Dynamic Storage Balancing in a Distributed Spatial Index*

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

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

Philippe Rigaux
Univ. Paris-Dauphine
Paris, France
philippe.rigaux@dauphine.fr

ABSTRACT

We propose a general framework to index very large datasets of spatial data in a distributed system. Our proposal is built on the recently proposed Scalable Distributed Rtree (SD-Rtree) [5] and addresses specifically the server allocation problem. In SD-Rtree, a new server is assigned to the network whenever a split of a full node is required. We describe a more flexible allocation protocol which copes with a temporary shortage of storage resources. Our algorithm is especially based on k-NN query processing we introduce as well. We analyze the cost of this protocol, describe its features, and propose practical hints to use it. We also present experiments validating our approach.

1. INTRODUCTION

Spatial indexing has been studied intensively since the early works on Rtrees and Quadtree at the beginning of the 80's [16]. 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*). Recently, the advent of popular distributed systems for sharing resources across large numbers of computers has encouraged research to extend centralized indexing techniques to queries in such contexts. Since index structures are central to efficient data retrieval, it is important to provide indexing support for distributed processing of common spatial queries.

In [5] we describe a distributed indexing framework named SD-Rtree, based on the Rtree principles, which supports all the crucial operations found in centralized systems: insertions (without duplication or clipping) and deletions; point and window queries. We measure their complexity as the number of messages exchanged between nodes, and show that this complexity is logarithmic in the number of nodes. Our structure is adapted from the centralized one 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" or hierarchical network topologies. We propose specialized components to deal with these specific constraints. In particular each node maintains in a local cache the *image* of the global tree, and bases the addressing of messages on this image.

The insertion algorithm proposed in [5] requires a split operation whenever a server becomes full, and the re-assignment of half of the objects from S to a newly allocated server S'. This assumes that a server is available each time it is requested. This may be difficult to achieve in settings where new storage availability cannot be guaranteed. Indeed, adding storage dynamically requires time, manpower, and a network management policy under control by a reliable institution.

Below we address this issue and extend the design of our SD-Rtree structure with a *storage balancing method* which redistributes objects from a full server and reorganizes accordingly the distributed tree. The method deals with situations where an incoming flow of insertions is sent to a network of servers that cannot, temporarily or definitely, extend itself by allocating more storage resources. In short, our contributions is:

1. we introduce a k-NN algorithm in the SD-Rtree framework; this algorithm turns out to be useful for our goal;
2. we describe a redistribution algorithm which saves the necessity of adding systematically new servers, at the cost of additional messages;
3. we discuss and analyze the impact of storage balancing on the insertion cost, and propose a simple parameter to achieve a trade-off between a full-split policy without balancing, and a full-balancing policy without split;
4. finally we report several experiments conducted on the SD-Rtree platform.

Our proposal constitutes a flexible storage allocation method for a distributed spatial index. The insertion policy can be tuned dynamically to cope with periods of storage shortage. In such cases storage balancing should be favored for better space utilization, at the price of extra message exchanges between servers. When server availability is not a problem,

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

storage balancing should be avoided or used very sparsely for better performance.

Related work

Until recently, most of the spatial indexing design efforts have been devoted to centralized systems [6] although, for non-spatial data, research devoted to an efficient distribution of large datasets is well-established [4, 13, 3]. Distributed index structure are either hash-based [14], or use a Distributed Hash Table (DHT) [4]. Some others are range partitioned, starting with RP* based [13], till BATON [9] most recently. There were also proposals based on quadtrees like hQT* [11]. [12] describes an adaptive index method which offers dynamic load balancing of servers and distributed collaboration. However the structure requires a coordinator which maintains the load of each server.

The work in [10] proposes an ambitious framework termed VBI. VBI is a distributed dynamic binary tree with nodes at peers. 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, we enlarge a region synchronously with any insert needing it. VBI framework advocates instead the storing of the corresponding point inserts in routing nodes, as so-called discrete data. It seems an open question how far one can apply this facet of VBI to spatial objects. Another recent theoretical contribution to the field is [1].

The rest of the paper recalls first the structure of the SD-Rtree and its construction (Section 2). The storage balancing method is described and discussed in Section 3. Experiments are reported in Section 4. Section 5 concludes the paper.

2. BACKGROUND: THE SD-RTREE

In this section we briefly describe the aspects of the SD-Rtree which are useful to understand our extensions. A longer presentation can be found in [5].

