A Framework for Distributed Spatial Indexing in Shared-Nothing Architectures*

Cédric du Mouza
CEDRIC, CNAM
Paris, France
dumouza@cnam.fr

Witold Litwin
Univ. Dauphine
Paris, France
witold.litwin@dauphine.fr

Philippe Rigaux
Univ. Dauphine and INRIA FUTURS
Paris, France
philippe.rigaux@dauphine.fr

Abstract

The paper presents a complete framework for spatial indexing support in a distributed setting. We consider a shared-nothing environment where a set of servers provides independent storage and computational services. Servers only communicate through point-to-point messaging, and constitute a non-structured network (*i.e.*, non-central server or "super peer"). These features cover two popular architectures, namely a strongly connected cluster of servers, and P2P networks.

Our proposal extends the recently proposed "Scalable Distributed Rtree (SD-Rtree)" structure with new algorithms and protocols. More specifically, we introduce a nearest-neighbors algorithm, a load balancing method and a termination protocol. The result constitutes a set of functionalities for distributed spatial indexing that matches those commonly found in centralized architectures.

Keywords.

Distributed index, storage balancing, share-nothing architecture.

1 Introduction

Spatial indexing has been studied intensively since the early works on Rtrees [9] and Quadtree [6] at the beginning of the 80's. For a recent survey see [17]. It constitutes now an integrated part of most database system engines which usually adopt the simple, flexible and efficient Rtree structure (*e.g., Oracle, MySQL*). Spatial indices support *point*, *window* and *nearest-neighbor* queries over multidimensional objects (*e.g.*, points in high-dimensional spaces) and spatial object (*e.g.*, linear or surfacic features approximated by their bounding boxes). A *point query* retrieves linear or surfacic objects that contain a point argument; a *window query* retrieves points, linear or surfacic objects which intersect (or are contained in) a window argument; a nearest-neighbor query or *kNN query* retrieves the k objects which are nearest to a point argument. These techniques are widely accepted as robust solutions for multidimensional and spatial data management in centralized settings

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 is such contexts. However, only a few work so far has 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 support for efficient processing of common spatial queries for distributed settings.

In the present paper we describe a distributed indexing framework, based on the Rtree principles, which supports all the crucial operations found in centralized systems : insertions (without duplication or clipping) and deletions; point, window and kNN queries. We measure their complexity as the num-

*Work partially funded by the WISDOM project, *http://wisdom.lip6.fr*

ber of messages exchanged between nodes, and show that this complexity is logarithmic in the number of nodes. Our algorithms are adapted from the centralized ones by assuming that (i) no central directory is used for data addressing and (ii) nodes communicate only through point-to-point messages. These are strong but necessary assumptions which allow us to address a wide range of shared-nothing architectures. It means in particular that we aim at an even distribution of the computing and storage load over the participating servers, and avoid to rely on "super peers" and hierarchical network topologies.

Our framework relies on specialized components to deal with these specific constraints. In short:

1. each node maintains in a local cache the *image* of the global tree, and bases the addressing of messages on this image; since an image may be outdated, *addressing errors may occur*, *i.e.*, a query is sent to a node different from the one the query should address; an opportunistic image refreshment method is then applied and guarantees that the addressing error does not repeat;

2. we introduce a *termination protocol*, intended to cope with the possible instability of the network, and which sends to a requester an acknowledgment message whenever its query is completed;

3. finally we propose a *load-balancing method* to deal with situations where an incoming flow of insertions is sent to a network of servers which cannot, temporarily or definitely, extend itself by allocating more storage resource.

Our proposal is based on the recently proposed SD-Rtree distributed index [5]. We rely on its general structure, which is that of a distributed balanced binary spatial tree where each node carries the minimal bounding box (mbb) of the area covered by the objects it stores. In the present paper we extend this proposal with several important contributions. First we consider a wider context where a node can carry both the client (sending queries) and server (providing computing and storage resources) components, being then a *peer*. Second we enrich accordingly the proposal with dedicated protocols: a load-balancing protocol, and a termination protocol. Finally we complement the point and window query described in [5] with a k-NN algorithm.

The core features of our method have been implemented and tested over several datasets. We refer to [5] for the presentation and the experimental validation of our architecture. We report only our new experiments concerning the storage balancing. The present paper aims at providing a complete specification for a distributed spatial indexing method in unstructured networks of servers.

Related work

Until recently, most of the spatial indexing design efforts have been devoted to centralized systems [7] although, for non-spatial data, research devoted to an efficient distribution of large datasets is well-established [4, 14, 3]. The architecture of the many SDDS schemes are hash-based, e.g., variants of LH* [15], or use a Distributed Hash Table (DHT) [4]. Some SDDSs are range partitioned, starting with RP* based [14], till BATON [10] most recently. There were also proposals based on quadtrees like hQT* [12]. [13] describes an adaptive index method which offers dynamic load balancing of servers and distributed collaboration. The structure requires a coordinator which maintains the load of each server.

