Distributed *k*-d trees[§]

Enrico Nardelli$^{(1,2)}$

(1) Dipartimento di Matematica Pura ed Applicata, Univ. di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy, nardelli@univaq.it.

(2) Istituto di Analisi dei Sistemi ed Informatica, C.N.R., Viale Manzoni 30, 00185 Roma, Italy.

Abstract

In this paper we present a generalization of the *k*-d tree data structure suitable for an efficient management and querying in a distributed framework. We present optimal searching algorithm for exact, partial, and range search queries. 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 set of *k*-d points is managed in a scalable way, i.e. it can be dynamically enlarged with insertion of new points. We consider distributed environments where multicast (i.e. restricted broadcast) is allowed but show how to use it only when necessary.

Key words and phrases: distributed data structure, multi-dimensional data, k-d-tree, multicast.

1. Introduction

With the striking advance of communication technology it is 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, in the order of 10-100MB/sec, and many powerwul and cheap workstations, in the order of 50-100 MIPS[1]. Many organizations have this kind of computing power: large organizations have easily a cumulative amount of main memory in the order of tenths of GB [LNS94b].

In this context it is usually available the *multicast* protocol. In this protocol with a single message 1 source reaches N destinations. This is to be intended in the sense that a multicast message has a communication cost which is the same of a point-to-point message. In other words the time used by a multicast message to reach all its destination is the same used by a point-to-point message to reach its unique destination.

This is possible since each site has a machine attached to the network by means of a *controller*. The task of the controller is to recognize messages that are of interest to the associated site. In the *point-to-point protocol* each message which is put on the network will reach only the site which is its destination. Other controllers will discard it without dispatching it to the associated site. In the *multicast protocol* messages which are put on the network may be recognized by a (possibly large) set of controllers. Each of these controllers will fetch the message and deliver it to the machine in the associated site.

The physical communication layer of most common networks, like Ethernet, is based on a bus architecture easily supporting a multicast protocol. Hence multicast is currently available in most networks also at the

[§] Parts of this work were done while the 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. The work of Enrico Nardelli is partially supported by the "Algoritmi, Modelli di Calcolo e Strutture Informative" 40%-Project of the italian Ministry for University and Scientific & Technological Research (MURST).

[1] Note that these assumptions are strongly realistic since recent versions of Ethernet already provide 100Mb/sec and optical networks, like FDDI, provide 100Mb-1Gb/sec, and with less than 10K$ it is now possible to buy a 50-100 MIPS workstation with 16-32 MB of main memory.

application level. Note that an uncontrolled use of multicast may somehow degrade performances of end-user applications, by forcing a machine to anyhow process the message delivered to it by its associated controller, even if it is useless for the application itself.

Many of the newest applications deals with very large amounts of complex (i.e. multi-attribute) data, that can be represented as k-dimensional points. An efficient searching of a k-dimensional space in a distributed framework is thus one of the main problems. Our objective is to be able to answer to the following queries, which are the fundamental ones for a data structure for k-d points: exact match, partial match, range. We also want a structure able to scale up to adapt itself to a growing number of points. Hence we want no centralized control, which is a serialization point.

Machines which manage the data structure and answer to queries are called *servers*. Machines which need to access or manipulate the k-d points for their application needs are called *clients*. In a network environment one cannot assume clients are always connected to the network to be kept up-to-date with the state of the data structure. The use of multicast is therefore essential in reaching overall efficiency. Due to its impact on overall performances, it should be used wisely and not in an uncontrolled manner. Moreover, a client has to be able to determine when the multicast query is terminated, i.e. when every server interested by the query has answered.

We do not make any particular assumption on how many clients are present and their frequency of access of the data structure. These can vary over the time. We do not consider issues regarding server failures. Hence we assume all servers are always on and working and we do not present strategies to cope with recovering the complete answer to a query when one (or more) of the servers are down.

