

SD-Rtree: A Scalable Distributed Rtree

Cédric du Mouza
LAMSADE, Univ. Paris-Dauphine
Paris, France

cedric.dumouza@dauphine.fr

Witold Litwin
CERIA, Univ. Paris-Dauphine
Paris, France

witold.litwin@dauphine.fr

Philippe Rigaux
LAMSADE, Univ. Paris-Dauphine
Paris, France

philippe.rigaux@dauphine.fr

Abstract

We propose a scalable distributed data structure (SDDS) called *SD-Rtree*. *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 The client addresses the tree through its image that the splits can make outdated.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 Specific messages towards the clients incrementally correct the outdated images.

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 [15]: (i) no central directory is used for data addressing, (ii) servers are dynamically added to the system when needed and (iii) the clients address the structure through a possibly outdated image.

An *SD-Rtree* avoids redundancy of objects references, like *Rtree* or *R*tree*. The general structure is that of a distributed balanced binary spatial tree where each node carries a mbb. 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, a split occurs, moving some data to a dynamically appended bucket.

An application addresses an *SD-Rtree* only through the client component. The address calculus requires neither a centralized component nor multicast messages. A client addresses the servers which are in its *image* of the structure. Some existing servers may not be in the image, due

to splits unknown from the client. The addressing may then send a query to a 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 gets then a specific *image adjustment message* (IAM). This improves the image at least so that the addressing error does not repeat.

We present in this paper the distributed structure of *SD-Rtree* and its algorithms for splitting, balancing and overlapping management. 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. Finally we assess the effectiveness of the structure through experimental results.

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.

Related work

Until recently, most of the spatial indexing design efforts have been devoted to centralized systems [5] although, for non-spatial data, research devoted to an efficient distribution of large datasets is well-established [3, 14, 2]. The architecture of the Many SDDS 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* [9] 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** [11] using *quadrees* for the same purpose. [8] 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. [12] 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 work [10] proposes an ambitious framework termed VBI. 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]. Next, VBI seems aiming at the efficient manipulation of multi-dimensional points. 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. It seems an open question how far one can apply this facet of VBI to spatial objects.

The rest of the paper presents first (Section 2) the structure of the SD-Rtree. Section 3 describes the insertion algorithm and the point and window queries. Section 4 shows the experimental performance analysis and Section 5 concludes the paper. More details on related work and other features discussed in this paper can be found in [4].

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 [7].

Kernel structure

The SD-Rtree is a binary tree, mapped to a set of servers. Each internal node, or *routing node*, refers to exactly two children whose heights differ by at most one. This ensures that the height of a SD-Rtree is logarithmic in the number of servers. A routing node maintains also left and right *directory rectangles* (dr) which are the minimal bounding boxes of, respectively, the left and right subtrees. Finally each leaf node, or *data node*, stores a subset of the indexed objects,

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.

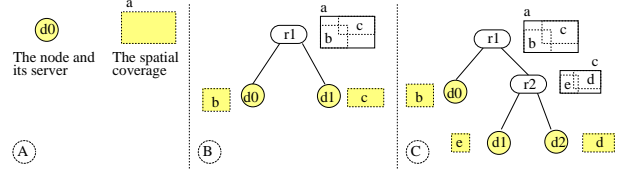


Figure 1. Basic features of the SD-Rtree

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. 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 $\text{Max}(\text{left.height}, \text{right.height})+1$. 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

The image

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 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.

An application that accesses an SD-Rtree maintains an *image* of the distributed tree. This image provides a view which may be partial and/or outdated. During an insertion, the user/application estimates from its image 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. Point and window queries also rely on the image to find quickly a server whose directory rectangle satisfies the query predicate. A message is then sent to this server which carries out a local search, and route the queries to other nodes if necessary.

An image is a collection of links, stored locally, and possibly organized as a local index if necessary. 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. 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.

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 [7, 6]. 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' .

The management and distribution of routing and data nodes are detailed on Figure 2 for the tree construction of

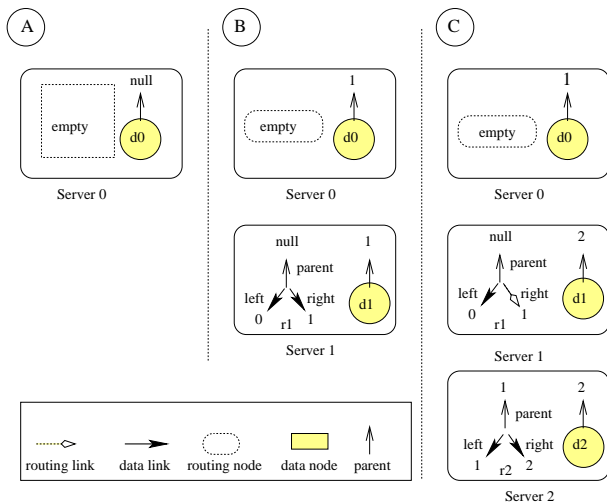


Figure 2. Split operations

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 $mbb(\text{left.dr}, \text{right.dr})$),
- 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 [7, 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.

