design a distributed *k*-d tree is like the one used to derive DRT from standard binary trees. Each server manages a different leaf of the tree, and each leaf corresponds to a bucket of data.

A *k*-d tree is a binary tree where each internal node *v* is associated to a

(bounded or not) k-d interval (or k-range) $I(v)$, a dimension index $D(v)$ and a value $V(v)$. The interval associated to the left (resp. right) son of v is made up by every point in $I(v)$ whose coordinate in dimension $D(v)$ has a value less than (resp. not less than) $V(v)$. $D(v)$ is called the *split dimension* for node v. $V(v)$ is the *split point* for node v. Leaves of the k-d tree are associated only to a k-d interval.

To each leaf w of a k-d tree one bucket exactly corresponds, denoted with the same name. Bucket w contains all points within $I(w)$. The k-d interval $I(v)$ of an internal node v is the initial k-range of the bucket which was associated to node v when v was inserted as a leaf into the k-d tree. When bucket v is split two leaves, say v' and y, are created and inserted in the k-d tree as sons of node v. Bucket v, with a new, reduced, k-range is associated to leaf v', and leaf y takes care of the new bucket y, so that $I(v) = I(v') \bigcup I(y)$ and $I(v') \bigcap I(y) = \emptyset$. Therefore, for each leaf w but one it exists a unique internal node z whose bucket's splitting created the k-range of bucket associated to w. Such a node z is called the *source node* of leaf w (and of bucket w) and is denoted as $\alpha(w)$. The leaf without source node, for which we let for completeness $\alpha(.) = \emptyset$ is the leaf managing the initial bucket of the k-d tree.

Clients may add k-d points, which go in the pertinent bucket. In this case a bucket b (and in the same way a server b) is pertinent with respect to point p if b is associated to the leaf node managing the portion of the k-d space containing p. Whenever a split is needed, it is done with a (k-1)-dimensional plane and various strategies can be used to select which dimension is chosen. A largely used strategy is the *round-robin* one, where at each level a different dimension is selected and after k levels the same sequence is used again and again.

Moreover clients can issue exact match, partial and range queries. An *exact match query* looks for a point whose k coordinates are specified. A *partial match query* looks for a (set of) point(s) for whom only $h < k$ coordinates are specified. A *range query* looks for all points such that their k coordinates are all internal to the (usually closed) k-dimensional interval specified by the query.

Search algorithm for exact, partial and range search is optimal. Optimality is in the sense that (1) only servers that could have k-dimensional points related to a query reply to it and that (2) the client issuing the query can deterministically know when the search is complete.

The latter property is very important for the multi-dimensional case. In fact while for a simple exact match query a client knows that the query is terminated whenever it receives an answer by the single pertinent server, in a partial match or a range query it is not true, in general, that there is exactly one pertinent server. Hence the client has the problem of checking that all pertinent servers have answered.

One way to perform the termination test is to calculate the volume V of the k-range of the query, then to calculate the volumes of the received k-d intervals and to consider the corresponding sum S. Whenever $V = S$ the termination test returns true. This approach is not good from a practical point of view, because

infinite precision multiplication is needed for the approach to be correct. In fact, if buckets covering very large and very small ranges exist at the same time in the data structure then, due to possible roundings, the termination test may fail.

An improved deterministic termination test, named "logical volumes" test, which does not suffer from this problem is presented in [30].

If multicast is available in the network, the protocol for search process is very simple and it is similar to the one of RP^*_n. A client issues a query by simple multicasting it in the network, and waiting for the answers of servers. If the use of multicast has to be reduced, like in RP^*_c, a client can use a local index to address only servers pertinent for its requests. In this case requests are sent using the point-to-point protocol. In case of address error, the server receiving the request multicasts it in the network.

