JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **49,** 111–134 (1998)
ARTICLE NO. PC981428

Distributed Searching of Multi-dimensional Data: A Performance Evaluation Study[1]

Enrico Nardelli[2]

Dipartimento di Matematica Pura ed Applicata, Univ. di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy; and Istituto di Analisi dei Sistemi ed Informatica, C.N.R., Viale Manzoni 30, 00185 Rome, Italy

and

Fabio Barillari and Massimo Pepe

Dipartimento di Matematica Pura ed Applicata, Univ. di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy

In this paper we present a data structure for searching in multi-dimensional point sets in distributed environments and discuss its experimental evaluation also through a comparison with previous proposals. The data structure is based on an extension of k-d trees. The technological reference context is a distributed environment where multicast (i.e., restricted broadcast) is allowed, but it is also shown how to avoid using it. The data structure supports exact, partial, and range search queries with a complexity that is optimal in a distributed sense. The set of multidimensional points is managed in a scalable way, i.e., it can be dynamically enlarged with insertion of new points. We also propose new performance measures for the comparative evaluation of the efficiency with which a data structure is distributed over a communication network.

1. INTRODUCTION

The impressive progress of communication technology makes it now easy and cost effective to set up distributed applications running on a network of workstations. The technological framework we make reference to is the so called *network computing*: fast communication networks and many powerful and cheap workstations. In this framework

[1]Parts of this work were done while the first author was visiting the International Computer Science Institute, Berkeley, the Hewlett-Packard Research Laboratory, Palo Alto, and the Eidgenössische Technische Hochschule, Zürich. The financial support from all these organizations is gratefully acknowledged. Research described here is partially supported by the "Algoritmi, Modelli di Calcolo e Strutture Informative" 40%—Project of the Italian Ministry for University and Scientific and Technological Research (MURST) and by the "Chorochronos" Research Network of the ESPRIT Research Programme of the European Union.

[2]E-mail: nardelli@univaq.it.

0743-7315/98 $25.00

it is possible and realistic to efficiently implement main memory applications using the main memory of distributed machines. Many advanced applications manage very large amounts of multi-attribute data. Such data can be considered as points in a k-dimensional space. Hence a key research issue in order to provide efficient implementations of these applications is the design of an efficient distributed data structure for searching in a k-dimensional space. The fundamental queries for a data structure for k-d points are exact match, partial match, and range. It is also important that the structure is able to scale up to adapt itself to a growing number of points.

In the considered technological framework, efficiency can be obtained by using the *multicast* protocol. In this communication protocol with a single message one source reaches N destinations. This means that a multicast message has a communication cost which is the same as a point-to-point message. In other words the time used by a multicast message to reach all its destinations is the same used by a point-to-point message to reach its unique destination. We call *servers* the machines which manage the data structure and answer to queries. *Clients* are the machines which need to access or manipulate the k-d points for their application needs. It is very realistic to assume that clients are not always connected to the network to be kept up-to-date with the status of the distributed data structure. Hence they have to be able to catch up whenever they connect back to the network. The use of multicast is therefore important in reaching overall efficiency. But note that an uncontrolled use of multicast may somehow degrade performances of end-user applications, by forcing a client to process in any case the message delivered to it by its associated network controller, even if it is useless for the application itself. Design issues related to the trade-off between the use of multicast and point-to-point queries are discussed in [NBP97a]. Further discussions on the technological background of multicast in relation to distributed data structures are in [NBP97b].

In our framework we are concerned with efficiency with respect to the communication network. Therefore performance is measured in terms of the overall number of messages on the network. This is the usual performance measure used for data structures in a distributed environment. Moreover, a client has to be able to determine when the multicast query is terminated, i.e., when every server interested in the query has answered. An important performance measure, proposed here for the first time and called *distribution efficiency*, is the overall efficiency of the communication protocol with respect to a nondistributed solution. In other words we measure the overall efficiency of having the servers distributed over a communication network instead of having a single server. We express this in terms of two indexes measuring how many computational resources are wasted for message processing. But note that the purpose of this performance measure is not to provide an absolute evaluation, since a distributed solution will always consume more computational resources than a nondistributed one. The main aim of distribution efficiency is to provide a means to compare different distributed data structures.

Litwin *et al.* were the first to present and discuss the paradigm of scalable distributed data structures, by proposing a distributed linear hashing, namely LH* [LNS93, LNS96], and a distributed 1-dimensional order-preserving data structure, namely RP* [LNS94a]. Extension of RP* to the k-dimensional case was first studied in [LNS94b] and subsequently published in [LN96]. A variant of LH* with explicit control of the cost–performance ratio was presented in [VBW94]. The use of hashing techniques for managing multi-dimensional data in a distributed framework was also studied in [Dev93].

When a multicast protocol is not supported by the given network, then everything needs to be done with point-to-point messages. In this case the first solution for multi-dimensional data was given by Kröll and Widmayer [KW94, KW95], who developed distributed random trees (DRT). Work related to the field of scalable distributed data structures was done on parallel distributed B-trees in the case of a fixed number of processors [MS91, JC92, JK93] and in the case of a fixed number of multiple disks [PK90, SL91]. The use of a fixed number of multiple disks to improve performances has also been considered in [KF92] for R-trees, a well-known data structure for spatial data.

The first scalable distributed data structure for managing k-dimensional points over a network where multicast is available to be published was presented in [Nar96b]. The solution was based on a data structure, named *lazy k-d tree*, for managing a collection of k-d trees, introduced in [Nar95] and refined in [Nar96a]. The solution featured optimal search algorithms for exact, partial, and range search. 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. In this paper we extend the definition of the scalable distributed data structure for k-dimensional points to the case of networks where multicast is not available and provide both a thorough experimental evaluation and a comparison with previous work. Our solution is able, like RP* (the 1-dimensional distributed data structure defined in [LNS94a]), to work both with and without using multicast, but our structure provides this capability also for points in a k-dimensional space. Experimental results show that our data structure is at least as efficient as previously known data structures for the 1-dimensional case, but it is also able to manage points in a multi-dimensional space.

The paper is organized as follows. In Section 2 the basic version of the data structure is briefly recalled. Section 3 deals with termination tests needed for multicast queries. In Section 4 we briefly recall the version of the structure using an index at client sites to reduce the use of multicast and in Section 5 the version avoiding the use of multicast at all. In Section 6 we present performance measures used in experimental evaluation. Experiments and results are discussed in Section 7, also in relation to previous work. Section 8 contains a final discussion and conclusions.

2. BASIC STRUCTURE AND BASIC ALGORITHMS

From a conceptual point of view our structure can be considered as a unique k-d tree. Each internal node x has only the function of maintaining track of the split $(k-1)$-d plane which divides in two the k-d space managed by node x itself. Each leaf node y has a bucket associated with it, containing all the k-d points which fall within the portion of the k-d space managed by leaf node y. Each server is managing a different leaf, hence each server manages a single bucket of data. We assume all buckets have the same size. Clearly the set of buckets is a partition of the whole k-d space managed by the structure.

Clients may add k-d points, which go in the pertinent bucket. A bucket 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. In a similar way it may be defined when a bucket b is *pertinent* with respect to any query. When a bucket overflows its point set is split in two (usually equally sized) parts. The split is done with a $(k-1)$-dimensional plane and various strategies can be used to select which dimension to use. A largely used strategy is the *round robin*,

where at each level a different dimension is selected and after k levels the same order is used again and again.

Clients can query the structure with any of the following queries: exact match, partial match, and range. 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.

