SD-Rtree: A Scalable Distributed Rtree

Cédric du Mouza
LAMSADE, Univ. Dauphine
cedric.dumouza@dauphine.fr

Witold Litwin
CERIA, Univ. Dauphine
witold.litwin@dauphine.fr

Philippe Rigaux
LAMSADE, Univ. Dauphine
philippe.rigaux@dauphine.fr

Abstract

We propose a scalable distributed data structure (SDDS) called SD-Rtree. We intend our structure for large spatial datasets over clusters of interconnected servers. SD-Rtree generalizes the well-known Rtree structure. It uses a distributed balanced binary spatial tree that scales with insertions to potentially any number of storage servers through splits of the overloaded ones. A user/application manipulates the structure from a client node. The client addresses the tree through its image that the splits can make outdated. This may generate addressing errors, solved by the forwarding among the servers. Specific messages towards the clients incrementally correct the outdated images.

We present the build-up of an SD-Rtree through insertions, focusing on the split and balancing algorithms. We follow with the point and window query algorithms. Experiments show additional aspects of SD-Rtree behavior. The results justify our various design choices and the overall utility of the structure.

Keywords.

Indexing, spatial data, distributed data structure.

1 Introduction

We aim at indexing large datasets of spatial objects, each uniquely identified by an *object id* (oid) and approximated by the *minimal bounding box* (mbb). We generalize the Rtree spatial index to a *Scalable Distributed Data Structure* (SDDS) that we call SD-Rtree. Our structure conforms to the general principles of an SDDS, in addition to its specific ones [15].

We store an SD-Rtree at interconnected *server* nodes, in a storage space usually termed *bucket*, each with some predefined capacity. The buckets may be entirely in the distributed RAM, providing potentially for much faster access than to disks. If a bucket overflows its capacity, a split occurs, moving some data to a dynamically appended bucket. An application addresses an SD-Rtree only through the client component. A client can be at the application node. The application may alternatively be at, or remotely address, a peer node carrying both a client and server components.

The address calculus does not require neither a centralized component nor multicast messages. A client addresses the servers which are in its *image* of the structure. Some existing servers may be not in the image, because of splits unknown from the client. The addressing may then send a query to an *incorrect* server that is different from the one the query should address. The servers recognize such *addressing errors* and forward the query among themselves, until it reaches the *correct* one. The client may get then a specific *image adjustment message* (IAM). This one improves the image at least so that the addressing error triggering the IAM does not repeat.

An SD-Rtree avoids redundancy of objects references, like Rtree or R*tree. The structure aims only on the addressing of the correct servers. The local search within a server is beyond our scope at present. The general structure of an SD-Rtree is that of a distributed balanced binary spatial tree where each node carries a mbb. We store both the leaves and the internal nodes in a distributed way at the servers. The client sends a query to the leaf it determines from its SD-Rtree image. The addressing is correct if the

object fits the actual mbb of the leaf, otherwise the forwarding traverses some upward path in the tree. It may end by reaching the leaf with the mbb including the one of the object. It may conclude alternatively that there is no such server. For a search it is an unsuccessful termination case. For an insert, next step is an enlargement of some internal nodes and leaf mbbs, as usual in an Rtree. The object is stored there. This may lead to a split, adding a new node to the SD-Rtree and triggering its possible rebalancing.

Below we present the distributed structure of SD-Rtree and its algorithms for splitting and balancing. Next, we discuss the search and insert processing at the servers and the client. The nodes communicate only through point-to-point messages. We then analyze the access performance of our scheme as the number of messages sent to the servers. In general, insert and point query operations into an SD-Rtree over N servers cost one message to contact the correct server. If the first message is *out of range* (i.e., the contacted server is not the correct one), the cost is in general within $2 \log N$, unless an infrequent split adds another $\log N$. The overlapping may add up to N messages but in practice it is relatively negligible. The processing of window queries is also efficient, as the maximal message path length to diffuse a window query is $O(\log N)$. All these properties prove the adequacy of the scheme to our goals.

Section 2 presents the SD-Rtree. Section 3 describes the insertion algorithm and Section 4 the point and window queries. Section 5 shows the experimental performance analysis. Section 6 discusses the related work and Section 7 concludes the paper.

2 The SD-Rtree

The structure of the SD-Rtree is conceptually similar to that of the classical AVL tree, although the data organization principles are taken from the Rtree spatial containment relationship [6].

Definition 1 (SD-Rtree). *The SD-Rtree is a binary tree, mapped to a set of servers, and satisfying the following properties:*

- *each internal node, or routing node, has exactly two children,*
- *each routing node has left and right directory rectangles (dr) which are the minimal bounding boxes of, respectively, the left and right subtrees,*
- *each leaf node, or data node, stores a subset of the indexed objects,*
- *at any node, the height of the two subtrees differs by at most one.*

The last property ensures that the height of a SD-Rtree is logarithmic in the number of servers.

2.1 Kernel structure

The tree has N leaves and $N - 1$ internal nodes which are distributed among N servers. Each server S_i is uniquely identified by an id i and (except server S_0) stores exactly a pair (r_i, d_i) , r_i being a routing node and d_i a data node. As a data node, a server acts as an objects repository up to its maximal capacity. The bounding box of these objects is the *directory rectangle* of the server.

Figure 1: Basic features of the SD-Rtree

Figure 1 shows a first example with three successive evolutions. Initially (part A) there is one data node d_0 stored on server 0. After the first split (part B), a new server S_1 stores the pair (r_1, d_1) where r_1 is a routing node and d_1 a data node. The objects have been distributed among the two servers and the tree $r_1(d_0, d_1)$ follows the classical Rtree organization based on rectangle containment. The directory rectangle of r_1 is a , and the directory rectangles of d_0 and d_1 are respectively b and c , with $a = mbb(b \cup c)$. The rectangles a , b and c are kept on r_1 in order to guide insert and search operations.

If the server S_1 must split in turn, its directory rectangle c is further divided and the objects distributed among S_1 and a new server S_2 which stores a new routing node r_2 and a new data node d_2 . r_2 keeps its directory rectangle c and the dr of its left and right children, d and e , with $c = mbb(d \cup e)$. Each directory rectangle of a node is therefore represented exactly twice: on the node, and on its parent.

A routing node maintains the id of its parent node, and *links* to its left and right children. Links are defined as follows:

Definition 2 (Link). A link is a quadruplet $(id, dr, height, type)$, where id is the id of the server that stores the referenced node, dr is the directory rectangle of the referenced node, $height$ is the height of the subtree rooted at the referenced node and $type$ is either *data* or *routing*.