2.1 Structure of the SD-Rtree

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

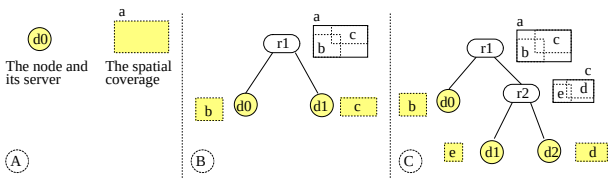


Figure 1: Basic features of the SD-Rtree

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

2.3 Overlapping coverage

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. So each data node N maintains all the directory rectangle of the sibling of its ancestors N_i (called in the following *outer nodes*) on the path from N to the root of the tree. This information is stored as an array of the form $[1 : oc_1, 2 : oc_2, \cdots, n : oc_n]$, such that oc_i is $outer_N(N_i).dr$. Note that we have to trigger a maintenance operation only when this overlapping changes.

2.4 Insertion

We now describe insertions in 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.

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[1]. 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 2. 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.

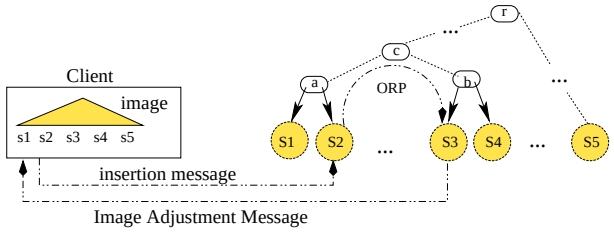


Figure 2: The insertion algorithm

In the worst case an insertion costs $3 \log n$ message, where n is the number of servers. 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. As shown by the experiments reported in [5], this strategy reduces drastically the workload of the root node.

2.5 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 [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'.

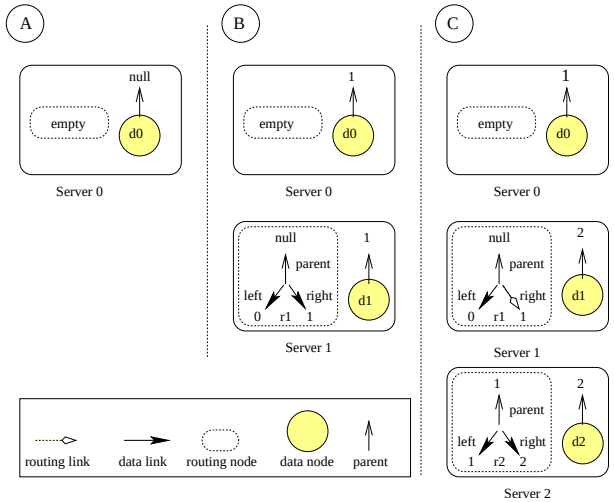


Figure 3: Split operations

An example of management and distribution of routing

[1]The image is initially empty. C must know at least one node N, and send the insertion request to N.

and data nodes is detailed on Figure 3 that corresponds to 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 (the tree root), 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 [8, 2]. When the server 1 becomes full again, the split generates a new routing node r_2 on the server 2, with its `left` and `right` data links pointing respectively to server 1 and to server 2, and its `parent_id` field referring to server 1. At this point the tree is still balanced. It may happen that splits make the tree imbalanced. In [5] we describe a rotation mechanism to preserve the balance of the tree at a very low extra-cost.

3. STORAGE BALANCING

The insertion algorithm described above 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 storage balancing scheme which allows a full node to transfer part of its data to lightly loaded servers whenever new storage resources cannot be allocated.

3.1 k-NN queries

Our technique is partially based on a k-NN querying algorithm 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 extra-message, using a classical k-closest neighbors algorithm [15]. The initial list of neighbors, found locally, is stored into an ordered list $neighbors(P, k)$.

Obviously, some closer 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 4 (for the sake of simplicity we represent on this figure objects as points). The local search in node N finds a set of neighbors, the farthest being object O. We assume here that k remains small compared to node capacity. So the distance from P to O determines the range of the query.

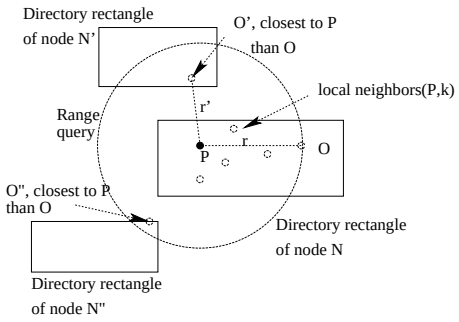


Figure 4: Range query to retrieve objects from other nodes

Now, still referring to Figure 4, 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 4, 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 5. 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''.

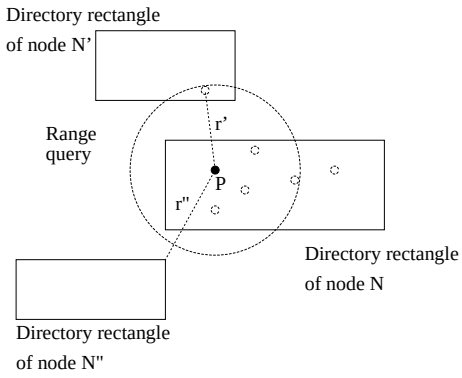


Figure 5: Improved k-NN query

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 [15] and illustrated in Figure 6.