In our environment we are concerned with efficiency with respect to the communication network. Therefore the *performance measure* is given in terms of the overall number of messages on the network.

Litwin, Neimat and Schneider were the first to present and discuss such a paradigm of scalable distributed data structures, by proposing a distributed linear hashing, namely LH* [LNS93, LNS94a], and a distributed 1-dimensional order-preserving data structure, namely RP* [LNS94b]. Extension of RP* to the k-dimensional case is a work still in progress [LNS94c, LN95].

Hence the work presented in this paper is the first scalable distributed data structure for k-dimensional point data, with optimal search algorithm 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.

When a multicast protocol is not available, then everything needs to be done with point-to-point messages. In this case the reader is referred to the work on Distributed Random Trees (DRT) of Kröll and Widmayer [KR94, KR95].

The paper is organized as it follows. In section 2 the basic version of the data structure is presented and basic algorithms for its management are discussed. In section 3 we describe how to reduce the use of multicast by means of an index at client sites. Section 4 describes the data structure used to build the index at client sites and proves that it is possible to correctly update it. In section 5 algorithms for updating client index are presented and their correctness is proved. Section 6 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 where each server is managing a different leaf. Hence each server manages a single bucket of data. We assume all buckets have the same size.

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.

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

2.1. Definitions

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) 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 $I(v)=I(v')\cup I(y)$ and $I(v')\cap I(y)=\emptyset$. 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 $\pi(w)$. The leaf without source node, for which we let for completeness $\pi(\cdot)=\emptyset$ is the leaf managing the initial bucket of the k-d tree.

2.2. Algorithm for insertions

It 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, depending on the kind of communication protocol available. In the one by Kröll and Widmayer [KL94], which assumes no multicast, there is a *split coordinator*. Its task is to create a new server s' and to provide to the splitting server s the address of s' Then server s transfers to s' half of its points. In the one by Litwin, Neimat, and Schneider [LNS94b] the splitting server s multicasts the possible address of the new server s' to be created. The action of s then depends on the received answer. If the creation of s' is denied (because it already exists) then the value of s' is increased and s tries again. Otherwise s transfers to s' half of its points.

2.3. Algorithms for queries

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 clients receive 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 the k-d volume of the intersection among its k-range and the query k-range then in the case of range query we can simply sum all the received k-d volumes. When they add up to the k-d volume of the range query we know all pertinent servers have answered. Even if theoretically good, from a practical point of view either infinite precision multiplication is available or this approach may be incorrect. 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.

In any case this approach cannot be applied for partial match and for range queries that specify an unbounded query interval, since the k-d volume is infinite. Hence either we use a time-out approach, which is not so desirable, or we need a more reliable termination test, which is shortly discussed in the next subsection.

2.4. Algorithms for termination test

Two approaches have been proposed. One based on combining adjacent ranges in the set of k-d ranges returned by buckets which answered [LNS94c]. One 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 (see [Nar95]) and has a worst case time of $O(kQ(\log Q)^k)$ for the whole termination test, using on overall 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 on overall worst-case space of $O(kQ)$. For more details see [Nar96].

3. A data structure with index at client sites

If clients have an index the use of multicast can be reduced. In this case a client can search in the index to individuate the server(s) which should answer to its query and can then issue a (set of) point-to-point query.

A crucial observation is that the client index needs not 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 behaviour it may happen that a client has an out-of-date index. This has two consequences. First, the server which received the point-to-point query may not be anymore the server managing the set of keys involved by the query itself (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.

3.1. Index adjustment

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 *loose collection* of generally unrelated *subsets* of a k-d tree. 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 for searching and updating.

In the following subsections we give algorithms for insertion and querying when a client has an index. In the next section we deal with issues regarding the building and maintenance of client index. We also show how to efficient adjust the index using ICMs.

3.2. Algorithm for insertions with index at client sites

The basic approach is to issue a point-to-point message towards the pertinent server if this is found in the index. Otherwise a multicast request is issued.