A client may receive ICM messages both from server s (managing a k-d interval I) it sent the request to, and from the real pertinent server s' (managing a k-d interval I'). In the overall index, I' may be associated to a node which is various levels down with respect to the current height in the client's local index of node associated to interval I. Interval I' certainly derives from splits of interval I (managed by s). But since servers do not maintain the story of splits, then the client receives intervals in ICMs without the related sequence of splits producing them.

While in RP^*_c a simple binary tree is used to manage such unrelated intervals in the local index of a client, in the multi-dimensional case the use of a standard k-d tree for the local index of a client may produce incorrect results.

Therefore in [28] a new structure, named *lazy k-d tree*, is used to manage the local index of clients. A lazy k-d tree is a k-d tree where:

- there are two types of nodes: *simple nodes* and *compound nodes*;
- a simple node may be a leaf or an internal node and it corresponds to a node of a k-d tree.
- a compound node u has no sons and no piece of information related to the global index is directly associated with it. u is a set $C(u)$ of lazy k-d trees, whose roots are simple nodes, such that for each couple of distinct roots v and w in $C(u)$ it is $I(v) \bigcap I(w) = \emptyset$. Complete details of the structure can be found in [30];

Each client index starts with a simple leaf node. The index is then built incrementally using answers arriving from servers.

A version of the structure with local index at client and server sites is defined in case only the point-to-point protocol is used to exchange messages. In this case the local indexes are local trees like in the DRT. The difference with respect to DRT is that now a local tree is a standard k-d tree and not a binary tree, but the behavior is the same. In fact now servers manage an index. The index of a server s contains at least the portion of the global tree built by the split of s. Moreover s can

send messages only to a server belonging to its index. The ICM arriving to a client as answer to a request contains the set of local indexes of the involved servers. The aggregation of these indexes corresponds at least to a contiguous portion of the global k-d tree connecting the node associated with the server addressed from the client to the node associated with the pertinent server.

The most important thing is that the correction technique used for the DRT* can be used also for this version of the distributed k-d tree. In [9] it is shown that the same analysis conducted for the mono-dimensional DRT* is valid also for this version of the distributed k-d tree and then the same results hold. In particular a request may require a linear number of messages in the worst-case and if we perform a sequence of m requests (that can be exact search and insertions) producing an n-servers distributed k-d tree, starting with one empty server, we have an amortized communication cost of $O\left(m\log_{(1+m/n)} n\right)$ messages.

4.2 k-RP$^*_s$

Another SDDS version of a k-d tree is k-RP$^*_s$ [19]. The paper presents the version used in case of point-to-point protocol. k-RP$^*_s$ is a k-dimensional version of the approach used to design family RP*. There are servers involved in the management of data (*servers bucket*), and servers involved in the management of address errors (*servers index*).

A server index maintains an internal index like in RP$^*_s$. The difference relies just in the fact that while in RP$^*_s$ a standard binary tree is used for the internal index, a k-d tree is used in k-RP$^*_s$. Techniques like "logical volumes" are used for the termination test after a range search, and in general the management of requests is very similar to the distributed k-d tree one.

Results for RP$^*_s$ holds for k-RP$^*_s$ as well. In particular, the worst-case logarithmic bound for the communication cost for an exact search and for an insertion. The main drawback is the use of additional servers to manage address errors, considering that a server is an expensive resource.

With respect to access performance, the comparison between distributed k-d tree and k-RP$^*_s$ is similar to the one between DRT* and RP$^*_s$.

k-RP$^*_s$ and RP$^*_s$ are better in the worst-case, where they guarantee an access cost logarithmic in the number of servers, while DRT* and the distributed k-d trees, under particular data distributions of the data domain, may have a cost linear in the number of servers.

We obtain the contrary in the amortized case. In fact, it is easy to prove that for both k-RP$^*_s$ and RP$^*_s$ the cost remain logarithmic also in the amortized case. More precisely, we have that for m intermixed exact searches and inserts, the amortized cost is $O\left(m\log_F n\right)$ messages, where n is the number of servers and F is related with the fanout of the servers in the kernel.