Whenever the type of a link is *data*, it refers to the data node stored on server id , else it refers to the routing node. Note that a node can be identified by its type (data or routing) together with the id of the server where it resides. When no ambiguity arises, we will blur the distinction between a node id and its server id .

The description of a routing node is as follows:

Type: ROUTINGNODE

$height, dr$: description of the routing node
 $left, right$: links to the left and right children
 $parent_id$: id of the parent routing node
 OC : the overlapping coverage

The routing node provides an exact local description of the tree. In particular the directory rectangle is always the geometric union of $left.dr$ and $right.dr$, and the height is $\text{Max}(left.height, right.height) + 1$. OC , the *overlapping coverage*, to be described next, is an array that contains the part of the directory rectangle shared with other servers. The type of a data node is as follows:

Type: DATANODE

$data$: the local dataset
 dr : the directory rectangle

$parent_id$: id of the parent routing node
 OC : the overlapping coverage

2.2 Node splitting

When a server S is overloaded by new insertions in its data repository, a split must be carried out. A new server S' is added to the system, and the data stored on S is divided in two approximately equal subsets using a split algorithm similar to that of the classical Rtree [6, 5]. One subset is moved to the data repository of S' . A new routing node $r_{S'}$ is stored on S' and becomes the immediate parent of the data nodes respectively stored on S and S' .

Figure 2: Split operations

The management and distribution of routing and data nodes are detailed on Figure 2 for the tree construction of Figure 1. Initially (part A), the system consists of a single server, with id 0. Every insertion is routed to this server, until its capacity is exceeded. After the first split (part B), the routing node r_1 , stored on server 1, keeps the following information (we ignore the management of the overlapping coverage for the time being):

- the $left$ and $right$ fields; both are data links that reference respectively servers 0 and 1,
- its height (equal to 1) and its directory rectangle (equal to $mbb(left.dr, right.dr)$),

- the parent id of the data nodes 0 and 1 is 1, the id of the server that host their common parent routing node.

Since both the `left` and `right` links are of type `data` links, the referenced servers are accessed as data nodes (leaves) during a tree traversal.

Continuing with the same example, insertions are now routed either to server 0 or to server 1, using a Rtree-like `CHOOSESUBTREE` procedure [6, 1]. When the server 1 becomes full again, the split generates a new routing node r_2 on the server 2 with the following information:

- its `left` and `right` data links point respectively to server 1 and to server 2
- its `parent_id` field refers to server 1, the former parent routing node of the splitted data node.

The right child of r_1 becomes the routing node r_2 and the height of r_1 must be adjusted to 2. These two modifications are done during a *bottom-up traversal* that follows any split operation. At this point the tree is still balanced.

2.3 Overlapping coverage

Any Rtree index presents some degree of overlapping in the space coverage. This sometimes leads to several paths during search operations. In a centralized structure, one discovers the paths that lead to the relevant leaves during the top-down traversal of the tree. We cannot afford this simple strategy in a distributed tree because it would overload the nodes near the tree root.

Informal presentation

Our search operations attempt to find directly, without requiring a top-down traversal, a data node d whose directory rectangle dr satisfies the search predicate. However, unlike hash-based or Btree-like distributed structures, this strategy is not sufficient with spatial structures that permit overlapping, because d does not contain *all* the objects covered by

dr . Once d is found on a server S , we must therefore be able to forward the query to the other servers that potentially match the search predicate. This requires the distributed maintenance of some redundant information regarding the parts of the indexed area shared by several nodes, called *overlapping coverage* (OC) in the present paper.

A simple but costly solution would be to maintain, on each data node d , the path from d to the root of the tree, including the left and right regions referenced by each node on this path. From this information we can deduce, when a point or window query is sent to d , the subtrees where the query must be forwarded. We improve this basic scheme with two significant optimizations. First, if a is an ancestor of d or d itself, we keep only the part of $d.dr$ which overlaps the sibling of a . This is the sufficient and necessary information for query forwarding. If the intersection is empty, we simply ignore it. Second we trigger a maintenance operation only when this overlapping changes.

Figure 3: Overlapping coverage examples

Figure 3 illustrates the concept. The left part shows a two-levels tree rooted at R . The overlapping coverage of A and B , $A.dr \cap B.dr$, is stored in both nodes. When a query (say, a point query) is transmitted to A , A knows from its overlapping coverage that the query must be routed to B if the point argument belongs to $A \cap B$.

Next, consider the node D . Its ancestors are A and R . However the subtrees which really matter for query forwarding are C and B , called the *outer subtrees* of, respectively, A and R with respect to D . Since $D.dr \cap C.dr = \emptyset$ and $D.dr \cap B.dr = \emptyset$, there is no need to forward any query whose argument (point or window) is included in $D.dr$. In other words, the overlapping coverage of D is empty.

An important feature is that the content of B , the outer subtree of R with respect to A , can evolve in-

dependently from A , as long as the rectangle intersection remains the same. Figure 3.b shows a split of the server B : its content has been partially moved to the new data node E , and a new routing node F has been inserted. Note that F is now the outer subtree of R with respect to A . Since, however, the intersection $A.dr \cap F.dr$ is unchanged, there is no need to propagate any update of the OC to the subtree rooted at A .

Finally the subtree rooted at A may also evolve. Figure 3.c shows an extension of D such that the intersection with F is no longer empty. However our insertion algorithm guarantees that no node can make the decision to enlarge its own directory rectangle without referring first to its parent. Therefore the object's insertion which triggers the extension of D has first been routed to A . Because A knows the space shared with F , it can transmit this information to its child D , along with the insertion request. The OC of D now includes $D.dr \cap F.dr$. Any point query P received by D such that $P \subseteq D.dr \cap F.dr$ must be forwarded to F .

Storage and maintenance of the overlapping coverage

Given a node N , let $anc(N) = \{N_1, N_2, \dots, N_n\}$ be the set of ancestors of N . Each node $N_i \in anc(N)$ has two children, One is itself an ancestor of N or N itself, while its sibling is not an ancestor of N and is called the *outer node*, denoted $outer_N(N_i)$. For instance the set of ancestors of d_2 in Figure 1 is $\{r_1, r_2\}$. The outer node $outer_{d_2}(r_2)$ is d_1 , the outer node $outer_{d_2}(r_1)$ is d_0 .

Definition 3 (Overlapping coverage). Let N be a node in the SD-Rtree, with $anc(N) = \{N_1, N_2, \dots, N_n\}$. The overlapping coverage of N is an array OC_N of the form $[1 : oc_1, 2 : oc_2, \dots, n : oc_n]$, such that oc_i is $N.dr \cap outer_N(N_i).dr$. Moreover an entry i is represented in the array only if $oc_i \neq \emptyset$.