If an addressing error is made, the server which has received the message issues a multicast request. In this request it includes the address of the client, so that the pertinent server will answer directly to the client itself. The client will surely be aware about the multicast since it is waiting the answer from the point-to-point message.

Algorithms executed at client and at server sites are described in figure 1.

```
[at client site]
IF      in the client index a server s exists whose range contains the k-d point p
        THEN      point-to-point(p,s)
        ELSE      multicast(p)
WHEN      answers arrive      {client expects only the answer with ack here, but a second
                               answers may arrive if it made an addressing error}
        THEN      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
                  ack and send back the current bucket parameters
        ELSEIF          the query was received through a point-to-point message
                  THEN      send back the current bucket parameters
                            multicast(p)
```

Figure 1: algorithms for insertions with index at client sites.

3.3. Algorithm for queries with index at client sites

In this case the basic approach depends on how many servers are involved by the query. It may be the case that, even if the client knows exactly which are the pertinent servers, it is more convenient to issue a single multicast query than many point-to-point queries. This of course depends from a cost function whose parameters are the current overall number of servers, the number of pertinent servers, some physical parameters depending on the network itself, and statistical parameters relative to the expected number of addressing errors.

If point-to-point messages are used, possible addressing errors are managed like for the insertion query. If a client decides to issue a multicast then a termination test algorithm has to be executed to know when all pertinent servers have answered. Answer from each server include also the current bucket parameters, that is the current range of the *k*-d space covered by the server.

Algorithms executed at client and at server sites are describe in figure 2 below.

4. How clients build their index

To make it possible the building of an index at client sites, a server managing bucket with name *x* sends back, with the answer to the query, also the following information:

- the actual *k*-range of bucket *x*, denoted I(*x*);
- the *k*-range of bucket managed by □(*x*) at the time of creation of *x*, called base-range of *x* and denoted BR(*x*); note that by definition it is BR(*x*)=I(□(*x*)).

Note that with our approach a server does not maintain a full record of its past history, but only the conditions under which it was created (which of course never change) and its current status (which may change).

We now prove that sending back the information above described makes it possible to clients to build incrementally an index from query answers, in a monotone and unique way. For the sake of clarity we first discuss a *static setting*, assuming all leaves are available. Afterwards, we analize what happens when leaves of the *k*-d tree are known only incrementally (*dynamic setting*).

```
[at client site]
IF    in the client index a set S of servers exists intersecting the whole k-d range q
      THEN      decide on the basis of the cost function
      ELSE multicast(q)
IF    multicast was used to issue the query
      THEN      execute the termination test
      ELSE wait for all acks    { if a server re-issue the query as multicast while client }
                                { is waiting then client switches to execute the termination test }
possibly update the index using received information

[at site of server s]
IF    the received k-d range q intersects the range covered by server s
      THEN      find the set P of k-d points covered by q
                ack with P and send back the current bucket parameters
      ELSEIF    the query was received through a point-to-point message
                THEN      send back the current bucket parameters
                          multicast(q)
```

Figure 2: algorithms for queries with index at client sites

4.1. The static setting

Tree T□is an *exact reconstruction* of *k*-d tree T if the set of internal nodes and the set of leaves of T□are exactly those of T. Having at disposal for all the buckets *x* of a *k*-d tree T the values I(*x*) and BR(*x*) we prove that it is indeed possible to reconstruct the exact tree-structure of T.

Theorem 1: Let L be the set of leaves of a *k*-d tree T with the function BR(·) above defined. Then it is possible to exactly reconstruct T from L.

