The VLDB Journal (2009) 18:933–958
DOI 10.1007/s00778-009-0135-4

REGULAR PAPER

# Large-scale indexing of spatial data in distributed repositories: the SD-Rtree

**Cédric du Mouza · Witold Litwin · Philippe Rigaux**

Received: 21 December 2007 / Revised: 8 December 2008 / Accepted: 5 January 2009 / Published online: 28 February 2009
© Springer-Verlag 2009

**Abstract** We propose a scalable distributed data structure (SDDS) called SD-Rtree. We intend our structure for point, window and *k*NN queries over large spatial datasets distributed on clusters of interconnected servers. The structure balances the storage and processing load over the available resources, and aims at minimizing the size of the cluster. SD-Rtree generalizes the well-known Rtree structure. It uses a distributed balanced binary 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 The client addresses the tree through its image that can be possibly outdated due to later split. 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 Specific messages towards the clients incrementally correct the outdated images.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 building of an SD-Rtree through insertions, focusing on the split and rotation algorithms. We follow with the query algorithms. We describe then a flexible allocation protocol which allows to cope with a temporary shortage of storage resources through data storage balancing. Experiments show additional aspects of SD-Rtree and compare its behavior with a distributed quadtree. The results justify our various design choices and the overall utility of the structure.

**Keywords** Spatial indexing · Distributed structure

C. du Mouza (✉)
Cedric, CNAM, Paris, France
e-mail: dumouza@cnam.fr

W. Litwin · P. Rigaux
Lamsade, University of Paris-Dauphine, Paris, France
e-mail: witold.litwin@dauphine.fr

P. Rigaux
e-mail: philippe.rigaux@dauphine.fr

## 1 Introduction

Spatial indexing has been studied intensively since the early works on Rtrees and Quadtrees at the beginning of the 1980s (see [21] for a recent in-depth coverage). It constitutes now an integrated part of most database systems engines which usually adopt the simple, flexible and efficient Rtree structure (e.g., *Oracle, MySQL*). Recently, the advent of popular distributed systems for sharing resources across large numbers of computers has encouraged research to extend centralized indexing techniques to support queries in such contexts. However, only a few works so far have considered the extension of spatial structures and algorithms to distributed environments. Since index structures are central to efficient data retrieval, it is important to provide indexing support for distributed processing of spatial queries.

In the present paper we describe a distributed spatial index, based on the Rtree principles, called *Scalable Distributed Rtree* (SD-Rtree). It supports all the crucial operations found in centralized systems : insertions (without duplication or clipping) and deletions; point, window and *k*NN queries. Nodes communicate only through point-to-point messages.

Our structure conforms to the general principles of *Scalable Distributed Data Structures* (SDDS) [17]: (i) no central directory is used for data addressing, (ii) servers are dynamically added to the system when needed and (iii) a client addresses the structure through its *image*, which is a local cache storing a possibly outdated representation of the distributed tree. Moreover the SD-Rtree aims at an even distribution of the processing and storage load over the participating servers, and does not rely on "super peers" or hierarchical network topologies.

We typically target a cluster of servers providing fast access to a very large collection of geographic objects. We

assume that the cluster incrementally adjusts itself to the size of the dataset, by adding or removing a server when necessary.

An SD-Rtree indexes large datasets of 2D objects, each uniquely identified by an *object id* (oid). The 2-densional extent of each object is approximated by the *minimal bounding box (mbb)*. Each object is referenced only once by the index. The general structure is a *distributed balanced binary tree* (like, e.g., [12]), adapted to 2D objects indexing, 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 much faster access than on disks. The structure is designed to fairly share the storage among the set of available servers.

If a server overflows, the index uses a flexible strategy to dynamically reorganize itself. A new server is allocated to the index. A *split* of the full server's bucket occurs, and half of its data is moved to the new server. This leads to a limited reorganization of the distributed tree, and to its possible *structure rebalancing*. Since, however, we cannot assume that a server is available each time it is requested, the SD-Rtree may also choose a *storage balancing method* which redistributes objects from a full server to other nodes without requiring the allocation of new resources. The insertion policy can therefore be tuned dynamically to cope with periods of storage shortage, at the cost of additional message exchanges between servers.

An application addresses an SD-Rtree only through a *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 an insert or search 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. The forwarding traverses some path in the tree. It ends by reaching the correct server(s), if any. The client gets then a specific *image adjustment message (IAM)*. This improves the image at least so that the addressing error does not repeat. Using an image avoids to systematically access the root node, or more generally the upper levels of the tree. This ensures that the processing and communication loads are uniformly distributed.

We analyze the access performance of our scheme as the number of messages sent to the servers. The performance analysis shows that on average, insert and point query over $N$ servers cost only 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 2 $\log N$, unless an infrequent split adds another $\log N$. The overlapping of Rtree nodes may add up to $N$ messages but in general the cost is negligible. The processing of window queries is also efficient, as the maximal message path length to evaluate a window query is $O(\log N)$. We also assess the effectiveness of the structure through experimental results, regarding in particular the role of images for load balancing, and the efficient and fair utilization of the available storage.

The present paper constitutes an extension of a short version published in [7], and integrates the flexible storage balancing method published in [6]. We provide a complete and consistent description of the SD-Rtree structure, including the full set of algorithms, and a termination protocol. We also extend the set of experiments presented in the two conference papers by testing larger synthetic and real datasets and checking the behavior of the structure with various load factors. The experimental section also provides an in-depth comparison with a distributed version of the MX-CIF quadtree, recently proposed in [22].

Below, after analyzing the related work (Sect. 2), we present the distributed structure of SD-Rtree and its algorithms for splitting, rotation and overlapping management (Sect. 3). Next, we discuss the algorithms for the insertion, deletion, point, window and $k$NN queries (Sect. 4). We describe our storage balancing method in Sect. 5. Section 6 introduces the system architecture and describes the implementation and variants of both the SD-Rtree and the distributed quadtree whose experimental behaviors are reported and discussed in Sect. 7. Finally we conclude our proposal and show some directions for future work.

## 2 Related work

Until recently, most of the spatial indexing design efforts have been focused on centralized systems [8] although, for non-spatial data, research devoted to an efficient distribution of large datasets is well-established [4,5,16]. Many SDDS schemes are hash-based, e.g., variants of LH* [17], or use a Distributed Hash Table (DHT) [5]. Some SDDSs are range partitioned, starting with RP* [16], till BATON [12] most recently. Among the few works that deal with a distributed index for spatial data, the ones which are closer to our proposal are [2,4,11–15,22]. Hambrusch and Khokhar [11] 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. Kriakov et al. [15] 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.

In [12], the authors propose a balanced tree for peer- to-peer networks, called *BATON*, which ensures query answers in $O(\log N)$. BATON is not designed for multi-dimensional

data but a variant of the approach, termed VBI [13], explicitly targets distributed indexing of multidimensional points. Like SD-Tree, the VBI-tree is a dynamic binary tree with nodes at peers. With respect to the differences, first, VBI aims 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 is an open question how far one can apply this facet of VBI to spatial objects. Second, 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 Sect. 3, it aims at minimizing the spatial overlap. Finally, SD-Rtree is primarily designed for clusters, and focuses on load and storage balancing, as well as an efficient and fair utilization of cluster resources. VBI rather targets P2P networks, and addresses specific issues such as nodes arrival/departure.

The work presented in [22] proposes a distributed quadtree-based dedicated to spatial objects with extents. This index is based on the MX-CIF quadtree. The basic principle is to store an object in the deepest quadrant that fully contains its *mbb*. The authors study an adaptation of this simple scheme to the requirements of a distributed structure, regarding in particular load-balancing. Unlike all the other distributed index discussed so far, the distributed quadtree supports indexing of surfacic objects, and for this reason we use it as a competitor for the SD-Rtree. A longer description is given in Sect. 6.

The P-tree [4] 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 P-tree does not index spatial objects. In [19], an organization called P2P-Rtree is proposed. Unlike in SD-Rtree, P2P-Rtree is based on R+tree. However the basic space partitioning is static. The space is divided into regular blocks at the top level, statically subdivided into next levels *groups* that can be subdivided into subgroups, etc. Also unlike the SD-Rtree, there is no attempt to balance the index tree that can become unbalanced. In [18], the authors propose a framework for indexing and querying multidimensional data in P2P systems. Each peer indexes its data with a R*tree and closest peers are gathered into a cluster. This strategy implies a large overlap of the mbb since the internal data organization of the peer remains unchanged while merging. Moreover the queries are always addressed to super-peers.

In summary, to our knowledge, SD-Rtree is the first attempt to address simultaneously the following challenges: (i) index very large datasets of surfacic objects with a distributed structure, (ii) provide low message costs for routing insertion and search requests, (iii) balance the load on the servers. Note that we aim at balancing both the processing load *and* the space load, i.e., we would like both the query processing and the data storage to be distributed evenly across the network.

## 3 The SD-Rtree

The structure of the SD-Rtree is conceptually similar to that of the classical AVL tree [1], although the data organization principles are taken from the Rtree spatial containment relationship [10].

### 3.1 Kernel structure

**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*, refers to exactly two children,
- each routing node maintains 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.

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 shows a first example. 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 $\mathtt{a} = mbb(\mathtt{b} \cup \mathtt{c})$. 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

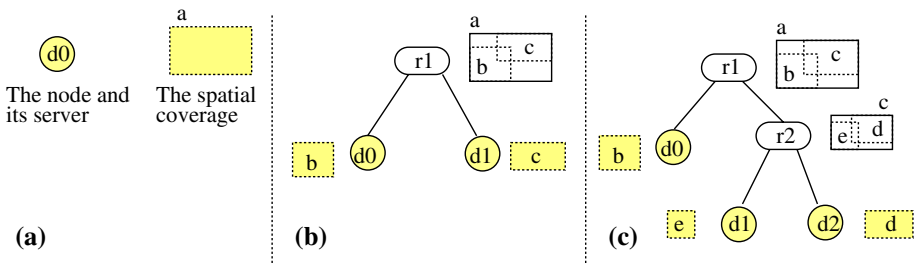



**Fig. 1** Basic features of the SD-Rtree

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 `Max(left.height, right.height)+1`. `OC`, the *overlapping coverage*, to be described in Sect. 3.4, 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

### 3.2 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 which sends back an 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 routes the query to other nodes if necessary.

An image is a collection of links, stored locally, and possibly organized as a local index. 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 client finally receives an IAM with $4n$ links. This information is later used to determine the server that can best satisfy an insertion or a search query.

### 3.3 Node splitting

When a server $S$ is overloaded by new insertions in its data repository, one of the following cases occurs:

1. a new server $S'$ can be added to the system, and $S$ transmits half of its data to $S'$;
2. no server is available, and $S$ must redistribute part of its data to its neighborhood.

The later case requires some sophisticated operations, and is presented in Sect. 5. The former case is the simpler one,

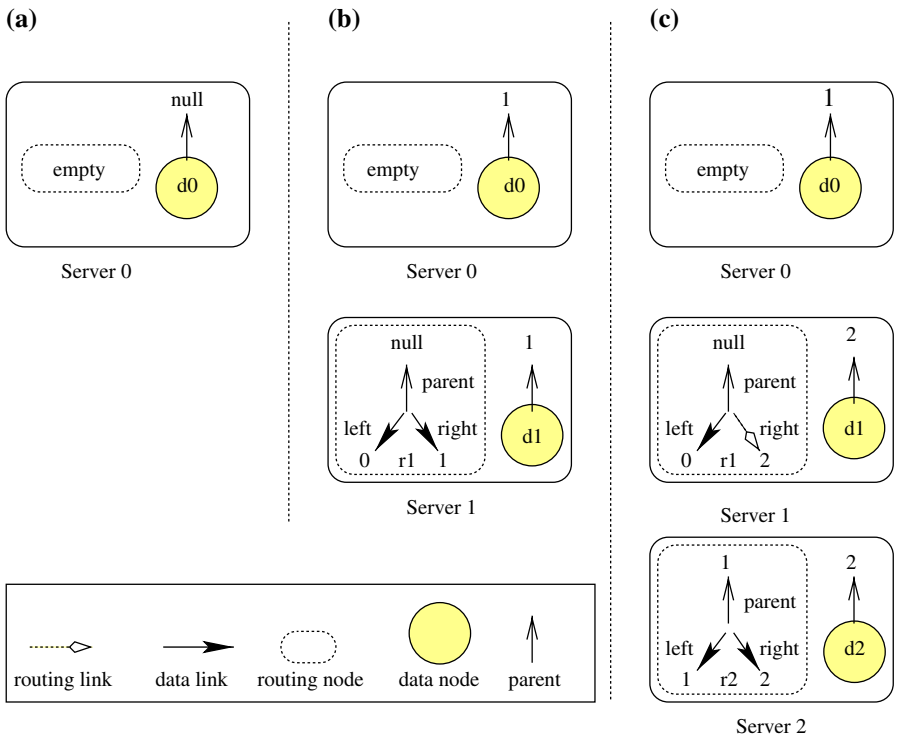



**Fig. 2** Split operations

and constitutes a major event with respect to the structure and organization of the index.

