Evaluation of spatial indices implemented with the DBMS O_2[1]

J.P. Peloux[a,b] – **G. Reynal de St Michel**[b] – **M. Scholl**[b,c]

[a] *Fleximage*
43 rue de la Brèche aux Loups, 75012 Paris, France

[b] *Cedric/CNAM*
292 Rue St Martin 75141 Paris Cedex 03, France

[c] *INRIA-Rocquencourt*
78153 Le Chesnay Cedex, France

(peloux,reynal,scholl)@asimov.cnam.fr

RÉSUMÉ. *Une évaluation de performances de l'opération de pointé est réalisée pour différentes structures de données spatiales. Parmi les quatre index testés, deux sont de type quadtree [Sam90], dont un est ressemble à la structure Excell Hiérarchique [Tam83]. Les deux autres structures sont des variantes connues de l'arbre-R appelées arbre-R* [BKSS90] et arbre-R$^+$ [SRF87]. L'implémentation est réalisée en O2C, le langage de développement du Système de Gestion de Base de Données Orienté Objet O_2 [O291]. L'objectif de l'expérimentation est double : (i) contribuer au choix d'une méthode d'accès pour l'accélération des requêtes spatiales dans une base de données géographique, les requêtes Pointé et Fenêtrage étant les plus courantes; (ii) examiner quels sont les problèmes posés par une implémentation avec un SGBD commercial.*

ABSTRACT. *We report a measurement experiment for evaluating the access performance of point queries with several spatial data structures implemented with O_2C, the development language of the object-oriented database management system (DBMS) O_2 [O291]. Four data structures were implemented and tested: the first one is a quadtree structure [Sam90], the second one is an extension which turned out to be very close to Hierarchical Excell [Tam83]. The third and the fourth data structures are the well known tree structures called R*tree [BKSS90] and R+tree [SRF87]. The objective we had in mind was twofold: (i) contribute to the choice of spatial access methods to be implemented in geographic DBMS in order to accelerate spatial queries among which point and window queries are the most common; (ii) understand where the performance bottlenecks are when implementing such structures with commercial*

[1] This work was supported in part by the Esprit BRA Amusing

object-oriented DBMS. Our goal was not to find the best data structure but to understand the behavior of each data structure with an actual implementation.

MOTS-CLÉS : *arbre, bases de données géographiques, index spatial, méthodes d'accès, objet, O2, requêtes spatiales, SGBD OO, SIG.*

KEY WORDS : *access methods, geographical data bases, GIS, object, O2, OO DBMS, spatial index, spatial queries, tree.*

---

1 Introduction

A large number of spatial data structures for representing 2 dimensional (2D) information have been proposed in the literature in the past ten years. For a comprehensive study, see for example [Gün88, LT93, Ooi90, Sam90]. CAD, medical and geographic applications are examples which require the management of 2D information. In this paper, we restrict our attention to geographic information systems (GIS). Such systems use as a kernel a geographic database management system (DBMS). Data handled by geographic DBMS has structured attributes as well as spatial attributes that describe the geometry and the topology of geographic objects. A fast access to such data, the volume of which is usually extremely large, requires the use of indices. Indices as provided by regular DBMS, e.g., B-tree like structures [BM72] are inefficient when the query is related to the location (geometry) of the objects. One major issue related to the choice of such indices, central to the design of geographic DBMS is the variety of geometric queries: one might argue that there is no index which performs best whatever the operation is. However we believe that all commercial systems will eventually provide a single simple-to-be-implemented data structure, although it might lead to uneven results depending on the query. This is the case with relational database technology: most of the commercial relational DBMS provide a unique B-tree-like index data structure.

The objective of this paper is to contribute to the choice of such a single index. The problem is quite challenging for two reasons:

1. there is no standard yet concerning the query language and therefore there is no agreement on which are the most important spatial operations which require the use of a spatial index for performance reasons. Geometric selection, clipping and map overlay are among the most common operations. Examples of geometric selection are point queries (find the objects that contain a given point) and window queries (objects overlapping, i.e., intersecting or containing a given rectangle). In this paper, we restrict our attention to point queries. A more complete study should include window queries and geometric operations required by the numerous algebras and relational query languages extended to geographic information proposed in the literature (see for example

[Fra82, SDMO87, Güt88, SPSW90, SV89, BDQV90]). In particular map overlay uses polygon intersection [KBS92, BKS93, Gün93].

2. a very large number of different data structures, applying to data represented in a vector format as well as in a raster format, are proposed in the literature. In the sequel we assume data to be handled by geographic DBMS to be in a vector format. Spatial data structures can schematically be classified into four categories: grid-like structures [TS82, NHS84, Fre87], k-d-trees [Ben75, Rob81, OMSD87, HSW89], quadtree-like structures [FB74, Ked82, AS83, Sam90, NP94] and Rtree-like structures [Gut84, RL85, SRF87]. We chose in this paper to concentrate on the two last categories because of their growing success in the past few years. A more complete study should however include other grid or tree structures. In particular the k-d-B-tree [Rob81] or the hB-tree [LS90] derived from the k-d-tree have been advocated to be efficient for point queries.

A large number of quadtree variants based on the recursive decomposition of the plane in four quadrants are proposed in [Sam90]. We study below two variants: the first one called Rquadtree is extremely simple. The second one called Nquadtree is a more complex one, where (a) instead of splitting an overflowing quadrant into 4 quadrants, it is partitioned into N squares, and (b) splitting of a square is adapted to the load. When N=4 the Nquadtree becomes the regular Rquadtree. As a matter of fact, the Nquadtree is a variant of Hierarchical Excell [Tam83]. The Nquadtree should outperform the Rquadtree. One objective of this paper is to check and quantify this performance improvement.

Since the seminal paper by Guttman [Gut84], there has been a large number of studies centered around Rtrees [RL85, FSR87, SRF87, BKSS90, KF93, TS94]. One advocated advantage of Rtree structures is that the tree (as B-trees do) evolves smoothly with data insertions in the database while keeping the tree balanced: access to any object requires to follow a path from the tree root to a leaf. This path has a constant length whatever object is searched and whatever the file size is. We chose a variant called R*tree [BKSS90]. One drawback of Rtrees is that they might require several leaf nodes to be visited, the visit of which might not be necessary for the solution. R+trees [SRF87, FSR87] avoid this drawback. Compared to Rtrees, they should significantly improve the access performance. Compared to Rquadtrees whose construction is trivial, R+tree design and implementation are more complex and depend on the 2D space partition technique. We chose a variant called PBR [BF90] and compared it to the R*tree.

The first goal of this paper is to compare R*(+)trees to R(N)quadtrees. This is done by measures on a dataset provided by IGN (french national geographic institute) that corresponds to a partition of the 2D space into regions in the area of Toulouse, France; it does not contain line objects such as roads or networks. At a first glance, the restriction to region objects should not significantly change the results, since in both cases the objects stored in the index are not the geometric objects themselves but their minimum bounding rectangle (MBR). However the fact that lines have longer and narrower MBR's might have some impact on the performance [Pel92, HS92].