A *k-d tree* [Ben75] is a binary tree where each internal node v is associated to a (bounded or not) k-d interval (or k-range) $\mathrm{I}(v)$, a dimension index $\mathrm{D}(v)$ and a value $\mathrm{V}(v)$. The interval associated to the left (resp., right) son of v is made up by every point in $\mathrm{I}(v)$ whose coordinate in dimension $\mathrm{D}(v)$ has a value less than (resp., not less than) $\mathrm{V}(v)$. $\mathrm{D}(v)$ is called the *split-dimension* for node v. $\mathrm{V}(v)$ 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 by the same name. Bucket w contains all points within $\mathrm{I}(w)$. The k-d interval $\mathrm{I}(v)$ 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 $\mathrm{I}(v) = \mathrm{I}(v') \cup \mathrm{I}(y)$ and $\mathrm{I}(v') \cap \mathrm{I}(y) = \emptyset$. Therefore, for each leaf w but one there exists a unique internal node z whose bucket's splitting created the k-range of the 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 $\alpha(\cdot) = \emptyset$ is the leaf managing the initial bucket of the k-d tree.

We now briefly recall the version of the structure without indexes. Every operation is performed through the use of multicast. For more details see [Nar96a, Nar96b]. The insertion algorithm is straightforward, since a client wanting to insert point p simply multicasts its request by putting the point coordinates in the message. The pertinent server, and there is exactly one, manages the insertion. If it overflows then it splits. Various algorithms have been proposed for split by Kröll and Widmayer [KL94] and by Litwin *et al.* [LNS94a]. The approach for exact match query is also very easy. A client wanting to access point p simply multicasts its request and puts the point coordinates in the message. The pertinent server, and there is exactly one, manages the query. If it finds the point in its bucket it answers with the required information. Otherwise it answers negatively. When the client receives an answer it knows the query is terminated.

The approach is somewhat more complex for partial match and range queries. In this case 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. If we assume that each server answers with its k-d range, then in the case of exact range query we can simply sum all the received k-d ranges and when they add up to the k-d volume of the exact range query we know the query is terminated. Even if theoretically good, from a practical point of view either infinite precision multiplication is available or this approach may be incorrect, if buckets covering very large and very small ranges exist at the same time in the data structure, due to possible roundings. And in any case this approach cannot be applied for partial match and partial range queries, where the k-d volume is infinite. Hence we need a more reliable termination test.

3. ALGORITHMS FOR A TERMINATION TEST

Two approaches have been proposed for a termination test. The first one (CAR) is based on *combining adjacent ranges* in the set of k-d ranges returned by buckets which answered [LNS94b]. The second one (LV) is based on the computation of the *logical volume* of the range query and on its comparison with the logical volume of k-d ranges returned by buckets which answered the query [Nar95]. The first approach can be efficiently implemented using k-d range trees [Nar95] and has a worst case time of $O(kQ(\log\ Q)^k)$ for the whole termination test, using overall a worst case space of $O(Q(\log\ Q)^{k-1})$. The second approach uses balanced binary search trees and has a worst case time of $O(kQ^2)$ for the whole termination test, using overall a worst case space of $O(kQ)$. This latter approach is discussed in more detail below.

3.1. Informal Description of the Logical Volume (LV) Algorithm for Termination Test

To explain the basic idea we use an example in 2-dimensional space (see Fig. 1a). Let us denote by $N_x(N_y)$ the number of distinct split points which divide buckets with a vertical (horizontal) segment. The split lines defined by these split points identify a partition of the whole space in $(N_x + 1)(N_y + 1)$ rectangular cells, called the *logical volume* of the space. A bucket B has a logical volume which is given by the number of cells which are within its range. In Fig. 1a the logical volume of the space is $(3 + 1)(3 + 1) = 16$ and the logical volume of buckets B_2, B_4, B_5, and B_6 is 3, 2, 2, and 1, respectively. We define the logical volume for a range query or a partial match query as the number of cells which are within or intersect its range. For queries Q_1 and Q_2 in the figure, the logical volume is 9 and 12, respectively. Assume that a client knows the logical volume of the query it issues. Assume also that for each bucket *involved* by the query the client is able to know the logical volume of the intersection between the bucket range and the query range. Then the termination test is simply the equality of the query logical volume to the sum of logical volumes of all buckets which have answered. This can be seen for the two queries in Fig. 1a. Query Q_1 can terminate when a logical volume of nine cells is reached; this is contributed by buckets B_1 (2 cells), B_2 (2), B_3 (2), B_5 (2), and B_6 (1). Whichever is the order of arrival of answers when the sum reaches 9 the client knows the query has terminated. For query Q_2 answers come by buckets B_1 (1 cell), B_2 (3), B_4 (2), B_5 (2), B_6 (1), B_7 (1), and B_8 (2); again, whichever is the order when the sum arrives at 12 cells the query terminates.

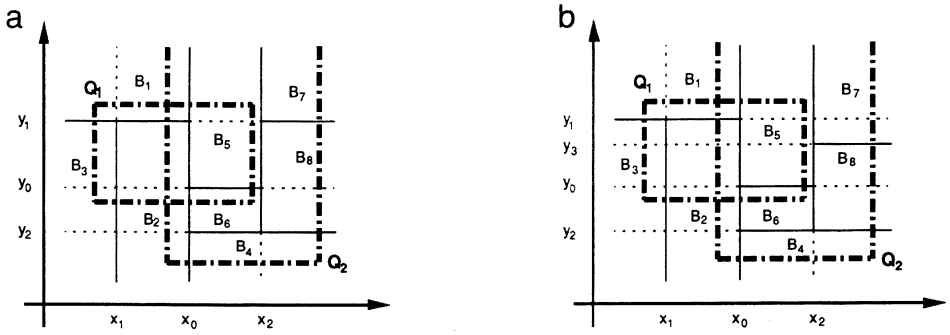

FIG. 1. (a) To the left, (b) to the right.

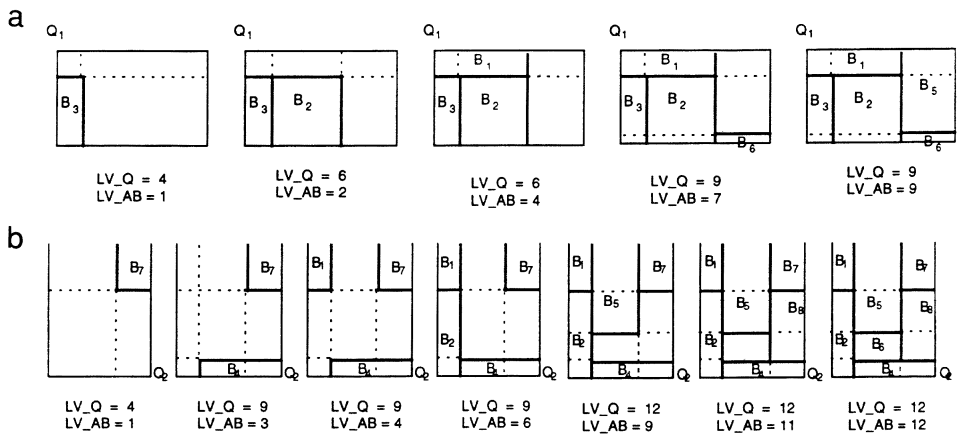

FIG. 2. (a) To the top, (b) to the bottom.

To compute the logical volumes above described is a straightforward process. After each bucket's answer the client simply adds, if they are not already present, the lower and upper limits along coordinate x (y) of the bucket to a current list of split points for coordinate x (y). By using these lists the client can evaluate the current logical volume of the query, LV_Q, and the current logical volume of all buckets which have answered, LV_AB. Whenever a new bucket answers the process is repeated, until LV_AB = LV_Q, when termination point is reached. To see this, consider that when the first bucket answers it is surely LV_Q $\geq$ LV_AB. The new split points possibly introduced by each new bucket do not increase LV_AB more than LV_Q. Therefore, when the equality is reached, this is due to the fact that the last bucket fills the gap with its own logical volume and all involved buckets have answered. As an example, consider the sequence of pictures in Figs. 2a and 2b, where for queries Q_1 and Q_2 of Fig. 1a the evolution of LV_Q and LV_AB after each bucket's answer is shown graphically.