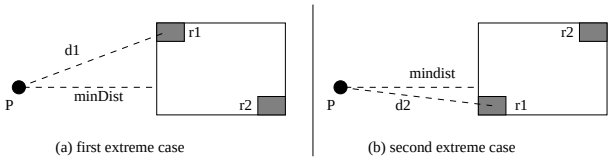


Figure 6: The minMaxDist

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 6 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 and d_2 guarantees there is an object within R at a distance less or equal to D. D 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 $neighbors(P, k)$. Recall from Section 2 that the overlapping coverage (OC) is 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 worst 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 7 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 P. 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 next node in the list.

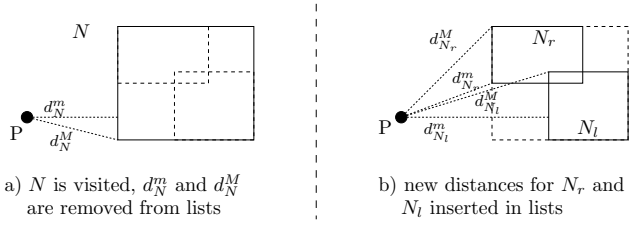


Figure 7: Updating L_{mmd} and L_{md}

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 $minDist$.

Althought this improved strategy is expected to reduce the cost of the kNN algorithm, its worst case is still that of the simple range query approach.

3.2 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 (Figure 8).

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.

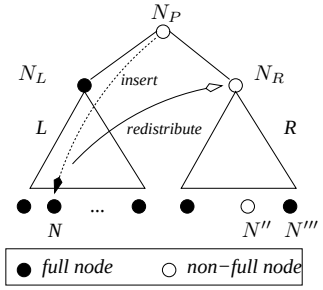


Figure 8: Overview of the load balancing strategy

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 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 8, 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 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) := ∅
      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 9 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 9 for the example with $H = 2$). In both cases the pivot node's

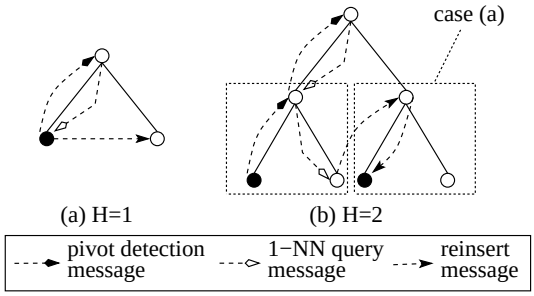


Figure 9: Redistributing data

height is at most $H - 1$. This yields the recursive formula: $cost(H) = 3H + 2 \times cost(H - 1)$. So finally, $cost(H) = \sum_{i=0}^{H-1} 2^i \times 3(H - i) = O(H.2^H)$. 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.

3.3 Discussion

The redistribution is costly, its main advantage being to save some splits, and thus the necessity to add servers to the structure. 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 split until all the servers are full. This 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, 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) algorithm 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, since a split occurs only when all the servers are full.

Parameter ν 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. ν 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 ν to 1. Depending on the time necessary to allocate new servers to the pool, ν can be progressively raised, as the servers get full, until new storage space becomes available. Then it can be set to 0 again to disable the

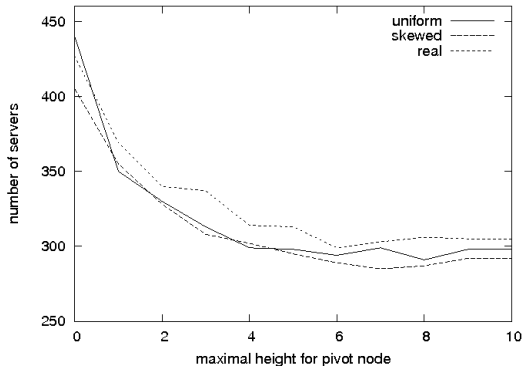
storage balancing option. So ν is a tuning parameter that brings flexibility since it allows to adapt the behavior of the system with respect to the available resources.

4. EXPERIMENTAL VALIDATION

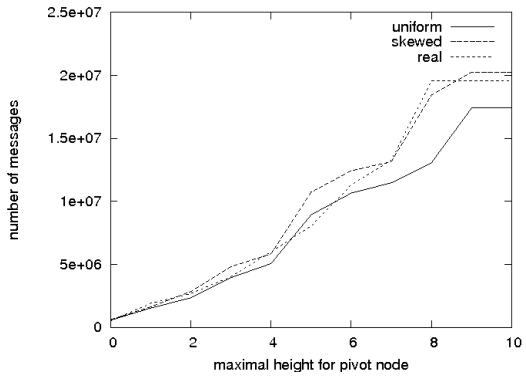
We report several experiments that evaluate the performance of our proposed architecture over large datasets of 2-dimensional rectangles, using a distributed structure simulator written in C. Our datasets include both data produced by the GSTD generator [17], and real data corresponding to the MBRs of 556,696 census blocks (polygons) of Iowa, Kansa, Missouri and Nebraska, provided by Tiger [7]. 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.