Our objective is not to find the best data structure among R(N)quadtrees and R(+)trees but to give some qualitative indications on the behavior of each data structure for point queries, when implemented with a commercial DBMS. We chose the object-oriented DBMS O_2 [O291, BDK92]. The measurement results are of course extremely dependent of this DBMS and we shall see that they suffer a little from the limitations of the current version of O_2. However we shall be able to draw some conclusions independent from the DBMS - and this is the second goal of this paper- on the design of spatial indices with object oriented DBMS. Following are some questions that motivated this work: should these indices be implemented at the system level, or is it sufficient to implement them with the application programming interface (API) provided by the DBMS? Which facilities are then required? Are point queries cpu or I/O bound? How do clustering and index (dataset) size influence the performance?

The paper is organized as follows: Section 2 presents the four data structures and algorithms for point queries. Section 3 describes our measurement experiment. Measurement results are given in Section 4 for the Rquadtree and the Nquadtree and in Section 5 for the R*tree and the R+tree. The four data structures are compared in Section 6. Section 7 describes some related work. Some conclusions drawn from our experiment are given in Section 8.

2 The data structures

The four data structures are trees utilized for indexing a set of geographic objects (whose geometry here is represented by polygons). Each node of the tree is associated with a rectangle included in the father node's rectangle. Geographic objects are referenced in the tree leaves. Only their geometry's MBRs are stored in the tree leaves together with the geographic object identifier (oid), (or that of its geometry) [2]. A point query then includes two phases: the first phase called *(index) search* traverses the tree from the root to the leaves and has for a result a set (or list) of tuples [MBR, oid]. The result list only includes tuples whose MBR includes the point. The second phase, called *selection* in the sequel, consists in actually testing the inclusion of the point in each object (polygon) of the resulting list. This test is refered to as *PointInPolygon* in the sequel.

[2] A variant not studied here would consist in storing the geographic object together with its MBR in the tree leaves. This would of course save one indirection and therefore possibly one page fault. But the increase in object size in the tree leaves would then result in leaf splitting and eventually in larger depth indices. Worse, storing the geometry together with its MBR would render the index dependent from the object geometry implementation.

2.1 Rquadtree

Quadtree-like tree structures utilize a recursive decomposition of the 2D space in quadrants [Sam90] : with each node of the tree is associated a quadrant. Various quadtree Spatial Access Methods (SAM) have been developed. All differ by the way spatial objects are mapped into quadrants. While some quadtree methods deal only with point data [FB74], others index polygon data; the latter deal with vertice and edge components of polygons [SW85], or with the MBR of polygons [Ked82], [Sha86]. We propose a simple variant data structure here, belonging to the latter category called Rquadtree first mentioned, to our knowledge, in [Cha91] and studied in [Pel92]. An internal node only contains pointers to its sons, one for each quadrant. A leaf node contains the set of geographic objects whose MBR *overlaps* the leaf quadrant. The number of tuples [MBR, oid] per leaf is limited by a *threshold t*. Upon the insertion of a tuple into a leaf already including *t* tuples, the leaf is split into four quadrants. Its tuples are distributed into the new leaves as follows: A tuple with MBR m is stored in any leaf l overlapping m. Therefore tuples are duplicated in all adjacent quadrants overlapping their MBR[3]. The leaf size *t* can be chosen such that the leaf exactly fits into a page. Figure 1 gives an illustration of the data structure.

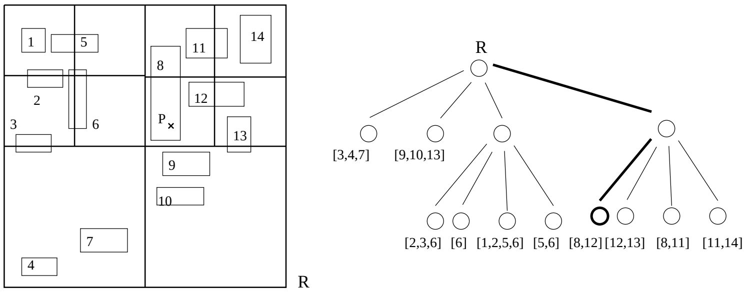


Figure 1: *Rquadtree Point Search - Threshold t=4. The result is 8*

As for other spatial tree structures which utilize a disjoint decomposition of the space (e.g. R+tree, see below), a single leaf is selected in the search phase of a point query: a single path is followed from the tree root to a leaf whose distance from the root is variable. The unbalance of the Rquadtree might lead to a variable access performance. The less uniform the MBR distribution in the 2D space, the less balanced the tree. Besides, since the splitting factor is only 4, quadtree depth increases with the number of indexed objects faster than

[3]If after splitting, there is a leaf, say l, with t+1 tuples, then nothing is done. If then a tuple is inserted into l, then l is split again, etc. Usually leaves are mapped onto pages with capacity t (this was not the case with our implementation). Then an overflowing technique should be chosen. Assuming that such overflows rarely happen, one might choose an overflow page common to all leaves. The same technique applies to the Nquadtree and the R+tree.

with other structures (whose internal node size might be much larger). Because of space limitations, we only give the point search algorithm here (Procedure LOCATE). Detailed insertion and deletion algorithms can be found in [Pel92].

```
Procedure LOCATE(P,M)
INPUT:    Point P, Tree Root M
OUTPUT:List                    /*list of [MBR,oid] such that P in MBR */
If        M is Leaf then
          For all N in MBR(M)  /* MBR(M): list of tuples in node M */
          if P in N then       /* If P included in the MBR of tuple N */
          List += N;
Else      LOCATE(P,quad(P,M));/*quad(P,M)is the son of M containing P*/
```

2.2 Nquadtree (Hierarchical Excell)

The second structure we implemented, can be seen as a Rquadtree extension, or as a variant of Hierarchical Excell [Tam83]. B-trees as opposed to binary search trees present the advantages of (i) tuning the number N of objects per node (the B-tree order) so that a node content is exactly mapped onto a page, (ii) decreasing the tree depth by increasing the nodes degree, and (iii) balancing the tree. Looking at the 2D case, quadtrees have the same characteristics as binary trees: unbalance, possibly high depth due to the small splitting factor (4) and possible performance discrepancy (too many page requests) due to a bad mapping of the tree into pages. The Nquadtree was designed to avoid such a performance degradation: instead of partitioning a quadrant into 4 subquadrants, we chose to divide it into $N \geq 4$ rectangles.