Proof: Since L is the complete set of leaves of the *k*-d tree T, then in L there exist at least two leaves which are sons of the same parent. This means that at least one leaf *x* exists in L for which a unique *y* exists in L such that BR(*y*)=I(*x*)□I(*y*). Therefore we can create a node *z* such that I(*z*)=BR(*y*), BR(*z*)=BR(*x*). We let *z* to be the parent of *x* and *y*. We can now produce from L a smaller size set L□=L-{*x*,*y*}□{*z*}. With L□we are in the initial conditions and this transformation can be applied again recursively. The sequence of transformations stops when the set is reduced to a single node, which results to be the root of the reconstructed *k*-d tree T. By the unicity of the choice of leaf *y* which is coupled with leaf *x*, the exactness of the reconstruction derives. •

The next corollary proves that all information we use in the reconstruction are necessary.

Corollary 2: If function BR(·) does not give information on each of the *k* 1-dimensional intervals covered by *x* then the exact reconstruction of T from L is not guaranteed.

Proof: If one of the *k* 1-dimensional interval misses in BR(·), there may be a non deterministic choice. In fact, it may happen that it is not possible for each node to uniquely identify its mate. See, as an example, figure 3 below, where if BR(·) does not contain information relative to the vertical coordinate both the subtrees shown can be constructed.

•

Note that each internal node of the overall *k*-d tree also knows its split dimension and its split point. These are not needed, in the static setting, to exact reconstruct the *k*-d tree, and indeed have not been used in the relative theorems. But since they are anyhow known to servers, we assume they are part of what each server sends back in its answer. Hence to the list above we also add, for a server *x*:

- the split-dimension of □(*x*) and the split-point of □(*x*), which are together denoted as sp(*x*); note that by definition it is sp(*x*)=[D(□(*x*)), V(□(*x*))] and we let D(∅)=V(∅)=∅.

sp(*x*) and BR(*x*), if distinct from ∅, together identifies the (*k*-1) dimensional splitting plane, called <u>base-line</u> of *x*, which divided the *k*-range of bucket assigned to □(*x*) to create the initial *k*-range of bucket *x*.

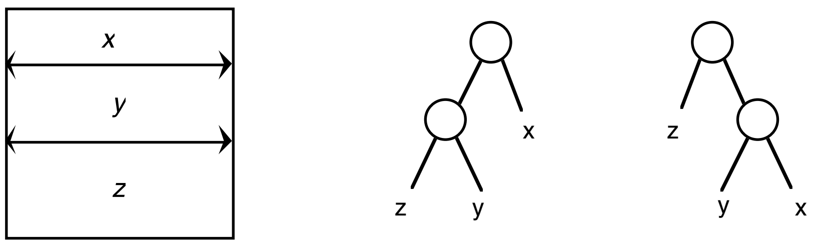

Figure 3.

Note that when a bucket *x* splits its *k*-range in two parts, say *x1* and *x2*, it may happen that only one of them, say *x2*, has a boundary lying on the base-line of *x*. In this case we force the bucket taking care of part *x2* to keep the name *x* (and therefore *x*'s base-line becomes its base-line). The other bucket receives a new name and its base-line is defined on the basis of *x*'s range and of the (*k*-1) dimensional plane which has split *x*'s range. In figure 4 below an example of the above definitions is shown.

4.2. The dynamic setting

What we have discussed up to here is the possibility of building a *k*-d tree assuming to have at disposal information from all its leaves. Since client queries retrieve each time only part of the whole index, a client index is generally built incrementally, from successive answers from buckets.