For DRT* and the distributed k-d trees in [8, 9] it is proved that a se-

quence of m intermixed exact searches and inserts, gives an amortized cost of $O\left(m \log_{(1+m/n)} n\right)$ messages, which is generally better than k-RP$^*_s$ and RP$^*_s$ one. Note that while this amortized cost decreases with m, this does not hold for the amortized cost of k-RP$^*_s$ and RP$^*_s$.

5 High availability

In this section we discuss some aspects regarding fault tolerance for SDDSs. The main consideration is that availability of a distributed file deteriorates with the number N of sites, and rather strongly in practice. Assume, in fact, that the probability p_d that a bucket is up is a constant, and rather high in practice, for example 99%. The probability p_c that key c is available is $p_c = p_d$. The probability p_F that the whole file F is available is $p_F = (p_d)^N$, under the usual assumption that bucket failures are mutually independent. If F scales moderately, to let us say 100 buckets, it leads to $p_F = 37\%$, i.e., most of the time F is not entirely available. For $N = 1000$, one gets $p_F = 0.00004$, i.e., zero in practice. For many applications, this may not be a problem. For other applications however, especially those needing a reliable very large database, these numbers may mean that an SDDS scheme simply does not scale-up to files large enough for their needs. A *k-availability scheme* preserves the availability of all records despite up to k bucket failures.

The first attempts in enhance file availability are based on the popular *mirroring* technique. In [21] a variant of LH* called LH*$_M$ is presented. This scheme mirrors every bucket and thus preserve full accessibility despite. The scalable and distributed generalizations of B$^+$-trees introduced in [4, 36] also use the replication. In both cases, the cost is the doubling of the storage requirements. This may be prohibitive for large files. High availability variants of LH* with smaller storage overhead have therefore been developed. The 1-availability scheme LH*$_S$ stripes every data record into m stripes, then places each stripe into a different bucket and stores the bitwise parity of the stripes in *parity* records in additional *parity* buckets [23]. The storage overhead for the high-availability is only about $1/m$ for m stripes per record. If a bucket is unavailable because of a missing stripe then, like in the RAID schemes, LH*$_S$ recovers the missing stripe from all the other stripes of the bucket, including the parity stripe. Striping typically produces meaningless record fragments. This prohibits or at best heavily impairs the parallel scans, especially with the function shipping. Those typically require entire records at each site. Efficient scans are decisive for many applications, especially web servers and parallel databases. In another 1-availability variant of LH* termed LH*$_g$ [22] the application record, called *data* record, remains entire in LH*$_g$. To obtain high availability, records are considered as forming m-member *record groups* each provided with the bitwise parity record. The resulting storage overhead is about $1/m$, as for the striping. The speed of searches is that of generic (0-available) LH*. It is unaffected by the

additional structure for the high-availability.

As the file grows, 1-availability or even k-availability for any static k is however not sufficient to prevent a decrease in reliability. To this aim one needs to dynamically increase k. The result is *scalable availability schemes*. The first scalable availability scheme was LH*$_{SA}$[25]. LH*$_{SA}$ retains the concept of record grouping, making the technique more elaborated. Each data record c is a member of k or $k+1$ 1-available groups that only intersect in c and are each 1-available. The value of k progressively increases with the file. For any k, LH*$_{SA}$ file is k-available. The storage overhead may vary substantially depending on the file size. It can be close to the minimal possible for k-availability which is known to be k/m. But it can also become over 50%.

In [24, 26] an alternative scalable availability scheme termed LH*$_{RS}$ is presented. Through record grouping, it retains the LH* generic efficiency of searches. Each record belongs to one group only, but with k or $k+1$ parity records. This provide the (scalable) k-availability of the file. The parity calculus uses the Reed Solomon Codes (RS-codes). This mathematically complex tool proves simple and efficient in practice. Current advantages of LH*$_{RS}$ are storage overhead always close to the minimal possible, and a more efficient recovery algorithm, accessing buckets only within one group. Moreover, the basic ideas in LH*$_{RS}$ may be also ported to other SDDS schemes, including the order preserving ones.