Note that it may happen that some split lines are defined by buckets which do not intersect the query range. This is the case of the boundary line between buckets B7 and B8 for query Q_1 in Fig. 1b. In cases such as this, since neither of the two buckets is involved by the query none of them will answer and such split lines will not be considered in computations of logical volumes. The algorithm maintains its validity, which will be proved after its formal definition.

3.2. A Formal Description of Algorithm LV

For the sake of clarity, the algorithm is presented assuming that buckets answer one at a time. It is straightforward to extend it and its complexity analysis to the case of more buckets answering at the same time.

Let AB denote the current set of buckets which answered query Q. For a bucket B in AB let $B' \neq \emptyset$ denote the range of $B \cap Q$ and let $\lambda_j^{B'}$ and $\Lambda_j^{B'}$ denote the lower and upper limits, respectively, for each coordinate j of B'. Let SP_j denote the ordered list of distinct split points for the jth coordinate ($j = 1, 2, \ldots, k$) defined by buckets in AB. $|SP_j|$ indicates the length of SP_j. Let δ_j and Δ_j ($j = 1, 2, \ldots, k$) denote the lower and upper limits, respectively, for each coordinate j of the query Q. Given two values d and D taken from the domain of the data type X_j, with $d \leq D$, let NSP(j, d, D) denote

Algorithm LV (Logical Volume termination test)

AB := ∅;
repeat
    B := new bucket to answer;
    for j:=1 to k do
(1)   add $\lambda_j^{B'}$ and $\Lambda_j^{B'}$ to SP_j, if not already present;
(2)   update CLS(j,·) of the buckets in AB whose range contains $\lambda_j^{B'}$ or $\Lambda_j^{B'}$
               if they have been added in line 1;
(3)   CLS(j,B) := NSP(j, $\lambda_j^{B'}$, $\Lambda_j^{B'}$)+1;
    AB := AB ∪ B;
(4) LV_AB := $\sum_{\forall B \in AB} \prod_{j=1}^{k}$ CLS(j,B);
(5) LV_Q := $\prod_{j=1}^{k}$ [NSP(j, δ_j, Δ_j)+1];
until LVQ = LV_AB

FIG. 3. Algorithm LV.

the number of split points along the coordinate j in the list SP_j which are internal to the open interval (d, D), open at both ends. Note that $NSP(j, -\infty, +\infty) = |SP_j|$ and if d and D are two consecutive split points in SP_j then $NSP(j, d, D) = 0$. For each bucket B in AB let $CLS(j, B)$ be the current *logical length* of open interval $(\lambda_j^{B'}, \Lambda_j^{B'})$ defined as $CLS(j, B) = 1 + NSP(j, \lambda_j^{B'}, \Lambda_j^{B'})$. Finally, let CLV(B) denote the current *logical volume* of B defined as $CLV(B) = \prod_{j=1}^{k} CLS(j, B)$. A formal statement of Algorithm LV is given in Fig. 3. Algorithm LV can be used with every data type X_j, as long as a total order can be defined on it. But this is always true if range searches on X_j are needed. Moreover the precision of integer multiplication is able to ensure a correct termination even in the case of many buckets of small range. Finally, the same approach based on logical volume can provide in all cases a deterministic termination test in the case of partial match query. In fact, for those coordinates whose range is not, or is only partially, specified in the query, the logical volume approach provide a means of computing their contribute to the logical volume of query space and to the logical volume of buckets which have answered.

3.3. Proof of Correctness Algorithm LV

The proof is by induction on the number of buckets which have answered.

Base step. The answer of the first bucket B, that is its lower und upper limits $\lambda_j^{B'}$ and $\Lambda_j^{B'}$ ($j = 1, 2, \ldots, k$), may introduce 0, 1, or 2 split points on each coordinate. If and only if no split point is introduced for any coordinate then LV_Q = 1 and LV_AB = 1; therefore the algorithm terminates. Otherwise it is LV_Q > LV_AB = 1.

Induction step. Each time a new bucket C answers to query Q, its lower and upper limits $\lambda_j^{C'}$ and $\Lambda_j^{C'}$ may either (i) leave LV_Q unaffected or (ii) increase it.

Case (i) occurs when all $\lambda_j^{C'}$ and $\Lambda_j^{C'}$ are already present in the various SP_js. In this case LV_AB is increased only by the logical volume of $C' = C \cap Q$; if equality between LV_Q and LV_AB is reached the algorithm terminates, otherwise LV_AB cannot become greater than LV_Q. To see this consider that (a) buckets are a partition of the whole space, (b) only buckets whose range is within or intersect with the range of the query answer, (c) the computation of LV_Q and LV_AB in lines 4 and 5 use the same SP_js. Therefore in this case either the algorithm terminates or the invariant LV_Q $>$ LV_AB is maintained.

Case (ii) occurs when new split points are introduced in some SP_js. In this case the increase of LV_AB is given by two terms. One term corresponds to the logical volume of $C' = C \cap Q$ and the same considerations of the previous case apply. The other one corresponds to the fact that new split lines cut buckets which answered in previous steps. But the increment in LV_AB due to new split points cannot be greater than the corresponding increment in LV_Q. This is due to the fact the whenever a new split point is introduced in a certain list SP_h the relative split lines cut a number of buckets which cannot be greater of $\Pi_{j=1,\, j\neq h}^{k}(|SP_j| + 1)$. Therefore also in this case either the equality between LV_Q and LV_AB is reached and the algorithm terminates or the invariant LV_Q $>$ LV_AB is maintained.

Since the number of buckets is finite the algorithm terminates after a finite number of answers with LV_Q $=$ LV_AB. Let us prove that whenever it terminates, only and all the buckets whose range is within or intersects the query range have answered. The "only" part is trivial. For the "all" part, if some bucket has not answered then, due to the fact that the cells define a partition of the whole space, at least one cell has to exist which contributes to LV_Q, since its boundaries are provided by the split lines defined by some other bucket, but which does not contribute to LV_AB, since the bucket which contains it did not answer. This would imply LV_Q $>$ LV_AB, which is a contradiction.

3.4. Complexity Analysis

Let M denote the total number of buckets and Q the number of buckets which answer the query. It is clearly $Q = O(M)$. Each split list can be managed as a balanced binary search tree, which has to be suitably modified to efficiently process insertions (Step 1), the update of CLS(j, $\cdot$) (Step 2), and the computation of NSP function (Steps 3 and 5). Hence each node x stores the number of split points in the subtree rooted at x and the minimum and maximum of their values. The set of all trees uses space $O(kQ)$. With these modifications Steps 1 and 3 can be executed, in the worst case, in time $O(\log Q)$, but Step 2 requires time $O(Q)$ (due to the fact that it may be necessary to update $O(Q)$ bucket ranges. Computation of LV_AB in Step 4 can be done in $O(kQ)$ worst case time and that of LV_Q in Step 5 in $O(k \log Q)$. Therefore each new answer can be processed in worst case time $O(kQ)$. Since the "repeat" cycle is executed $O(Q)$ times we have a worst case time of $O(kQ^2)$ for the whole termination test, using overall a worst case space of $O(kQ)$.

Algorithm LV has a better worst case space complexity than Algorithm CAR, but at the price of a worse worst case time complexity. From a practical point of view, though, it has to be pointed out that: (i) both algorithms are working in main memory, therefore their impact on the overall performance of the distributed data structure is significant only for very large range queries; (ii) the range tree is a more complex data structure than the binary tree used by Algorithm LV; (iii) the worst case analysis should not be

the only argument used in choosing among the algorithms; their average case behavior should also be investigated through simulation or analysis.

4. DATA STRUCTURE WITH INDEX AT CLIENT SITES

In this section we shortly recall the version having an index at client sites to reduce the use of multicast. For more details see [Nar96b, NBP97b] and for a discussion of design issues see [NBP97a].