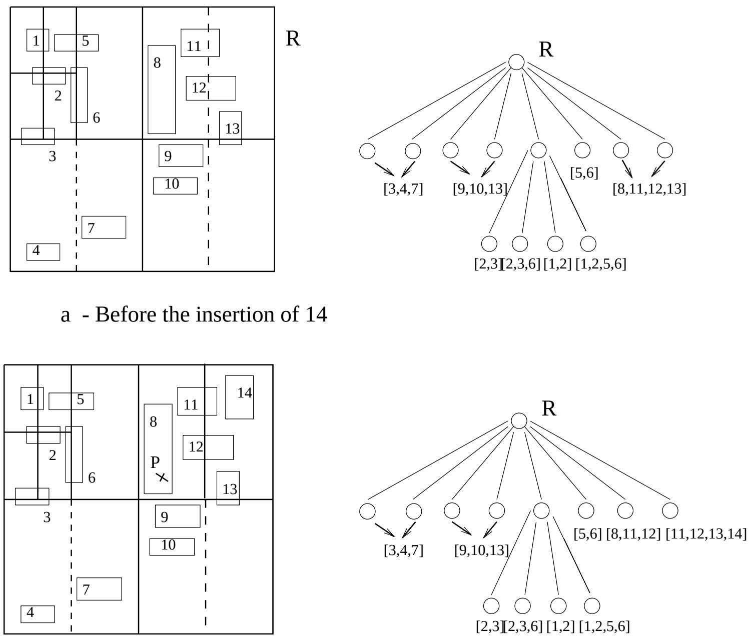
The Nquadtree as the Rquadtree remains unbalanced. Besides, a large value of N may lead to a very low node (storage) utilization. To avoid the two latter drawbacks, a gradual splitting is defined as follows:

Instead of splitting right away an overflowing leaf into N new leaves, we first divide it into four leaves l1, l2, l3, l4[4] as for the Rquadtree. Then, when in turn, one of these new leaves, say l1 is full, all leaves (l1, l2, l3, l4) are themselves divided into two so that there are now 8 leaves. However only for the overflowing leaf l1 the tuples are distributed among the two new leaves. The sons of another leaf, say l2 (that gives birth to l21 and l22) share the same list of tuples L as their father, until one of them, say l21, in turn overflows. Then the tuples in L are actually distributed among l21 and l22.

As an example, look at Figure 2. Before the insertion of tuple 14, the two right leaves (depth 1) contain the same list of tuples: [8,11,12,13]. Upon the

[4] With the Hierarchical Excell structure, the leaf is first divided into 2 leaves.

insertion of 14, this list is actually split into two lists: [8,11,12] is associated with one leaf, while the other one receives [11,12,13,14].



b - After the insertion of 14

Example of point search (P) ; result : rectangle 8

Figure 2: *An example of Nquadtree, N=8, t=4*

Note that, before the insertion of tuple 6 (assuming tuple j is inserted before tuple i, where j< i), the tree has one root and 4 leaves among which the third one called l3 contains [1,2,3,5]. The insertion of 6 (in l3) triggers the splitting of the 4 leaves into 8 leaves: l3 is divided into l31 and l32. l31 (resp. l32) receives [1,2,3,5,6] (resp. [5,6]). l31 is still overflowing. Then, since the maximal splitting factor N=8 is reached, l31 is split into 4 leaves (depth 2) : the list [1,2,3,5,6] is split into [2,3], [2,3,6], [1,2] and [1,2,5,6] (see Figure 2 a).

More generally, if $N = 2^n$, a leaf is successively split into 4,8,..,2^n leaves, where 2 or 4 or,..., or 2^{n-1} leaves might include the same list of tuples. On any path from the root to a leaf, each internal node except perhaps the last one has degree N. Such a gradual splitting a la Extendible Hashing [FNPS79] guarantees a better node utilization in the presence of a non uniform MBR distribution. The splitting from 2^{n-1} to 2^n is done in dividing the rectangle with a median line alternately vertical or horizontal.

We now give the point search algorithm for the Nquadtree (see [Pel92] for the insertion and deletion algorithms):

```
Procedure LOCATE(P,M)
INPUT :   Point P, Root M
OUTPUT List
If        M is Leaf then
          For all N in MBR(M)/* list of tuples in node M */
          if P in N then        /* P in the MBR of tuple N */
          List += N
Else      LOCATE(P,f(P,M))
```

f(P,M) returns the son in M whose MBR contains P. Node M includes *de*+1 fields. The first one gives the number of sons *de* (between 4 and N); it is followed by *de* fields, one for each son.

Only one parameter has to be taken into account for the choice of N, the page size: one might choose N so that a node with N cells exactly fits into a page. However recall (i) the search time includes a traversal from the root to a leaf and is therefore linear in the leaf depth; the larger N, the smaller the depth; (ii) this traversal time also depends on the time for processing a node: the computation of the unique cell i among N which includes the point is trivial; the time to access the ith element of the node might be significant when N is large depending on its implementation.

2.3 Rtree-like indices

Two Rtree structures were implemented and evaluated:

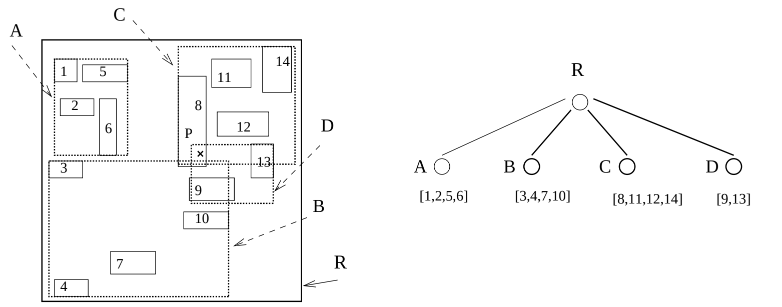
1. we used a R*tree structure as described in [BKSS90] and already implemented with O_2 by IGN [Ray90]; compared to other Rtree proposals, the R*tree construction has a more sophisticated splitting policy which attempts to minimize the overlap between brother nodes. From now on, for short the R*tree is refered to as Rtree. With each node of an Rtree is associated a rectangle. Only subtrees whose root rectangle includes the point are scanned. The major drawback of Rtrees is that twin rectangles may strongly overlap. Therefore if the point happens to be in two overlapping rectangles r and r', subtrees with roots r and r' have to be scanned.

2. R+trees overcome such a drawback: Only one path from the root to a leaf is followed. The price to be paid for this is a possibly significant object duplication. There are several variants of R+tree construction depending on the splitting policy [Gre89, HS92, FSR87]. We chose to implement the PBR R+tree as described in [BF90] because of its better node storage utilization. Therefore, compared to other R+trees, the index depth might be decreased. The PBR R+tree refered to as R+tree in the sequel uses a two step static construction:

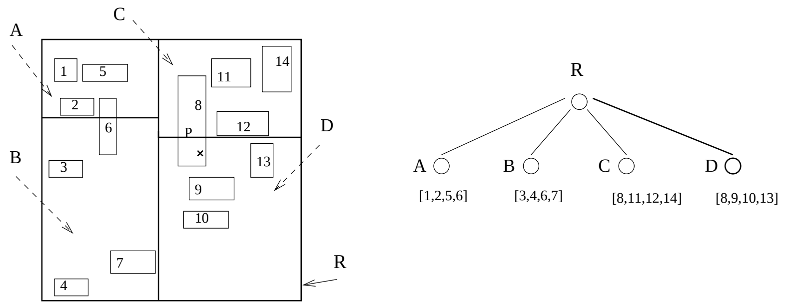
(a) A top down process: leaves are obtained using a recursive binary space partition a la spatial k-d-tree [OMSD87];

(b) A bottom-up process: 2^k leaves MBR's are grouped into an internal node (their father) where $2^k < t < 2^{k+1}$ and t is the node size (the number of tuples [MBR,leaf oid] that can fit into a page). 2^k internal first level nodes are in turn grouped into second level nodes, etc, until the root is reached.