The P-tree [3] is an interesting distributed B+-tree that has a concept similar to our image with a best-effort fix up when updates happen. A major difference lies in the view correction which is handled by dedicated processes on each peer in the P-tree, and by IAMs triggered by inserts in our framework.

The work [11] proposes an ambitious framework termed VBI. The framework is a distributed dynamic binary tree with nodes at peers. VBI shares this and other principles with our proposal. VBI seems aiming at the efficient manipulation of multi-dimensional points. Our framework rather targets the spatial (non-zero surface) objects, as R-trees specifically. Consequently, as we enlarge a region synchronously with any insert needing it. VBI framework advocates instead the storing of the correspond-

ing point inserts in routing nodes, as so-called discrete data. It seems an open question how far one can apply this facet of VBI to spatial objects.

The rest of the paper presents first the structure of the SD-Rtree (Section 2) and its construction (Section 3). Section 4 presents the algorithms supported by the structure. The load balancing method is described and discussed in Section 5. Experiments about storage balancing are reported in Section 6. Section 7 concludes the paper.

2 Background: The SD-Rtree

The SD-Rtree is conceptually similar to that of the classical AVL tree, although the data organization principles are taken from the Rtree spatial containment relationship [9].

2.1 Structure of the SD-Rtree

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

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

Figure 1 shows a first example with three successive evolutions. Initially (part A) there is one data node d_0 stored on server 0. After the first split (part B), a new server S_1 stores the pair (r_1, d_1) where r_1 is a routing node and d_1 a data node. The objects have been distributed among the two servers

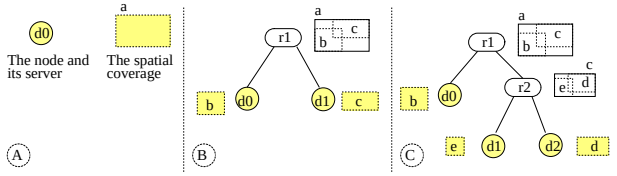


Figure 1: Basic features of the SD-Rtree

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 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 $\mathtt{c} = mbb(\mathtt{d} \cup \mathtt{e})$. 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 di-

rectory 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 next, is an array that contains the part of the directory rectangle shared with other servers. The type of a data node is as follows:

Type: DATANODE
`data`: the local dataset
`dr`: the directory rectangle
`parent_id`: id of the parent routing node
`OC`: the overlapping coverage

2.2 The image

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

An application that accesses an SD-Rtree maintains an *image* of the distributed tree. This image provides a view which may be partial and/or outdated. During an insertion, the user/application estimates from its image the address of the *target server* which is the most likely to store the object. If the image is obsolete, the insertion can be routed to an incorrect server. The structure delivers then the insertion to the correct server using its actual routing node at the servers. The correct server sends back an image adjustment message (IAM) to the requester. Point and window queries also rely on the image to find quickly a server whose directory rectangle satisfies the query predicate. A message is then sent to this server which carries out a local search, and route the queries to other nodes if necessary.

An image is a collection of links, stored locally, and possibly organized as a local index if necessary. Each time a server S is visited, the following links can be collected: the data link describing the data node of S; the routing link describing the routing node of S, and the left and right links of the routing node. These four links are added to any message forwarded by S. When an operation requires a chain of n messages, the links are cumulated so that the application finally receives an IAM with $4n$ links.

2.3 Overlapping coverage

We cannot afford the traditional top-down search in a distributed tree because it would overload the nodes near the tree root. Our search operations attempt to find directly, without requiring a top-down traversal, a data node d whose directory rectangle dr satisfies the search predicate. However this strategy is not sufficient with spatial structures that permit overlapping, because d does not contain *all* the objects covered by dr. We must therefore be able to forward the query to all the servers that potentially match the search predicate. This requires the distributed maintenance of some redundant information regarding the parts of the indexed area shared by several nodes, called *overlapping coverage* (OC) in the present paper.

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

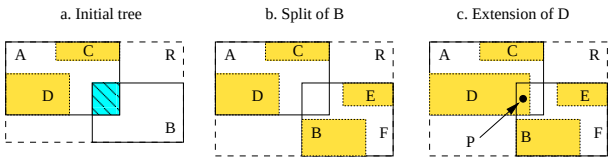


Figure 2: Overlapping coverage examples

Figure 2 illustrates the concept. The left part shows a two-levels tree rooted at R. The overlap-

ping 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.

Figure 2.b shows a split of the server B: its content has been partially moved to the new data node E, and a new routing node F has been inserted. Note that F is now the outer subtree of R with respect to A. Since, however, the intersection $A.dr \cap F.dr$ is unchanged, there is no need to propagate any update of the OC to the subtree rooted at A. Finally the subtree rooted at A may also evolve. Figure 2.c shows an extension of D such that the intersection with F is no longer empty. However our insertion algorithm guarantees that no node can make the decision to enlarge its own directory rectangle without referring first to its parent. Therefore the object's insertion which triggers the extension of D has first been routed to A. Because A knows the space shared with F, it can transmit this information to its child D, along with the insertion request. The OC of D now includes $D.dr \cap F.dr$. Any point query P received by D such that $P \subseteq D.dr \cap F.dr$ must be forwarded to F.