If clients have an index the use of multicast can be reduced. In fact, a client can search in its index to individuate the server(s) which should answer to its query and can then issue a (set of) point-to-point query. The key observation is that the client index does not need to cover the whole data space, since a *lazy* approach can be taken. Namely, if a client has the pertinent part of the data space covered by its index then search queries can be managed by issuing a (set of) point-to-point query. Otherwise the query is multicasted. The same holds for insertions.

Given the assumption we have on clients behavior it may happen that a client has an out-of-date index. This has two consequences. First, the server which receives the point-to-point query may not be the server managing the whole set of keys involved by the query itself anymore (in this case we say the client has done an *addressing error*). Second, the client index has to be adjusted to avoid repeating the same addressing error again and again. The adjustment of a client index is done by means of index correction messages (ICMs) from the pertinent servers. Adjustment is required when server s that a client considers as the one that is managing a certain set S of keys has split an unknown number of times. Therefore s cannot any more, in general, manage all queries regarding S. But s has some knowledge about the subtree generated by its split. This knowledge is given back to the client in the ICM so that it can avoid repeating the same error. This means that, in general, an ICM contains a part of the overall k-d tree that has to be added to the client index or to substitute a part of it.

A client index is therefore a *collection* of loosely related *subsets* of a k-d tree. We call such a collection a lazy k-d tree (lkd-tree for short). This means that the client knows only some nodes and some paths of the overall k-d tree and has the problem of efficiently managing such a collection. Algorithms for insertion and querying when a client has an index and more details on how to build an lkd-tree and how to efficiently adjust it using ICMs are contained in [Nar96b, NBP97b].

5. DATA STRUCTURE WITH INDEX ALSO AT SERVER SITES

If we also allow servers to have an index, then we can distribute the overall k-d tree (with some possible replications) among all servers. Each server s keeps track of the part of the tree in which s was involved when it was created. Hence multicast can be avoided completely since if a server is not currently pertinent to a query it knows how to forward the query toward the pertinent server(s) anyway. When the pertinent server is reached the answer is sent to the client by following the chain of forwarding servers backward. In this process index adjustment information is added to the answer message by each server in the chain.

The management and behavior of an lkd-tree when an index is also maintained at server sites is conceptually similar to the DRT of Kröll and Widmayer [KW94]. We only note

[at client site]
FIND in the client index the server s whose range contains the k-d point p
point-to-point(p,s)
WHEN answer arrives
  EXECUTE possibly update the index using received information
[at site of server s]
IF the received k-d point p is in the range covered by server s
  THEN insert(p) and possibly split
  ELSE FIND in the index of server s the server s' whose range contains the k-d point p
    point-to-point(p,s')
    WHEN answer arrives
      EXECUTE possibly update server s index using received information
answer to the sender with ack and the current index of server s

FIG. 4. Algorithms for insertions with index both at client and server sites.

that a faster convergence of a client index can be obtained if index adjustment information is always sent back to the client, even if the server has been able to completely answer the query without forwarding. This approach also allows a larger degree of fault tolerance in the case of servers' failure, since it increases the degree of replication of the various parts of the overall index managed by each server. We give, for completeness, algorithms for insertion, exact query and range query respectively in Figs. 4, 5, and 6.

6. PARAMETERS FOR PERFORMANCE EVALUATION

We experimentally investigated the behavior of our data structure under different conditions. As representative classes of queries we considered, beyond insertions, both exact and range queries. Concerning data distributions we analyzed synthetically generated data, studying both uniform and gaussian random distribution, in a 1-dimensional space as well as in a 2-dimensional space. The performance measures used for comparison are:

— *load factor*, i.e., percentage of bucket space used in the average over all servers;

[at client site]
FIND in the client index the server s whose range contains the k-d point p
point-to-point(p,s)
WHEN answer arrives
  EXECUTE possibly update the index using received information
[at site of server s]
IF the received k-d point p is in the range covered by server s
  THEN retrieve(p)
  ELSE FIND in the index of server s the server s' whose range contains the k-d point p
    point-to-point(p,s')
    WHEN answer arrives
      EXECUTE possibly update server s index using received information
answer to the sender with ack, info on p, and the current index of server s

FIG. 5. Algorithms for exact queries with index both at client and server sites.

[at client site]
FIND in the client index the set S of servers containing the whole k-d range q
point-to-point(q',s) for each s in S
{ *q' is the intersection between q and range of s in the client index* }
WHEN answers arrive
EXECUTE possibly update the index using received information
[at site of server s]
find the set P of k-d points covered by the received server k-d range q'
IF the received server k-d range q' strictly contains the range covered by server s
THEN FIND in the index of server s the set S of servers containing the whole k-d range q'
point-to-point(q'',s') for each s' in S
{ *q'' is the intersection between q' and range of s' in the index of server s* }
WHEN answers arrive
EXECUTE possibly update server s index using received information
answer to the sender with ack, P, and the current index of server s

FIG. 6. Algorithms for range queries with index both at client and server sites.

— *average number of messages required for each insertion*, including split messages and ICMs when addressing errors are made;

— *average number of messages required for each query*, including ICMs when addressing errors are made;

— *convergence*, that is the average number of index correction messages required for a new client to obtain an up-to-date version of the overall k-d tree.

We also introduced and studied the *distribution efficiency* of our data structure both from the point of view of a single server and from an overall point of view. With this approach we want to measure how large the waste of computational resources is deriving from the fact that servers are distributed over a communication network. The main purpose of distribution efficiency measures is the comparison between different distributed data structure, since a distributed solution always consumes more computational resources than a nondistributed one. Hence we defined the following two parameters:

— *local overhead* (l_ovh) that measures 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* (g_ovh) that measures the 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 by s the overall number of servers, with $\#rec\_msg(i)$ and $\#pert\_msg(i)$, respectively, the number of messages received by server i and the number of pertinent messages received by server i. Then it is

$$l\_ovh = \frac{\sum_{i=1}^{s} \frac{\#rec\_msg(i) - \#pert\_msg(i)}{\#rec\_msg(i)}}{s}$$

and

$$g\_ovh = \frac{\left(\sum_{i=1}^{s} \#rec\_msg(i)\right) - \#queries}{\sum_{i=1}^{s} \#rec\_msg(i)}.$$

Note that both formulas compute values between zero (the highest distribution efficiency) and one. Also, note that while for insert or exact queries we have $\#queries \equiv \sum_{i=1}^{s} \#pert\_msg(i)$, for range queries the item on the right-hand side of this formula is generally much larger than the one on the left. A local overhead value closer to zero means that the server's CPU is spending less time in processing useless messages. A global overhead value closer to zero means that each query is processed by the distributed data structure with a lower number of overhead messages. Finally, we also investigated the behavior of the distributed indexes both in terms of distribution of heights of client and server indexes and in terms of the largest number of messages required to answer a query.

7. EXPERIMENTS

In this section we report results of the simulation of the behavior of our distributed data structure. We implemented it using the CSIM simulation software package [Sch92]. This approach was followed also by Litwin *et al.* for LH* and RP* and by Kröll and Widmayer for DRT. We compared the behavior of our data structure with that of LH*, DRT, and RP*. We did not implement any of these structures, hence to make significant comparisons we have used, for performing our experiments, the same parameter space and have chosen the same sample points in the parameter space which are reported in papers describing them. The experimental setting is therefore fully comparable. Note, though, that the two performance parameters introduced to measure distribution efficiency, namely *global overhead* and *local overhead*, are, to our knowledge, not known in the literature, hence we have no data about it for other data structures.

7.1. Performance Results for Insert

For load factor (Tables 1 and 2) and insertion (Tables 3–6) we took as a reference the results reported in [KW94], where DRT and LH* are compared. We hence performed a sequence of 10,000 insertions of distinct integer keys on an empty file, under different bucket capacities and number of clients (of different activity levels). Integer keys to be inserted were drawn independently from two different distributions: a uniform distribution in the range from 0 to 10^6 and a gaussian distribution generated according to the polar method of Knuth [Knu81], with a mean of 10^3. Each experiment was repeated five times (for the uniform distribution) or seven times (for the gaussian distribution).