In other words the overlapping coverage of a node N consists of all the non-empty intersections with the outer nodes of the ancestors of N . Each node

stores its overlapping coverage which is maintained as follows.

When an object obj must be inserted in a subtree rooted at N , one first determines with CHOOSESUBTREE the subtree I where obj must be routed. O , the sibling of I , is therefore the outer node with respect to the leaf where obj will be stored. The node I must possibly be enlarged to accommodate obj and this leads to check whether the intersection $I.dr \cap O.dr$ has changed as well, in which case the overlapping coverage must be modified as follows:

1. the OC entry $[O.id : I.dr \cap O.dr]$ is added to the insertion message routed to the child I
2. a OC update message, containing the OC entry $[I.id : I.dr \cap O.dr]$, is sent to the child O .

The operation is called recursively until a data node is reached. The insertion message contains then the updated information regarding the OC of d . The top-down traversal (if any) necessary to find d accesses some ancestors for which the possible changes of the overlapping coverage must be propagated to the outer subtrees, thanks to the UPDATEOC procedure below.

```

UPDATEOC ( $N, id, rect$ )
Input: a node  $N$ , the  $id$  of an ancestor node,
        and  $rect$ , the directory rectangle of  $outer_N(id)$ 
Output: update the local OC and forward to subtrees
begin
    // Check whether  $N.OC[id]$  has changed
    if ( $rect \cap N.dr$  differs from  $N.OC[id]$ )  $N.OC[id] := rect$ 
    // Diffuse to subtree if the OC is non-empty
    if ( $N$  is not a leaf)
        // Compute and send to the left subtree
        if ( $rect \cap N.left.dr \neq \emptyset$ )
            UPDATEOC ( $N.left.id, id, rect \cap N.left.dr$ )
        // Compute and send to the right subtree
        if ( $rect \cap N.right.dr \neq \emptyset$ ) then
            UPDATEOC ( $N.right.id, id, rect \cap N.right.dr$ )
    endif
end

```

The cost of the OC maintenance through calls to UPDATEOC depends both on the length of the inser-

tion path to the chosen data node d , and on the number of enlargements on this path. In the worst case, the insertion path starts from the root node, and all the overlaps between d and its outer nodes are modified, which result at worse in $N - 1$ UPDATEOC messages. However, in practice, the cost is limited because the insertion algorithm avoids in most cases a full traversal of the tree from the root to a data node d , and reduces therefore the number of ancestors of d that can possibly be enlarged. Moreover the number of node's enlargements lowers as soon as the union of the servers directory rectangles cover the embedding space.

Regarding the second aspect, it suffices to note that no enlargement is necessary as soon as there exists a server whose directory rectangle fully contains the inserted object o . It can be shown, assuming an almost uniform size of objects, that it is very unlikely that a new inserted object cannot find a server's directory rectangle where it is fully contained. Our experiments confirm that the overlapping coverage remains stable when the embedding space is fully covered, making the cost of OC maintenance negligible.

2.4 Balancing

Figure 4: Rotation issues in a binary Rtree

In order to preserve the balance of the tree, a rotation is sometimes required during the bottom-up traversal that adjusts the heights. Balancing techniques for binary trees over ordered domains are very well-known. They must be adapted with care when applied to a spatial index. Consider for instance Fig-

ure 4. The left part shows that the node a is unbalanced: the height of its left child is $n + 2$ whereas the height of its right child is n . A straightforward balancing is shown in Figure 4.b. The resulting tree is balanced, but putting nodes c and d in the same subtree results in a very bad spatial organization, with a large dead space and uncontrolled overlapping (note that node e is the sibling of node b , yet it is fully included in the directory rectangle of b). Finally this gives rise to an imbalance between the respective coverages of sibling nodes (see again nodes e and b in Figure 4.b). Figure 4.c shows that associating c and e would yield a much better spatial organization. Unfortunately this results in an unbalanced tree.

The balancing of the SD-Rtree takes advantage of the absence of order on rectangles which gives more freedom for reorganizing an unbalanced tree, compared to classical AVL trees. The technique is described with respect to a *rotation pattern*, defined as follows:

Definition 4. A rotation pattern is a subtree of the form $a(b(e(f, g), d), c)$ which satisfies the following conditions for some $n \geq 1$:

- $height(c) = height(d) = height(f) = n - 1$
- $height(g) = \max(0, n - 2)$

An example of rotation pattern is shown on Figure 5. Note that a , b and e are routing nodes.

Now, assume that a split occurs in a balanced SD-Rtree at node s . A bottom-up traversal is necessary to adjust the heights of the ancestors of s . Unbalanced nodes, if any, will be detected during this traversal. The following holds:

Proposition 1. Let a be the first unbalanced node met on the adjustment path that follows a split. Then the subtree rooted at a matches a rotation pattern.

Proof. First, since the subtrees are not ordered, we can consider, without loss of generality, that any node, when it becomes unbalanced, can be mapped to the form $a(b(e(f, g), d), c)$ by exchanging some of the left and right children. Second, note that the split that initiated the imbalance

Figure 5: Balancing in the SD-Rtree

occurred necessarily in the subtree rooted at e (if $n = 1$) or at f (if $n > 1$). Since a is the first unbalanced node on this bottom-up path, b and e are balanced. An easy case-by-case reasoning shows that the respective heights of f , g , c and d satisfy the conditions of Definition 4. $\square$

The proposition shows that the management of unbalanced nodes always reduces to a balancing of a rotation pattern $a(b(e(f,g), d), c)$. The operation is as follows:

1. b becomes the root of the reorganized subtree,
2. The routing node a becomes the right child of b ; e remains the left child of b and c the right child of a ,
3. One determines which one of f , g or d should be the sibling of c in the new subtree. The chosen node becomes the left child of a , the other pair constitutes the children of e .

The choice of the moved node should be such that the overlapping of the directory rectangles of e and

a is minimized. Tie-breaking can be done by considering the minimization of the dead space as second criteria. This rotation mechanism can somehow be compared to the forced reinsertion strategy of the R*tree [1], although it is here limited to the scope of a rotation pattern.

Definition 4 shows that any pairwise combination of f , g , d and c yields a balanced tree. The three possibilities, respectively called $\text{move}(g)$, $\text{move}(d)$ and $\text{move}(f)$ are shown on Figure 5. The choice $\text{move}(g)$ (Figure 5.b) is the best one for our example. Note that all the information that constitute a rotation pattern is available from the left and right links on the bottom-up adjust path that starts from the splitted node.