6 Related work

Many papers related to SDDS analyzed, as discussed in the introduction, distributed data structures under the assumption that the number of processors (nodes, sites) is fixed. Typically, the main problem addressed here is how to place data among the fixed number of sites in order to balance the workload. We discuss some of the papers belonging to this area, without the goal of being exhaustive. Consider also that some of the structures presented in this section could be transformed into scalable ones.

One important aspect under which structures here discussed and SDDSs differ is the management of load balancing, that is one of the main goal for both of the categories. In order to achieve load balancing a non scalable structure distributes data among all the server from the beginning, while a scalable structure starts with a minimum number of servers (typically one) and balances the load among them to provide adequate performances. When this is no more possible using the existing servers, new servers are called in the structure and now the load balancing technique considers also the new servers.

In [10] a (non scalable) distributed version of extendible hashing is presented. The directories of the table are replicated among several sites (the *directory managers*). Data buckets are distributed among sites (the *bucket managers*). Every update to the directories must be distributed to every copy. Other issues discussed

in the paper regard changes in the structure to allow replications, used to increase availability of the data structure in presence of failure or to improve performance by allowing more concurrency.

In [27], the problem of implementing Bounded Disordered files in multiprocessors multi-disk environments consisting of a fixed number of processor-disk pairs is considered. The used schema is valid both for tightly coupled (shared memory) and loosely coupled (local network) processors. The use of Bounded Disordered files is motivated by the fact that it achieves good performance for single-key operations (almost as good as that of hash based methods), and unlike hashing schemes, range searches are performed with a low cost.

The straightforward solution is to equally partition the file records among processors, each of which maintains its part of "local" Bounded Disorder file (stored in the processor's main memory and disk). This method is highly parallel and achieves good performance due to the use of Bounded Disorder files.

An alternative method, called Conceptual Bounded Disorder file, is presented, which obtains performance similar to the above straightforward solution, and in addition, obtains a significant cut down in main memory space consumption.

In [31] and [11] complexity issues related with the distribution of a dictionary over a fixed number of processors on a network are considered. Communication cost takes into account the topology of the network, in the sense that a message has a cost given by the number of links traversed from the source to destination. In this case one basic lower bound for the communication cost of an operation in the worst-case is $\Omega(D)$ traversed links, where D is the diameter of the graph associated to the network. Let m be the number of information items stored in a structure at a given point in time, and n be the fixed number of sites. The main objective in [31] and [11] is to have *memory-balanced* structures, that is structures where the amount of storage required at the various sites in the system is roughly the same, more precisely it has to be $O(Load)$, where $Load = \left\lfloor \frac{m}{n} \right\rfloor$.

Special attention is devoted to networks having a tree topology. In order to apply results for this kind of networks to networks with a generic topology, the problem of embedding a virtual network with a tree topology in a given generic network is studied.

Various structures are proposed and analyzed in detail with respect to topologies of the network and some conditions on m. Some structures present communication complexity close to optimality (i.e. $O(D)$) either in the worst-case or in the amortized case. In some of these cases to obtain this communication cost results either a central directory manager or directory structure replicated on each site is assumed. Both these approaches do not allow extensions of the structures to a scalable environment.