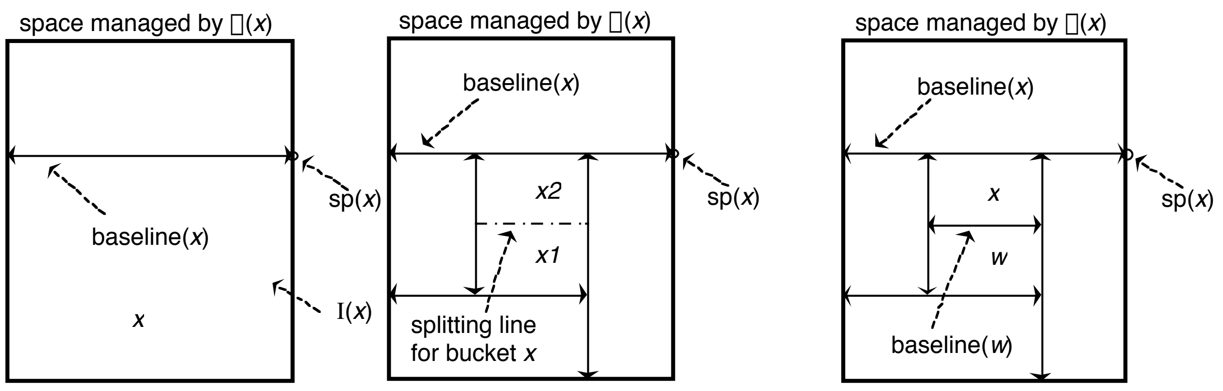

Figure 4.

This incremental building may lead, if it is not done properly, to construct a partial *k*-d tree that needs later to be dismounted. For example, consider two buckets B1 and B2 which are "far away" in the *k*-d tree (i.e. whose lowest common ancestor in the *k*-d tree is at least 2*k*+1 levels higher than the highest of the two leaves). In this example if we build a single *k*-d tree with these two buckets then it will surely have to be dismounted. In fact, whichever root we may choose on the basis of the received information is not an ancestor of both leaves in the original *k*-d tree.

In the dynamic context it is therefore important to avoid dismounting and rebuilding a partial *k*-d tree when a new answer arrives. The amount of work which may be needed in restructuring is of the order of the size of the tree and a restructuring may be needed each time the height of the tree is increased.

The only viable strategy to avoid restructuring is to avoid building a *k*-d tree which may be later dismounted, that is to realize a *monotone* building of the overall *k*-d tree. This means that an answer from a bucket can be used in the reconstruction only if we are sure that the relative decisions we take in *k*-d tree building will not be rejected at a later stage. But also we do not want to wait until the arrival of the last answer to start to process the set of answers. This is for two reasons. First, if a client has to do many queries in a short amount of time, it is more efficient to use information returned from already terminated queries than to issue the query using the out-of-date index. This is true even if returned information is only partial. Second, if this partial information is used then it is not less efficient to search in a tree than to search in a list, as it would be if client waited until the arrival of the last answer.

Hence we want to be able to manage a forest of *k*-d trees, all of them part of the overall *k*-d tree. We therefore introduce a new data structure, called lazy *k*-d tree and presented in the next section, which is an extension of the *k*-d tree to efficiently cope with this case.

5. Client index building and maintenance

In this section we provide an algorithm that makes it possible to build and update an index at each client site. The discussion is rooted in the framework presented in the previous section. We first illustrate what is the index built by a client. Then we show how each client builds the index. Finally we prove the correctness of the index building algorithm.

5.1. Definitions

A lazy *k*-d tree (*lkd-tree* in the following) is defined as it follows:

- it has two types of nodes, namely simple nodes and compound nodes;
- a simple node *v* may be external (i.e., a leaf) or internal; I(*v*) is associated to simple node *v*;

- BR(*v*) is also associated to leaf *v*;
- sp(*v*) (i.e., the couple D(*v*) and V(*v*)) is also associated to a simple internal node *v*;
- a simple internal node *v* has one or two sons, which may be simple or compound, denoted left_son(*v*) and right_son(*v*);
- a compound node *u* has no sons and no piece of information is directly associated to it;
- a compound node *u* is a set C(*u*) of lkd-trees, whose roots are simple nodes, such that for each couple of distinct roots *v* and *w* in C(*u*) it is I(*v*)∩ I(*w*)=∅; we say the roots (or the lkd-trees) *v* and *w* are not comparable;
- given a simple internal node *v*, the interval associated to its left son, if the left son is a simple node, or to each simple root in its left son, if the left son is a compound node, contains only points in I(*v*) whose coordinate in dimension D(*v*) has a value less than V(*v*); for the right son definitions are simmetric.
- •