The balancing can be obtained in exactly 6 messages for $\text{move}(f)$ and $\text{move}(g)$, and 3 messages for $\text{move}(d)$ because the subtree rooted at e remains in that case the same. When a node a receives an adjust message from its modified child (b in our example), it knows the right link c and gets the links for e , d , f and g which can be maintained incrementally in the chain of adjustment messages. If a detects that it is unbalanced, it takes account of the information represented in the links to determine the subtree f , g or d which becomes the sibling of c . Let s be this node and (s_1, s_2) be the remaining pair. Then the following messages are sent to the servers b, c, d, e, f and g and to the parent of a :

1. Send to the parent of a : its child a is replaced by b . Note that, thanks to the balancing operation, the bottom-up adjustment path stops there, because the directory rectangle and the height of the reorganized subtree remain unchanged.
2. Send to b : its parent is now the former parent of a , its children e and a ,
3. Send to e : its children are s_1 and s_2 , compute its overlapping coverage with a ;
4. Send to s_1 and s_2 : their parent is e ,
5. Send to s : its parent is a ,

In addition, the routing node a which drives the rotation must self-adjust its own representation. In

particular, its parent is now b , its children c and s , it overlapping coverage with e is $e.dr \cap a.de$. When the `move(d)` rotation is chosen, the messages for e and its children are not necessary, and this reduces the cost to three messages.

The overlapping coverage must also be updated for the subtrees rooted at f , d , g and c . Consider again Figure 5, assuming that the chosen rotation is 5.b.

1. since $f.dr \cap a.dr \neq \emptyset$, a message `UPDATEOC` is sent to the children of f ,
2. since $g.dr \cap e.dr \neq \emptyset$, a message `UPDATEOC` is sent to the children of g ,
3. no update of the OC information is required for both d and c .

Updating the overlapping coverage in a subtree may result, at worse, in a dissemination to all the leaves. If a balancing occurs at the tree root, the whole tree may be affected. In practice the impact is limited to the nodes N that overlap either with $e.dr$ or $b.dr$, depending on which one is outer with respect to N .

3 Index construction

An important concern when designing a distributed tree is the load of the servers that store the routing nodes located at or near the root. These servers are likely to receive proportionately much more messages. In the worst case all the insertion messages must be first routed to the root. This is unacceptable in a scalable data structure which must distribute evenly the work over all the servers.

3.1 The image

The application that requests insertions maintains an *image* of the distributed tree. This image provides a view which may be partial and/or outdated. Using the image, the user/application estimates the address of the *target server* which is the most likely to store the object. If the image is obsolete, the insertion can

be routed to an incorrect server. The structure delivers then the insertion to the correct server using its actual routing node at the servers. The correct server sends back an image adjustment message (IAM) to the requester.

An image is a collection of links (see Definition 2). Each time a server S is visited, the following links can be collected: the data link describing the data node of S ; the routing link describing the routing node of S , and the left and right links of the routing node.

Recall that the type of a link can be `data` or `routing`. Therefore visiting a server may result in acquiring more than one data link. These four links are added to any message forwarded by S . When an operation requires a chain of n messages, the links are cumulated so that the application finally receives an IAM with $4n$ links.

When the insertion of an object o has to be performed, the image is examined according to the following procedure.

CHOOSEFROMIMAGE (I : *image*, mbb : *rectangle*)

Input: I , a list of data or routing links, a rectangle mbb

Output: a link to a target server

1. all the data links are considered first; if a link is found whose directory rectangle contains mbb , it is kept as a candidate; when several candidates are found, the one with the smallest dr is chosen;
 2. if no data link has been found, the list of routing links are considered in turn; among the links whose dr contains mbb , if any, one chooses those with the minimal height (i.e., those which correspond to the smallest subtrees); if there are still several candidates, the one with the smallest dr is kept;
 3. finally, if the above investigations do not find a link that covers mbb , the data link whose dr is the closest to mbb is chosen.
-

The rationale for these choices is that one aims at finding the data node which can store o without any enlargement. If it happens that several choices are possible, the one with the minimal coverage is chosen because it can be estimated to be the most accurate one. This heuristic is motivated by the fact that the coverage of the servers shrinks at each split. Therefore one may suspect that if, in the image, a data node d_1 covers partially and is larger than a data node d_2 , its representation is outdated and was received before a split. A second heuristic is to choose the data node d whose coverage is closest to o , when all the other attempts fail. Indeed one can expect to find the correct data node in the neighborhood of d , and therefore in the local part of the SD-Rtree.

3.2 The algorithm

The main SD-Rtree variant considered in what follows maintains an image on the client component, although we shall investigate in our experiments another variant that stores an image on each server component. Initially a client C knows only its *contact server*. The IAMs allow to extend this knowledge and avoid to overflow this server with insertions that must be forwarded later on. In order to insert an object o , C searches its local image, following the above procedure, to find the link to the target server S . If the link is of type data, C addresses a message INSERT-IN-LEAF to S ; else the link refers to a routing node and C sends a message INSERT-IN-SUBTREE to S .

- (INSERT-IN-LEAF message) S receives the message; if the directory rectangle of its data node d_S covers actually $o.mbb$, S can take the decision to insert o in its local repository; there is no need to make any other modification in the distributed tree (if no split occurs); else the message is *out of range*, and a message INSERT-IN-SUBTREE is routed to the parent S' of d_S ;
- (INSERT-IN-SUBTREE message) when a server S' receives such a message, it first consults its routing node $r_{S'}$ to check whether its directory rectangle covers o ; if no the message is forwarded to the parent until a satisfying subtree is

found (in the worst case one reaches the root); if yes the insertion is carried out from $r_{S'}$ using the classical Rtree top-down insertion algorithm. During the top-down traversal, the directory rectangles of the routing nodes may have to be enlarged.

If the insertion could not be performed in one hop, the server that finally inserts o sends an acknowledgment to C , along with an IAM containing all the links collected from the visited servers. C can then refresh its image.

The insertion process is shown on Figure 6. The client chooses to send the insertion message to S_2 . Assume that S_2 cannot make the decision to insert o , because $o.mbb$ is not contained in $d_2.dr$. Then S_2 initiates a bottom-up traversal of the SD-Rtree until a routing node whose dr covers o is found (node c on the figure). A classical insertion algorithm is performed on the subtree rooted at c . The *out-of-range path* (ORP) consists of all the servers involved in this chain of messages. Their routing and data links constitute the IAM which is sent back to C .