When a split is carried out, the data stored on $S$ is divided in two approximately equal subsets using a split algorithm similar to that of the classical Rtree [10]. 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 in Fig. 2 for the tree construction of Fig. 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 $mbb(left.\mathtt{dr}, right.\mathtt{dr})$),
- 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 [3,10]. 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 data node that splits.

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.

### 3.4 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.

#### *3.4.1 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 this strategy

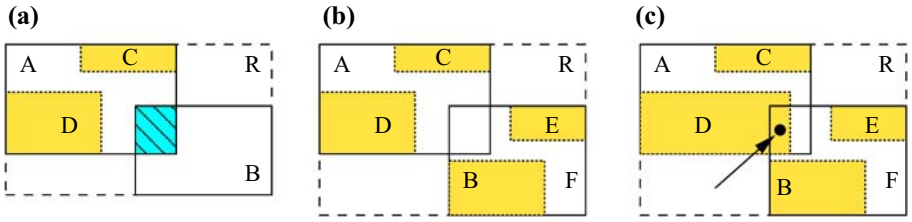



**Fig. 3** Overlapping coverage examples. **a** Initial tree, **b** Split of B, **c** Extension of D

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 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 $a$. A query sent to $d$ is forwarded to this sibling if and only if its argument concerns this part. If the intersection is empty, we simply ignore it. Second we trigger a maintenance operation only when this overlapping changes.

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 independently from $A$, as long as the rectangle intersection remains the same. Figure 3b 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 3c 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 $F$.

### 3.4.2 Storage and maintenance of the overlapping coverage

Formally, given a node $N$, let $anc(N) = \{N_1, N_2, \ldots, N_n\}$ be the set of ancestors of $N$. Each node $N_i \in anc(N)$ has two children. One is 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 Fig. 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 $[1:oc_1, 2:oc_2, \ldots, n:oc_n]$, 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 $[O.id : I.\text{dr}\cap O.\text{dr}]$ is added to the insertion message routed to the child $I$
2. a OC update message, containing the OC entry $[I.id : I.\text{dr}\cap O.\text{dr}]$, 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

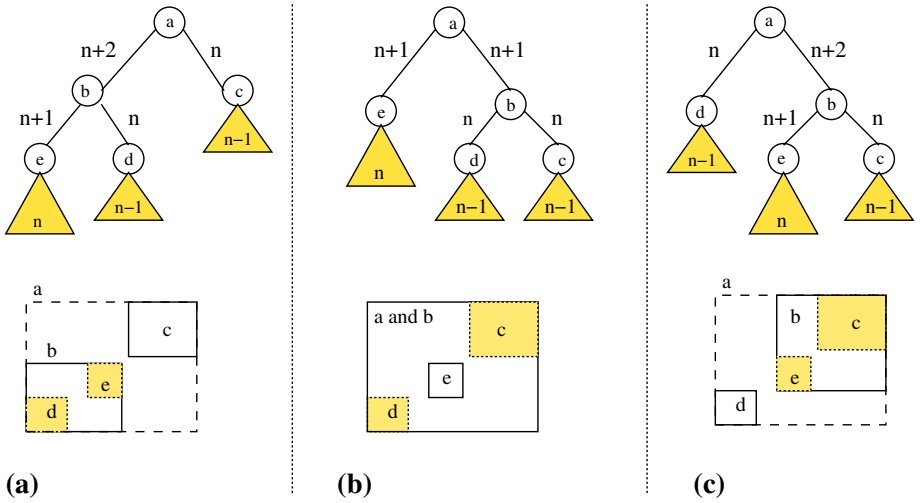



**Fig. 4** Rotation issues in a binary Rtree. **a** Unbalanced node a, **b** Straightforward rotation bad overlapping, **c** Incorrect rotation good overlapping

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** ($rect \cap N.\mathtt{dr}$ differs from $N.OC[id]$) $N.OC[id] := rect$
// Forward to subtree if the OC is non-empty
**if** ($N$ is not a leaf)
// Compute and send to the left subtree
**if** ($rect \cap N.left.\mathtt{dr} \neq \emptyset$)
UPDATEOC $(N.left.id, id, rect \cap N.left.\mathtt{dr})$
// Compute and send to the right subtree
**if** ($rect \cap N.right.\mathtt{dr} \neq \emptyset$) **then**
UPDATEOC $(N.right.id, id, rect \cap N.right.\mathtt{dr})$
**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, where $N$ is the number of servers. 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.

Regarding the number of enlargements, it suffices to note that no enlargement is necessary as soon as there exists a server whose directory rectangle fully contains the inserted object $o$. Our experiments confirm that the overlapping coverage remains stable when the embedding space is fully covered, making the cost of OC maintenance negligible.

### 3.5 Balancing

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. 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 Fig. 4b. 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 overlapping. Note that node e is the sibling of node b, yet it is fully included in the directory rectangle of b, which actually covers the same space at its parent, a. Finally this gives rise to an imbalance between the respective coverages of sibling nodes (see again nodes e and b in Fig. 4b). Figure 4c 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 exploits the fact that the rectangle containment relationship 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 2** A *rotation pattern* which is a subtree rooted at level $n$ of the form a(b(e(f,g),d),c) satisfying the following conditions for some $n \geq 1$:

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

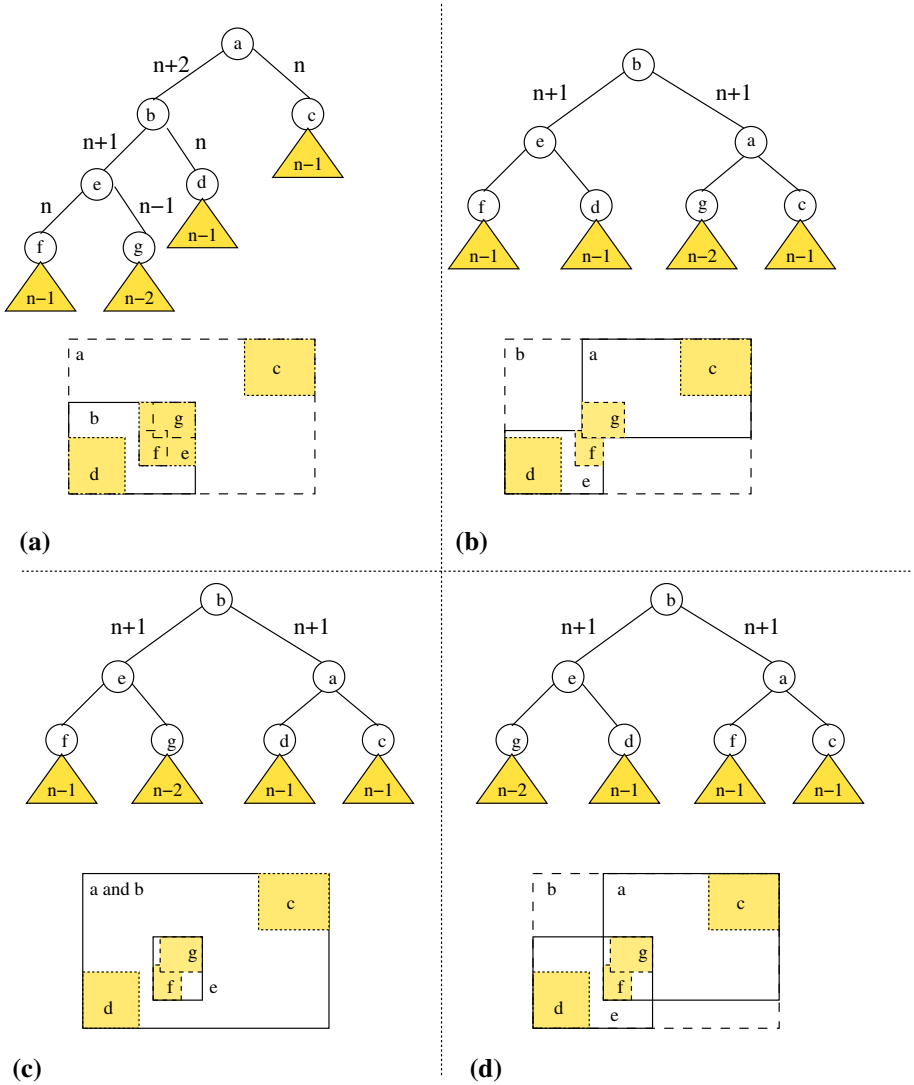



**Fig. 5** Balancing in the SD-Rtree. **a** The rotation pattern a(b(e(f,g), d), c), **b** Choice move(g)=b(e(f,d), a(g,c)), **c** Choice move(d)=b(e(f,g), a(d,c)), **d** Choice move(f)=b(e(g,d), a(f,c))

An example of rotation pattern is shown in Fig. 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:

**Lemma 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 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 2. □

The lemma shows that the management of unbalanced nodes always reduces to a balancing of a rotation pattern a(b(e(f,g),d),c), performed as follows:

1. b becomes the root of the reorganized subtree,
2. 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 be compared to the forced reinsertion strategy of the R*tree [3], 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 in Fig. 5. The choice move(g) (Fig. 5b) 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 split 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 node 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 $s$, its overlapping coverage with $e$ is $e.\mathtt{dr} \cap a.\mathtt{dr}$. 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 Fig. 5, assuming that the chosen rotation is Fig. 5b.

1. since $f.\mathtt{dr} \cap a.\mathtt{dr} \neq \emptyset$, a message UPDATEOC is sent to the children of f,
2. since $g.\mathtt{dr} \cap e.\mathtt{dr} \neq \emptyset$, 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 worst, in a dissemination to all the leaves. If a balancing occurs at the tree root, the whole tree may be affected, and a message sent to each server. 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 $N$.

## 4 Algorithms

We present now the algorithms for the insertion, deletion, point, window and $k$NN queries. Recall that all these operations rely on an *image* of the structure in order to remain as much as possible near the leaves in the tree, avoiding the root overloading. These operations also adjust the image through IAMs to better reflect the current 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 extend this knowledge and avoid to flood this server with insertions that must be forwarded later on.

### 4.1 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; a link whose directory rectangle contains $mbb$ 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 server $S$; else the link refers to a routing node and $C$ sends a message INSERT- IN- SUBTREE to $S$.

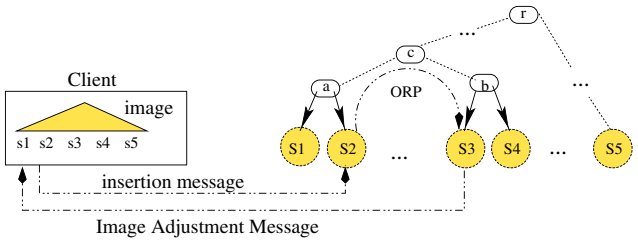



**Fig. 6** The insertion algorithm

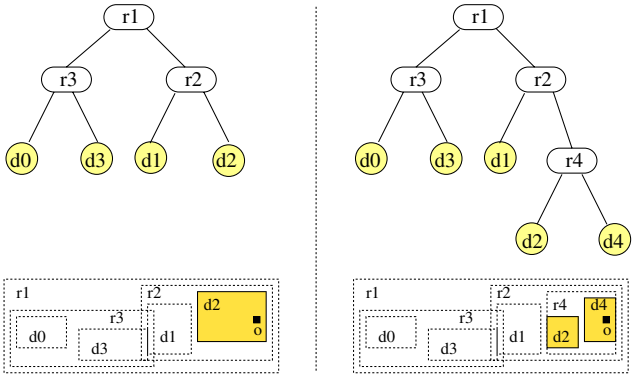



**Fig. 7** An example of out-of-range query, due to an outdated client image

– (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 not the message is forwarded to the parent until a satisfying subtree is found (in the worst case one reaches the root); otherwise 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 in Fig. 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 $d_2.\texttt{dr}$. 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 $C$.

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 join the structure. We expect 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 in Fig. 7. The image of the client $C$ (left part) contains a 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 $d_{S_2}.\texttt{dr}$. 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.