Overlapping coverage

We cannot afford the traditional top-down search in a

distributed tree because it would overload the nodes near the tree root. 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 this strategy is not sufficient with spatial structures that permit overlapping, because d does not contain *all* the objects covered by dr . We must therefore be able to forward the query to all the 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.

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 .

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 insertion 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.

Balancing

In order to preserve the balance of the tree, a rotation is sometimes required during the bottom-up traversal that adjusts the heights. 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* which is a subtree of the form $a(b(e(f, g), d), c)$ satisfying the following conditions for some $n \geq 1$:

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

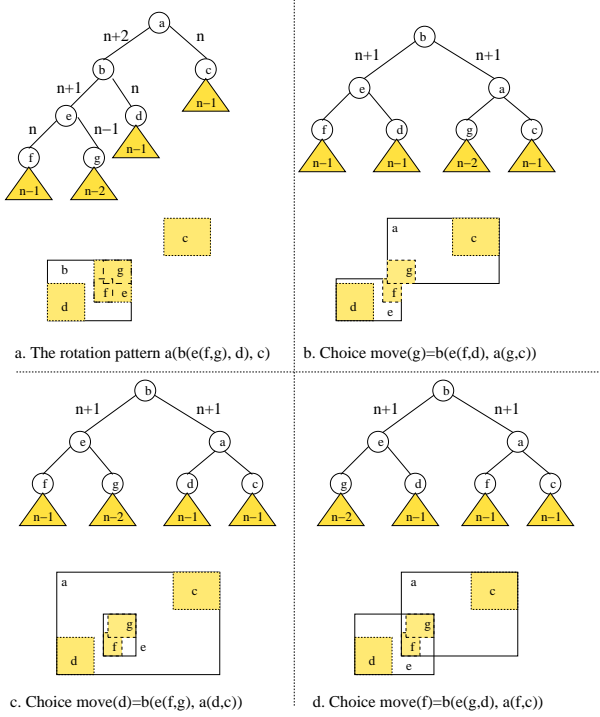


Figure 3. Balancing in the SD-Rtree

An example of rotation pattern is shown on Figure 3. 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.*

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.

Any pairwise combination of f , g , d and c yields a balanced tree. The three possibilities, respectively called $move(g)$, $move(d)$ and $move(f)$ are shown on Figure 3. The choice $move(g)$ (Figure 3.b) is the best one for our example. 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 $move(f)$ and $move(g)$, and 3 messages for $move(d)$ 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 .

The overlapping coverage must also be updated for the subtrees rooted at f , d , g and c .

3 Algorithms

We present now the main algorithms of the SD-tree, namely insertion, deletion, and point and window queries. Recall that all these operations rely on an *image* of the structure (see above) which helps to remain as much as possible near the leaves level in the tree, thereby avoiding root overloading. Moreover, as a side effect of these operations, the image is adjusted through IAMs to better reflect the current state of the structure.

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.

Insertion

In order to insert an object o with rectangle mbb , C searches its local image as follows:

1. all the data links in the image 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;

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. If the above investigations do not find a link that covers mbb , the data link whose dr is the closest to mbb is chosen. 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.

If the selected 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 4. 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 .

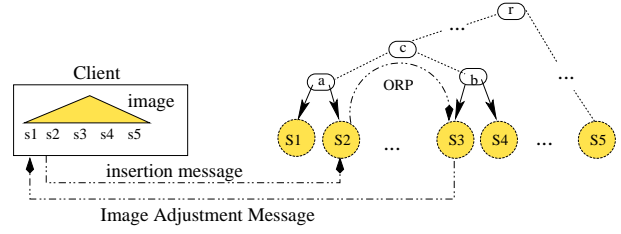


Figure 4. The insertion algorithm

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. 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.

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 host 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 boundaries of the most-upper directory rectangles.

Deletion

Deletion is somehow similar to that in an R-Tree [7]. 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.

Point queries

The point query algorithm uses a basic routine, PQ-TRAVERSAL, 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. 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.

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 := CHOOSEFROMIMAGE(*Client.image*, W)

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

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

while ($W \not\subseteq \text{node.dr}$ **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 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 [16]. 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 SD-Rtree structure, and corrects incrementally this image through adjustment messages. We recall that 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.

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.

Figure 5 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.

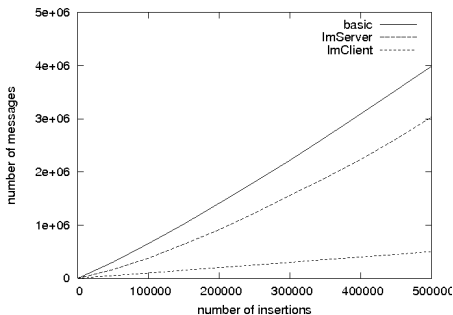


Figure 5. Number of messages for insertion

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 decreases to 1 message on average. The con-

vergence 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.