Table 1
Load Factor for the *Uniform* Distribution

						1-dimensional		2-dimensional	
Number	BC	NC	INS	LH* load	DRT load	NS	Load	NS	Load
1	17	1	10,000	0.570	0.707	837.2	0.70264	834.2	0.70521
2	33	1	10,000	0.566	0.701	431.2	0.70277	435.2	0.69633
3	40	1	10,000	0.532	0.697	356.8	0.70071	362.6	0.68953
4	40	2	5,000 5,000	0.531	0.697	356.8	0.70071	362.6	0.68953
5	40	3	3,333 3,333 3,333	0.532	0.697	356.8	0.70064	362.6	0.68946
6	40	3	6,977 2,326 697	0.532	0.697	356.8	0.70071	362.6	0.68953

BC, capacity of each bucket; NC, number of clients; INS, number of insert for each client; NS, average number of servers.

The three variants of the data structure discussed in this paper all reported the same value of load factor, hence results are reported only once (Table 1 for the uniform distribution and Table 2 for the gaussian one). Load factor has practically the same values for our data structure as for DRT (which is better than LH*).

Results in Tables 3 and 4 regard the average number of messages for each insert for all 1- and 2-dimensional distributions. We report, for the experiments regarding more than one client, each client value and the average value (shown with boldface numbers). The average number of messages for each insert is slightly better for our data structure without

Table 2
Load Factor for the *Gaussian* Distribution

						1-dimensional		2-dimensional	
Number	BC	NC	INS	LH* load	DRT load	NS	Load	NS	Load
1	40	1	10,000	0.264	0.697	359.6	0.69533	356.4	0.70142
2	40	2	5,000 5,000	0.260	0.694	359.6	0.69533	356.4	0.70142
3	40	3	3,333 3,333 3,333	0.250	0.697	359.6	0.69526	356.4	0.70135
4	40	3	6,977 2,326 697	0.257	0.695	359.6	0.69533	356.4	0.70142

BC, capacity of each bucket; NC, number of clients; INS, number of insert for each client; NS, average number of servers.

Table 3
Average Number of Messages for Each Insert (*Uniform* Distribution)

			No index		Client index		Server index	
Number	LH*	DRT	1-dim.	2-dim.	1-dim.	2-dim.	1-dim.	2-dim.
1	1.4265	1.2932	1.33448	1.33328	1.47840	1.48152	1.55576	1.55636
2	1.2301	1.1578	1.17208	1.17368	1.25360	1.25632	1.29436	1.29764
3	1.2121	1.1371	1.14232	1.14464	1.21032	1.21320	1.24432	1.24748
4	1.2262	1.2315	1.14384	1.14208	1.27416	1.27296	1.33932	1.33840
			1.14080	1.14720	1.27264	1.27832	1.33892	1.34424
			1.14232	**1.14464**	**1.27340**	**1.27564**	**1.33912**	**1.34132**
5	1.2389	1.3183	1.14329	1.14569	1.33243	1.33339	1.42700	1.42814
			1.13345	1.13945	1.30651	1.31299	1.41518	1.42280
			1.15026	1.14881	1.33411	1.33771	1.43270	1.43234
			1.14233	**1.14465**	**1.32435**	**1.32803**	**1.42496**	**1.42776**
6	1.2329	1.2874	1.14230	1.14425	1.23804	1.24119	1.28539	1.28858
			1.14308	1.14377	1.38160	1.37524	1.52046	1.53001
			1.14003	1.15151	1.48379	1.46313	2.01119	1.99455
			1.14232	**1.14464**	**1.28856**	**1.28784**	**1.39066**	**1.39394**

index (both for 1-dimensional and 2-dimensional data) than for DRT and LH*, while it is slightly worse for our data structure with index on clients only or on both clients and servers (both for 1-dimensional and 2-dimensional data) than for DRT and LH*.

Tables 5 and 6 present distribution efficiency results for the two considered data distributions. As you can see, overhead of the data structure is almost independent from the considered number of dimensions and from the specific distribution of data. Overhead is greatest for the version without index but very low for the version with index at server

Table 4
Average Number of Messages for Each Insert (*Gaussian* Distribution)

			No index		Client index		Server index	
Number	LH*	DRT	1-dim.	2-dim.	1-dim.	2-dim.	1-dim.	2-dim.
1	1.4534	1.1394	1.14343	1.14217	1.21157	1.21020	1.24564	1.24421
2	1.5351	1.2401	1.14114	1.14537	1.27246	1.27497	1.33811	1.33977
			1.14571	1.13897	1.27606	1.26971	1.34123	1.33509
			1.14343	**1.14217**	**1.27426**	**1.27234**	**1.33967**	**1.33743**
3	1.5999	1.3384	1.14847	1.14624	1.33586	1.32969	1.43033	1.42643
			1.14521	1.14299	1.33209	1.33295	1.42784	1.42806
			1.13664	1.13733	1.32249	1.32378	1.41824	1.41970
			1.14344	**1.14219**	**1.33015**	**1.32880**	**1.42547**	**1.42473**
4	1.5634	1.3188	1.14505	1.13792	1.24104	1.23350	1.28872	1.28098
			1.14224	1.15158	1.39405	1.39885	1.52328	1.53292
			1.13117	1.15331	1.46854	1.51117	1.99959	2.02132
			1.14343	**1.14217**	**1.29249**	**1.29131**	**1.39283**	**1.39119**

Table 5
Global and Local Overhead during Insert Operation (*Uniform* Distribution)

		No index		Client index		Server index	
Number		Global	Local	Global	Local	Global	Local
1	1-dim.	0.99760	0.99814	0.97541	0.98096	0.19433	0.35519
	2-dim.	0.99760	0.99811	0.97145	0.97769	0.19433	0.35017
2	1-dim.	0.99539	0.99639	0.89920	0.92138	0.11269	0.23998
	2-dim.	0.99540	0.99638	0.90159	0.92345	0.11376	0.24117
3	1-dim.	0.99443	0.99557	0.86004	0.88914	0.09532	0.21230
	2-dim.	0.99443	0.99562	0.86211	0.89236	0.09649	0.21895
4	1-dim.	0.99443	0.99557	0.92331	0.93940	0.12046	0.22610
	2-dim.	0.99443	0.99562	0.92361	0.94012	0.12133	0.23231
5	1-dim.	0.99443	0.99557	0.95199	0.96197	0.14207	0.23754
	2-dim.	0.99443	0.99562	0.95308	0.96308	0.14303	0.24352
6	1-dim.	0.99443	0.99557	0.95431	0.96390	0.14099	0.24430
	2-dim.	0.99443	0.99562	0.95787	0.96690	0.14096	0.24858

sites. This is not surprising, since it is exactly the purpose of the introduction of indexes. Also, it is reasonable that the lowest overhead is reached with the highest bucket capacity and the lowest number of clients (Experiment 3 in Table 5). Remember that there are no measures in the literature about this performance parameter for other data structures.