The insertion procedure HANDLEMESSAGE runs on a server (called "the current server") and takes as input the address of the client that requested the insertion, a message, and the object that must be inserted. Each message features an attribute `orp` which contains the out-of-range path, i.e., the list of servers which have been visited since the client issued the insertion message. Depending on the message type, the procedure carries out the appropriate action: a new message is forwarded to another server, or the object is inserted in the current server.

```
HANDLEMESSAGE (C : client, m : Message, o : object)
Input: an object that must be inserted
Output: the appropriate action, depending on the message's type
begin
  // Add the links stored on the current server to the message
  Add {data_link, left_link, right_link, local_link} to m.orp
  switch (m.type)
  begin
     case INSERT- IN- LEAF:
        if (o.mbb ⊆ current.dr)
           // o can be inserted in the current server
           Insert o in the current server repository
           Update the local OC
           // If the server capacity is exceeded: split
           if (current.nb_objects > current.capacity) then
              SPLITANDADJUST(current)
           endif
```

```
            Send the IAM message to C
        else
            // Out-of-range: forward to the parent
            m.type := INSERT- IN- SUBTREE
            Send m to the server parent
        endif
        break
    case INSERT- IN- SUBTREE:
        if (o.mbb ⊆ current.dr or current is the root)
            // Insert in the subtree rooted at current
            Go to the choice TOP- DOWN- TRAVERSAL
        else
            // Continue to forward the message bottom-up
            Send m to the parent server
        endif
        break
    case TOP- DOWN- TRAVERSAL:
        // Let l and r be the left and right links of the current
        // server's routing node. Apply the CHOOSESUBTREE
        // algorithm which returns the inner and outer nodes
        (I, O) := CHOOSESUBTREE (l.dr, r.dr, o.mbb)
        // Enlarge the directory rectangle of I
        I.mbb = I.mbb ∪ o.mbb
        if (I is a data link) then
            // The chosen child is the leaf where o must be inserted
            m.type = INSERT- IN- LEAF;
            Send m to the server I.id
        else
            // Compute and maintain the overlapping coverage
            if (I.mbb ∩ O.mbb has changed) then
                // Update the OC in the outer node
                UPDATEOC (O, I, I.mbb ∩ O.mbb)
                // Update the OC in message
                m.Rc := [O.id, I.mbb ∩ O.mbb]
            endif
            // Recursive insertion message
            Send m to the server I.id
        break
  end
end
```

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. The worst-case results thus in $3 \log N$ messages. However, if the image is reasonably accurate, the insertion is routed to the part of the tree which should host the inserted object. This results in a short out-of-range path with few messages. The strategy reduces the workload of the root that is accessed only for objects that fall outside the boundaries of the most-upper directory rectangles.

### 4.2 Deletion

Let $\tau$ be the minimal space occupancy, expressed as a percentage of a server's full capacity, A simple choice is to let $\tau = 50\%$, but choosing a lower value brings more flexibility to the structure. A server becomes *under-full* is its space occupancy falls below the $\tau$ threshold.

Deletion of an object $o$ is similar to that in an R-Tree [10]. First, the data server $S$ that stores $o$ is found, using the standard WINDOWQUERY procedure (described next). Three cases must then be considered:

(a) $S$ is not under-full after deletion of $o$;
(b) $S$ becomes under-full, but the space occupancy of its sibling is strictly above $\tau$;
(c) $S$ becomes under-full, and the space occupancy of its sibling equals $\tau$.

Case (a) is the simplest. Object $o$ is simply removed from $S$'s storage. It may be necessary to adjust its directory rectangle, with a recursive bottom-up propagation of the adjustment. In the worst case, all the directory rectangles on the path from $S$ to the root are adjusted. Each adjustment possibly modifies the OC of the nodes. This is handled by the UPDATEOC procedure presented in Sect. 3.4.

The analysis of the OC maintenance cost still holds. However, in the case of deletion, the update of the OC can be delayed, and carried out with the next insertion or query that affects this OC. This lazy strategy saves the calls to UPDATEOC at deletion time, but deteriorates the query performance because an outdated OC may lead to unnecessarily route a query to a server.

We consider now case (b). $S$ contacts its sibling $S'$ which sends to $S$ the object that leads to the maximal reduction of their common spatial overlap. $S$ and $S'$ both update their OC and overlapping coverage, and send an update message to their parent, as in case (a). We can reduce the probability of $S$ to become under-full again by relocating several objects in one shot.

Finally, case (c) leads to a merge of the two siblings. Objects of $S$ are relocated on $S'$, and $S$ returns to the free servers pool. Node $S'$ becomes the child of its grandparent. An adjustment of the height is propagated upward as necessary. This may require a rotation, performed by the technique presented in Sect. 3.5.

### 4.3 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 so 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 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 applies locally to the data repository. 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 then applied from $r$, and from the outer nodes in the overlapping coverage array $r.OC$ whose directory rectangle contains $P$.

In this way all the parts of the SD-Rtree which may contain $P$ are visited. The overlapping coverage information stored at each node usually avoids to traverse the whole path starting at the root as it will appear.

With an up-to-date client image, the target server is correct. The number of PQTRAVERSALs 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 worst, all the nodes must be visited. 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.