Figure 6: The insertion algorithm

Initially the image of C is empty. The first insertion query issued by C is sent to the contact server. More than likely this first query is out of range and the contact server must initiate a path in the distributed tree through a subset of the servers. The client will get back in its IAM the links of this subset which serve to construct its initial image.

An image becomes obsolete as splits occur and new servers are added to the system. In that case one expects that the out-of-range path remains local and involves only the part of the tree that changed with respect to the client image. This is illustrated on Figure 7. The image of the client C (left part) contains a

Figure 7: An example of out-of-range query, due to an outdated client image

link to the data node d_2 which has actually been split in the current SD-Rtree (right part of the figure). The directory rectangle of d_2 known by C is obsolete. When C wants to insert an object o , it considers S_2 as the target server. S_2 gets the message and finds that o falls out of $d_{S_2}.dr$. The insertion message is routed to the parent routing node r_4 which is stored on S_4 .

Since the directory rectangle of r_4 is the union of the dr of d_2 and d_4 , it is likely to contain o . The insertion of o will therefore be initiated, starting from r_4 , and will be routed to d_2 or to d_4 by CHOOSESUBTREE (for our example, the choice is d_4). The number of messages in this case is 2 (because r_4 and d_4 reside on the same server). C gets an IAM describing the adjustment of the local part of its image.

In the worst case a client C sends to a server S an out-of-range message which triggers a chain of unsuccessful INSERT-IN-SUBTREE messages from S to the root of the SD-Rtree. This costs $\log N$ messages. Then another set of $\log N$ messages is necessary to find the correct data node. Finally, if a split occurs, another bottom-up traversal might be required to adjust the heights along the path to the root. So the worst-case results in $O(3 \log N)$ messages. However, if the image is reasonably accurate, the insertion is routed to the part of the tree which should hosts the inserted object, and this results in a short out-of-range path with few messages. This strategy reduces the workload of the root since it is accessed only for objects that fall outside the bound-

aries of the most-upper directory rectangles.

3.3 Deletion

Deletion is somehow similar to that in an R-Tree [6]. A server S from which an object has been deleted may adjust covering rectangles on the path to the root. It may also eliminate the node if it has too few objects. The SD-Rtree relocates then the remaining objects to its sibling S' in the binary tree. Node S' becomes the child of its grandparent. An adjustment of the height is propagated upward as necessary, perhaps requiring a rotation. We do not elaborate on deletions further, as they are rare in practice.

4 Point and range queries

Point and window queries are similar in their principles to the insertion algorithm described in the previous section. The main features of the point and window query algorithms is the combined use of the client image and of the overlapping coverage information in order to remain as much as possible near the leaves level in the tree, thereby avoiding root overloading.

4.1 Point queries

The point query algorithm uses a basic routine, PQTRAVERSAL, which is the classical point-query algorithm for Rtree: at each node, one checks whether the point argument P belongs to the left (resp. right) child's directory rectangle. If yes the routine is called recursively for the left (resp. right) child node.

First the client searches its image for a data node d whose directory rectangle contains P , according to its image. A point query message is then sent to the server S_d (or to its contact server if the image is empty). Two cases occur: (i) the data node rectangle on the target server contains P ; then the point query can be applied locally to the data repository, and a PQTRAVERSAL must also be routed to the outer nodes o in the overlapping coverage array $d.OC$ whose rectangle contains P as well; (ii) an out-of-range occurs (the data node on server S_d

does not contain P). The SD-Rtree is then scanned bottom-up from S_d until a routing node r that contains P is found. A PQTRAVERSAL is applied from r , and from the outer nodes in the overlapping coverage array $r.OC$ whose directory rectangle contains P .

This algorithm ensures that all the parts of the SD-Rtree which may contain the point argument are visited. The overlapping coverage information stored at each node avoids to visit the root for each query.

With an up-to-date client image, the target server is correct, and the number of PQTRAVERSAL which must be performed depends on the amount of overlapping with the leaf ancestors. It is well known in the centralized case that a point might be shared by all the rectangles of an Rtree, which means that, at worse, all the nodes must be visited. With a decent distribution, a constant (and small) number of leaves must be inspected. The worst case occurs when the data node dr overlaps with all the outer nodes along its root path. Then a point query must be performed for each outer subtree of the root path. In general the cost can be estimated to 1 message sent to the correct server when the image is accurate, and within $O(\log N)$ messages with an outdated image.

4.2 Window queries

Window queries are similar to point queries. Given a window W , the client searches its image for a link to a node that contains W . The CHOOSEFROMIMAGE procedure can be used. A query message is sent to the server that hosts the node. There, as usual, an out-of-range may occur because of image inaccuracy, in which case a bottom-up traversal is initiated in the SD-Rtree. When a routing node r that actually covers W is found, the subtree rooted at r , as well as the overlapping coverage of r , allow to navigate to the appropriate data nodes. The algorithm is given below. It applies also, with minimal changes, to point queries. The routine WQTRAVERSAL is the classical Rtree window query algorithm adapted to a distributed context.

WINDOWQUERY (W : rectangle)

Input: a window W

Output: the set of objects whose mbb intersects W

begin

 // Find the target server

$targetLink := \text{CHOOSEFROMIMAGE}(Client.image, W)$

 // Check that this is the correct server. Else move up the tree

$node :=$ the node referred to by $targetLink$;

while ($W \not\subseteq node.dr$ **and** $node$ is not the root) // out of range
 $node := parent(node)$

endwhile

 // Now $node$ contains W , or $node$ is the root

if ($node$ is a data node)

 Search the local data repository $node.data$

else

 // Perform a window traversal from $node$

 WQTRAVERSAL ($node, W$)

end

 // Always scan the OC array, and forward

for each (i, oc_i) **in** $node.OC$ **do**

if ($W \cap oc_i \neq \emptyset$) **then**

 WQTRAVERSAL ($outer_{node}(i), W$)

endif

end for

end

The analysis is similar to that of point queries. The number of data nodes which intersect W depends on the size of W . Once a node that contains W is found, the WQTRAVERSAL must be broadcasted towards these data nodes. The maximal length of each of these broadcasted message paths is $O(\log N)$. Since the requests are forwarded in parallel, and result each in an IAM when a data node is finally reached, this bound on the length of a chain guarantees that the IAM size remains small.

4.3 Termination protocol

The termination protocol lets an SD-Rtree client issuing a point or window query figure out when to end the communication with the servers and return the result to the application. As in general for an SDDS, SDR-tree clients and servers may follow a probabilistic or deterministic protocol. The probabilistic protocol means here that (i) only the servers with data relevant to the query respond, (ii) the client considers as established the result got within some timeout. In unreliable configuration such protocol

may lead to a miss, whose probability may though be negligible in practice.