Let a left son *w* of a simple internal node *x* be such that I(*w*) contains *all* the points in I(*v*) whose coordinate in dimension D(*v*) has a value less than V(*v*). Then *w* is said to be direct son of *v* (and correspondingly *v* is said to be direct parent of *w*). Otherwise *w* is said to be a descendent of v (corr. ancestor). For a right son definitions are simmetric. When we use just son we are not taking into consideration the containment relation between intervals. When we do not want to distinguish between direct sons (resp. direct parents) and descendants (resp. ancestors) we speak of successors (resp. predecessors).

5.2. Algorithms

The index built by each client is a lazy *k*-d tree built incrementally using answers arriving from buckets.

To make treatment of answer from bucket *x* easier we transform the answer in a very simple tree, made up by just two nodes. A leaf, representing the range of bucket *x*, and a root b(*x*), which represents the base range of *x*. Remember that bucket *x* answers also with I(*x*), BR(*x*), and sp(*x*), with the above defined meanings. Hence b(*x*) is such that I(b(*x*))=BR(*x*) and [D(b(*x*)), V(b(*x*))]=sp(*x*). I(*x*) is the *k*-range managed by the bucket *x* and [D(*x*), V(*x*)] is empty.

Let us assume the client has already an index in the form of a lazy *k*-d tree whose root is the simple node *y* (we shall later remove the restriction that the root is a simple node). A new answer from bucket *x* arrives. If this is the first answer to arrive and the client has no index then the above described tree made up by b(*x*) and *x* becomes the client index. Otherwise algorithm INSERT below is invoked.

The high level description of algorithm INSERT considers these four cases:

```
INSERT(x: bucket just arrived; y: root of a lazy k-d tree):
    CASE
        A: I(b(x)) is not comparable with I(y);
        B: I(b(x)) coincides with I(y);
        C: I(b(x)) strictly contains I(y);
        D: I(b(x)) is strictly contained in I(y).                    •
```

We now provide a more detailed analysis of actions taken in each of the above cases.

A: I(b(*x*)) is not comparable with I(*y*), that is BR(*x*)□ I(*y*)=∅;
A compound node is built, made up by two lazy *k*-d trees. The roots are b(*x*), whose only son is *x*, and *y*.

B: I(b(*x*)) coincides with I(*y*), that is BR(*x*)=I(*y*);
This implies that *y* is precisely that predecessor of *x* whose splitting created *x*, that is *y*=□(*x*). Hence an answer from bucket *x* already arrived to this client. Since after that answer *x* may have further split we invoke SIMPLE_ INSERT(*x*,*y*), to be described later, to find the right place for *x* in the lkd-tree rooted at *y*.

C: I(b(*x*)) contains I(*y*), that is BR(*x*)   I(*y*);
IF *y* is a successor on the *same* side of *x* with respect to b(*x*)
THEN
We have to check the containment relation between I(*y*) and I(*x*). Since *x* is a leaf in the index, it may only be that either I(*y*) and I(*x*) are not comparable or I(*y*) contains I(*x*). In the former case *y* and *x* are simply put together to make up a compound node, which substitutes *x* as the only son of b(*x*). In the latter case *y* is made son of b(*x*) and *x* has to be inserted in the lkd-tree rooted at *y*. This is done by invoking SIMPLE_INSERT(*x*,*y*).
ELSE {*y* is a successor on the *other* side of *x*}
y is simply added as a son of b(*x*) on the other side of *x*.

D: I(b(*x*)) is contained in I(*y*), that is BR(*x*) ℘ I(*y*);
IF I(b(*x*)) is contained in the left subtree of *y* :
THEN INSERT(b(*x*), left_son(*y*))
ELSE INSERT(b(*x*), right_son(*y*)). •

We now describe algorithm SIMPLE_INSERT.