### 4.4 Window queries

Given a window $W$, the client searches its image for a link to a node that contains $W$. A query message is sent to the server that hosts the node. An out-of-range may occur because of image inaccuracy, in which case a bottom-up traversal is initiated. 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 the navigation to the appropriate data nodes. The algorithm is given below. The routine WQTRAVERSAL is the classical Rtree window query algorithm adapted to our 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 ⊈ 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 ∩ oc_i ≠ ∅) then
        WQTRAVERSAL (outer_node(i), W)
     endif
  end for
end
```

The number of data nodes which intersect $W$ depends on the size of $W$. Once a node that contains $W$ is found, the WQTRAVERSAL is broadcast towards these data nodes. The maximal length of each of these message paths is $O(\log N)$. The requests are forwarded in parallel, and result each in an IAM when a data node is finally reached.

### 4.5 $k$-NN queries

Our $k$-NN query algorithm is decomposed in two steps:

1. find a data node $N$ that contains the query point $P$ and evaluate *locally* the $k$-NN query;
2. from $N$, explore all the other data nodes which potentially contain an object closer to $P$ than those found locally.

The first step is a simple point-query. Assume that a data node $N$ containing $P$ is found. The $k$-NN query is evaluated locally, i.e., without any additional message, using a classical $k$-closest neighbors algorithm [20]. The initial list of neighbors, found locally, is stored into an ordered list $neighbors(P, k)$.

Obviously, some closest neighbors may also be located in other nodes, which may lead to an update of $neighbors(P, k)$.

A naïve approach to determine the servers to contact consists in a range query over the tree, the range $r$ being the distance between $P$ and the farthest object in $neighbors(P, k)$. However, with this strategy some messages are useless, as illustrated in Fig. 8. Assume node $N'$ is contacted first. It sends back to $N$ object $O'$. This reduces the maximal radius to $r'$. Let $r''$ be the distance between $P$ and the directory rectangle of $N''$. Since $r'' > r'$, there is clearly no hope to get any closer neighbor from $N''$.

A $k$-NN query generates messages that contact the nodes in an order which is estimated to deliver the quickest convergence towards the final result. This is likely to limit the number of servers to contact. We first recall some useful distance measures for $k$-NN queries [20], illustrated in Fig. 9.

Let $R$ be the bounding box of a set of rectangles. First, let min $Dist$ denote the minimal distance between a point $P$ and $R$. Second, note that each of $R$'s edges shares at least one point with one of its inner rectangles. Figure 9 shows the two extreme positions for a rectangle $r_1$ inside $R$ with respect to the edge closest to $P$. The maximal distance $D$ among $d_1$

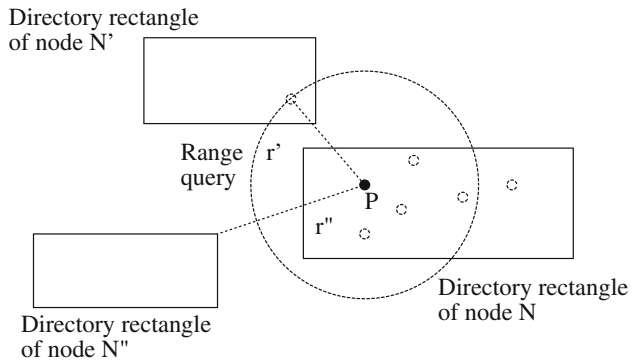



**Fig. 8** Improved $k$-NN query

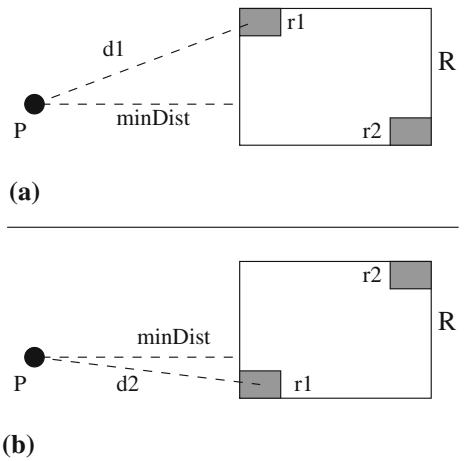



**Fig. 9** The minMaxDist. **a** first extreme case, **b** second extreme case

and $d_2$ guarantees there is an object within $R$ at a distance less or equal to $D$, denoted $\min\text{Max}Dist(P, R)$.

Now, given a list $neighbors(P, k)$, let $d_{\max}$ be the distance between $P$ and the last (farthest) object in $neighbors(P, k)$. To support our $k$-NN querying strategy, we extend the notion of overlapping coverage (OC) presented in Sect. 3, to keep information about *all* outer nodes, whether there is an overlap or not. So the overlapping coverage (OC) consists now of an array of the form $[1 : oc_1, 2 : oc_2, \ldots, n : oc_n]$, such that $oc_i$ is $outer_N(N_i).\texttt{dr}$. If $N'$ is a node in the overlapping coverage of $S$, the object closest to $P$ in $N'$ is at best at distance $\min Dist(P, N.\texttt{dr})$, and at worst at distance $\min\text{Max}Dist(P, N.\texttt{dr})$. Our algorithm relies on the two following observations:

(i) if $d_{\max} < \min Dist(P, N_i.\texttt{dr})$ we do not have to visit node $N_i$;
(ii) if $\min\text{Max}Dist(P, N_i.\texttt{dr}) < d_{\max}$ we must visit node $N_i$ because it contains an object closer to $P$ that the farthest object from $neighbors(P, k)$.

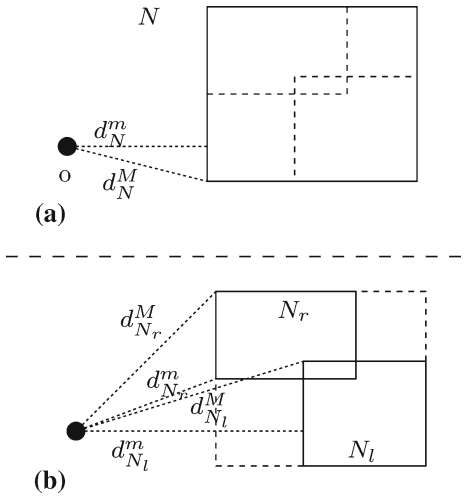



**Fig. 10** Updating $L_{mmd}$ and $L_{md}$. **a** $N$ is visited, $d_N^m$ and $d_N^M$ are removed from lists, **b** new distances for $N_r$ and $N_l$ inserted in lists

When processing a $k$-NN query, we compute the $\min Dist$ and $\min\text{Max}Dist$ distances from $P$ to all the mbb of the nodes in the OC list. We build a list $L_{md}$ (resp. $L_{mmd}$) of pairs $(id_i, d_i)$ where $id_i$ is the id of the node $N_i$ and $d_i$ the value of $\min Dist(P, N_i.\texttt{dr})$ (resp. $\min\text{Max}Dist(P, N_i.\texttt{dr})$). $L_{md}$ and $L_{mmd}$ are sorted on $d_i$ in ascending order. To determine which server must be contacted we simply compare the distance from $P$ to the farthest object in $neighbors(P, k)$, denoted $d_{\max}$, and the distances in $L_{md}$ and $L_{mmd}$.

The algorithm maintains an ordered list of the nodes to visit. The first node of the list is then contacted. If it is a data node, a local search is carried out, which possibly modifies $neighbor(P, k)$, as well as the lists $L_{md}$ and $L_{mmd}$. If it is a routing node, it computes for its two children the distances $\min Dist$ and $\min\text{Max}Dist$, and updates $L_{md}$ and $L_{mmd}$ accordingly. This is illustrated in Fig. 10 where the routing node $N$ has to be visited, regarding either its $\min\text{Max}Dist d_N^M$ or its $\min Dist\ d_N^m$ with query point $o$. When $N$ is contacted, it removes first $d_N^M$ (resp. $d_N^m$) from the list $L_{mmd}$ (resp. $L_{md}$). Since $N$ knows the mbb of its two children, $N_r$ and $N_l$, it is able to compute the $\min\text{Max}Dist\ d_{N_r}^M$ (resp. $d_{N_l}^M$) and $\min Dist d_{N_r}^m$ (resp. $d_{N_l}^m$) for $N_r$ (resp. $N_l$). It inserts each of these distances in the appropriate ordered list ($L_{mmd}$ or $L_{md}$) and visits w.r.t. these lists the next node in the list.

In both cases, the next server in the list is contacted in turn, until no possible improvement of the nearest neighbors can be obtained. The message transmitted from one server to another contains $P$, the list $neighbor(P, k)$ at the current step of the algorithm, and the lists $L_{md}$ and $L_{mmd}$. An heuristic consists in contacting first the server with the minimal $\min\text{Max}Dist$ since we know for sure that this modifies $neighbor(P, k)$ and reduces $d_{\max}$. When all the servers have been explored with respect to $\min\text{Max}Dist$, the remaining ones are contacted with respect to $\min Dist$.

KNN_EVAL_INIT $(P, k)$
**Input**: a point $P$, the number of nearest-neighbors $k$
**Output**: the list $neighbors(P, k)$, if no other server needs to be contacted
**begin**
  Find the node $N$ that contains $P$
  Perform locally the $k$-NN search;
  Store the result in $neighbors(P, k)$, sorted on the distance to $P$
  **for each** $(id_i, mbb_i) \in N.OC$ **do**
    Insert $(id_i, \min \text{Max}Dist(P, mbb_i))$ in $L_{mmd}$
    Insert $(id_i, \min Dist(P, mbb_i))$ in $L_{md}$
  **endfor**
  Let $d_{\max}$ be the distance between $P$ and $last(neighbor_k(P))$
  $(id, mmd) := first(L_{mmd})$; $(id', md) := first(L_{md})$;
  **if** $(mmd < d_{\max})$ **then**
    // Continue the $k$-NN search on the server that hosts $N_{id}$
    KNN_NEXTSERV $(N_{id}, P, neighbors(P, k), L_{mmd}, L_{md})$
  **else if** $(md < d_{\max})$ **then**
    // Continue the $k$-NN search on the server that hosts $N_{id'}$
    KNN_NEXTSERV $(N_{id'}, P, neighbors(P, k), L_{mmd}, L_{md})$
  **else**
    // The current list of neighbors can no longer be improved
    **return** $neighbors(P, k)$
  **endif**
**end**

KNN_NEXTSERV$(N, P, neighbors(P, k), L_{mmd}, L_{md})$
**Input**: a node $N$, the point query $P$, a list of neighbors, two lists of candidate servers
**Output**: the list $neighbors(P, k)$, if no other server needs to be contacted
**begin**
  Remove $N$ from $L_{mmd}$ and $L_{md}$ (if present)
  **if** ($N$ is a routing node) **then**
    Add $N$'s children to $L_{mmd}$ and $L_{md}$
  **else**
    // $N$ is a data node
    Update $neighbor(P, k)$ with objects from $N$
  **endif**
  // Forward the search as in KNN_EVAL_INIT
**end**

We expect thus to reduce the cost of $k$-NN algorithm. Neverthless, its worst case is still that of the simple range query approach.

## 5 Storage balancing

The insertion algorithm described in Sect. 3 requires a split each time a server is full. This adds systematically a new server to the network. The institution managing the network must be ready to allocate resources at any moment (in the case of a cluster of servers), or a free server has to be available (in the case of an unsupervised network). We present now a storage balancing scheme which allows a full node to redistribute overflowing data to lightly loaded servers whenever new storage resources cannot be allocated. Our technique is partially based on the $k$-NN querying algorithm.

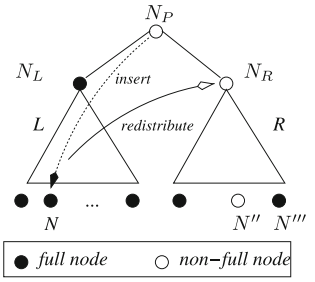



**Fig. 11** Overview of the load balancing strategy

### 5.1 Revisited insertion

Assume that an object $o$ must be inserted in a full node $N$. $N$ may require a redistribution balancing as follows. First, it contacts, by sending bottom-up messages, its nearest ancestor $N_P$ which has at least one non-full descendant, called the *pivot node*. If no such ancestor is found, the tree is full. We must then proceed to a node split of $N$, as presented previously. Otherwise, since $N_P$ is the nearest non-full ancestor, one of its two subtrees (say, $L$) contains only full nodes, including $N$. The other subtree (say, $R$) contains at least one non-full node (Fig. 11).

$N_P$ must move some objects from $L$ rooted at $N_L$ to $R$ rooted at $N_R$. For simplicity we describe the technique for a single object but it may be extended for redistributing larger sets.

We need additional information to detect the pivot node. We use two flags, $Full_{left}$ and $Full_{right}$, one for each child, added to the routing node type. Flag $Full_i$ is set to 1 when the subtree rooted at child $i$ is full. Flag values are maintained as follows. When a leaf becomes full, a message is sent to the server that hosts its parent node to set the corresponding flag to 1. This one may in turn, if its companion flag is already set to 1, alert its parent, and so on, possibly up to the root, i.e., $\log(n)$ messages (effectively, quite unfrequent since this implies a full tree). The flags are maintained similarly for deletions.

In order to minimize the overlap between the mbb of $L$ and $R$, we need to redistribute the object indexed by $L$ which is the *closest* to $R$. This can be achieved with a 1-NN query in $L$.

The query point $P$ is the centroid of the directory rectangle of $N_R$. With that choice the query will return an object producing a small enlargement of the overlapping coverage. The 1-NN algorithm is then initiated by $N_R$, which sets initially $L_{mmd}$ and $L_{md}$ to $N_L$ (we want our algorithm visits $N_L$), and $neighbors(P, 1)$ to $\emptyset$. $N_R$ re-inserts then the object obtained

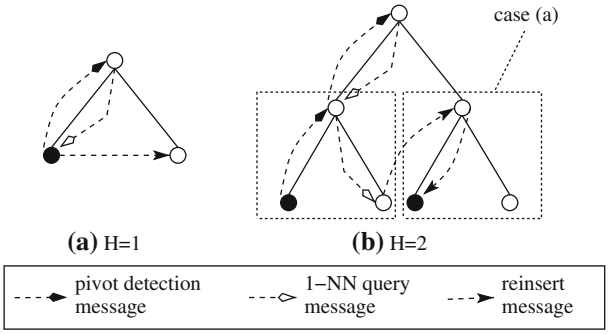



**Fig. 12** Redistributing data. **a** $H = 1$, **b** $H = 2$

as the result of this query in its subtree. This redistribution impacts the two subtrees $N_L$ and $N_R$ in two ways.

First, the reinserted object may have to be assigned to a full data node in the subtree rooted at $N_R$. This triggers another redistribution. However, since $N_R$ was marked as non-full, we know that the pivot node will be found *in* the subtree rooted at $N_R$. Consider for instance Fig. 11, and assume that a reinserted object must be put in the full node $N'''$. Since there exists at least one non-full node in $N_R$ ($N''$ for our example), the pivot node has to be a descendant of $N_R$.

Second the node $N$ where the initial object $o$ had to be put may become full in turn. So the redistribution algorithm has to be iterated. For the very same reasons, we know that the pivot node has to be a descendant of $N_L$. The load balancing stops when the chain of redistribution stops.