A straightforward deterministic protocol is in our case the *reverse path* protocol. Each contacted server other than the initial one, i.e., getting the query, sends the data found to the node from which it got the query. The initial server collects the whole reply and sends it to the client. The obvious cost of the protocol is that each path followed by the query in the SD-Rtree has to be traversed twice. The alternate *direct reply* protocol does not have this cost, at the expense of processing at the client. Our performance evaluation considers this protocol. In the nutshell, each server getting the query responds to the client, whether it found the relevant data or not. Each reply has also the description of the query path in the tree between the initial server and the responding one. The traversed servers accumulate this description including into it also all its outer nodes intersecting the query window. It includes the OC tables found. The client examines the graph received and figures out whether the end of every path in the graph is the server it got the reply from. It resends messages to the servers whose replies seem lost.

5 Experimental evaluation

We performed several experiments to evaluate the performance of our proposed architecture over large datasets of 2-dimensional rectangles, using a distributed structure simulator written in C. Our datasets are produced by the GSTD generator [18]. The experimental study involves the following variants of the SD-Rtree:

BASIC. This variant does not use an image on the client nor on the servers. Each request, whether it is an insertion, a point query or a window query, is sent to the server that maintains the root node. From there we proceed to a top-down traversal of the tree to reach the adequate server. This variant is implemented for comparison purposes, since the high load of the root levels makes it unsuitable as a SDDS.

IMCLIENT. This is the main variant described in the previous sections. Each client component builds an image of the SR-Rtree structure, and corrects incre-

(a) Insertions of uniformly distributed data

(b) Insertions of skewed data

Figure 8: Number of messages for data insertion

mentally this image through adjustment messages. We recall that the servers have their actual routing nodes they use for query forwarding.

IMSERVER. The third variant maintains an image on each server component and not on the client component. This corresponds to an architecture where many light-memory clients (e.g., PDA) address queries to a cluster of interconnected servers. We simulate this by choosing randomly, for each request (insertion or query) a contact server playing the role of a services provider. The contact server uses its own image.

We study the behavior of the different variants for insertions ranging from 50,000 to 500,000 objects (rectangles). We also execute against the structure 0-3,000 point and window queries. The cost is measured as the number of messages exchanged between server. The size of the messages remains, as expected, so small (at most a few hundreds of bytes) that this can be considered as negligible. The data node on each server is stored as a main memory R-tree, and the capacity of the servers is set to 3,000

objects.

5.1 Cost of insertions

For the three variants we study the behavior after an initialization of the SD-Rtree with 50,000 objects. This avoids partially the measures distortion due to the cost of the initialization step which affects primarily the first servers. The comparisons between the different techniques are based on the total number of messages received by the servers, and on the load balancing between servers.

Messages

Figure 8(a) shows the total number of messages for insertions of objects following a uniform distribution. It illustrates the role of the images. While BASIC requires on average 8 messages when the number of insertions is 500,000, IMSERVER needs 6 messages on average, thus a 25% gain. The cost of each insertion for the BASIC variant is approximately the length of a path from the root to the leaf. The final, maximal, height of the tree is here 8. Additional messages are necessary for height adjustment and for OC maintenance, but their number remains low.

With IMSERVER, each client routes its insertions to its contact server. When the contact server has an up-to-date image of the structure, the correct target server can be reached in 2 messages. Otherwise, an out-of-range occurs and some forwarding messages are necessary along with an IAM. We experimentally find that the average number of additional messages after an out-of-range is for instance 5 with 252 servers and 500,000 insertions. The gain of 25% is significant compared to the BASIC variant, but even more importantly this greatly reduces the unbalanced load on the servers (see below).

Maintaining an image on the client ensures a drastic improvement. The average number of messages to contact the correct server, always for 500,000 insertions, decreases to 1 message on average, i.e., we quickly need only one message. The convergence of the image is naturally much faster than with IMSERVER because a client that issues m insertions will get an IAM for the part of these m insertions

which turns out to be out-of-range. Using the IMSERVER variant and the same number of insertions, a server will get only $\frac{m}{N}$ insertions requests (N being the number of servers), and much less adjustment messages. Its image is therefore more likely to be outdated. Our results show that the IMCLIENT variant leads to a direct match in 99.9% of the cases.

Figure 8(b) shows the behavior of the variants when the distribution of the data is skewed. The main difference concerns IMSERVER. Indeed, the gain reaches now 40% compared to BASIC. This improvement was expected since most of the inserted data belong to the same part of the space, and affect therefore the same servers. Consequently these servers obtain quickly an up-to-date image, and forward more efficiently an insertion message.

Table 1 summarizes the characteristics of the SDR-tree variants, initialized as above, for a large number of insertions. With a uniform distribution, the tree grows regularly and its height follows exactly the rule $2^{height-1} < N \leq 2^{height}$. The average load factor is around 70%, i.e., around the well-known typical $\ln 2$ value. The BASIC variant requires a few more messages than the height of the tree, because of height adjustment and overlapping coverage maintenance. On the average, the number of messages per insertion is equal to the final height of the tree. With IMSERVER the number of messages is lower because (i) a few forwarding messages are sufficient if the contacted node has split, in which case the correct server can be found locally, and (ii) if no information regarding the correct server can be found in the image, an out-of-range path is necessary.

The length of an out-of-range path should be the height of the tree on the average. But the heuristics that consists in choosing the “closest” server in the image (i.e., the one with the smallest necessary directory rectangle enlargement) turns out to be quite effective by reducing in most cases the navigation in the SD-Rtree to a local subtree. Table 1 shows for instance that with a tree of height 7 with 127 servers, only 4 additional messages are necessary to reach the correct server (2 bottom-up, and 2 top-down messages).