3 Insertion and deletions

We now describe insertions in and deletions from the distributed tree. Recall that all these operations rely on an *image* of the structure (see above) which helps to remain as much as possible near the leaves level in the tree, thereby avoiding root overloading. Moreover, as a side effect of these operations, the image is adjusted through IAMs to better reflect the current state of the structure.

3.1 Insertion

Assume that a client application C requires the insertion of an object o with rectangle mbb in the distributed tree. C searches its local image I and determines from the links in I a data node which can store o without any enlargement. If no such node exist, the 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. Note that the image is initially empty. C must know at least one node N, and send the insertion request to N. C will receive in return an initialization of its image.

If the selected link is of type `data`, C addresses a message INSERT-IN-LEAF to S; else the link refers to a routing node and C sends a message INSERT-IN-SUBTREE to S.

- (INSERT-IN-LEAF message) S receives the message; if the directory rectangle of its data node d_S covers actually $o.mbb$, S can take the decision to insert o in its local repository; there is no need to make any other modification in the distributed tree (if no split occurs); else the message is *out of range*, and a message INSERT-IN-SUBTREE is routed to the parent S' of d_S;
- (INSERT-IN-SUBTREE message) when a server S' receives such a message, it first consults its routing node $r_{S'}$ to check whether its directory rectangle covers o; if no the message is forwarded to the parent until a satisfying subtree is found (in the worst case one reaches the root); if yes the insertion is carried out from $r_{S'}$ using the classical Rtree top-down insertion algorithm. During the top-down traversal, the directory rectangles of the routing nodes may have to be enlarged.

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

The insertion process is shown on Figure 3. The client chooses to send the insertion message to S_2.

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

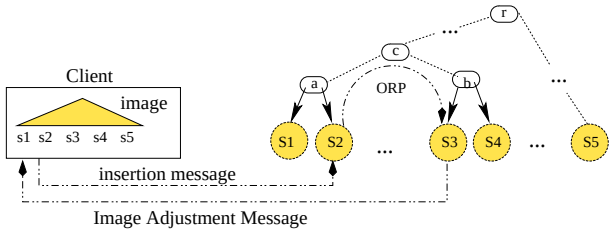


Figure 3: The insertion algorithm

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

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

3.2 Node splitting

When a server S is overloaded by new insertions in its data repository, a split must be carried out. A new server S' is added to the system, and the data stored on S is divided in two approximately equal subsets using a split algorithm similar to that of the classical Rtree [9, 8]. One subset is moved to the data repository of S'. A new routing node $r_{S'}$ is stored on S' and becomes the immediate parent of the data nodes respectively stored on S and S'.

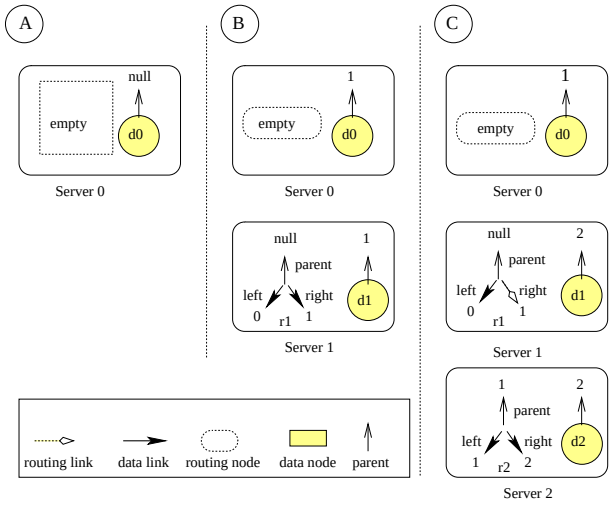


Figure 4: Split operations

The management and distribution of routing and data nodes are detailed on Figure 4 for the tree construction of Figure 1. Initially (part A), the system consists of a single server, with id 0.

Every insertion is routed to this server, until its capacity is exceeded. After the first split (part B), the routing node r_1, stored on server 1, keeps the following information (we ignore the management of the overlapping coverage for the time being):

- the `left` and `right` fields; both are data links that reference respectively servers 0 and 1,
- its height (equal to 1) and its directory rectangle (equal to $mbb(left.dr, right.dr)$),
- the parent id of the data nodes 0 and 1 is 1, the id of the server that host their common parent routing node.