```
REDISTRIBUTE (N : node)
Input: a full node N
begin
  if (no ancestor N_P such that N_P.Full_right = 0) then
      Split(N)
  else
     while (N is full) do
        // Assume withour loss of generality that right nodes are
           outer nodes
        Find an ancestor N_P such that N_P.Full_right = 0
        Set N_i.Full_left to 1 on that path
        // Find the object to transfer
        Determine P the centroid of the dr of (N_R)
        L_md := L_mmd := N_L; neighbors(P, 1) := ∅
        o := result of k-NN query with these parameters;
        // Reinsert o in N_R.
        INSERT- IN- SUBTREE(o, N_R)
        Update flags in N_R if needed
     endwhile
  endif
end
```

Figure 12 illustrates the algorithm when the height of the pivot node is respectively 1(a) and 2(b). The algorithm analysis is as follows. When a node $N$ gets full, we need $H$ messages to find the pivot node $N_P$, where $H$ is the height of $N_P$. Then we need $H$ messages to find the object that must be moved from the left subtree in $N_P$, $N_L$, to the right one, $N_R$. The reinsertion of an object costs $H$ messages.

We possibly have to iterate the algorithm for the two insertions, respectively in $N_L$ and $N_R$ (see Fig. 12 for the example with $H = 2$). In both cases the pivot node's height is at most $H-1$. This yields the recursive formula: $cost(H)=3H+2\times cost(H-1)$. So finally, $cost(H) = \sum_{i=0}^{H-1} 2^i \times 3(H-i) = O(H \cdot 2^H)$. In the worst case, $H = \log_2 n$ (i.e., the pivot node is the root) and this results in $O(n \log_2 n)$ messages, where $n$ is the number of servers.

### 5.2 Discussion

The redistribution avoids to add new servers to the structure. However its cost is significant. Note that we may simply not have the choice, if no storage resource is available! In general, we can choose between two extreme strategies: one that performs systematically a split (and adds a new server) and one that delays any splits until all the servers are full. The latter clearly requires a lot of exchanges, whereas the former is not always possible. It seems convenient to provide a mean to adopt a trade-off between splits and redistributions.

The basic idea is to proceed only to "local" reorganization by limiting data redistribution to the close siblings of a full node, and not to the whole structure. As shown by the above analysis, the redistribution cost is in the worst case exponential in the height of the pivot node. By bounding this height, and thus the locality of the redistribution, we limit its cost, accepting more frequent splits, and less effective storage utilization. Let $\nu$ represent the maximal height for a pivot node. The REDISTRIBUTE($N$) algorithm is modified as follows:

- if $height(N_P) > \nu$ then split $N$
- else, apply REDISTRIBUTE($N$)

The choice $\nu = 0$ corresponds to a strategy where we split whenever a node is full, without any data redistribution. This minimizes the number of messages exchanged. An opposite choice is to set $\nu$ to $\infty$, allowing to choose any ancestor of a full node as a pivot, including the root, which results in a likely perfect storage utilization, but to a maximal number of messages, since a split occurs only when all the servers are full.

Parameter $\nu$ can easily be changed *dynamically*, which makes possible to adapt the behavior of the distributed tree to specific circumstances (i.e., a highly loaded period). Assume for instance that an initial pool of $m$ servers is available. $\nu$ can be set to 0, thereby minimizing the number of network exchanges. When the $m$ servers are in use, and a new one is required by a split, storage balancing can be enabled by setting $\nu$ to 1. Depending on the time necessary to allocate new servers to the pool, $\nu$ can be progressively raised, as the servers get full, until new storage space becomes available.

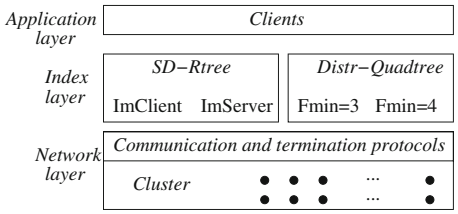



**Fig. 13** System architecture

Then it can be set to 0 again to disable the storage balancing option. So $\nu$ is a tuning parameter that brings flexibility since it allows to adapt the behavior of the system with respect to the available resources.

## 6 System architecture

We implemented in C++ a simulator which consists in three layers (Fig. 13). The *network layer* simulates a cluster of interconnected servers that exchange messages through a communication protocol and collects statistics on the server exchanges. The *index layer* implements spatial indices, and the *application layer* supports clients.

### 6.1 The network layer

We report experiments based on a communication protocol, where each server can directly send a message to any other server in the pool. Although the simulator supports P2P protocols (e.g., Chord) we did not investigate P2P simulations since we mostly target clusters of servers. Our results focus on the number of messages required by a structure to satisfy an operation, and not on the complexity of transmitting a message between two servers. The size of the messages always remains negligible (at most a few hundreds of bytes) and its impact is thus ignored. We also measure how the structures balance the space and processing loads.

The termination protocol lets a client issuing a point or window query figure out when to end the communication with the servers and return the result to the application. SDDS clients 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 an unreliable configuration such protocol may lead to a miss.

### 6.2 Implementation of spatial indices

We implemented two variants of the SD-Rtree, IMCLIENT and IMSERVER, according to the description given in the previous sections.

**IMCLIENT**. Each client builds an image, and maintains it according to the IAMs. We recall that the servers forward the operations through their routing nodes. We believe this proposal particularly fits applications where clients dispose of some memory and computation abilities (e.g., computers or laptops).

**IMSERVER**. There is an image on each server component, instead of on the client one. This corresponds to an architecture for many light-memory clients (e.g., PDA) addressing queries to a cluster of interconnected servers. We simulate it by choosing randomly, for each request (insertion or query) a contact server playing the role of a services provider.

In addition, we occasionally compare the performance of our two variants with a basic implementation that does not use an image at all. Each request is forwarded to the server that maintains the root node. The top-down traversal of the tree to reach the adequate server follows. BASIC is only implemented for comparison purposes, since the high load of the root levels makes it unsuitable as a SDDS.

We compare the SD-Rtree with the only other dynamic distributed structure that, to our knowledge, supports indexing of surfacic objects approximated by their *mbb*, namely the distributed quadtree recently proposed in [22]. Each object is associated to the deepest quadrant that entirely covers its bounding box. This is illustrated in Fig. 14, which shows both the space partition (left part) and the quadtree organization (right part). The deepest quadrant that fully contains object $X$ for instance is $A$. The deepest quadrant that contains $Y$ is $A.C.A$.

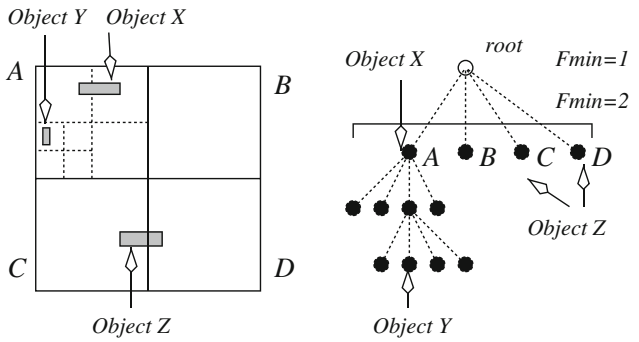



**Fig. 14** The distributed quadtree

Each server stores exactly one node, and is connected to its four sub-nodes which can be reached in one message. Compared to [22], we avoid the DHT indirection that would require $O(\log n)$ additional routing messages. This makes the structure behavior rather different, since nodes are randomly distributed by the DHT on the P2P servers in [22], whereas we use a one-to-one mapping between nodes (or control points") and servers in our implementation. This may lead to a non-efficient storage usage. It is unclear how grouping nodes for better space utilization could be achieved without hindering load balancing. Unlike SD-Rtree, there is no distinction between routing and data nodes, since objects may be stored in non-leaf nodes.

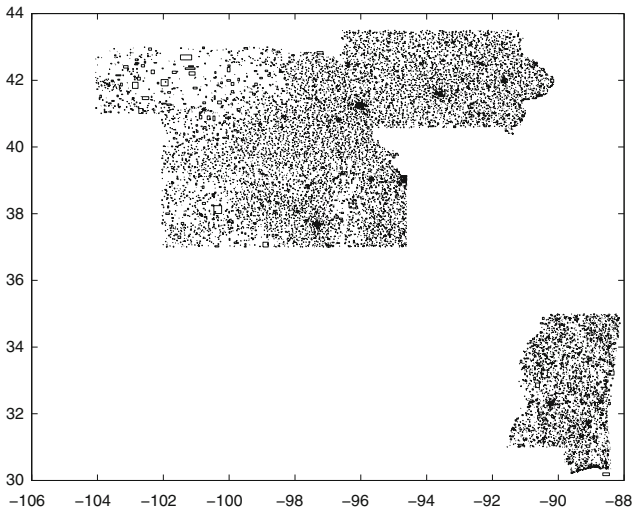



**Fig. 15** MBB of Tiger Census data concerning Iowa, Kansas, Missouri and Nebraska

The authors of [22] introduce the *fundamental minimum level*, $F_{\min}$, to improve load balancing among the servers. The rule is that all objects must be inserted in servers/nodes at level $l \geq F_{\min}$. Figure 14 shows an example with $F_{\min} = 2$ ($F_{\min} = 1$ sets the root as the minimum insertion level). Objects $X$ and $Y$ can be stored as before, because their respective storage levels are 2 and 4. The rule prevents object $Z$ from being stored at the root level, as it should in a standard quadtree. It must be divided in two parts, assigned respectively to the subtrees rooted at $C$ and $D$.

All the operations of the distributed quadtree can be routed directly to the servers at level $F_{\min}$. Since the space covered by each of these servers is fixed, the client that requests an operation only needs to know the server address associated to each cell in the space partition at level $F_{\min}$. The higher the value of $F_{\min}$, the better the balance achieved with this mechanism. On the other hand, $4^{F_{\min}-1}$ servers must be initially assigned to the structure. We use in our experiments $F_{\min} = 3$ ($4^2 = 16$ servers at level $F_{\min}$) and $F_{\min} = 4$ ($4^3 = 64$ servers).