As a further experimental analysis for the version with index at server sites we studied (Table 7) the height reached by the whole index, when considered as a single tree. Since no balancing operation is supported in the index, it is theoretically possible to have a tree which has degenerated to a linear list. The values of the height provide an indication about the overall shape of a tree. Height is compared both to the logarithmic lower bound for the case of a nondistributed balanced search tree and to the square root lower bound for the case of a distributed search tree where balancing is explicitly maintained [KW95]. Even if lower bounds are worst case values and height values in our experiment are

Table 6
Global and Local Overhead during Insert Operation (*Gaussian* Distribution)

		No index		Client index		Server index	
Number		Global	Local	Global	Local	Global	Local
1	1-dim.	0.99442	0.99564	0.86050	0.89167	0.09583	0.22496
	2-dim.	0.99442	0.99558	0.86017	0.88991	0.09527	0.20989
2	1-dim.	0.99442	0.99564	0.92205	0.93961	0.12075	0.23859
	2-dim.	0.99442	0.99558	0.92153	0.93806	0.12001	0.22349
3	1-dim.	0.99442	0.99564	0.94788	0.95947	0.14232	0.24980
	2-dim.	0.99442	0.99558	0.94931	0.96004	0.14199	0.23489
4	1-dim.	0.99442	0.99564	0.95181	0.96260	0.14276	0.25779
	2-dim.	0.99442	0.99558	0.95236	0.96241	0.14118	0.24228

Table 7
Behavior of the Version with Index at Server Sites during Insert Operation
(U, Uniform; G, Gaussian)

	Number of servers		*lg* (number of servers)		Index height		$\sqrt{\text{Number of servers}}$		Max number of messages	
	1-dim.	2-dim.	1-dim.	2-dim.	1-dim.	2-dim.	1-dim.	2-dim.	1-dim.	2-dim.
U-1	837.2	834.2	9.71	9.70	14.6	14.8	28.93	28.88	10.0	9.4
U-2	431.2	435.2	8.75	8.77	13.0	12.4	20.77	20.86	8.6	8.4
U-3, 4, 5, 6	356.8	362.6	8.48	8.50	12.2	12.2	18.89	19.04	8.8	8.6
G-1, 2, 3, 4	359.6	356.4	8.49	8.48	12.3	12.0	18.96	18.88	8.8	8.6

average measures, Table 7 shows that the data structure behaves well in practice. This is not surprising since for a uniform distribution the expected height of a search tree under random insertion is logarithmic in the number of insertions [Knu73, CLR90]. Once again, the behavior of the data structure is almost independent from the considered number of dimensions and from the specific distribution of data. A further measure shown in the same table is the maximum number of messages required to complete an operation; remember that the theoretical worst case is two times the height of the tree. In the first column U-1 to U-6 refers to the six cases considered in Table 1, while G-1 to G-4 refers to the four cases considered in Table 2.

The study of the behavior of index for the version with index at server sites is completed by an analysis of the average distribution of the number of leaf nodes (i.e., the number of pointers to other servers) in the index of each server. In a certain sense this measures the average distribution of the branching factor of servers. It is shown in Table 8 for the 1-dimensional case. A new server has one leaf (that is, the root). On each row the first line refers to the uniform distribution and the second line to the gaussian one.

The good behavior in practice of the version with index at server sites discussed in Table 7 with reference to the height of the index is confirmed by Table 9. This shows, for the 2-dimensional case, the same parameters of Table 7 but refers to a larger number of insertions. Bucket capacity is 40 and only one client is active.

We also studied the behavior of the load factor of the version with index at server sites for a larger number of insertions. Results are shown in Table 10, confirming the good behavior of the data structure for a larger number of insertions. Bucket capacity is 40, only one client is active, and a 2-dimensional space is considered.

7.2. Performance Results for Exact Search

For an exact search (Table 11) we also took as a reference the results reported in [KW94]. Hence we performed, under different bucket capacities and number of clients (of different activity levels), a sequence of 10,000 searches on a file with 10,000 keys, randomly generated by a different client. We used, like Kröll and Widmayer [KW94], a uniform distribution in the range from 0 to 10^6. Keys to be searched for were drawn independently from a uniform distribution in the range from 0 to 10^6. Each experiment was repeated five times. We also report results for the 2-dimensional case, showing that the same performance as the 1-dimensional case is obtained. The version of our data

Table 8
Distribution of Servers with Respect to Number of Leaves in Their Index (1–dimensional)

		Number of insert operations							
		10,000		30,000		50,000		100,000	
Leaves		Servers	%	Servers	%	Servers	%	Servers	%
1	U[a]	177.40	48.92	527.20	48.98	883.33	49.15	1766.33	49.47
	G[b]	174.60	49.04	529.60	49.23	882.00	49.10	1772.33	49.44
2	U	95.60	26.37	284.40	26.42	467.33	26.00	912.00	25.54
	G	94.40	26.52	274.60	25.53	462.33	25.74	910.67	25.40
3	U	44.80	12.36	133.40	12.39	225.00	12.52	447.00	12.52
	G	42.60	11.97	141.80	13.18	233.00	12.97	463.00	12.92
4	U	22.40	6.18	64.40	5.98	111.00	6.18	231.67	6.49
	G	22.80	6.40	64.20	5.97	111.67	6.22	227.00	6.33
5	U	12.20	3.36	34.00	3.16	60.33	3.36	103.00	2.88
	G	9.80	2.75	34.20	3.18	56.00	3.12	99.33	2.77
6	U	5.20	1.43	17.00	1.58	20.67	1.15	58.67	1.64
	G	6.40	1.80	16.20	1.51	25.33	1.41	56.00	1.56
7	U	2.20	0.61	8.40	0.78	19.00	1.06	25.67	0.72
	G	3.00	0.84	7.40	0.69	13.00	0.72	30.67	0.86
8	U	1.80	0.50	4.20	0.39	4.33	0.24	14.33	0.40
	G	1.40	0.39	3.40	0.32	6.67	0.37	12.67	0.35
9	U	0.40	0.11	1.40	0.13	2.67	0.15	6.00	0.17
	G	1.00	0.28	3.00	0.28	3.00	0.17	6.33	0.18
10	U	0.60	0.17	1.00	0.09	2.33	0.13	1.67	0.05
	G	0.00	0.00	0.20	0.02	1.33	0.07	3.67	0.10
11	U	0.00	0.00	1.00	0.09	0.67	0.04	2.67	0.07
	G	0.00	0.00	1.00	0.09	1.33	0.07	1.67	0.05
12	U	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.03
	G	0.00	0.00	0.20	0.02	0.67	0.04	0.66	0.02
13	U	0.00	0.00	0.00	0.00	0.67	0.04	0.33	0.01
	G	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.01
14	U	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.01
	G	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.01

[a]Uniform.
[b]Gaussian.

structure with index only at client sites always has a better behavior than both DRT and LH*. The version with index both at client and server sites is always better than DRT (both structures do not use multicast). We did not test the version without index, since in this case a single multicast request is always sufficient, hence each query is answered with exactly two messages.

Table 12 presents results for distribution efficiency measures. Again, the version without index was not tested. In fact, by manipulating formulas in Section 6 it can

Table 9
Behavior of the Version with Index at Server Sites for Large Numbers of Insert Operations

Number of inserts	Number of servers		*lg* (number of servers)		Index height		$\sqrt{\text{Number of servers}}$		Max number of messages	
	Uniform	Gaussian	Uniform	Gaussian	Uniform	Gaussian	Uniform	Gaussian	Uniform	Gaussian
10,000	362.6	356.0	8.50	8.48	12.2	12.0	19.04	18.87	8.6	8.6
30,000	1076.4	1075.8	10.07	10.07	14.0	14.0	32.81	32.80	10.2	10.4
50,000	1797.3	1796.3	10.81	10.81	14.7	15.0	42.39	42.38	11.3	10.7
100,000	3570.7	3584.7	11.80	11.81	16.0	16.3	59.76	59.87	12.3	11.7

be seen that both the global overhead and local overhead are equal to $1 - 1/s$, where s is the number of servers.

Concerning convergence (Table 13), we took as a reference the results reported in [LNS94a]. Hence we measured, under different bucket capacities, the speed of a new client to bring its index up-to-date with the overall k-d tree. Clearly, here LH* (that is not order-preserving) has the fastest convergence, being upper-bounded by the logarithm of the overall number of buckets [LNS93, LNS94a]. But our data structure with index only at client sites has practically the same behavior of RP*$_C$ (i.e., RP* with index only at client sites), both for 1-dimensional and 2-dimensional data. The version of our data structure with index also at server sites has a slower convergence than RP*$_S$, which is the variant of RP*$_C$ where some servers are fully dedicated to the management of the index. This is reasonable since by sacrificing some servers exclusively for index management RP*$_S$ obtains a much higher branching factor, but a lower degree of fault tolerance against server failures.