4.1 Cost of the redistribution

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



(a) Number of servers



(b) Number of messages

Figure 10: Impact of the maximal height for the pivot node

First note that with our setting, whatever the distribution is, the height of the tree is 10. Figure 10a 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 distribution, 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 11).

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 11). 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.

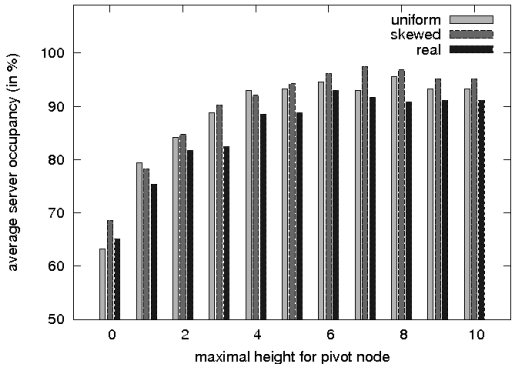


Figure 11: Average server's occupancy rate w.r.t. maximal pivot's height

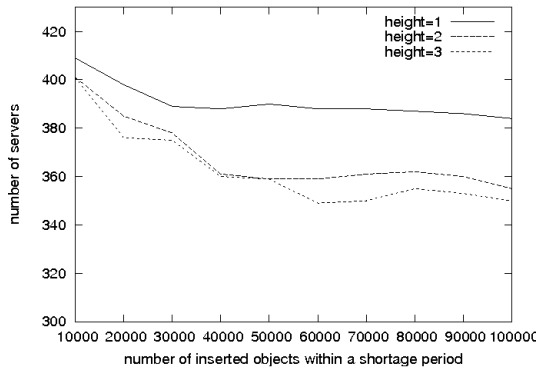
Figure 10b shows the cost in number of messages of the redistribution strategy. Depending on the distribution, a rebalancing with a maximal pivot's height set to 1 requires between 2 and 4 times more messages. If we allow the pivot node to be at any height, possibly up to the root, the number of messages 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 the worst case. Analysis of Figures 10a-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.

4.2 Analysis of server allocation profiles

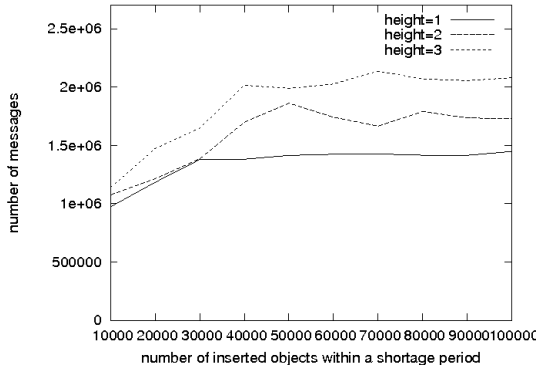
The second set of experiments illustrates how our solution may be deployed in architectures supporting a mixed strategy. We make now the practical assumption that the "traditional" insertion mechanism (without redistribution) is used when there are servers available. If, at some point, the system lacks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

scribe 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 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 12: 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 12.a). Obviously, the impact is opposite on the number of messages (Figure 12.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 12a 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.

5. CONCLUSION

Our framework enables a Rtree-based indexing structure for large spatial data sets stored over interconnected servers. We present a k-NN querying algorithm and a data distribution algorithm that limits the number of servers and improve the storage utilization, at the cost of additional messages. We believe that the scheme should fit the needs of new applications using endlessly larger sets of spatial data.

6. REFERENCES

[1] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-Webs: Efficient Distributed Data Structures for Multi-Dimensional Data Sets. In *PODC*, pages 69–76, 2005.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*tree : An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *WebDB*, pages 25–30, 2004.

[4] R. Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In *FODO*, 1993.

[5] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtre: A Scalable Distributed Rtree. In *ICDE*, pages 296–305, 2007.

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

[7] U. C. B. Geography Division. Tiger/Line files, 2007. URL:http://www.census.gov/geo/www/tiger/.

[8] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 45–57, 1984.

[9] H. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *VLDB*, pages 661–672, 2005.

[10] H. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. In *ICDE*, 2006.

[11] J. S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. In *FODO*, 1998.

[12] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. In *EDBT*, pages 748–764, 2004.

[13] W. Litwin, M.-A. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *VLDB*, pages 342–353, 1994.

[14] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *TODS*, 21(4):480–525, 1996.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.

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

[17] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *SSD*, 1999.