### 6.3 Datasets and performance indicators

We produced synthetic datasets with the GSTD generator [23]. We also use the MBRs of 556,696 census blocks (polygons) of Iowa, Kansas, Missouri and Nebraska based on Tiger Census's data [9], represented in Fig. 15. This dataset is highly skewed.

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. For these experiments we fixed the capacity of each server to 3,000 objects, unless stated otherwise.

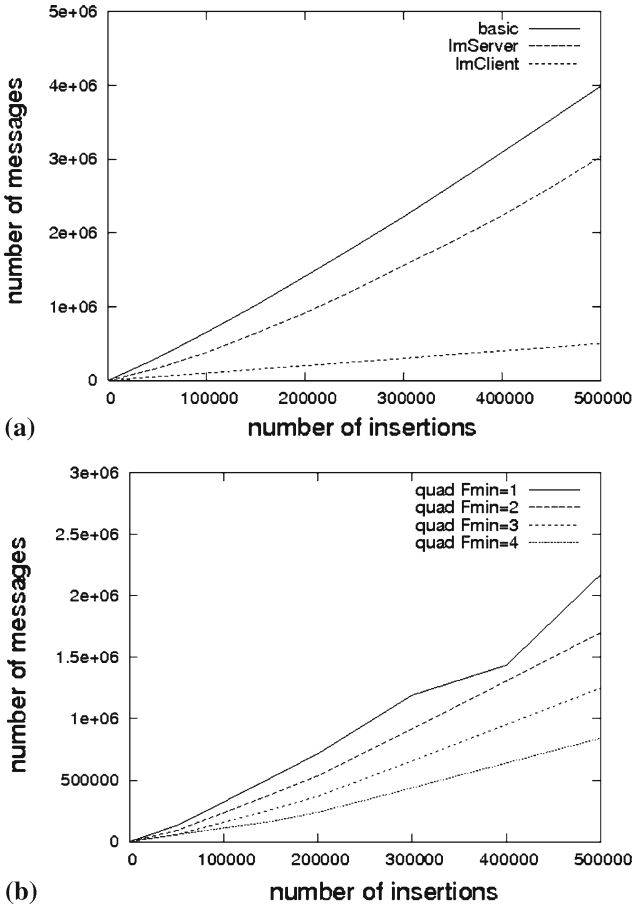



**Fig. 16** Insertions of uniformly distributed data. **a** SD-Rtree variants, **b** Quadtree structure with various $F_{\min}$

## 7 Experimental results

### 7.1 Insertions

The results reported in this section study the behavior after an initialization of the indices with 50,000 objects. This avoids partially the measures distortion due to the cost of the initialization step which affects primarily the first servers. These initial insertions follow the specific distribution of the dataset.

#### *7.1.1 Comparison of insertion costs*

Starting with the SD-Rtree, Fig. 16a shows the total number of messages for insertions of objects following a uniform distribution. We use BASIC (which does not use images) as a yardstick to measure the improvement brought by images. The cost of each insertion for the BASIC variant is approximately the length of a path from the root to the leaf, 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. It needs 6 messages on average, thus a 25% gain. 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.

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 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.

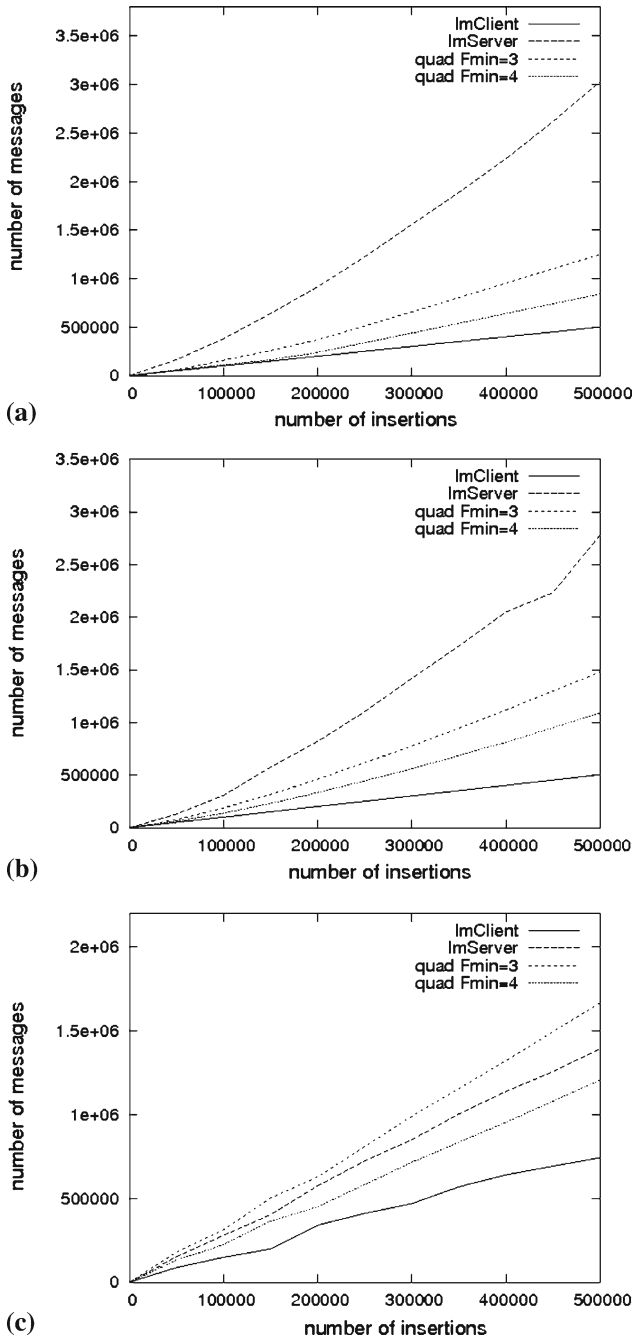



**Fig. 17** Comparison SD-Rtree/Quadtree for insertions **a** Uniformly distributed data, **b** Skewed data, **c** Real data

We turn now to the quadtree. Figure 16b compares the number of insertions with $F_{\min}$ ranging from 1 to 4. Recall that $F_{\min} = 1$ denotes a standard quadtree where all queries are sent first to the root, whereas $F_{\min} = 4$ corresponds to 64 servers, each acting as root of a local quadtree covering a cell of the space decomposition at level 4. Lower values for $F_{\min}$ lead to a higher number of messages because an insertion starts at level $F_{\min}$ and needs to access more levels.

Table 1 compares the SD-Rtree (variants ImClient and ImServer) with the quadtree approach (for $F_{\min}$ equal to 3 or 4). Figure 17a redraws four curves from Fig. 16 for easier analysis. It shows the total number of messages for insertions, with various distributions. ImClient performs better than the quadtree for all the $F_{\min}$ values. The quadtree still requires the traversal of a partial top-down path in the tree, while ImClient directly contacts the (likely correct) server. The quadtree behaves better than ImServer whose performance is affected by outdated images, although the ratio depends on the dataset. With uniform data the quadtree needs up to 3.6 less messages than ImServer. For skewed and real data, the results are closer because the SD-Rtree is a balanced tree which better adapts to the data distribution.

Figure 17b with skewed distribution and Fig. 17c with real data confirm the best performances of ImClient while the quadtree approach with $F_{\min} = 4$ still outperforms the ImServer. Both approaches are more costly for skewed data, although the number of messages increases more slowly for the SD-Rtree. For instance, in Fig. 17b, the quadtree with $F_{\min} = 4$ requires 120% more messages than ImClient, and 70% with uniform data. ImServer costs 250% more messages than the quadtree with $F_{\min}$ set to 4 for uniform data, but 150% with skewed data. Again this is explained by the data-driven approach of the SD-Rtree.

ImServer exhibits a lower number of messages than the basic tree without image. First a few forwarding messages are sufficient if the contacted node has been split, in which case the correct server can be found locally. Second 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

**Table 1** Average number of messages per insertion for variants of SD-Rtree and Quadtree with various distributions

| Distribution | nb objects | SD-Rtree | | | Quadtree | |
|---|---|---|---|---|---|---|
| | | BASIC | IMSERVER | IMCLIENT | $F_{\min} = 3$ | $F_{\min} = 4$ |
| Uniform | 100,000 | 6.5 | 3.8 | 1.0 | 1.6 | 1.1 |
| | 200,000 | 7.1 | 4.6 | 1.0 | 1.8 | 1.2 |
| | 300,000 | 7.4 | 5.2 | 1.0 | 2.2 | 1.5 |
| | 400,000 | 7.7 | 5.6 | 1.0 | 2.4 | 1.6 |
| | 500,000 | 8.0 | 6.1 | 1.0 | 2.5 | 1.7 |
| Skewed | 100,000 | 6.9 | 3.1 | 1.0 | 1.9 | 1.4 |
| | 200,000 | 7.3 | 4.1 | 1.0 | 2.3 | 1.7 |
| | 300,000 | 7.7 | 4.7 | 1.0 | 2.6 | 1.9 |
| | 400,000 | 8.1 | 5.1 | 1.0 | 2.8 | 2.0 |
| | 500,000 | 8.3 | 5.5 | 1.0 | 3.0 | 2.2 |
| Real | 100,000 | 7.2 | 2.8 | 1.5 | 3.2 | 2.2 |
| | 200,000 | 7.8 | 2.9 | 1.7 | 3.1 | 2.3 |
| | 300,000 | 8.3 | 2.8 | 1.6 | 3.3 | 2.4 |
| | 400,000 | 8.6 | 2.8 | 1.6 | 3.3 | 2.4 |
| | 500,000 | 8.9 | 2.8 | 1.5 | 3.3 | 2.4 |

effective by reducing in most cases the navigation in the SD-Rtree to a local subtree. Table 1 shows for instance that, for a 200,000 objects dataset (corresponding to a tree of height 7 with 125 servers), only 4 additional messages are necessary on average to reach the correct server (2 bottom-up, and 2 top-down messages).

Finally, the average number of messages for IMCLIENT does not 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 tree height is slightly larger.

### *7.1.2 Load balancing*

The gain obtained with images is significant. Even more importantly this evenly spreads the load on the servers. Figure 18a analyzes the distribution of messages with respect to the position of a node in the tree for the different variants of the SD-Rtree. Looking at the BASIC variant, still used for comparison purposes, the servers that store the root or other high-level internal nodes are overloaded. 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 receives 6.38%. Figure 18

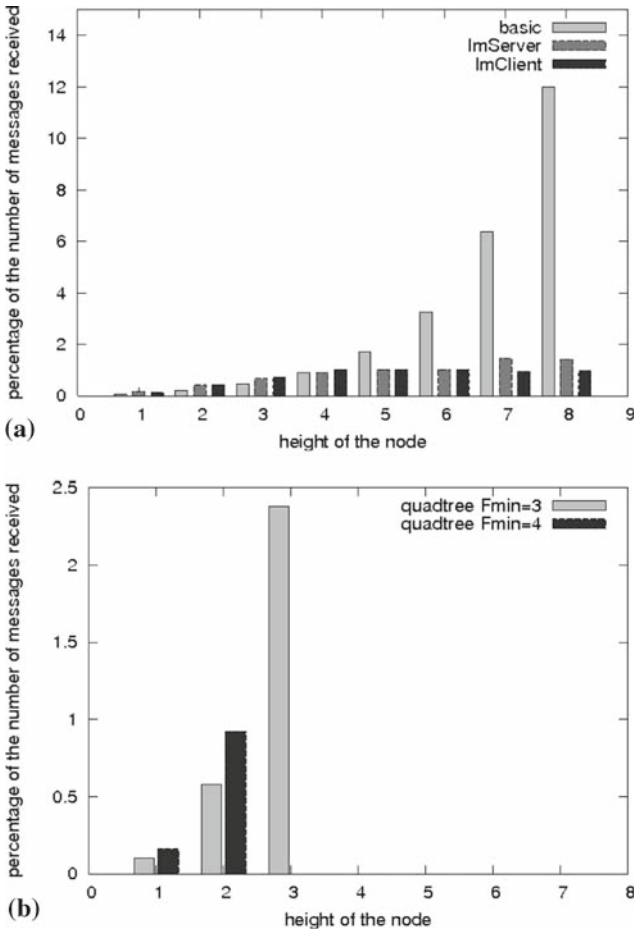



**Fig. 18** Messages distribution for insertions. **a** Uniformly distributed data with SD-Rtree, **b** Uniformly distributed data with Quadtree

shows that maintaining an image (either with ImServer or ImClient) not only allows to save messages, but also distributes almost uniformly 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. With ImClient, since a client acquires quickly a complete image, it contacts in most cases the correct server.

Regarding the distributed quadtree, we observe that the impact of the fundamental level is not sufficient to uniformly balance the load. Figure 18b illustrate the unbalanced distribution of messages for uniform data. For instance, with $F_{\min}$ set to 3, a node at level $i$ receives 4 times more messages than a node at level $i-1$. With $F_{\min}$ set to 4, this factor decreases to 2.5.

Improving the load balance is hindered by the fact that the top-down traversal method remains the basic principle of the quadtree insertion algorithm. Therefore the unbalance effect still holds for the levels below $F_{\min}$. Higher values for $F_{\min}$ improve the load balancing but require more object splits, so more duplicates and more servers. This is an important difference with our solution based on images, which does not impose the level of the tree where insertions are routed. Rather, an accurate image allows insertions to be sent directly at the leaf level. Since, with our structure, each server acts as a leaf, we potentially reach a near-perfect balance.

#### *7.1.3 Storage utilization*

Table 2 shows the storage utilization statistics. With a uniform distribution, the SD-Rtree grows regularly and its height follows the rule $2^{\text{height}-1} < N \leq 2^{\text{height}}$. The storage utilization factor is around 70%, i.e., around the well-known typical ln 2 value. The quadtree-based approach needs twice more servers for the same number of insertions, so an average load factor of 45% for uniform data, and 30% for real data. The SD-Rtree benefits from the mapping of its structure to the set of available servers. This mapping defines a one-to-one association between the servers and the leaves of the binary Rtree, thereby ensuring that the structure exploits the storage capacity of each server. There is no such guarantee with the quadtree because its organization is quite different. The assignment of objects to servers is not based on storage utilization, but on spatial relationships between the quadrants and the objects *mbb*. A split creates four (4) new servers, with an initial low storage utilization. Consequently, SD Rtree allows to index more data for a given number of servers.

Another factor that explains the lower storage efficiency, although less important, is the duplication of objects that cannot be fully contained in a cell at a level greater than $F_{\min}$, or that must be pushed down the tree when a server is full. Table 3 reports the storage redundancy with $F_{\min} > 1$, due to subdivisions of objects. It can reach up to almost 10% of the number of objects with $F_{\min}$ set to 4.