Finally, the average number of messages for IM-

nb objects	UNIFORM DISTRIBUTION						SKEWED DISTRIBUTION					
	nb serv.	height	load(%)	BASIC	IMSERV	IMCL	nb serv.	height	load(%)	BASIC	IMSERV	IMCL
50,000	58	6	57.5	6	3	5	49	7	68.0	6	3	8
100,000	64	6	78.1	6	3	3	73	7	68.5	6	3	4
150,000	108	7	61.7	6	3	3	96	8	69.4	7	4	3
200,000	125	7	66.7	7	4	3	122	8	68.3	7	4	3
250,000	127	7	78.7	7	4	3	149	9	67.1	7	5	3
300,000	166	8	70.3	7	4	3	176	9	66.3	8	5	3
350,000	207	8	64.4	8	5	3	200	9	66.7	8	5	3
400,000	233	8	64.4	8	5	3	224	9	66.7	8	5	3
450,000	240	8	69.4	8	5	3	246	9	67.8	8	5	3
500,000	243	8	75.4	8	5	3	267	10	68.7	8	5	3

Table 1: Average number of messages per insertion for variants BASIC, IMSERVER and IMCLIENT

CLIENT does no longer depend on the height of the tree. After a short acquisition step (see the analysis on the image convergence below), the client has collected enough information in its image to contact either directly the correct server, or at least a close one. The difference in the number of messages with the IMSERVER version lies in the quality of the image, since a client quickly knows almost all the servers.

We observe the same behavior with a skewed distribution, except that for the same number of servers the height is slightly higher. This leads to an increase of the average number of additional messages for BASIC and IMSERVER.

Figure 9: Messages distribution for insertions

Load balancing

Figure 9 analyzes the distribution of messages with respect to the position of a node in the tree. Using the BASIC variant, the servers that store the root or other high-level internal nodes have much more work than the others. Basically a server storing a

routing node at level n receives twice more messages than a server storing a routing node at level $n - 1$. This is confirmed by the experiments, *e.g.*, the server that manages the root handles 12.67% of the messages, while the servers that manage its children received 6.38%. Figure 9 shows that maintaining an image (either with IMSERVER or IMCLIENT) not only allows to save messages, but also distributes much more evenly the workload.

The distribution depends actually on the quality of the image. With IMSERVER, each server S is contacted with equal probability. If the image of S is accurate enough, S will forward the message to the correct server S' which stores the object. Since all the servers have on average the same number of objects, it is expected that each server receives approximately the same number of messages. At this point, for a uniform distribution of objects, the load is equally distributed over the server. The probability of having to contact a routing node N decreases exponentially with the distance between the initially contacted data node and N . The initial insertion of 50,000 objects results in a tree whose depth is 5, hence the lower number of messages for the nodes with height 1, 2 or 3, since they are newer.

Finally the last column illustrates the balancing of the workload when the client keeps an image. Since a client acquires quickly a complete image, it can contact in most case the correct server. The same remark holds for nodes whose level is 1, 2 or 3 as above.

(a) Uniformly distributed data

(b) Skewed data

Figure 10: Adjustment and rotation messages

Cost of balancing

There is a low overhead due to the balancing of the distributed tree. Figure 10(a) shows the average number of additional messages required to balanced the tree depending on the number of insertions. With our 3000-objects capacity and 500,000 insertions of uniformly distributed data for instance, we need only 440 messages for updating the heights of the subtrees and 0 for rotations, to maintain the tree balanced, *i.e.*, around 1 message for every 1000 insertions.

For Figure 10(a), the tree grows uniformly because of the uniform distribution and because of the large capacity of the servers. This explains the absence of balancing operations. With skewed data more messages are necessary for maintaining the height (640 instead of 440 for 500,000 insertions) and additional messages are required to balance the tree (310). Nonetheless on average, only 1 message per 500 insertions is necessary for maintaining the tree.

Figure 11 illustrates the convergence speed of the client image. With less that 200 query messages for

Figure 11: Convergence of the client image

instance, the client builds an image referencing 80% of the servers. With only 30 messages it knows half of the servers. The logarithmic aspect of the curve is due to the fact that at the beginning the client acquires quickly a lot of information, since each query explores a path not yet recorded in the image. It becomes more and more uncommon for a query to address an unknown server. For instance after 30 messages the client knows half of the server, thus the 31st message may address with a probability 0.5 a server already known. Hence the logarithmic behaviour.

5.2 Cost of queries

We turn now to the point and window queries. The following experiments create first a tree by inserting 200,000 objects uniformly distributed. One obtains a tree composed of 107 servers with a maximal height of 7. Then we evaluate 0-3,000 queries.

Figure 12(a) shows the gain of the image-aware variants compared to the BASIC one. Since the tree remains stable (no insertions), we need on average a constant number of messages to retrieve the answer in BASIC. As expected the variants which rely on an image outperform BASIC. For these variants, the results follow quickly a linear growth with respect to the number of queries. After the acquisition step, either all the servers or the client, depending on the variant, have a complete image and thus can contact directly the correct server(s). The convergence is much faster for IMCLIENT than for IMSERVER since with 200 queries for instance, IMSERVER requires as many messages as BASIC, whereas IMCLIENT needs

Figure 12: Cost of query answering

only the half. This shows that the IMCLIENT is very efficient even for a small number of queries. The IMSERVER saves 50% (resp. 40%) of the messages and IMCLIENT 65% (resp. 60%) for 3,000 (resp. 1,000) queries, compared to BASIC. Figure 12(a) also shows that only 3 messages per query suffice, on the average.

Figure 12(b) summarizes the window queries experiments. The extend of the query rectangle on each axis is randomly drawn up to 10% of the space extend. The cost of a window query tends to be about twice that of a point query for all the three variants. The higher cost of a window query is due to the overlap between the window and the dr of the servers. The figure shows also the good influence of the images on the window queries which is that the costs of IMCLIENT tend to be a half of those of BASIC. The reason is again the acquisition of images. This one is faster, incidentally, for window queries than for the point queries, since the former contact more servers. Finally, we see also that we get the answer with about 8 messages, for IMCLIENT variant. This

number would increase for a larger query window than in our setting. It would cover a more significant portion of the embedding space, hence would address more servers and solicitate more the upper levels of the tree.

Figure 13: Rate of good matches for point queries

Figure 13 shows the ratio of correct matches when an image is used for point query. With IMSERVER, after 1500 (resp. 2500) queries, any server has an image that permits a correct match in 80% (resp. 95%) of the cases. For IMCLIENT, only 600 queries are necessary, and with 200 queries the structure ensures a correct match for 80% of the queries. This graph confirms the results of Figure 12, with very good results for IMCLIENT even when the number of queries is low.

Figure 14: Messages distribution for queries

Finally Figure 14 confirms that using an image serves to obtain a satisfying load balancing, for the very same reasons already given in the analysis of the insertion algorithm.