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

Continuing with the same example, insertions are now routed either to server 0 or to server 1, using a Rtree-like CHOOSESUBTREE procedure [9, 2]. When the server 1 becomes full again, the split generates a new routing node r_2 on the server 2 with the following information:

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

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

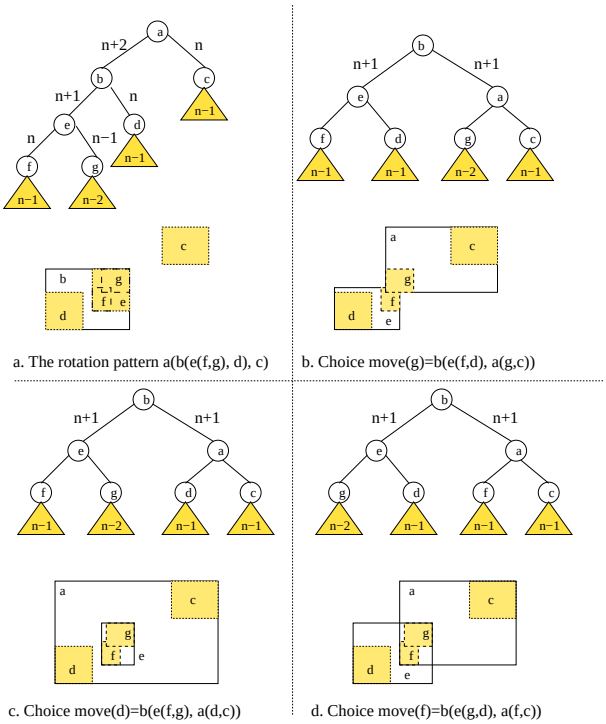


Figure 5: Rotation in the SD-Rtree

3.3 Rotation

In order to preserve the balance of the tree, a rotation is sometimes required during the bottom-up traversal that adjusts the heights. The rotation of the SD-Rtree takes advantage of the absence of order on rectangles which gives more freedom for reorganizing an unbalanced tree, compared to classical AVL trees. The technique is described with respect to a *rotation pattern* which is a subtree of the form `a(b(e(f,g),d),c)` satisfying the following conditions for some $n \geq 1$:

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

An example of rotation pattern is shown on Figure 5. Note that `a`, `b` and `e` are routing nodes. Now, assume that a split occurs in a balanced SD-Rtree at node s. A bottom-up traversal is necessary to adjust the heights of the ancestors of s. Unbalanced nodes, if any, will be detected during this traversal.

The management of unbalanced nodes always reduces to a rotation of a rotation pattern `a(b(e(f,g),d),c)`. The operation is as follows:

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

The choice of the moved node should be such that the overlapping of the directory rectangles of `e` and `a` is minimized. Any pairwise combination of `f`, `g`, `d` and `c` yields a balanced tree. The three possibilities, respectively called `move(g)`, `move(d)` and `move(f)` are shown on Figure 5. The choice `move(g)` (Figure 5.b) is the best one for our example. All the information that constitute a rotation pattern is available from the `left` and `right` links on

the bottom-up adjust path that starts from the splitted node. The rotation 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.

3.4 Deletion

Deletion is somehow similar to that in an R-Tree [9]. We can then consider two different strategies, depending on the frequency of the deletions: (i) a server S from which an object has been deleted may adjust covering rectangles on the path to the root or (ii) if the deletions are frequent, a lazy adjustment of the covering rectangles that does not require any additional messages, *i.e.*, a node may have covering rectangles for its children that became false due to deletions; the updates will be encapsulated within the following message from its children. In both cases, it may also eliminate the node if it has too few objects. The SD-Rtree relocates then the remaining objects to its sibling S' in the binary tree. Node S' becomes the child of its grandparent. An adjustment of the height and of the bounding boxes that is propagated upward as necessary, perhaps requiring a rotation.

4 Query algorithms

We present in this section the algorithms for the most popular queries in spatial databases area, namely point, window and k-NN queries. We also introduce a termination protocol generally required in asynchronous distributed environment.

4.1 Point queries

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

First the client searches its image for a data node d whose directory rectangle contains P, according to its image. A point query message is then sent to the server S_d (or to its contact server if the image is empty). Two cases occur: (i) the data node rectangle on the target server contains P; then the point query can be applied locally to the data repository, and a PQTRAVERSAL must also be routed to the outer nodes o in the overlapping coverage array $d.OC$ whose rectangle contains P as well; (ii) an out-of-range occurs (the data node on server S_d does not contain P). The tree is then scanned bottom-up from S_d until a routing node r that contains P is found. A PQTRAVERSAL is applied from r, and from the outer nodes in the overlapping coverage array $r.OC$ whose directory rectangle contains P.

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

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

4.2 Window queries

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

queries. The routine WQTRAVERSAL is the classical Rtree window query algorithm adapted to a distributed context.