#### *7.1.4 SD-Rtree rotations and image maintenance*

We now examine some results which are specific to the SD-Rtree, starting with the cost of rotations. There is a low overhead due to the balancing of the distributed tree. Figure 19a shows the average number of additional messages required to balancing the tree depending on the number of insertions. With our 3,000-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 1,000 insertions.

In Fig. 19a, the tree grows uniformly due to the uniform distribution, and because of the large capacity of the servers. This explains the absence of balancing operations.

Experiments with skewed data (Fig. 19b) 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 balancing the tree (310). Nonetheless, on average, only 1 message per 500 insertions is necessary for maintaining the tree. With real data (Fig. 19c) more messages are required for maintaining the height and for rotations (1,000 adjust and rotate messages). But here again this overhead remains quite low compared to the high number of insertions (i.e., 2 messages now for every 500 insertions).

Figure 20 illustrates the convergence speed of the client image. With less that 200 query messages for 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 behavior.

**Table 2** Number of nodes, average storage load and tree height for quadtree and SD-Rtree

| Disturbution | Objects | Quadtree $F_{min} = 3$ | | | Quadtree $F_{min} = 4$ | | | SD-Rtree (all variants) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Servers | Load (%) | Height | Servers | Load (%) | Height | Servers | Load (%) | Height |
| Uniform | 150,000 | 85 | 58.8 | 4 | 85 | 39.2 | 4 | 64 | 78.1 | 6 |
| | 250,000 | 341 | 24.4 | 5 | 341 | 24.4 | 5 | 125 | 66.7 | 7 |
| | 350,000 | 341 | 34.2 | 5 | 341 | 34.2 | 5 | 166 | 70.3 | 8 |
| | 450,000 | 341 | 44.0 | 5 | 341 | 44.0 | 5 | 233 | 64.4 | 8 |
| | 550,000 | 341 | 53.8 | 5 | 341 | 53.8 | 5 | 243 | 75.4 | 8 |
| Skewed | 150,000 | 149 | 33.6 | 7 | 169 | 29.6 | 7 | 73 | 68.5 | 7 |
| | 250,000 | 229 | 36.4 | 7 | 245 | 34.1 | 7 | 122 | 68.3 | 8 |
| | 350,000 | 329 | 35.5 | 8 | 349 | 33.4 | 8 | 176 | 66.3 | 9 |
| | 450,000 | 425 | 33.6 | 8 | 453 | 33.1 | 8 | 224 | 66.7 | 9 |
| | 550,000 | 525 | 34.9 | 9 | 553 | 33.1 | 9 | 267 | 68.7 | 10 |
| Real | 150,000 | 165 | 30.3 | 8 | 225 | 22.2 | 8 | 78 | 64.1 | 8 |
| | 250,000 | 277 | 30.1 | 8 | 329 | 25.3 | 8 | 126 | 66.1 | 9 |
| | 350,000 | 377 | 30.9 | 8 | 417 | 28.0 | 8 | 175 | 66.7 | 9 |
| | 450,000 | 475 | 31.6 | 8 | 505 | 29.7 | 8 | 225 | 66.7 | 9 |
| | 550,000 | 565 | 32.4 | 9 | 601 | 30.1 | 9 | 281 | 65.2 | 10 |

**Table 3** Average number of duplicates with various $F_{min}$ for the distributed Quadtree

| Objects | $F_{min} = 1$ | $F_{min} = 2$ | $F_{min} = 3$ | $F_{min} = 4$ |
|---|---|---|---|---|
| 100,000 | 0 | 1,642 | 4,468 | 12,557 |
| 200,000 | 3,400 | 4,408 | 8,767 | 19,300 |
| 300,000 | 7,817 | 8,644 | 12,970 | 25,598 |
| 400,000 | 12,236 | 12,922 | 17,499 | 32,293 |
| 500,000 | 16,714 | 17,739 | 25,683 | 44,310 |

### 7.2 Queries

The following experiments report the results of point and window queries over a dataset of 250,000 objects, uniformly distributed. We then evaluate 3,000 queries.

Figure 21a compares the cost of point and window queries for all variants. The quadtree-based index provides better results with $F_{min} = 4$. Indeed, when the $F_{min}$ level is close to the leaves, the tree traversal requires a fewer messages. The quadtree also benefits from its non-overlapping structure. For point queries, a single path is followed from $F_{min}$ to the leaf that contains the objects of interest to the query. With $F_{min} = 3$, the length and the number of paths increase, and eventually leads to worse performances than IMCLIENT. Recall however that this gain is obtained at the cost of twice more servers.

Focusing on the SD-Rtree behavior, the role of the image can be analyzed as follows. Without image, and assuming no overlapping, the number of messages to retrieve the answer is expected to be 7, the height of the tree. The overlap costs here 2 additional messages on average. As expected for the SD-Rtree variants, which rely on an image, 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). 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 a basic implementation that does not use an image.

Figure 22 shows the ratio of correct matches when an image is used for a point query. It illustrates further, in the wake of Fig. 21, the convergence of the image toward the actual structure of the distributed tree. With IMSERVER, after 1,500 (resp. 2,500) queries, every 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 thus the results of Fig. 21. The results are especially good for IMCLIENT when the number of queries is low.

It is important to note that the SD-Rtree is a structure that improves it efficiency as each of its component learns about the overall organization. Figure 22 shows that, asymptotically, a client that submitted enough queries and got back enough information will reach a state where it almost always directly contacts the correct server. In other words,

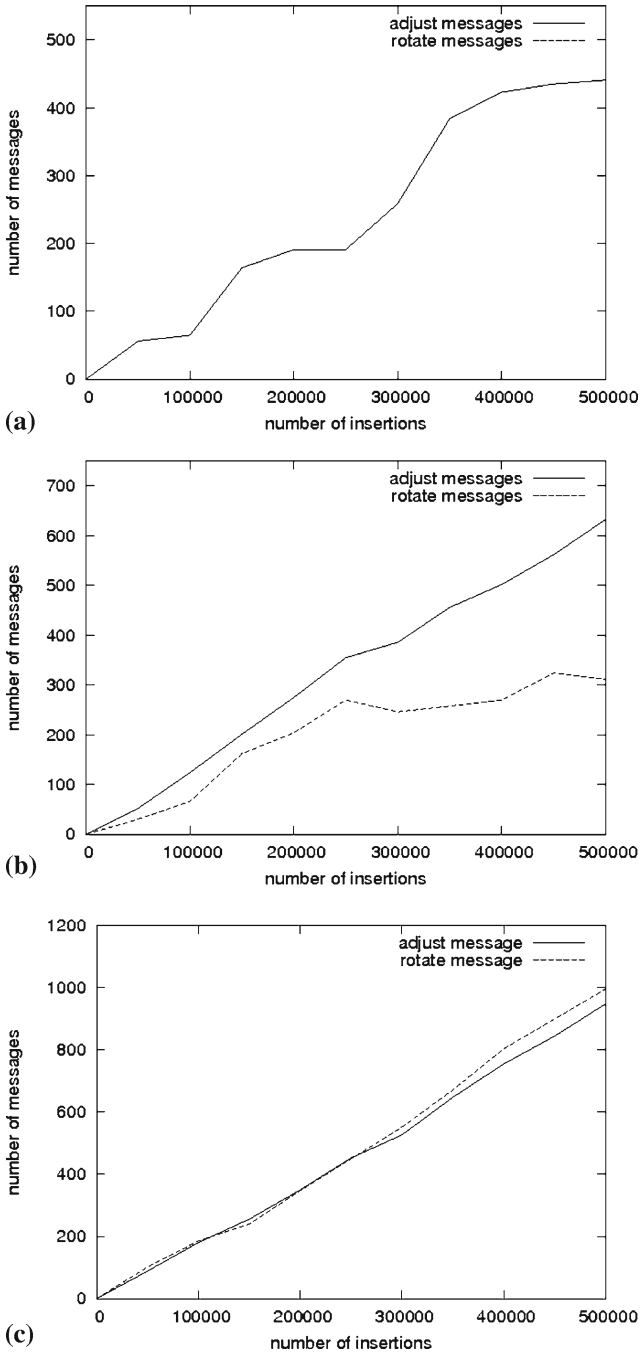



**Fig. 19** Adjustment and rotation messages. **a** Uniformly distributed data, **b** Skewed data, **c** Real data

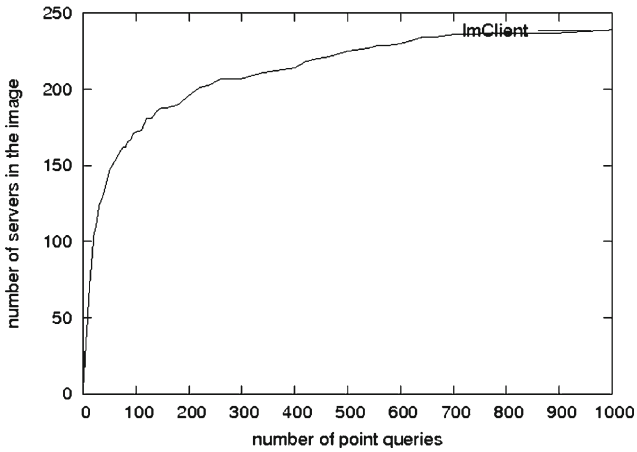



**Fig. 20** Convergence of the client image

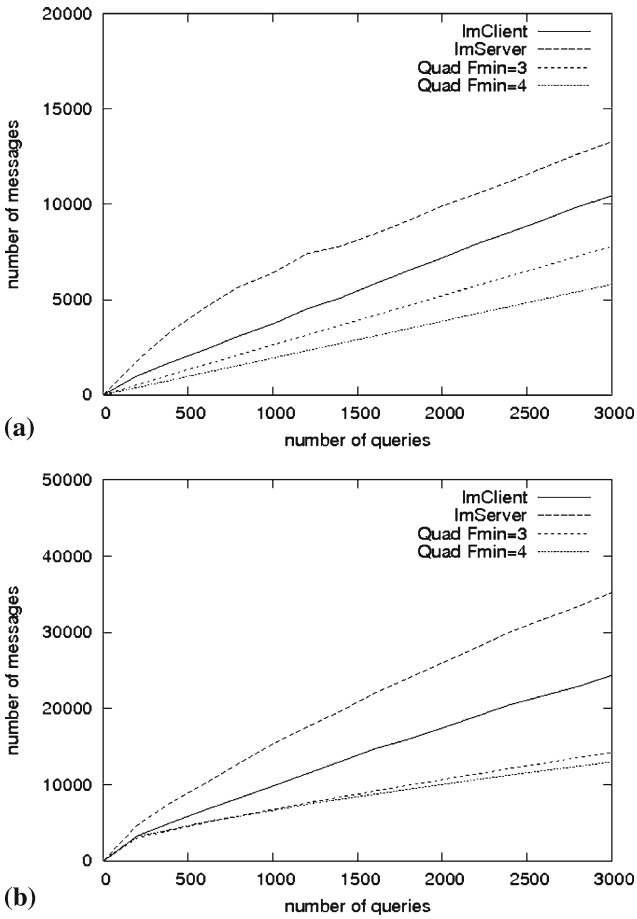



**Fig. 21** Query answering cost. **a** Point queries, **b** Window queries

the number of messages per point query decreases towards about 1. Quadtree does not present similar learning behavior.

Figure 21 b summarizes the window queries experiments. The extent of the query rectangle on each axis is a randomly chosen value, which can be up to 10% of the space extent. The cost of a window query tends to be about twice that of a point query for all variants. The higher cost of a window query is due to the overlap between the window and the *dr* of the servers. The figure also shows the benefit of images for window queries. The performance of ImClient tends to be a half that of Basic (not shown in Fig. 21). The image acquisition is faster, incidentally, for window queries than

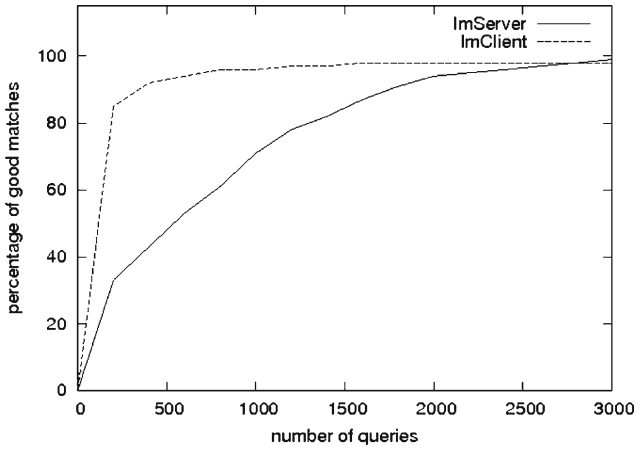



**Fig. 22** Good matches for point queries

for the point queries, since the former contact more servers.

Here again the quadtree-based proposal presents better result for window queries. With $F_{\min} = 4$, and a quadtree height of 5, only two servers need to be contacted when a window argument is fully contained in one the the quadtree leaves quadrant. The average cost is actually slightly higher (3.6 messages), because several paths must be followed for windows that intersect quadrants boundaries. The same comments as above hold: the price to pay for this efficiency is a much larger number of servers and object storage redundancy. Moreover the efficiency of the SD-Tree with the number of queries, as for point queries, improve whereas the quadtree performance remain identical.

Finally, Fig. 23a confirms that using an image achieves a satisfying load balancing, for reasons already given for the insertion algorithm. The message distribution is unbalanced for quadtrees since nodes at level $l$ receive on average 2–4 times more messages, depending on the value of $F_{\min}$, than nodes at level $l - 1$ (Fig. 23b,c). These results suggest that SD-Rtree should better support query hot spots because it tends to use more servers to cover dense regions.

## 7.3 Storage balancing

We now report several experiments that evaluate the performance of our dynamic storage balancing. Since the technique is specific to the SD-Rtree, from now on we do not compare with the quadtree. We use three data sets (uniform and skewed generated data, and our real data provided by Tiger) composed of 556,696 rectangles. In order to emphasize the impact of storage limitations and the effect of balancing operations, we fix the capacity of each server to 2,000 objects for these experiments.

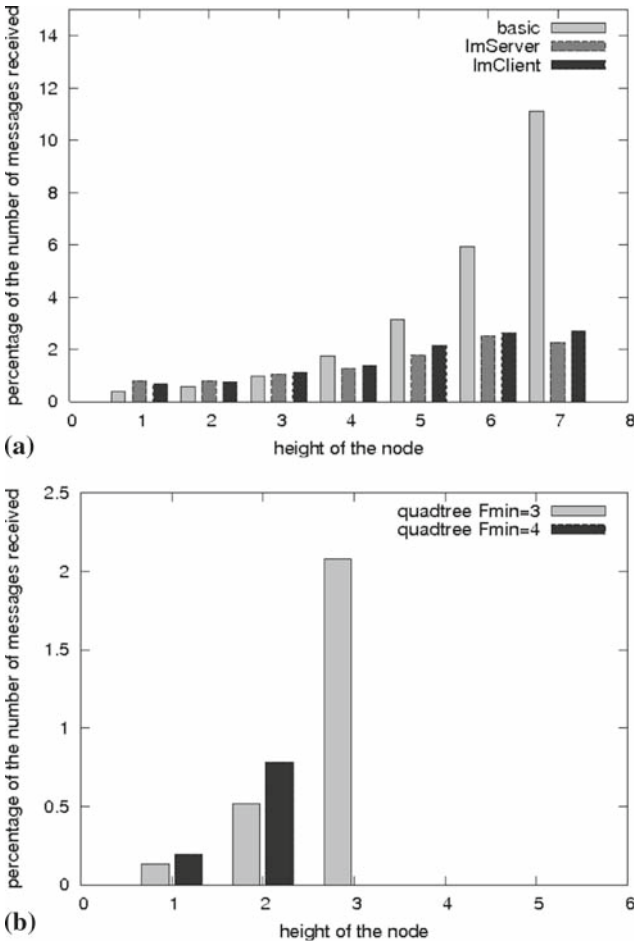



**Fig. 23** Messages distribution for point queries. **a** Uniformly distributed data with SD-Rtree, **b** Uniformly distributed data with Quadtree

### 7.3.1 Redistribution cost

A first experiment stresses the impact of the maximal pivot height allowed for the data redistribution. First note that with our setting the height of the tree is 10 for all distributions. Figure 24a shows that our redistribution algorithm highly reduces the number of servers requested, even for small values of the maximal height $\nu$ of a pivot node. For instance with a uniform distribution, the number of servers without redistribution is 440. With $\nu$ set to 1, this number falls to 350, so a gain of 21% of the required resources. With higher values for $\nu$, the number of servers can reach 291, so a gain of 34%. We observe similar results with other distributions.