Paper [12] introduces dB-tree, a distributed memory version of the B-link tree. A B-link tree is a B^+-tree where every node has a pointer to its right sibling. When a node overflows due to an insertion, a *half-split* operation is performed. First the node creates its sibling and pass it half of its keys, and second it inserts in its

parent a pointer to the sibling. Between first and second step keys moved to the sibling may still be reached through the link between siblings and due to the fact that each node stores the *highest* value reachable in the subtree rooted at the node. The dB-tree distributes nodes of a B-link tree among a fixed number of processors according to the following strategy: each leaf node is owned by exactly one processor, the root node is owned by all processors, each processor owns all internal nodes linking the root node to its leaf node. Each processor has a local copy of all nodes it owns. Each node is linked to both its siblings. Each node has a queue of pending operations to be performed on its copies of nodes. These operations may be requested either locally or by other processors to make all the copies of the same node consistent. Search may start at any processor by descending the tree. When access is needed to a node the processor does not have a copy of, the request is forwarded and it eventually reaches the processor holding the unique copy of the leaf node possibly containing the key. Key insertions and deletions that do not require restructuring (i.e. node split or node merge) are similarly managed.

A *split* is carried out in three steps: first, a new node is created and half of the keys are transferred to it (*half-split*); second, a *link-change* suboperation is performed on the sibling to make it point to the new node; third an insert of a pointer to the new node is performed on the parent. A *merge* operation is carried out in four steps: first, keys are moved to the siblings; second, a link-change suboperation is performed on both siblings to make them to point each other; third, a delete is performed on the parent to cancel the pointer to the node; fourth, after all pointers to the node have been properly changed the node is deleted. Split and merge operations may trigger similar operation on parent nodes.

Correct serialization of restructuring operations on neighboring nodes is implemented by requiring that a processor wishing to restructure a node first obtain acknowledgment to a *restructure-lock* request from all processors holding a copy of the neighboring nodes, then carries out the required modifications, and finally releases the restructure lock. This will allow the locked neighbors to proceed, if needed, with their restructuring operations. This discussion assumed an operation is performed atomically on all copies of the same node. But only restructuring operations (i.e. a split or a merge) require *synchronizing* between all processors having a copy of the node. Other operations are either *lazy*, that is they can be carried out independently by each processor, or *lazy/synchronizing*, that is they simply require the processor which triggered them is advised when they have been completed.

This line of research is further extended in [13], where a *lazy update* approach is proposed to lower the cost of updating the replicated nodes in a dB-tree.

References

[1] F.Barillari, E. Nardelli, M. Pepe: Fully Dynamic Distributed Search Trees Can Be Balanced in $O(\log^2 n)$ Time, *Technical Report 146*, Dipartimento di Matematica Pura ed Applicata, Universita' di L'Aquila, July 1997, accepted for publication on *Journal of Parallel and Distributed Computation (JPDC).*

[2] J.L. Bentley: Multidimensional binary search trees used for associative searching, *Comm. ACM*, 18:509-517, 1975.

[3] P. Bozanis, Y. Manolopoulos: DSL: Accomodating Skip Lists in the SDDS Model, *Workshop on Distributed Data and Structures (WDAS 2000)*, L'Aquila, Carleton Scientific, June 2000.

[4] Y. Breitbart, R. Vingralek: Addressing and Balancing Issues in Distributed B^+-Trees, *1st Workshop on Distributed Data and Structures (WDAS'98)*, Carleton Scientific, 1998.

[5] R.Devine: Design and implementation of DDH: a distributed dynamic hashing algorithm, *4th Int. Conf. on Foundations of Data Organization and Algorithms (FODO)*, LNCS Vol. 730, Springer Verlag, Chicago, 1993.

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

[7] A.Di Pasquale, E. Nardelli: Improving Search Time in Balanced and Distributed Search Trees, *WSDAAL 99*, September 1999.

[8] 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.

[9] A.Di Pasquale, E. Nardelli: Distributed searching of *k*-dimensional data with almost constant costs, *ADBIS 2000*, Prague, LNCS Vol. 1884, Springer Verlag, September 2000.

[10] C.S.Ellis: Distributed data structures: a case study, *IEEE Trans. on Computing*, C-34(12):1178-1185, 1985.