Figure 3 describes the R*tree and the R+tree as constructed on the same set of objects as for the Rquadtree (Figure 1); in both cases the part of the tree which has been scanned for a solution is in bold lines.



A) Rtree Point Search - Threshold t=4. B, C and D are scanned - The result is 8



A) R+tree Point Search - Threshold t=4. Only D is scanned - The result is 8

Figure 3: *An example of Rtree and R+tree*

We now give the point search algorithm for both structures:

```
Procedure LOCATE(P,M)
INPUT :   Point P, Root M
OUTPUT:List
If        M is Leaf then
          For all N in MBR(M) /* list of tuples in node M */
          if P in N then
          List += N;
Else
          For all N in MBR(M)
          if P in N then        /* SON(N) is the son*/
          LOCATE(P,SON(N));/* component of tuple N */
```

With the Rtree, at each step of the recursion possibly more than one node contains P (more than one subtree is then traversed). For the R+tree, all tests in internal nodes are useful. At each (recursion) step only one subtree is scanned: once P in N is true, it is not necessary to continue scanning MBR(M). There is a single path from root to the leaf. However all MBR's of a leaf are scanned. Recall the price to be paid is duplication of objects. We call *duplication rate, the ratio of number of tuples [MBR,oid] actually stored in the index leaves over the number of indexed objects in the database.*

3 Measurement environment

The four data structures described in Section 2 were implemented with version 4.4 of the object-oriented DBMS O_2 [BDK92, O291], on a Sun SPARCstation 10, model 30 (30Mhz, 86.1 MIPS, 32 Mbytes RAM). The indices were coded in O_2C, the O_2 development language [BDK92].

We started from a prototype called GeO2 [DRSM93] written in O_2C and provided by IGN which already included the implementation of the Rtree [Ray90]. GeO2 was upgraded with the three other SAM: Rquadtree, Nquadtree and R+tree. As a dataset, we used a thematic map called ACPH and provided by IGN (Figure 4).

ACPH is a partition of the 2D space in 6865 disjoint polygons. Its size in the IGN exchange format is 1.9 Megabytes. The distribution of the polygon locations in the 2D space is uniform (or at least very regular) but the polygons have varying size. Since datasets larger than ACPH were not available in the GeO2 version utilized, in order to check whether the results obtained from the experiment with ACPH remain valid with larger datasets, we generated from ACPH a 500 Megabytes dataset refered to as 256ACPH (Figure 5) as follows: 2ACPH includes ACPH polygons as well as overlapping objects obtained from ACPH by a translation on the x-axis such that 10 % of the objects MBR's overlap. 4ACPH is obtained from 2ACPH by a translation on the y axis such

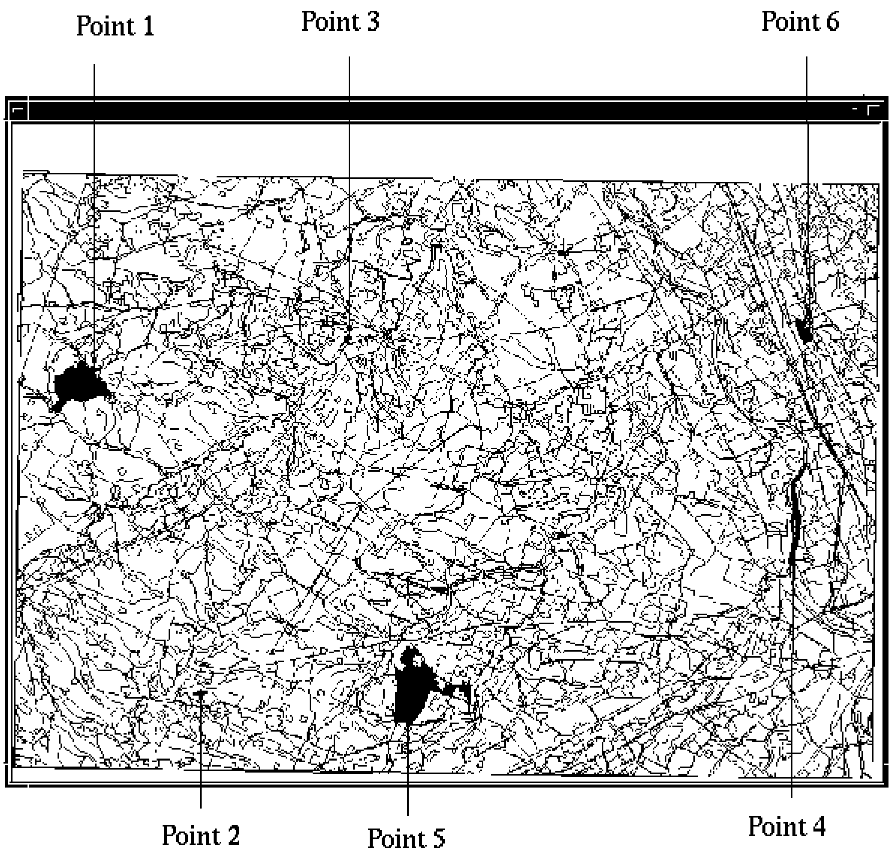


Figure 4: *Dataset ACPH*

that 10 % of the objects MBR's overlap, and so on. The final percentage of overlapping objects with 2^iACPH when i is large approaches 10 %. 256ACPH contains 1 757 440 polygons and has a size of almost half a Gigabyte (486 Megabytes in ASCII format).

Clearly, our only objective was to easily get *any* dataset whose size approaches half a gigabyte. The distribution of the polygons location and their size might have some impact on the results, however we believe that they would not change significantly (by an order of magnitude) the results reported below.

Each of the 4 index structures was constructed from ACPH for 6 node sizes. Recall the node size, also called *threshold* and denoted t is a) the maximal number of tuples in a leaf of the Rquadtree or the Nquadtree, or b) the maximal number of tuples in leaves as well as in internal nodes of the Rtree and the R+tree. For 256ACPH a single index was constructed for the R(N)quadtree

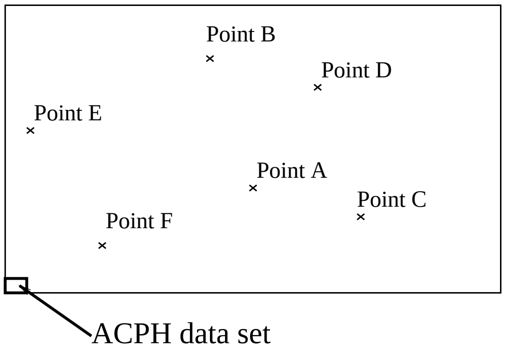


Figure 5: *Dataset 256ACPH - Location of points used for point querying*

and the R+tree: a node size t= 130 was chosen since it is the closest to the O_2 page capacity (4 Kbytes).