6 Related work

Until recently, most of the spatial indexing design efforts have been devoted to centralized systems [4] although, for non-spatial data, research devoted to an efficient distribution of large datasets is well-established [3, 14, 2, 16]. The architecture of the SD-Rtree respects closely the design requirements of *Scalable Distributed Data Structures* (SDDS) [15]. They can be summarized as follows: (i) no central directory is used for data addressing, (ii) servers are dynamically added to the system when needed and (iii) the clients addresses the structure through a possibly outdated image. Many schemes are hash-based, e.g., variants of LH* [15], or use a Distributed Hash Table (DHT) [3]. Some SDDSs are range partitioned, starting with RP* based [14], till BATON [8] most recently. There were also proposals for the k-d partitioning, e.g. k-RP [13] using distributed kd-trees for points data, or hQT* [10] using quadrees for the same purpose. Parallel execution of spatial queries, based on a simple partitioning scheme, is also proposed in [17]. [7] presents a distributed data structure based on orthogonal bisection tree (2-d KD tree). Each processor has an image of the tree. The balancing needs to fully rebuild the tree using multicast from all the servers. [11] describes an adaptive index method which offers dynamic load balancing of servers and distributed collaboration. The structure requires a coordinator which maintains the load of each server.

The P-tree [2] is an interesting distributed B+-tree that has a concept similar to our image with a best-effort fix up when updates happen. Each node maintains a possibly partially inconsistent view of its neighborhood in the distributed B+-tree. A major difference lies in the correction which is handled by dedicated processes on each peer in the P-tree, and by IAMs triggered by inserts in the SD-Rtree.

The recent work [9] proposes an ambitious framework termed VBI that “can be used to implement a variety of ... region-based index structures, including .. R-tree ..., in a peer-to-peer system”. The framework is a distributed dynamic binary tree with nodes at peers. VBI shares this and other principles with

the SD-Rtree. With respect to the differences, first, SD-Rtree is a data structure worked out to its full extent. It is partly in VBI scope, but fully roots instead in the more generic SDDS framework [15]. An SD-Rtree can also have the client component (our IMCLIENT variant) that in particular lets usually for a substantial access performance gain, as Section 5.1 shows. Next, VBI seems aiming at the efficient manipulation of multi-dimensional points. An SD-Rtree rather targets the spatial (non-zero surface) objects, as R-trees specifically. Consequently, an SD-Rtree enlarges a region synchronously with any insert needing it. VBI framework advocates instead the storing of the corresponding point inserts in routing nodes, as so-called discrete data. These are subject of delayed (lazy) processing, ultimately leading to a batch enlargement. The approach has important consequences for the VBI conform distributed data structures. Beyond, it seems an open question how far one can apply this facet of VBI to spatial objects.

By the same token, VBI advocates four specific AVL-tree rotations for its tree balancing. The SD-Rtree also uses rotations for the balancing, but some are different of those in VBI. Its rotation algorithm is also more specific to spatial objects. As discussed in Section 2, it aims at minimizing the spatial overlap. Likewise, since SD-Rtree is a specific distributed data structure, we could formulate the termination protocols for its range queries. VBI does not propose any such protocol and it is not sure that one could even formulate it at any framework level. How all these similarities and differences will finally please the applications, largely remains to be studied. One reason is that ours and VBI experimental performance evaluation environments are highly dissimilar. VBI reports a study of messaging in the tree basically with 10 objects per leaf node, but scaling to 10,000 nodes, i.e., up to 100 000 objects. Our setup involved, we recall, 500 000 objects leading to about 270 leaf nodes, hence loaded each with about 2100 objects. This, leading to our average and about typical load factor of 70 %, given our choice of maximal capacity of 3 000 objects per leaf, in line with application requirements we were aware of. Besides,

neither VBI nor we, nor anyone else to the best of our knowledge, yet has the wall-clock timing results.

Finally, our study, and every other related one above, yet remains about entirely open with respect to the clever design of the additional features like concurrency, transactions, node failures, fault tolerance, security. We plan, in particular, to deal with the node failures by applying the erasure correcting code in [12].

7 Conclusion

The SD-Rtree provides the Rtree capabilities for large spatial data sets stored over interconnected servers. The distributed addressing and specific management of the nodes with the overlapping coverage avoid any centralized calculus. The analysis, including the experiments, confirmed the efficiency of our design choices. The scheme should fit the needs of new applications of spatial data, using endlessly larger datasets, e.g., Google Earth or SkyServer to cite the most visible ones.

Future work on SDR-tree should include other spatial operations: kNN queries, distance queries and spatial joins. One should study also more in depth the concurrent distributed query processing. As for other well-known data structures, additions to the scheme may perhaps increase the efficiency in this context. Another direction is the analysis of the R*tree type of splitting, or the related ones that we did not address in our experiments.

Acknowledgment: We are grateful to Prof. N. Rousopoulos for his useful comments on this manuscript.

References

- [1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*tree : An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.
- [2] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, pages 25–30, 2004.
- [3] R. Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In *Foundations of Data Organization and Algorithms (FODO)*, pages 101–114, 1993.
- [4] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.
- [5] Y. Garcia, M. Lopez, and S. Leutenegger. On Optimal Node Splitting for R-trees. In *VLDB*, 1998.
- [6] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 45–57, 1984.
- [7] S. E. Hambrusch and A. A. Khokhar. Maintaining Spatial Data Sets in Distributed-Memory Machines. In *Proc. Intl. Parallel Processing Symposium (IPPS)*, pages 702–707, 1997.
- [8] H. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *VLDB*, pages 661–672, 2005.
- [9] H. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes., 2006. <http://www.comp.nus.edu.sg/ooibc/>. To appear at ICDE’06.
- [10] J. S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In *Foundations of Data Organization and Algorithms (FODO)*, pages 37–46, 1998.
- [11] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *EDBT*, pages 748–764, 2004.

- [12] W. Litwin, R. Moussa, and T. Schwarz. LH*RS-A Highly-Available Scalable Distributed Data Structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.
- [13] W. Litwin and M.-A. Neimat. k-RP*S: A Scalable Distributed Data Structure for High-Performance Multi-Attribute Access. In *Proc. Intl. Conf. on Parallel and Distributed Inf. Systems (PDIS)*, pages 120–131, 1996.
- [14] W. Litwin, M.-A. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *VLDB*, pages 342–353, 1994.
- [15] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM Trans. on Database Systems*, 21(4):480–525, 1996.
- [16] A. D. Pasquale, E. Nardelli, and G. Proietti. An RP* Extension with Almost Constant Amortized Costs. In *Proc. Intl. Workshop on Distributed Data and Structures*, pages 1–15, 2003.
- [17] J. M. Patel, J.-B. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. E. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, D. J. DeWitt, and J. F. Naughton. Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *SIGMOD*, pages 336–347, 1997.
- [18] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, 1999.