SIMPLE_INSERT(*v*: simple node with no sons; *y*: root of a lazy *k*-d tree):
CASE
A□ I(*v*) is not comparable with I(*y*);
A compound node is built, made up by two lazy *k*-d trees. The two roots are *v*, which is a simple node without sons, and *y*.
B□ I(*v*) coincides with I(*y*);
Nothing has to be done.
{Note that the case "I(*v*) is strictly contained in I(*y*)" cannot happen. In fact, *v* is a leaf and the index cannot contain the refinement of a leaf before the arrival of the leaf itself. Moreover, after its split the leaf will never be inserted again with its original value};
D□ I(*v*) coincides with I(*y*);
IF I(*v*) is contained in the left subtree of *y*:
THEN SIMPLE_INSERT(*v*, left_son(*y*))
ELSE SIMPLE_INSERT(*v*, right_son(*y*)). •

To complete the analysis we have only to consider what happens if I(*v*) has to be compared with I(*w*) but *w* is a compound node. We only need to describe what happens either when *v* has no sons or when it has exactly one son.

COMPARE(*v*: simple node with one or zero sons; *w*: compound node):

CASE

P: no root in *w* is comparable with *v*;
node *v* is simply added to the compound node *w*.

```
Q: exactly one root $r$ in $w$ is comparable with $v$;
IF $v$ has no sons
    THEN  SIMPLE_INSERT($v$, $r$)
    ELSE  INSERT($v$, $r$).

R: the set $R$ of roots in $w$ which are comparable with $v$ is such that $|R|>1$;
{In this case clearly each root $r$ in $R$ is such that $I(r) \wp I(v)$}
delete $R$ from the set of sons of $w$ and add $v$ in the set of sons of $w$;
IF $v$ has no sons
    THEN $R$ is transformed in a compound node which is added as a son of $v$
    ELSE
    let $v'$ be the son of $v$;
    let $R'$ be the subset of $R$ made up by those roots in $R$ which are
                              on the same side as $v'$ with respect to $v$;
    let $R''$ be the set of remaining roots;
    $R''$ is transformed in a compound node which is added as the other son of $v$;
    IF all roots in $R'$ are not-comparable with $v'$
        THEN
        build a compound node $z$ containing $R'$ and $v'$;
        delete $v'$ as a son of $v$ and add $R'$ as a son of $v$;
    {note that $v'$ cannot contain a root in $R'$ since $v'$ is a leaf};
    IF $\exists! r$ in $R'$ such that $I(v') \wp I(r)$
        THEN
        build a compound node $z$ containing roots in $R'$;
        delete $v'$ as a son of $v$ and add $R'$ as a son of $v$;
        SIMPLE_INSERT($v'$, $r$).                                          •
```

5.3. Correctness

The correctness of algorithms described in previous section is ensured by the following invariance lemmas. Correctness of lemmas directly derives from the actions taken by above algorithms in the various cases.

Lemma 3: For each new arriving bucket x and for each bucket y in the lazy k-d tree T x is made a successor of y in T iff $I(x) \wp I(y)$ and y is made a successor of x in T iff $I(y) \wp I(x)$. •

Lemma 4: If y is a lazy k-d tree then after INSERT(x, y) y is still a lazy k-d tree. •

Lemma 5: If y is a lazy k-d tree then after SIMPLE_INSERT(x, y) y is still a lazy k-d tree. •

The following theorem ensures above algorithms provide an exact-reconstruction of the overall k-d tree. Let t be any given point in time. Without loss of generality we can assume no buckets split exactly at time t. Let $B(t)$ be the set of buckets existing at time t and let $S(t)$ the set of those buckets that have split or have been created after time t.

Theorem 6: If client C has received at time t at least one answer from all buckets in $B(t)\backslash S(t)$ and also it has received after time t at least one answer from each bucket in $S(t)$ then its lazy k-d tree coincides with the overall k-d tree.