The first part of the experiment consisted in measuring for each constructed index some figures such as index depth, number of nodes, duplication rate,... (see Sections 4 and 5). The second part evaluates the indices' access performance for point queries on the 2 data sets (ACPH, 256ACPH). The same 6 points were used throughout all experiments for each dataset. The points (1 to 6) for ACPH are identified on the hard copy of ACPH in Figure 4 by the surrounding polygons (solutions of the point query). The points have been chosen so that the subtrees containing any two points have as few nodes in common as possible (hopefully only the root). For 256ACPH, measurements are shown for the six points (A to F) identified in Figure 5.

For each tuple [dataset, index, node size, point] several measurements were made such as cpu time, total time (cpu + I/O time), number of page requests in the two phases of the point query, namely search and selection. Depending on the point, the number of polygons to be tested (PointInPolygon) in the selection phase with the given datasets is at most three.

The O_2 version used was implemented as two processes on the same processor: the client and the (page) server processes. For a discussion on the O_2 DBMS architecture and its memory management, see [O291]. When accessing an object, two cases may occur: (i) either the object is already in the client memory (ii) or not: then the client first looks for the page containing the object in the client cache (4Mbytes). If it is not in this cache, then a *client page* request is delivered to the server. The latter then looks into its cache. If the page is there, there is a memory to memory copy of the page from the server cache into the client cache. Else there is a server page fault and the page is read from disk and transfered to the client. Three types of measurements were performed:

1. cpu time: the time measured is that of the client process. We had no access to the source code of the server process and therefore could not measure the server cpu time. However, the latter is included in the measured total time. Furthermore, compared to the client cpu time the server cpu time should be much less important.

2. total time: this time is the sum of the client cpu time plus the time to deliver objects when there is a (client) page fault (i.e. server cpu time + I/O in case of server page fault). Note that the client cpu time includes the time for processing a page read from the server in the case of page fault (creating an entry in the objects table, reading the object,...).

3. number of client page requests: we checked in our experiment that the number of client page faults is always equal to the number of server page faults, i.e., each time a page is requested from the server, it was not in the latter's cache and therefore each client page fault generated a disk I/O.

Mapping tree nodes to pages and clustering with O_2C

With the current version of O_2C, the programmer has *no control at all* on which page a newly created (O_2) object will be located. Worse the programmer has *no primitive for clustering* in a single page a given set or list of objects or values (two neighbour elements in a list are not guaranteed to be in a single page). This has the following consequences: (i) it is impossible to map the contents of an index node onto a single page, and (ii) it is impossible to implement any clever paging algorithm for data structures such as the Rquadtree whose internal node size is by far smaller than the page capacity[5].

Later versions of O_2C will provide the user with such clustering facilities. But this current limitation of the O_2 system is a serious drawback for developing performant indices on top of the kernel and it implies that the actually implemented indexes are not properly the indexes as refered to in the literature, since except for the Rquadtree, all SAM imply that a node is mapped onto a page. This is not a drawback when there is no page fault (depending on the load, the paging policy and the system architecture accessing a node -page request- does not automatically imply a page fault: the page could be memory-resident). Then under light load conditions, the indices behave as "central memory" data structures and of course the Rquadtree is expected to perform best. Under high load conditions on the contrary, we shall observe a discrepancy in the response time since *the number of page faults might be much larger than the tree depth*. Indeed, there might be a page fault not only when one goes from one node to its son, but also several page faults inside a node occur when one sequentially scans the list of tuples of the node.

[5]We shall see however that for the R(N)quadtrees we were able to perform some limited node clustering by "packing" the index after its construction.

With clustering facilities and a good mapping of nodes onto pages, the number of visited nodes during point query gives an upperbound on the number of page faults (and therefore on the I/O time: with the current configuration, a page fault approximately takes 35ms). Then with the Rquadtree, the number of page requests (and therefore the number of page faults) should be much smaller than the number of visited nodes. For the other SAM, the number of page requests should exactly be the number of visited nodes.

A consequence of this is that (i) even for this very limited experiment (a single operation, a single data set), no conclusion can be drawn on comparing data structures on response time, but (ii) *relevant performance criteria are the tree depth and the number of visited nodes*.

Measurement methodology

Last but not least we describe our measurement methodology. The memory allocated to the client is far from being saturated and no swapping occurs. Then page faults occur only upon the first reference to an object and not because there is no more available space in main memory. Each result given below for the indices access is the average of five consecutive experiments (a less than fifteen per cent standard deviation was observed) with the *same parameters* (dataset, threshold):

Each of the five experiments was done in the same experimental conditions on the same single user stand alone processor: The client process was connected. A point query search time was performed on the first point and the associated measures were collected. Then the same measures were collected on the second point query, and so on. Then the transaction and the O_2 application were ended, the client and server processes were disconnected and connected again (a simple commit does not empty the client cache) for the second of the five experiments which includes the same point queries (parameters unchanged) as the first experiment, and so on. As expected, a singular behavior is observed for the first point (point 1) for which the methods code is loaded as well as the upper levels of the tree (the 4Mbytes client cache is far from being saturated). The five following point queries (points 2 to 6) have similar behaviors. This is why, because of space limitations, only measures for points 3,4 and 5 are reported. Results for the other points can be found in [Rey93]. As we shall see below for all indices, query on point 5 is a bit faster. This is due to the loading of more nodes prior to the execution of point 5 query (point 5 is close to point 2). It was checked, of course, that upon repeating the same point query twice without ending the transaction, the number of page requests is null.

4 Rquadtree and Nquadtree evaluation

4.1 Index construction

Figure 6 summarizes the measurement results related to the Rquadtree construction.

a) ACPH (2 MO)
polygons nb : 6865

b) 256 ACPH (512 MO)
polygons nb : 1 757 440

Threshold t	15	30	80	130	200	130 (b)
Nb of stored objects	23993	14778	10373	9255	8531	2 442 843
Duplication Rate R	3,49	2,15	1,51	1,34	1,24	1,39
Max depth	10	8	6	5	5	10
leaf rate / level 4 (%)			1 %	25 %	27 %	
leaf rate / level 5 (%)	0,2 %	1 %	88 %	75 %	73 %	
leaf rate / level 6 (%)	22 %	75 %	11 %			
leaf rate / level 7 (%)	57 %	16 %				
leaf rate / level 8 (%)	18 %	1 %				22 %
leaf rate / level 9 (%)	2 %					78 %
leaf rate / level 10 (%)	0,3 %					0,3 %
Nb of Nodes	3552	1208	352	188	104	52 643
Nb of leaves	2665	907	265	142	79	39 487
Nb of objects /leaf	9	16	39	65	108	62

Figure 6: *Rquadtree construction for different thresholds and datasets*

For the ACPH dataset, for each node size ranging from 200 to 15, an Rquadtree index was constructed. For 256ACPH, only one Rquadtree was constructed for a node size of 130 (corresponding to an O_2 page size). In each case, the total number of objects (MBR) actually stored in the tree leaves, the duplication rate R, the depth distribution, the number of nodes, the number of leaves and the average number of objects per leaf are given.