[11] K.Z.Gilon, D.Peleg: Compact deterministic distributed dictionaries, *10th ACM Symp. on Principles of Distributed Computing*, 81-94, 1991.

[12] T.Johnson, A.Colbrook: A distributed data-balanced dictionary based on the B-link tree, *Int. Parallel Processing Symp.*, 319-325, 1992.

[13] T.Johnson, P.Krishna: Lazy updates for distributed search structures, *ACM SIGMOD Int. Conf. on Management of Data*, Washington, 1993.

[14] D. Knuth: Sorting and searching, *"The Art of Computer Programming"*, Vol. 2, Addison Wesley, Reading, MA, 1973

[15] B. Kröll, P. Widmayer: Distributing a search tree among a growing number of processor, in *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, 1994.

[16] B. Krll, P. Widmayer. Balanced distributed search trees do not exists, in *4th Int. Workshop on Algorithms and Data Structures(WADS'95)*, Kingston, Canada, (S. Akl et al., Eds.), Lecture Notes in Computer Science, Vol. 955, pp. 50-61, Springer-Verlag, Berlin/New York, August 1995.

[17] 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.

[18] 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.

[19] W. Litwin, M.A. Neimat, D.A. Schneider: *k*-RP$^*_s$ - A High Performance Multi-Attribute Scalable Distributed Data Structure, in *4th International Conference on Parallel and Distributed Information System*, December 1996.

[20] W. Litwin, M.A. Neimat, D.A. Schneider: LH* - A Scalable Distributed Data Structure, *ACM Trans. on Database Systems*, 21(4), 1996.

[21] W. Litwin, M.A. Neimat: High-availability LH* Schemes with Mirroring, *International Conference on Cooperating Information Systems (COOPIS)*, IEEE Press 1996.

[22] W. Litwin, T. Risch: LH*g: a High-availability Scalable Distributed Data Structure through Record Grouping, *Res. Rep. U. Paris 9 & U. Linkoping* (Apr. 1997). Submitted.

[23] W. Litwin, M.A. Neimat, G. Levy, S. Ndiaye, S. Seck: LH*s: a High-availability and high-security Scalable Distributed Data Structure through Record Grouping, *IEEE Workshop on Res. Issues in Data Eng. (RIDE)*, 1997.

[24] W. Litwin, T.J.E. Schwarz, S.J.: LH*$_{RS}$: a High-availability Scalable Distributed Data Structure using Reed Solomon Codes, *ACM SIGMOD Int. Conf. on Management of Data*, 1999.

[25] W. Litwin, J. Menon, T. Risch: LH* with Scalable Availability, IBM Almaden Res. Rep. RJ 10121 (91937), may 1998.

[26] W. Litwin, J. Menon, T. Risch, T.J.E. Schwarz, S.J.: Design Issues for Scalable Availability LH* Schemes with Record Grouping, *Workshop on Distributed Data and Structures (WDAS'99)*, Princeton, NJ, May 1999.

[27] G.Matsliach, O.Shmueli: Maintaining bounded disorder files in multiprocessor multi-disk environments, *Int. Conf. on Database Theory (ICDT'90)*, LNCS, Springer Verlag, 1990.

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

[29] 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.

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

[31] D.Peleg: Distributed data structures: a complexity oriented view, *4th Int. Workshop on Distributed Algorithms*, 71-89, Bari, 1990.

[32] M. Sloman, J. Kramer, Distributed System and Computer Networks, Prentice Hall 1987.

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

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

[35] R.Vingralek, Y.Breitbart, G.Weikum: Distributed file organization with scalable cost/performance, *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, 1994.

[36] R.Vingralek, Y.Breitbart, G.Weikum: SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load, *Distr. and Par. Databases*, 6, 2, 1998.

Adriano Di Pasquale is with Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia. E-mail: dipasqua@univaq.it

Enrico Nardelli is with Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia and with Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italia. E-mail: nardelli@univaq.it