WINDOWQUERY ($W : rectangle$)
Input: a window W
Output: the set of objects whose mbb intersects W
begin
  // Find the target server
  Choose in Image the $targetLink$ corresponding to W
  // Check if correct server. Else move up the tree
  $node :=$ the node referred to by $targetLink$;
  while ($W \not\subseteq node.dr$ **and** $node$ is not the root)
      // out of range
    $node := parent(node)$
  endwhile
  // Now $node$ contains W, or $node$ is the root
  if ($node$ is a data node)
    Search the local data repository $node.data$
  else
    // Perform a window traversal from $node$
    WQTRAVERSAL ($node$, W)
  end
  // Always scan the OC array, and forward
  for each (i, oc_i) **in** $node.OC$ **do**
    if ($W \cap oc_i \neq \emptyset$) **then**
      WQTRAVERSAL ($outer_{node}(i)$, W)
    endif
  end for
end

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

4.3 k-NN queries

We introduce now an algorithm that supports the search for the k nearest neighbors of a point P (k-NN query). It can be decomposed in two steps:

1. find a data node N that contains 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 as presented previously. Assume that a data node N containing P is found. The k-NN query is evaluated locally, *i.e.*, without any extra-message, using a classical k-closest neighbors algorithm [16]. Note that the choice depends on the data structure used locally, and is therefore beyond the scope of the present paper. 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 $neighbor(P, k)$. In order to determine which servers must be contacted, the following strategies can be considered.

Range querying

The first approach performs a range query over the tree, the range r being the distance between P and the farthest object in $neighbors(P, k)$.This is illustrated on Figure 6. The local search in node N finds a set of neighbors, the farthest being object O. The distance from P to O determines the range of the query.

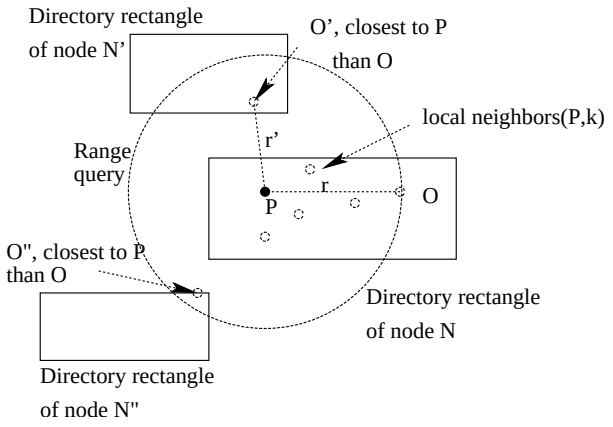


Figure 6: Range query to retrieve objects from other nodes

Now, still referring to figure 6, nodes N' and N'' fall into this range. Thus N sends to N' and N'' a query with P and the radius r. The contacted nodes send back to N the object(s) that can replace one or many element(s) of $neighbors(P, k)$. For example object O', found in the space covered by N' turns out to be closer to P than O. N' searches its local space for all objects whose distance to P is less than r. Object O' is found and returned to N. The same process holds for N'', which returns object O''. Given the list of all objects retrieved through this process, server N determines the final list of nearest neighbors. Still referring to figure 6, and assuming $k = 5$, the final list contains O' but neither O nor O''.

Improved k-NN search

The above algorithm broadcasts the query to all the nodes that may potentially improve the kNN list. However some messages tend to be useless, as illustrated on Figure 7. Assume node N' is contacted first. It sends back to N object O' which improves the nearest neighbors list, and 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''.

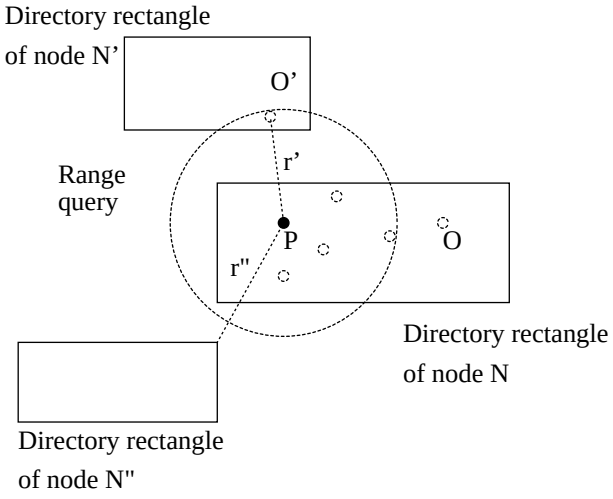


Figure 7: Improved k-NN query

The improved strategy, presented below, generates *a chain* of messages which 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, as proposed in [16] and illustrated in Figure 8.

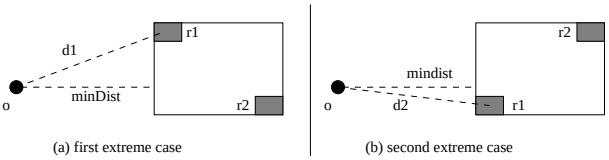


Figure 8: The minMaxDist

Let R be the bounding box of a set of rectangles. First let $minDist$ 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 8 shows the two extreme positions for a rectangle r_1 inside R with respect to the edge closest to P. The maximal distance among d_1 and d_2 represents the minimal distance from P to a rectangle contained in R. This distance is denoted $minMaxDist(P, R)$.