With ACPH, when the node size decreases, the depth and the rate of duplication increase. The maximal depth ranges from 5 (node size 200) to 10 (node size of 15). The unbalance of the tree (difference between the minimal depth and the maximal one) is of course larger when the node size is small: 5 when the node size is 15. But we shall see below that this unbalance has no influence on the performance with these datasets and the current implementation. When the node size is large, the unbalance is reduced to 1. The duplication rate ranges from 3.5 when the node size is small to 1.25 when it is large.

256ACPH contains 1 757 440 polygons. The actual number of stored MBR's is 2 442 843 (duplication rate R= 1.39). The depth ranges from 8 to 10 with 78% of the leaves at depth 9. This relative balance however is due to the way 256ACPH was generated. As expected, when the dataset size is multiplied by 256, the depth is only doubled.

a) ACPH (2 MO) **N=256**
polygons nb : 6865

Threshold t	15	30	80	130	200
Nb of stored objects	20102	13019	9578	8823	8367
Duplication Rate R	2,92	1,89	1,39	1,28	1,21
Max depth	4	3	3	2	2
leaves rate / level 2	0,5 %	13 %	92 %	100 %	100 %
leaves rate / level 3	99 %	87 %	8 %		
leaves rate / level 4	0,5 %				
leaves rate / level 5					
Nb of Nodes	2027	774	178	95	63
Nb of leaves	1774	594	170	94	62
Nb of objects /leaf	11	22	56	94	135

b) 256 ACPH (512 MO)
polygons nb : 1 757 440

Threshold t	130
Nb of stored objects	2 302 247
Duplication Rate R	1,31
Max depth	4
leaves rate / level 2	
leaves rate / level 3	98 %
leaves rate / level 4	2 %
leaves rate / level 5	
Nb of Nodes	25 503
Nb of leaves	25 229
Nb of objects /leaf	91

c) ACPH (2 MO) **N=64**
polygons nb : 6865

Threshold t	15	30	80	130	200
Nb of stored objects	20110	13023	9579	8727	8319
Duplication Rate R	2,92	1,89	1,39	1,27	1,21
Max depth	4	4	3	3	3
leaves rate / level 2			2 %	41 %	84 %
leaves rate / level 3	82 %	99 %	98 %	59 %	16 %
leaves rate / level 4	18 %	1 %			
leaves rate / level 5					
Nb of Nodes	1977	659	231	119	68
Nb of leaves	1787	592	170	92	62
Nb of objects /leaf	11	22	56	95	134

Figure 7: *Nquadtree construction (N=64 and N=256) for different thresholds and datasets*

Figure 7 gives the measurement results for the Nquadtree construction. We constructed Nquadtrees with N=64 and N=256 for ACPH and with N=256 for 256ACPH. For all values of N, one index was constructed per node size. The same node size values were chosen as for the Rquadtree. Note that for ACPH, with N=64 the tree is one level deeper than with N=256. As for the Rquadtree, the maximal depth is only doubled from ACPH to 256ACPH (N=256).The Nquadtree (as Hierarchical Excell) was designed so that a node is mapped onto a single page. Therefore N (a power of 2) should be such that the size of N pointers is closest to the page capacity. In the O_2 environment, this is achieved with N=256.

For a given couple (dataset, node size value), as could have been expected, the depth is significantly lower than with the Rquadtree. As an example, with ACPH and a node size of 130, the maximal depth is 5 (ACPH) or 10 (256ACPH) with the Rquadtree, while the depth is 2 (ACPH, N=256), 3 (ACPH, N=64) or 4 (256ACPH, N=256) with the Nquadtree.

4.2 Rquadtree search performance

Figure 8 gives the index search performance on ACPH for points 3, 4 and 5. The tables give for each point and for each threshold two measures for the cpu time, two measures for the total time, the number of page requests and the depth of the leaf.

The first noteworthy result is that the number of page requests is extremely large compared to the depth! For a threshold equal to 200 it is even 5 or 6 times as large as the depth. This is due to the fact that we had no control at all on the clustering of neighbour nodes in the tree. In particular when a leaf overflows, its list of tuples gives birth to 4 new lists which are not necessarily stored in the same page as their father. Besides, since an internal node size is that of 4 pointers and therefore is much smaller than a page, a clever index internal nodes clustering (a mapping of the tree nodes onto pages) is necessary.

Point 3

Threshold t	15	30	80	130	200
CPU exp. 1	30	50	40	60	60
CPU exp. 2	30	30	30	60	60
Total time exp. 1	310	600	430	400	850
Total time exp. 2	90	180	110	170	150
page requests	8	16	11	10	21
depth	7	6	6	5	4

Point 4

Threshold t	15	30	80	130	200
CPU exp. 1	30	40	40	40	60
CPU exp. 2	20	20	40	40	60
Total time exp. 1	380	350	430	1230	1050
Total time exp. 2	90	90	110	320	160
page requests	10	9	10	31	24
depth	6	6	5	4	4

Point 5

Threshold t	15	30	80	130	200
CPU exp. 1	30	35	40	50	50
CPU exp. 2	20	35	30	30	50
Total time exp. 1	250	490	360	240	660
Total time exp. 2	110	160	80	120	100
page requests	6	13	7	5	17
depth	6	5	5	5	4

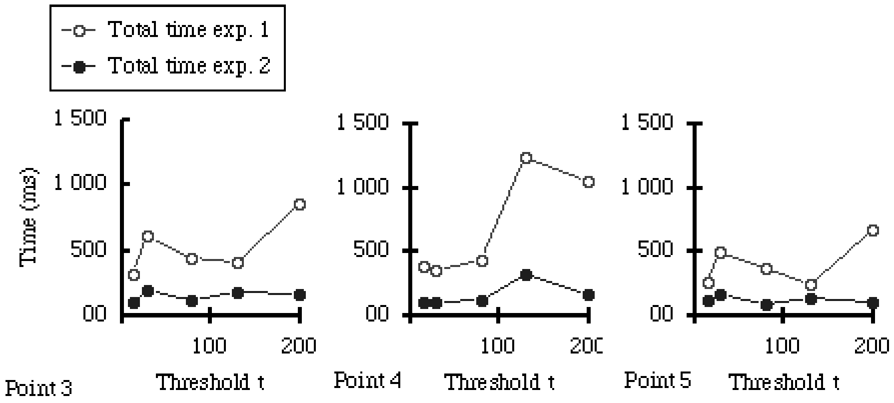


Figure 8: *Rquadtree search performance, ACPH dataset, Total time versus threshold t. Experiment 1: The server is disconnected between two series of measures. Experiment 2: The server is not disconnected*

Two measures were performed for each point and each threshold: in the first case as explained in section 3, the figures shown are an average of 5 series of measures: each series consists in a sequence of 5 point queries (one for each point) with the same index (the same threshold). Between two series, not only the transaction and the O_2 application are ended but also the O_2 server process is disconnected. In the second case, the only difference is that between two consecutive series of measures on a single threshold, we do not disconnect the server. Then *pages are kept in the server's cache* between two series of measures. In the latter case we discard the first series (we keep for each point and each threshold only four series of measures).