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.

nb objects	nb serv.	height	load(%)	BASIC	IMSERV	IMCL
50,000	58	6	57.5	6	3	5
100,000	64	6	78.1	6	3	3
150,000	108	7	61.7	6	3	3
200,000	125	7	66.7	7	4	3
250,000	127	7	78.7	7	4	3
300,000	166	8	70.3	7	4	3
350,000	207	8	64.4	8	5	3
400,000	233	8	64.4	8	5	3
450,000	240	8	69.4	8	5	3
500,000	243	8	75.4	8	5	3

Table 1. Number of messages per insertion

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 IMCLIENT 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.

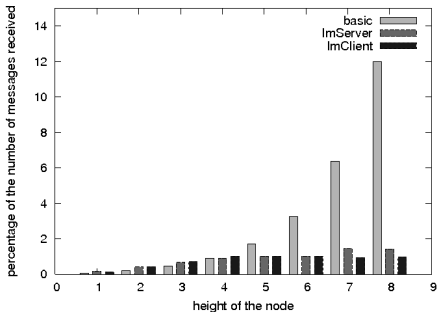


Figure 6. Messages distribution for insertions

Figure 6 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 6 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 approximatively 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.

There is a low overhead due to the balancing of the distributed tree. We perform several experiments to evaluate this overhead. 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.*,

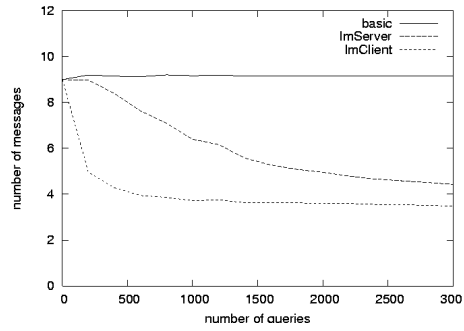


Figure 7. Cost of query answering

around 1 message for every 1000 insertions.

Experiments with skewed data (not reported here due to space limitations) show a similar behavior of the structure. The only noticeable difference is that 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. (in the worst case $\log(n\text{servers})$ and for balancing since the reorganization remains local.

Cost of 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 7 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. If there were no overlap, this number could be predicted to be 7, the height of the tree. The overlapping costs here 2 additional messages on average. As expected the variants which rely on an image outperform BASIC. The number of messages per query decreases with the number of queries, as the server or the client, depending on the variant, acquires a more faithful image and thus contacts more and more frequently the correct server(s) directly. The convergence is faster for IMCLIENT than for IMSERVER. IMCLIENT appears very efficient even for a small number of queries. After 3,000 queries, the search with both variants become almost three times faster than with BASIC.

Window queries experiments give similar results. The higher cost reported, for all variants, is due to the overlap between the window and the dr of the servers.

Figure 8 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

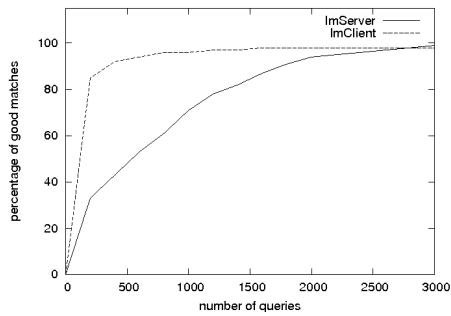


Figure 8. Good matches for point queries

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 7, with very good results for IMCLIENT even when the number of queries is low.

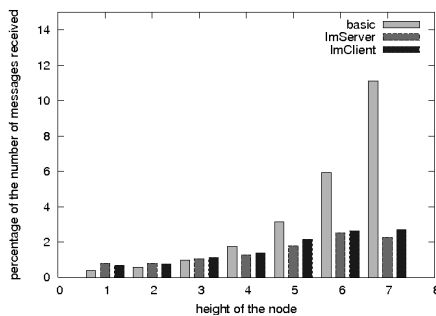


Figure 9. Messages distribution for queries

Finally Figure 9 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.

5 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.

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. A final issue relates to the fanout

of our structure. The binary choice advocated in the present paper favors an even distribution of both data and operations over the servers. A larger fanout would reduce the tree height, at the expense of a more sophisticated mapping scheme. The practicality of the related trade-offs remains to be determined.

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)*, 1993.
- [4] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree, a Scalable Distributed Rtree. Technical report, Lamsade, 2006. <http://www.lamsade.dauphine.fr/rigaux/sdrtree.pdf>.
- [5] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.
- [6] Y. Garcia, M. Lopez, and S. Leutenegger. On Optimal Node Splitting for R-trees. In *VLDB*, 1998.
- [7] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 45–57, 1984.
- [8] S. E. Hambrusch and A. A. Khokhar. Maintaining Spatial Data Sets in Distributed-Memory Machines. In *Proc. Intl. Parallel Processing Symposium (IPPS)*, 1997.
- [9] 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.
- [10] 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. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2006.
- [11] J. S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In *Foundations of Data Organization and Algorithms (FODO)*, 1998.
- [12] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *EDBT*, pages 748–764, 2004.
- [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] 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.