The skewed distribution leads to a lower number of servers if we do not use redistribution, compared to other datasets. The reason is that insertions are concentrated on a specific part of the indexing area, hence mostly concern a subset of the pool. These nodes fill in up to their capacity, then split and, since they still cover the dense insertion area, remain subject to high insertions load. With uniform distribution,

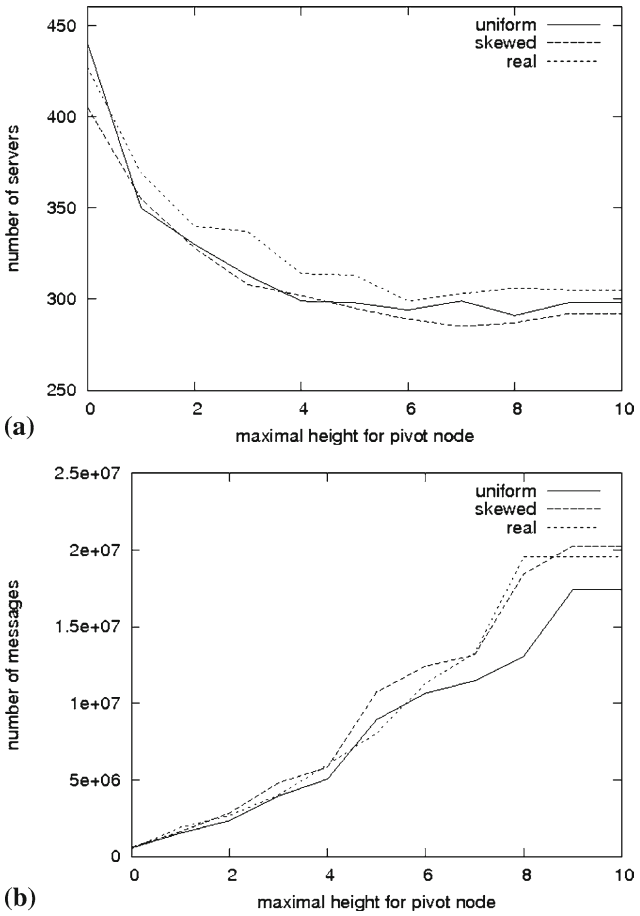



**Fig. 24** Impact of the maximal height for the pivot node. **a** Number of servers, **b** Number of messages

each node newly created is initially half-empty, and its probability to receive new insertion requests is similar to that of the other nodes. This leads to a lower average space occupancy (63%) than with skewed data (69%), and therefore so a higher number of servers (Fig. 25).

Using the redistribution algorithm, one achieves a high fill-in rate for the servers, i.e., up to 96% with uniform distribution, 98% with skewed distribution, and 93% with real data (Fig. 25). This value is already reached with a medium value of $\nu$ like 4 or 5. With $\nu$ set to 1, the improvement is still noteworthy, e.g., 79, 78 and 75% for respectively uniform, skewed and real datasets.

Figure 24b shows the cost in number of messages of the redistribution strategy. Depending on the distribution, a rebalancing with a maximal pivot's height set to 1 requires between 2 and 4 times more messages. If we allow the pivot node to be at any height, possibly up to the root, the number of messages reaches a value 30 times higher with our data! Indeed, with this complete flexibility, the tree is almost full and a new insertion generally leads to a costly iterative redistribution process, that may affect all the nodes of the tree in

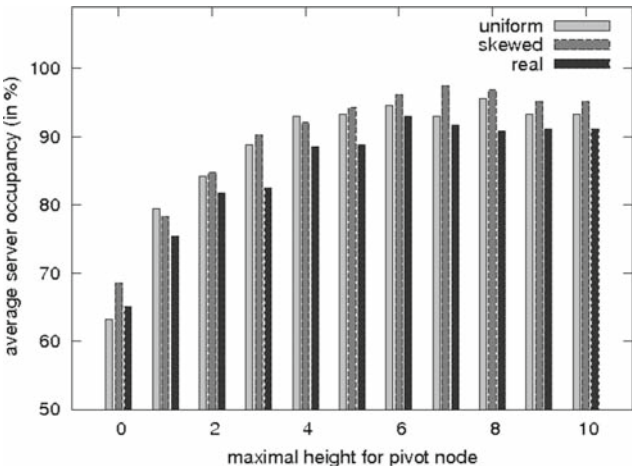



**Fig. 25** Average server occupancy w.r.t. maximal pivot's height

the worst case. Analysis of Fig. 24a,b suggests that setting $\nu$ to 4 provides generally a number of servers very close to the best possible space occupancy, with a number of messages "only" 10 times higher than without redistribution.

### *7.3.2 Analysis of server allocation profiles*

The second set of experiments illustrates how our solution may be deployed in architectures supporting a mixed strategy. We now make the practical assumption that the insertion mechanism without redistribution is used when there are servers available. If, at some point, the system lacks of storage resource, it dynamically switches to the redistribution mode, until new servers are added. We call "server allocation profile" the set of parameters that describe this evolution, including the initial size of the server pool, the average time period necessary to extend the pool, and the number of servers added during an extension.

The experiment assumes that the system consists initially of 200 servers. When new resources are requested, a set of additional 25 servers is allocated. The *shortage period* between the request for new servers and their effective allocation (represented in our profile by the number of insertion requests) corresponds to a forced redistribution mode and constitutes the variable parameter that we analyze. We measure the behavior of the system as the total number of servers and messages required, the maximal height for the pivot node being set to 1, 2 or 3.

Figure 26a shows the final number of servers generated by the splits. As expected, the higher the allowed height for the pivot node is, the lower is the number of requested servers. The impact is opposite on the number of messages (Fig. 26b). Both figures show that the system can handle a shortage of servers during a period corresponding to up to 100,000 insertions with a limited cost (here at most 3 times the cost

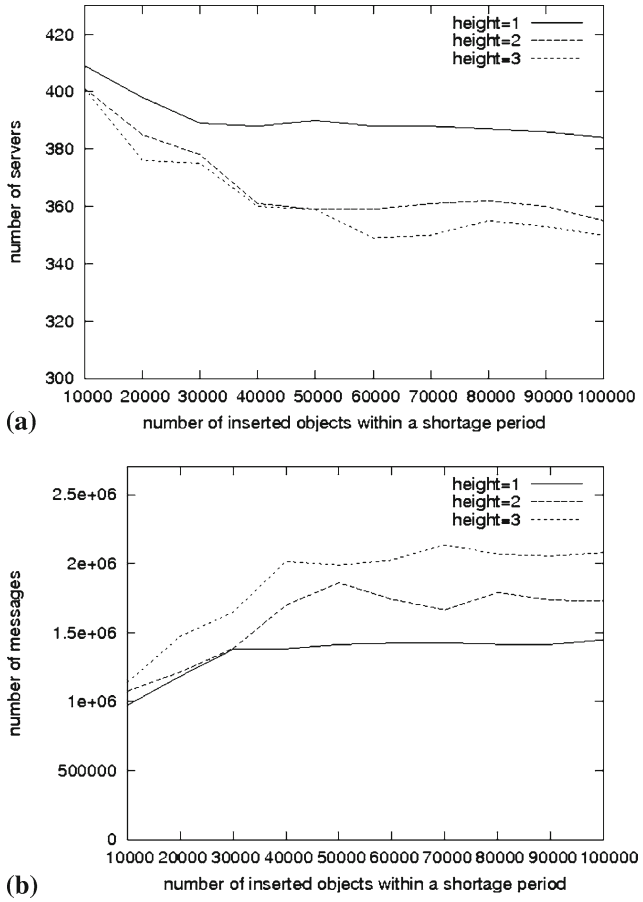



**Fig. 26** Impact of the number of insertions for a shortage period. **a** Number of servers, **b** Number of messages

without shortage). The decreasing aspect of the curves in Fig. 26a is due to the storage balancing effect: with a long shortage period, many objects are inserted and they trigger a redistribution, thereby optimizing servers capacity. The amount of storage balancing increases, and so does the total number of messages.

## 8 Conclusion

SD-Rtree appears as a practical structure for large-scale indexing of spatial data. The experiments confirm its good operational behavior. It constitutes a scalable, compact and efficient structure that evenly exploits the processing and storage space of a pool of distributed data servers. These properties are important in environments which do not enjoy unlimited and free computing power. It should be particularly important in clusters of servers, the primary context for an SDDS application. These configurations should also benefit from mechanisms for storage balancing such as the one proposed.

The distributed quadtree is simple to implement, and behaves efficiently. It constitutes a convenient solution in P2P networks. Its drawback in a more resource-restricted context could be its non-balanced behavior, both for space and processing utilization. The $F_{\min}$ parameter also requires an initial allocation of $4^{F_{\min}-1}$ servers to the cluster. The SD-Rtree approach does not present such a constraint and more tightly adapts the size of the pool of server to the indexing dataset.

Future work on SD-Rtree should address the issues we have already enumerated for future studies. It should also study 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.

**Acknowledgments** We thank Nick Roussopoulos and anonymous referees for helpful suggestions.

## References

1. Adelson-Velskii, G.M., Landis, E.M.: An algorithm for the organization of information. Doklady Akademii Nauk SSSR **146**, 263–266 (1962)
2. Arge, L., Eppstein, D., Goodrich, M.T.: Skip-Webs: efficient distributed data structures for multi-dimensional data sets. In: Proc. Intl. Symp. on Principles of Distributed Computing (PODC), pp. 69–76 (2005)
3. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger B.: The R*tree: an efficient and robust access method for points and rectangles. In: Proc. ACM Symp. on the Management of Data (SIGMOD), pp. 322–331 (1990)
4. Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: Querying peer-to-peer networks using P-trees. In: Proc. Intl. Workshop on the Web and Databases (WebDB), pp. 25–30 (2004)
5. Devine, R.: Design and implementation of DDH: a distributed dynamic hashing algorithm. In: Proc. Intl. Conf. on Foundations of Data Organization and Algorithms (FODO), pp. 101–114 (1993)
6. du Mouza, C., Litwin, W., Rigaux, P.: Dynamic storage balancing in a distributed spatial index. In: Proc. Intl. Symp. on Geographic Information Systems (ACM-GIS) (2007)
7. du Mouza, C., Litwin, W., Rigaux, P.: SD-Rtre: a scalable distributed Rtree. In: Proc. Intl. Conf. on Data Engineering (ICDE), pp. 96–305 (2007)
8. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
9. US Census Bureau Geography Division. Tiger/Line files (2007). http://www.census.gov/geo/www/tiger/
10. Guttman, A.: R-trees : A dynamic index structure for spatial searching. In: Proc. ACM Symp. on the Management of Data (SIGMOD), pp. 45–57 (1984)
11. Hambrusch, S.E., Khokhar, A.A.: Maintaining spatial data sets in distributed-memory machines. In: Proc. Intl. Parallel Processing Symp. (IPPS), pp. 702–707 (1997)

12. Jagadish, H.V., Ooi, B.C, Vu, Q.H.: BATON: a balanced tree structure for peer-to-peer networks. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 661–672 (2005)
13. Jagadish, H.V., Ooi, B.C., Vu, Q.H., Zhang, R., Zhou, A.: VBI-Tree: a peer-to-peer framework for supporting multi-dimensional indexing schemes. In: Proc. Intl. Conf. on Data Engineering (ICDE) (2006)
14. Karlsson, J.S.: hQT*: a scalable distributed data structure for high-performance spatial accesses. In: Proc. Intl. Conf. on Foundations of Data Organization and Algorithms (FODO), pp. 37–46 (1998)
15. Kriakov, V, Delis, A., Kollios, G.: Management of highly dynamic multidimensional data in a cluster of workstations. In: Proc. Intl. Conf. on Extending Data Base Technology (EDBT), pp. 748–764 (2004)
16. Litwin, W., Neimat, M.-A., Schneider, D.A.: RP*: a family of order preserving scalable distributed data structures. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 342–353 (1994)
17. Litwin, W., Neimat, M.-A., Schneider, D.A.: LH*: a scalable, distributed data structure. ACM Trans. Database Syst. (TODS) **21**(4), 480–525 (1996)
18. Liu, B., Lee, W.-C., Lee, D.L.: Supporting complex multi-dimensional queries in P2P systems. In: Proc. Intl. Conf. on Distributed Computing Systems (ICDCS), pp. 155–164 (2005)
19. Mondal, A., Lifu, Y., Kitsuregawa, M.: P2PR-Tree: an R-tree-based spatial index for peer-to-peer environments. In: EDBT Workshops, pp. 516–525 (2004)
20. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: Proc. ACM Symp. on the Management of Data (SIGMOD), pp. 71–79 (1995)
21. Samet, H.: Foundations of Multi-dimensional Data Structures. Kaufmann, Morgan (2006)
22. Tanin, E., Harwood, A., Samet, H.: Using a distributed quadtree index in peer-to-peer networks. VLDB J. **16**(2), 165–178 (2007)
23. Theodoridis, Y., Silva, J.R.O., Nascimento, M.A.: On the generation of spatiotemporal datasets. In: Proc. Intl. Conf. on Large Spatial Databases (SSD), pp. 147–164 (1999)