The number of page requests is the same in both experiments but the total times are extremely different. In the former experiment (the server process is disconnected between two series), to each page request corresponds a default page and the total time is significantly higher than in the latter case, since there are actually more disk accesses. The CPU time slightly changes also because part of it accounts for processing page requests. Both total times are plotted versus the threshold for each point. In the sequel of the paper whatever index is measured, in order to stick as much as possible to worst case conditions (the index has to be completely load from disk), we shall always disconnect and connect again the server process in between two series of measures.

Point 3

Threshold t	15	30	80	130	200
CPU	30	50	40	60	60
CPU with post-packing	30	30	40	60	70
Total time	310	600	430	400	850
Tot.time with post-pack:	240	270	285	320	390
page requests	8	16	11	10	21
page requests with p.-p.	6	6	7	7	8
path depth	7	6	6	5	4

Point 4

Threshold t	15	30	80	130	200
CPU	30	40	40	40	60
CPU with post-packing	30	30	40	50	60
Total time	380	350	430	1230	1050
Tot.time with post-pack:	270	270	280	360	380
page requests	10	9	10	31	24
page requests with p.-p.	7	7	7	8	8
path depth	6	6	5	4	4

Point 5

Threshold t	15	30	80	130	200
CPU	30	30	40	50	50
CPU with post-packing	20	40	30	30	50
Total time	250	490	360	240	660
Tot.time with post-pack:	90	160	65	150	220
page requests	6	13	7	5	17
page requests with p.-p.	2	3	1	3	4
path depth	6	5	5	5	4

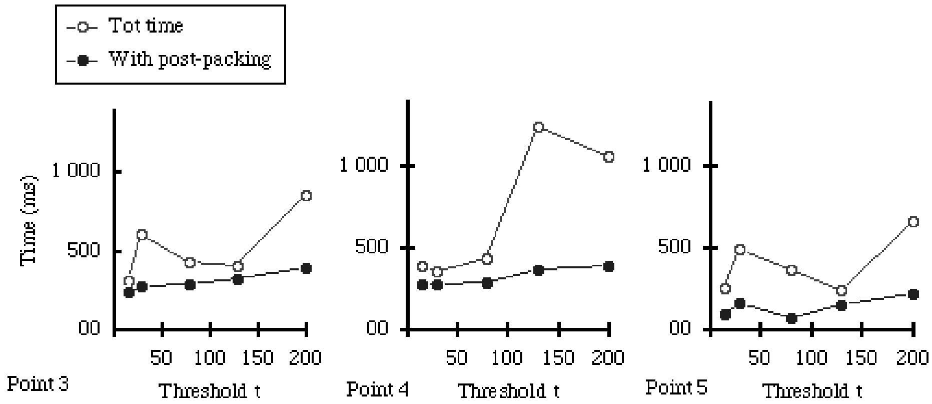


Figure 9: *Impact of Rquadtree post-packing on the tree search performance*

From this experiment it is clear that eventually either the index is developed under the API interface or the capability of mapping objects (nodes, tuples, lists,...) to pages must be provided to the O_2C developer. However we were able to significantly improve clustering by some "post-packing" of the index after its construction. The resulting scheme is called clustered Rquadtree. We start from an already constructed index. The tree is then traversed breadth first and copied node per node into a new tree (new O2 objects created in sequence have better chances to be clustered in a single page). Figure 9 displays for ACPH and points 3, 4 and 5 the total search times in both cases: post-packed and original Rquadtree.

The number of page requests is now of the order of the leaf depth except for high values of the threshold (e.g. for point 4 and t= 200 the number of page requests is 8 while the depth is 4). For point 5 it is even smaller than the depth. Compared to the unpacked Rquadtree, the saved number of page requests is variable: e.g. three times less for point 4, t=200, but only .5 times less for point 3, t=130. Recalling that some CPU time is devoted to page request processing, then clustering also improves CPU time. The total time (CPU + I/O) gain can be as large as almost 400% (point 4, t=130)!
Such a post-packing of course cannot be done with dynamic datasets. It provides a fast prototyping in O_2C of an Rquadtree with an acceptable performance for point querying. We shall see below that this performance remains acceptable even with a large dataset. From now on we assume the Rquadtree is post-packed.

Point 3

Threshold t	15	30	80	130	200
CPU	30	30	40	60	70
Total time	240	270	285	320	390
nr	7	19	19	113	177
path depth	7	6	6	5	4
page requests	6	6	7	7	8

Point 4

Threshold t	15	30	80	130	200
CPU	30	30	40	50	60
Total time	270	270	280	360	380
nr	14	14	38	118	118
path depth	6	6	5	4	4
page requests	7	7	7	8	8

Point 5

Threshold t	15	30	80	130	200
CPU	20	40	30	30	50
Total time	90	160	65	150	220
nr	6	20	20	20	141
path depth	6	5	5	5	4
page requests	2	3	1	3	4

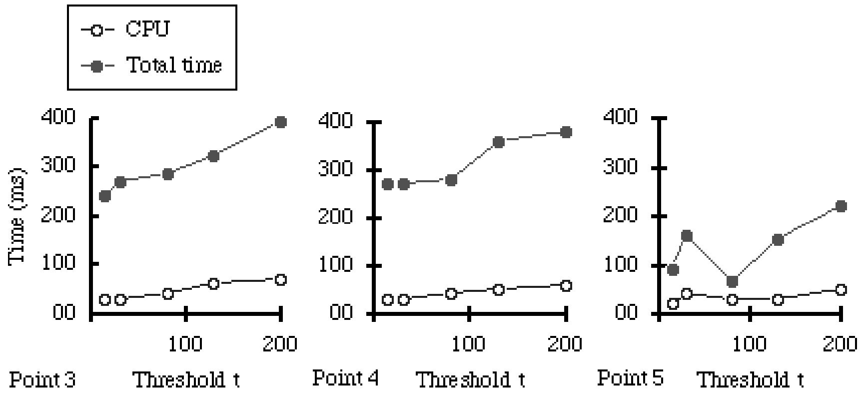


Figure 10: *Rquadtree search access performance, ACPH dataset, CPU and total time versus threshold t*

The search CPU and total times are compared in Figure 10. The CPU and total times are plotted versus the threshold t. The performance is I/O bound.

Although small thresholds provide a better cpu time, the latter is not clearly monotonic in t. As a matter of fact - and this is true for all indices - the important parameter is the number of scanned leaf tuples nr. The CPU times are increasing with nr.

Points	Pt A	Pt B	Pt C	Pt D	Pt E	Pt F
CPU	70	50	60	50	60	50
Total time	340	370	230	400	300	320
nr	91	98	107	122	51	114
path depth	8	8	8	8	8	8
page requests	7	9	8	7	5	6

Figure 11: *Rquadtree search access performance, 256ACPH dataset, CPU and total time*

Figure 11 gives the search performance of the Rquadtree constructed from 256ACPH for various points (see Figure 5). A threshold t= 130 corresponding to the maximal number of tuples [MBR, oid] that could be stored in a page has been chosen.