Proof: From the three invariance lemmas above and from the fact that the algorithm never restructures the lazy k-d tree (in the sense of altering the successor-predecessor relations) we have that when answers from all buckets in the k-d tree are arrived to client C then its lazy k-d tree has become the exact k-d tree. •

6. Conclusions

We have introduced and discussed in this paper the lazy *k*-d tree. This is the first scalable distributed data structure for *k*-dimensional point data, with optimal search algorithm 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. 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 proved that the lazy *k*-d tree is a generalization of the *k*-d tree data structure that is suitable for an efficient management and querying in a distributed framework. We have considered distributed environments where multicast (i.e. restricted broadcast) is allowed but have also shown how to reduce its use whenever possible.

An experimental analysis of the performances of the structure here described, including the study of trade-offs between using point-to-point and multicast, and a comparison with previously introduced structure (LH*, RP* and DRT) is in progress [Bar96]. A variant where an index is used at server sites to improve overall performances, suitable also for the cases when multicast is not available at all, is also under experimental analysis [Pep96]. Note that in this case the lazy *k*-d tree, from a theoretical point of view, behaves in a similar way to the Distributed Random Tree of Kröll & Widmayer.

Work in progress is also dedicated to the extension to other multi-dimensional data structures and to extended objects. Candidates under considerations for this are R^+ -trees, quad-trees, and grid-files.

Acknowledgments

I wish to thank Witold Litwin, Marie-Anne Neimat, and Donovan Schneider for having introduced me 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 structure were very useful and provided me with many valuable insights. Fabio Barillari e Massimo Pepe read various drafts of this work and always provided comments which helped to improve the presentation. Their work on the implementation of lkd-tree provided also valuable feedback on various design decisions.

References

[Bar96] F.Barillari, Analisi sperimentale 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. ACM, 18:509-517, 1975.

[Ben79] J.L.Bentley, Decomposable searching problems, Information Processing Letters, 8(5):244-251, June 1979.

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

[KW95] B.Kröll, P.Widmayer, Balanced distributed search trees do not exist, 4th Int. Workshop on Algorithms and Data Structures (WADS'95), Kingston, Canada, 50-61, August 1995, Lecture Notes in Computer Science 955, S.Akl et al. (eds), Springer Verlag.

[Lue78] G.S.Lueker, A data structure for orthogonal range queries, 19th IEEE Symp. on Foundations of Computer Science, Ann Arbor, Mi., 0ct.1978, 28-34.

[LNS93] 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.

[LNS94a] W.Litwin, M.-A.Neimat, D.A.Schneider, LH* - A scalable distributed data structure, ACM Trans. on Database Systems, to appear.

[LNS94b] W.Litwin, M.-A.Neimat, D.A.Schneider, RP* - A family of order-preserving scalable distributed data structures, 20th Conf. on Very Large Data Bases, Santiago, Chile, 1994.

[LNS94c] W.Litwin, M.A.Neimat, and D.Schneider, k-RP$^*_N$: a spatial scalable distributed data structure, manuscript, July 1994.

[LN95] W.Litwin and M.A.Neimat, k-RP$^*_S$: a high performance multi-attribute scalable data structure, manuscript, March 1995.

[Nar95] E.Nardelli, Some issues on the management of k-d trees in a distributed framework, Technical Report n.76, Dipartimento di Matematica Pura ed Applicata, Universita' di L'Aquila, January 1995.

[Nar96] E.Nardelli, Efficient management of distributed k-d trees, Technical Report n.101, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, March 1996.

[Pep96] 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, 1996.

[Wil85] D.E.Willard, New data structures for orthogonal range queries, SIAM J. on Computing, 14(1):232-253, 1985.

[WL85] D.E.Willard and G.S.Lueker, Adding range restriction capability to dynamic data structures, J. of the ACM, 32(3):597-617, July 1985.