7.3. Performance Results for Range Query

Concerning range queries there are no experimental studies in the literature, since both LH* [LNS93, LNS96] and RP* [LNS94a] and DRT [KW94] only considered insertion

Table 10
Behavior of the Version with Index at Server Sites for Large Numbers of Insert Operations

Number	Number of inserts		Number of servers	Load factor	Number of messages
1	10,000	uniform	362.6	0.68953	1.24748
		gaussian	356.0	0.70227	1.24406
2	30,000	uniform	1076.4	0.69682	1.24591
		gaussian	1075.8	0.69722	1.24533
3	50,000	uniform	1797.3	0.69549	1.24625
		gaussian	1796.3	0.69588	1.24605
4	100,000	uniform	3570.7	0.70015	1.24537
		gaussian	3584.7	0.69743	1.24651

Table 11
Average Number of Messages for Each Exact Search (*Uniform* Distribution)

						Client index		Server index	
Number	BC	NC	QUE	LH*	DRT	1-dim.	2-dim.	1-dim.	2-dim.
1	17	1	10,000	2.0006	2.1127	2.00010	2.00010	2.05136	2.05148
2	33	1	10,000	2.0002	2.0587	2.00010	2.00010	2.02744	2.02752
3	40	1	10,000	2.0004	2.0500	2.00010	2.00010	2.02368	2.02388
4	40	2	5,000	2.0008	2.0873	2.00020	2.00020	2.03944	2.04248
			5,000			2.00020	2.00020	2.03888	2.03600
						2.00020	**2.00020**	**2.03916**	**2.03924**
5	40	3	3,333	2.0012	2.1200	2.00030	2.00030	2.05005	2.05413
			3,333			2.00030	2.00030	2.05160	2.05101
			3,333			2.00030	2.00030	2.05028	2.05065
						2.00030	**2.00030**	**2.05065**	**2.05193**
6	40	3	6,977	2.0012	2.1200	2.00014	2.00014	2.03039	2.03044
			2,326			2.00043	2.00043	2.06965	2.07206
			697			2.00143	2.00143	2.20373	2.20488
						2.00030	**2.00030**	**2.05160**	**2.05228**

BC, capacity of each bucket; NC, number of clients; QUE, number of queries for each client.

and exact search. We performed range queries with three different ratios between query area and total area, namely 0.1, 1, and 10%. In the 2-dimensional case we tested range queries with two different shape ratios: *square* (that is 1:1) and *rectangular* (1:3). These

Table 12
Global and Local Overhead during Exact Search Operation (*Uniform* Distribution)

		Client index		Server index	
Number		Global	Local	Global	Local
1	1-dim.	0.98593	0.98592	0.32124	0.26383
	2-dim.	0.98583	0.98582	0.32905	0.26532
2	1-dim.	0.94896	0.94895	0.19254	0.15728
	2-dim.	0.94984	0.94983	0.18366	0.14956
3	1-dim.	0.92716	0.92718	0.15707	0.12777
	2-dim.	0.92930	0.92931	0.16756	0.13634
4	1-dim.	0.96210	0.96211	0.20323	0.17108
	2-dim.	0.96326	0.96326	0.21758	0.18083
5	1-dim.	0.97439	0.97439	0.23307	0.19827
	2-dim.	0.97518	0.97518	0.23455	0.19853
6	1-dim.	0.97305	0.97305	0.25066	0.20999
	2-dim.	0.97393	0.97393	0.25525	0.21540

Table 13
Average Number of Messages Required to a New Client to Build an Up-to-Date Index

					Client index		Server index	
	BC	RP^*_C	RP^*_S	LH*	1-dim.	2-dim.	1-dim.	2-dim.
1	50	2867	22.9	8.9	2865	2865	634.57	634.57
2	100	1438	11.4	8.2	1447	1447	316.71	316.71
3	250	543	5.9	6.8	543	543	120.57	120.57
4	500	258	3.1	6.4	256	256	58.42	58.42
5	1000	127	1.5	5.7	127	127	29.71	29.71
6	2000	63	1.0	5.2	63	63	13.00	13.00

BC, capacity of each bucket.

experiments were done using two different frameworks: in the first (*static*), a sequence of 10,000 searches was executed on a file with 10,000 keys, randomly generated by a different client; in the second (*dynamic*) the structure was first filled with only 1,000 points then 1 range query was done every 10 further insertions, until all the 10,000 points were inserted (so the number of queries was 900). Bucket capacity in all cases had the value of 40. Each experiment was repeated five times. In Table 14 you can see the results regarding the average number of messages required to answer each range query, in the dynamic framework, for the version without index and the version with index at server sites. Regarding the version with index at client sites remember that if the client has all servers in its index it always issues point-to-point queries. For this version we report in Table 15 the results in the static framework also.

In Tables 16 and 17 we present overhead results for the considered distribution. Remember that there are no measures in the literature about this performance parameter for other data structures. As it is expected, distribution efficiency is highest in the version with index at server sites. Table 16 reports global and local overhead for the dynamic framework and for the version without index and with index at server sites. Table 17 reports results for the version with index at client sites, both for static and dynamic frameworks.

Table 14
Average Number of Messages for Each Range Query
(*No Index* and *Server Index—Dynamic*)

	No index			Server index		
		2-dimensional			2-dimensional	
%	1-dimensional	Square	Rectangular	1-dimensional	Square	Rectangular
0.1	2.20667	3.02556	3.24822	2.53489	4.12133	4.55622
1.0	3.95867	6.23311	6.79333	5.82489	10.42156	11.52667
10.0	20.62422	21.70622	21.85333	38.93889	41.23200	41.53467

Table 15
Average Number of Messages for Each Range Query (*Client Index*)

	Static			Dynamic		
		2-dimensional			2-dimensional	
%	1-dimensional	Square	Rectangular	1-dimensional	Square	Rectangular
0.1	2.57067	4.69556	5.31667	2.48511	4.00133	4.42911
1.0	8.54889	14.81156	16.43956	5.80244	10.10978	11.26111
10.0	68.73733	68.65289	68.29511	39.06267	40.71044	40.76111

8. CONCLUSIONS

We have discussed in this paper a scalable distributed data structure for k-dimensional point data, with optimal search algorithm for exact, partial, and range searches. The set of k-d points is managed in a scalable way, i.e., it can be dynamically enlarged with insertion of new points. We have also proposed new measures to evaluate the efficiency of distributing the management of a data structure over a communication network.

Experiments discussed in previous section show that the load factor is as good for our data structure as it is for DRT and RP* (these structures being based on key comparison) and it is better than LH* (this structure being based on hashing). Values of the average number of messages for insert and query operations show that if multicast is available we obtain the same performance of RP*, not only in the 1-dimensional case, but also for a multi-dimensional point set. If multicast is not available we are not worse than DRT. For range queries there was no previous result in the literature. Experiments we have reported prove that our data structure makes possible an efficient management and querying of a set of multi-dimensional points in a distributed framework. Experiments also seem to

Table 16
Global and Local Overhead during Range Query
(*No Index* and *Server Index—Dynamic*)

		No index		Server index	
%		Global	Local	Global	Local
0.1	1-dim.	0.99447	0.99549	0.15451	0.24910
	square	0.99447	0.99520	0.20642	0.25311
	rectangular	0.99447	0.99352	0.36269	0.20121
1.0	1-dim.	0.99447	0.99470	0.24386	0.21562
	square	0.99447	0.99510	0.21746	0.24876
	rectangular	0.99447	0.98675	0.64356	0.12325
10.0	1-dim.	0.99447	0.98709	0.62953	0.10933
	square	0.99447	0.99377	0.34313	0.20619
	rectangular	0.99447	0.98671	0.64515	0.12466

Table 17
Global and Local Overhead during Range Query (*Client Index*)

		Static		Dynamic	
%		Global	Local	Global	Local
0.1	1-dim.	0.89464	0.98821	0.91081	0.92755
	square	0.86178	0.97089	0.91629	0.92738
	rectangular	0.85191	0.96476	0.92101	0.90766
1.0	1-dim.	0.78250	0.91868	0.91833	0.92151
	square	0.74262	0.83566	0.91654	0.92613
	rectangular	0.72589	0.80734	0.92891	0.83120
10.0	1-dim.	0.71921	0.20023	0.92800	0.83023
	square	0.74010	0.29907	0.92032	0.91058
	rectangular	0.74474	0.32308	0.92785	0.82903

suggest that using a pure multicast approach is not a good solution for application with tight time requirements.