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 must slightly extend the notion of overlapping coverage (OC) presented in Section 2 to keep information about *all* outer nodes, whether there is an overlap or not. So the OC consists now of an array of the form $[1 : oc_1, 2 : oc_2, \cdots, n : oc_n]$, such that oc_i is $outer_N(N_i).dr$. If N' is a node in the overlapping coverage of S, the object closest to P in N' is at best at distance $minDist(P, N.dr)$, and at worse at distance $minMaxDist(P, N.dr)$. Therefore if $d_{max} < minDist(P, N.dr)$, there is no hope to improve the current list of neighbors by contacting N'.

When processing a k-NN query, we compute the $minDist$ and $minMaxDist$ 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 $minDist(P, N_i.dr)$ (resp. $minMaxDist(P, N_i.dr)$). 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}. Our algorithm relies on the two following observations:

i) if $d_{max} < minDist(P, N_i.dr)$ we do not have to visit node N_i;

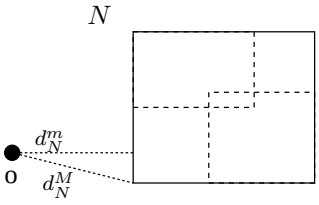
ii) if $minMaxDist(P, N_i.dr) < d_{max}$ we must visit node N_i because it contains an object closer to P that the farthest object from $neighbors(P,k)$.

The algorithm maintains an ordered list of the nodes that need to be visited. 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 $minDist$ and $minMaxDist$, and updates L_{md} and L_{mmd} accordingly. This is illustrated in figure 9 where the routing node N has to be visited, regarding either its $minMaxDist$ d_N^M or its $minDist$ 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 $minMaxDist$ $d_{N_r}^M$ (resp. $d_{N_l}^M$) and $minDist$ $d_{N_r}^m$ (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 following node.

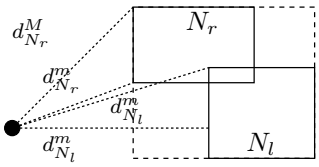
In both cases, the next server in the list is contacted in turn, until no possible improvement of the nearest neighbors can be obtained. The message transmitted from one server to another contains P, the list $neighbor(P,k)$ at the current step of the algorithm, and the lists L_{md} and L_{mmd}.

An heuristic consists in contacting first the server with the minimal $minMaxDist$ 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 $minMaxDist$, the remaining ones are contacted with respect to $minDist$. The pseudo-code of the algorithm is given below.

KNN_EVAL_INIT (P, k)
Input: a point P, the number of nearest-neighbors k



a) N is visited, d_N^m and d_N^M removed from lists



b) new distances for N_r and N_l inserted in lists

Figure 9: Updating L_{mmd} and L_{md}

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 $(id_i, minMaxDist(P, mbb_i))$ in L_{mmd}
Insert $(id_i, minDist(P, mbb_i))$ 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

The cost of this k-NN algorithm is in the worst case that of the simple range query approach. However in the average case the number of message may be drastically reduced.

4.4 Termination protocol

The termination protocol lets a client issuing a point or window query figure out when to end the communication with the servers and to return the result to the application. The protocol may be probabilistic or deterministic. The probabilistic protocol means here that (i) only the servers with data relevant to the query respond, (ii) the client considers as established the result got within some timeout. In unreliable configuration such protocol may lead to a miss, whose probability may though be negligible in practice.

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

5 Load Balancing

The insertion algorithm described in Section 3 requires a split each time a server is full. This adds systematically a new server to the network. The institution that manages the network must be ready to allocate resource 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 in this section a load balancing scheme which allows a full node to transfer part of its data to lightly loaded servers whenever new storage resources cannot be allocated.

This enables a more flexible framework where a node or a subtree that gets full may choose between a split or a data redistribution, depending on contextual parameters (availability and cost of new servers, vs. higher messages exchanges rate if a redistribution is chosen). We first present the technique, analyze its cost and finally discuss how a convenient tradeoff can be chosen between a full-split and a full redistribution strategy.

5.1 Revisited insertion

Assume that an object o must be inserted in a full node N. N may require a load 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 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 (Figure 10).

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

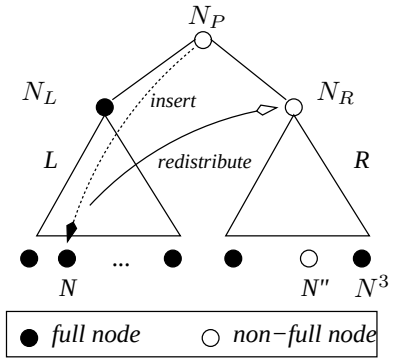


Figure 10: Overview of the load balancing strategy

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). Note that these messages take place during the redistribution process and serve to determine the pivot node. 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 *closest* object indexed by L to the set of objects indexed by 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, and $neighbors(P, 1)$ to $\emptyset$. N_R re-inserts then the object obtained 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 Figure 10, and assume that a reinserted object must be put in the full node N^3. 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 still be full. 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 there is room for inserting o in N.