Our structure has the advantage over its competitors of being able to manage both 1-dimensional and multi-dimensional data (like DRT is able to do) but can operate both using multicast and avoiding its use (like RP* is able to provide). We also argued that our solution is able to provide a better fault tolerance against server failures.

Work in progress is concerned with the extension to other multi-dimensional data structures and to extended objects [Bar96, Pep97]. Candidates under considerations for this are R^+-trees, quad-trees, and grid-files. A further line of investigation is considering a fully dynamic (i.e., also supporting deletions) version of the distributed data structure. A first proposal is described in [BNP97]. Finally, an explicit analysis of fault tolerance issues is highly desirable.

ACKNOWLEDGMENTS

The first author thanks Witold Litwin, Marie-Anne Neimat, and Donovan Schneider for having introduced him to the field of distributed spatial data structures and for many interesting discussions on the topics here presented. Discussions with Brigitte Kröll and Peter Widmayer on various issues regarding the efficient management of distributed spatial data structures were very useful and provided him with many valuable insights.

REFERENCES

[Bar96] F. Barillari, "Definizione ed Analisi di Strutture di Dati Distribuite," Master Degree Thesis in Computer Science (in Italian), Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, 1996.

[Ben75] J. L. Bentley, Multidimensional binary search trees used for associative searching, *Comm. Assoc. Comput. Mach.* **18** (1975), 509–517.

[BNP97] F. Barillari, E. Nardelli, and 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, Università di L'Aquila, July 1997, submitted for publication.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," McGraw-Hill, New York, 1990.

[Dev93] R. Devine, Design and implementation of DDH: a distributed dynamic hashing algorithm, *in* "4th Int. Conf. on Foundations of Data Organization and Algorithms (FODO)," Chicago, 1993.

[Gra88] J. Gray, The cost of messages, *in* "7th ACM Symp. on Principles of Distributed Systems," 1988.

[JC92] T. Johnson and A. Colbrook, A distributed data-balanced dictionary based on the B-link tree, *in* "Int. Parallel Processing Symp.," pp. 319–325, 1992.

[JK93] T. Johnson and P. Krishna, Lazy updates for distributed search structures, *in* "ACM SIGMOD Int. Conf. on Management of Data," Washington, 1993.

[KF92] I. Kamel and C. Faloutsos, Parallel R-trees, *in* "ACM SIGMOD Int. Conf. on Management of Data," CA, 1992.

[Knu73] D. Knuth, Sorting and searching, "The Art of Computer Programming," Vol. 3, Addison Wesley, Reading, MA, 1973.

[Knu81] D. Knuth, Seminumerical algorithms, "The Art of Computer Programming," Vol. 2, Addison Wesley, Reading, MA, 1969. [2nd edition, 1981]

[KW94] B. Kröll and P. Widmayer, Distributing a search tree among a growing number of processors, *in* "ACM SIGMOD Int. Conf. on Management of Data," pp. 265–276, Minneapolis, MN, 1994.

[KW95] B. Kröll and P. Widmayer, Balanced distributed search trees do not exist, *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.

[LN96] W. Litwin and M. A. Neimat, "*k*-RP$^*_S$: A High Performance Multi-Attribute Scalable Data Structure," *in* "4th International Conference on Parallel and Distributed Information System," December 1996.

[LNS93] W. Litwin, M.-A. Neimat, and D. A. Schneider, LH*—Linear hashing for distributed files, *in* "ACM SIGMOD Int. Conf. on Management of Data," Washington, D.C., 1993.

[LNS94a] W. Litwin, M.-A. Neimat, and D. A. Schneider, RP*—A family of order-preserving scalable distributed data structures, *in* "20th Conf. on Very Large Data Bases," Santiago, Chile, 1994.

[LNS94b] W. Litwin, M. A. Neimat, and D. Schneider, "*k*-RP$^*_N$: A Spatial Scalable Distributed Data Structure," manuscript, July 1994.

[LNS96] W. Litwin, M.-A. Neimat, and D. A. Schneider, LH*—A scalable distributed data structure, *ACM Trans. Database Systems* **21,** No. 4 (1996), 480–525.

[MS91] G. Matsliach and O. Shmueli, An efficient method for distributing search structures, *in* "1st Int. Conf. on Parallel and Distributed Information Systems (PDIS'91)," Miami Beach, 1991.

[Nar95] E. Nardelli, "Some Issues on the Management of *k*-d Trees in a Distributed Framework," Technical Report 76, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, January 1995.

[Nar96a] E. Nardelli, "Efficient Management of Distributed *k*-d Trees," Technical Report 101, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, March 1996.

[Nar96b] E. Nardelli, Distributed *k*-d trees, *in* "XVI Int. Conf. of the Chilean Computer Science Society (SCCC'96)," Valdivia, Chile, November 1996.

[NBP97a] E. Nardelli, F. Barillari, and M. Pepe, Design issues in distributed searching of multi-dimensional data, *in* "3rd International Symposium on Programming and Systems," Algiers, Algeria, to be published in the "Lecture Notes in Artificial Intelligence," Springer-Verlag, Berlin/New York.

[NBP97b] E. Nardelli, F. Barillari, and M. Pepe, "Distributed Searching of Multi-Dimensional Data," Technical Report 137, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, April 1997.

[Pep97] M. Pepe, "Prestazioni del *k*-d Tree Distribuito Senza Multicast," Master Degree Thesis in Computer Science (in Italian), Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, 1997.

[PK90] S. Pramanik and M. H. Kim, Parallel processing of large node B-trees, *IEEE Trans. Comput.* **39,** 9 (Sept. 1990), 1208–1212.

[Sch92] "CSIM Reference Manual," Technical Report, MCC, Austin, Texas, 1992.

[SL91] B. Seeger and P.-Å. Larson, Multi-disk B-trees, *in* "ACM SIGMOD Int. Conf. on Management of Data," 1991.

[VBW94] R. Vingralek, Y. Breitbart, and G. Weikum, Distributed file organization with scalable cost/performance, *in* "ACM SIGMOD Int. Conf. on Management of Data," Minneapolis, MN, 1994.

ENRICO NARDELLI is an associate professor of computer science at the University of L'Aquila and an invited research scientist at the Institute for System Analysis and Computer Science of the Italian National Research Council in Rome. He carries out research in various fields of theoretical and applied computer science, namely the design and analysis of algorithms and data structures, the definition and analysis of data models, the definition and design of tools, and environments for interaction with information sytems. In particular, he is working on physical data structures for spatial/geometric data in both centralized and distributed environments and on the design and analysis of graph and network algorithms. As far as the database field is concerned, he works on the definition and analysis of data models, with particular attention to spatial and geographic data.

FABIO BARILLARI and MASSIMO PEPE are Ph.D. students at the University of L'Aquila. Their main area of interest is the design and performance analysis of distributed data structures for spatial and geometrical data.

Received April 7, 1997; revised January 23, 1998; accepted January 26, 1998