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 wlog 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)
      $L_{md} := L_{mmd} := N_L$; $neighbors(P, 1) := \emptyset$
      $o :=$ 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

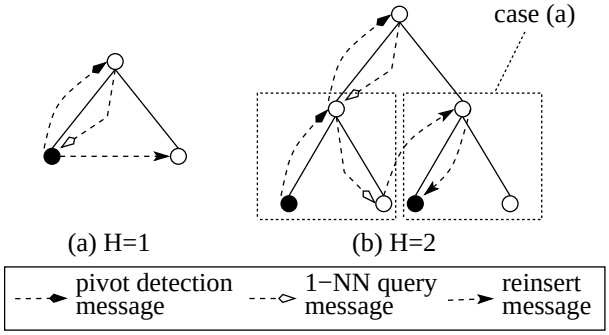


Figure 11: Redistributing data

Figure 11 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. After this first step, we possibly have to iterate the algorithm for the two insertions, respectively in N_L and N_R (see Figure 11 for the example with $H = 2$). In both cases the pivot node's height is at worse $H - 1$. This yields the recursive formula: $cost(H) = 3H + 2 \times cost(H - 1)$. So finally, $cost(H) = \sum_{i=0}^{H-1} 2^i \times 3(H - i) = O(H.2^H)$. 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 Finding a convenient tradeoff between splits and redistribution

The redistribution is costly, its main advantage being to save some splits, and thus the necessity to add servers to the structure. An extreme strategy would be to delay any split until all the servers are full. This would clearly requires a lot of exchanges. It seems more convenient to adopt a trade-off between splits and redistributions.

The basic idea is to proceed only to "local" data reorganization by limiting data redistribution to a subtree of a full node, and not to the whole structure. As shown by the above analysis, the redistribution cost is exponential in the height of the pivot node. By bounding this height, we limit the cost of redistribution at the price or more frequent splits, and less effective storage utilization. Let ν represent the maximal height for a pivot node. The REDISTRIBUTE(N) is simply 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 ν to ∞, 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. The choice of parameter ν highly depends on the kind of applications our architecture is deployed for.

6 Experimental validation

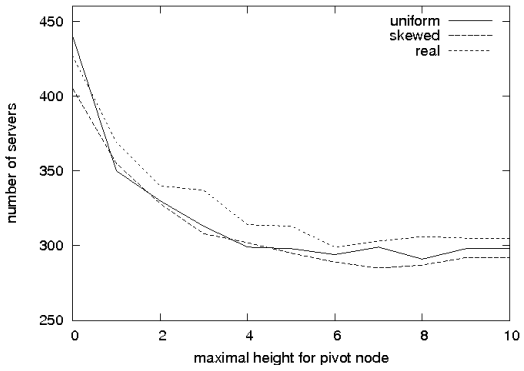
We report several experiments that evaluate the performance of our proposed architecture over large datasets of 2-dimensional rectangles, using a distributed structure simulator written in C. Our datasets include both data produced by the GSTD generator [18], and real data corresponding to the MBRs of 556,696 census blocks (polygons) of Iowa, Kansa, Missouri and Nebraska, provided by Tiger [1]. For comparison purpose the number of synthetic objects, generated following a uniform or a skewed distribution, is also set to 556,696. The capacity of each server is set to 2,000 objects.

6.1 Cost of the redistribution

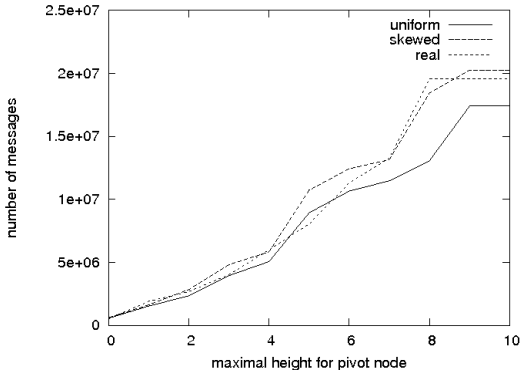
A first experiment is performed to stress the impact of the maximal pivot height allowed for the data redistribution.

First note that with our setting, whatever the distribution is, the height of the tree is 10. Figure 12a shows that our redistribution algorithm highly reduces the number of servers requested, even for small values of the maximal height ν of a pivot node. For instance with a uniform distribution, the number of servers without redistribution is 440. With ν set to 1, this number falls to 350, so a gain of 21% of the required resources. With higher values for ν, 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 concerns mostly 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 distribu-



(a) Number of servers



(b) Number of messages

Figure 12: Impact of the maximal height for the pivot node

tion, 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 (Figure 13).

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 (Figure 13). This value is already reached with a medium value of ν like 4 or 5. With ν set to 1, the improvement is still noteworthy, e.g., 79%, 78% and 75% for respectively uniform, skewed and real datasets.

Figure 12b 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

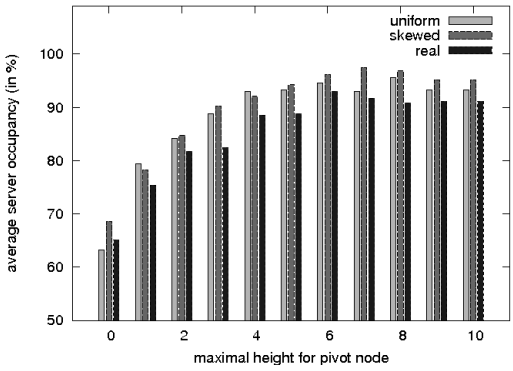


Figure 13: Average server's occupancy rate w.r.t. maximal pivot's height

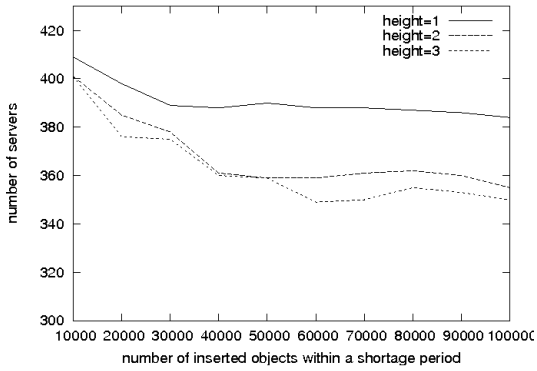
messages. If we allow the pivot node to be at any height, possibly up to the root, the number of messages reaches a value 30 times higher with our data! Indeed, with this complete flexibility, the tree is almost full and a new insertion generally leads to a costly iterative redistribution process, that may affect all the nodes of the tree in the worst case. Analysis of figures 12a-b suggests that setting ν 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.

6.2 Analysis of server allocation profiles

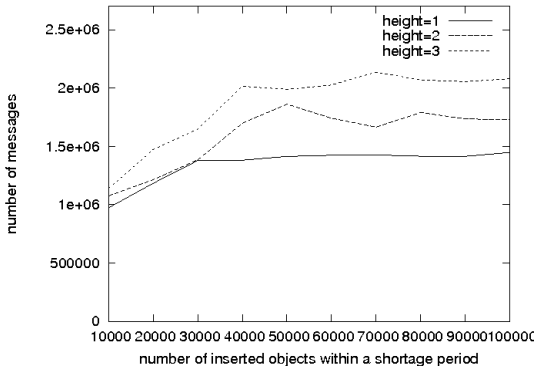
The second set of experiments illustrates how our solution may be deployed in architectures supporting a mixed strategy. We make now the practical assumption that the "traditional" insertion mechanism (without redistribution) is used when there are servers available. If, at some point, the system lacks 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 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.



(a) Number of servers



(b) Number of messages

Figure 14: Impact of the number of inserted objects during a shortage period

As expected, the higher the allowed height for the pivot node is, the lower is the number of requested servers (Figure 14.a). Obviously, the impact is opposite on the number of messages (Figure 14.b). 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 without shortage). The decreasing aspect of the curves in Figure 14a 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.

7 Conclusion

Our framework provides the Rtree capabilities for large spatial data sets stored over interconnected servers. The distributed addressing and specific management of the nodes with the overlapping coverage avoid the centralized calculus. We provide algorithms for traditional spatial queries that limits the number of required messages. Our termination protocol permits to decide in our asynchronous context when the result is set. We also present a data distribution balancing algorithm that limits the splits at the cost of additional messages. We believe that the scheme should fit the needs of new applications of spatial data, using endlessly larger datasets.

Future work on our framework should include other spatial operations like spatial joins. One should study also more in depth the concurrent distributed query processing. As for other well-known data structures, additions to the scheme may perhaps increase the efficiency in this context. A final issue relates to the fanout of our structure. The binary choice advocated in the present paper favors an even distribution of both data and operations over the servers. A larger fanout would reduce the tree height, at the expense of a more sophisticated mapping scheme. The practicality of the related trade-offs remains to be determined.

References

[1] Tiger/Line Technical Documentation, Geography Division, U.S. Census Bureau, 2007. URL:http://www.census.gov/geo/www/tiger/.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*tree : An Efficient and Robust Access Method for Points and Rectangles. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 322–331, 1990.

[3] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, pages 25–30, 2004.

[4] R. Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In *Foundations of Data Organization and Algorithms (FODO)*, 1993.

[5] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtre: A Scalable Distributed Rtree. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 296–305, 2007.

[6] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[7] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[8] Y. Garcia, M. Lopez, and S. Leutenegger. On Optimal Node Splitting for R-trees. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 1998.

[9] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 45–57, 1984.

[10] H. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 661–672, 2005.

[11] H. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2006.

[12] J. S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In *Foundations of Data Organization and Algorithms (FODO)*, 1998.

[13] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *Proc. Intl. Conf. on Extending Data Base Technology (EDBT)*, pages 748–764, 2004.

[14] W. Litwin, M.-A. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 342–353, 1994.

[15] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM Trans. on Database Systems*, 21(4):480–525, 1996.

[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, 1995.

[17] H. Samet. *Foundations of Multi-dimensional Data Structures*. Morgan Kaufmann Publishing, 2006.

[18] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, 1999.