Recall that 256ACPH holds 1 757 440 objects and its size is almost half a gigabyte. The (not so) surprising result is that the performance is the same as that of the Rquadtree on ACPH (see Figure 10, except for point 5: total search time= 150ms for t= 130). The leaf depth for all points is 8, i.e. twice the depth of the points queried with ACPH (increasing the dataset size by 256 results in doubling the tree depth). Clearly the depth has less importance than the number of page requests which is almost the same as for ACPH and of the order of the leaf depth. It is even much smaller for point G (5) than the depth (8). Compared to ACPH, the CPU time has not changed.

The search phase performance is compared to the selection phase performance in Figure 12. Various measures are displayed versus the node size (threshold t) for points 3,4 and 5: (i) the CPU time, (ii) the total search time, (iii) the total search time plus the selection CPU time and the index response time (search + selection).

Recall the selection time (PointInPolygon) is index independent: it depends only on the point and the polygon size. Looking at point 3 the result list from the search phase includes 3 polygons to be tested. Only one includes the point. The selection time is very significant: it is twice as large as the index search time on the average. Its complexity is of course dependent on the number of polygons tested and the number of edges in each polygon. For Point 3, the CPU time (PointInPolygon test) is of the same order as the search CPU time but the number of page requests is extremely high. Once again, this is due to the fact that no clustering is possible among component objects (edges of a polygon).

For point 5, although only one polygon has to be tested, the selection time

Point 3

Threshold t	30	80	130	200
CPU time search	30	40	60	70
Total time search	270	285	320	390
CPU polygon selection	80	80	80	80
Tot time selection	640	640	640	640
page requests search	6	7	7	8
page requests selection	17	17	17	17
polygons tested	4	4	4	4

Point 4

30	80	130	200
30	40	50	60
270	280	360	380
110	110	110	110
700	700	700	700
7	7	8	8
22	22	22	22
3	3	3	3

Point 5

30	80	130	200
40	30	30	50
160	65	150	220
90	90	90	90
600	600	600	600
3	1	3	4
17	17	17	17
1	1	1	1

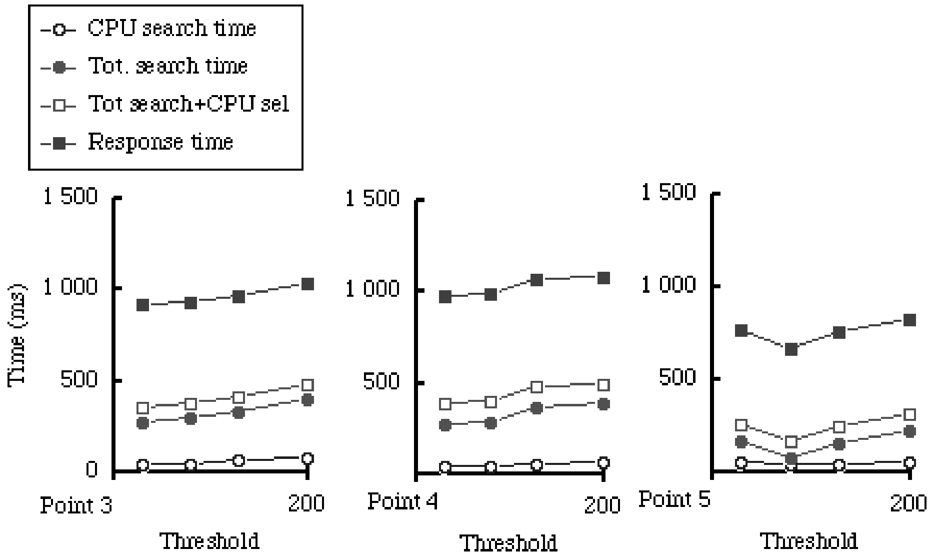


Figure 12: *Rquadtree, comparaison between search and selection, ACPH dataset*

is approximately the same as for point 3: 17 page requests and 600ms: PointIn-Polygon is linear in the polygon number of edges. The polygon tested for Point 5 has approximately as many edges as the three polygons tested for inclusion for Point 3.

Among all above results which are of course dependent (i) on the implementation, (ii) on the dataset, the most salient one is the need for clustering. Not only tree nodes need to be clustered but also the geometric objects components of the polygons searched for. The above results should of course be conforted by experiments on datasets with other distributions of MBR's in the 2D plane. In particular, the dataset used for our experiment corresponds to a very regular distribution of the MBR's. What happens when the distribution is skewed? For the time being we have performed the same experiment on the much larger dataset 256ACPH. As ACPH, 256ACPH is extremely regular and therefore the above results should be looked at with extreme care. However they show at least that when the size of the dataset is multiplied by 256, the performance degradation is negligible.

4.3 Nquadtree search performance

Figures 13 and 14 give the Nquadtree search performance for ACPH with N=64 and with N=256 for points 3,4 and 5. CPU and total search times are plotted versus the threshold.

Point 3

Threshold t	15	30	80	130	200
CPU	30	30	40	40	40
Total time	330	330	320	340	360
nr	13	13	48	123	177
path depth	3	3	3	3	2
page requests	8	8	8	8	9

Point 4

Threshold t	15	30	80	130	200
CPU	30	30	30	20	20
Total time	290	340	330	270	230
nr	14	17	69	118	118
path depth	3	3	3	2	2
page requests	7	8	8	7	6

Point 5

Threshold t	15	30	80	130	200
CPU	30	30	30	30	40
Total time	180	190	180	180	150
nr	13	20	53	53	141
path depth	3	3	3	3	2
page requests	4	4	4	4	3

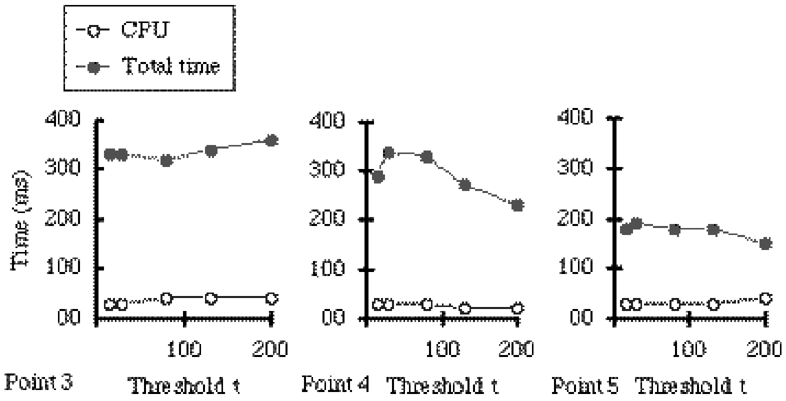


Figure 13: *Nquadtree N=64, search access performance, ACPH dataset, CPU and total time versus threshold t*

The Nquadtree was clustered in the same way as the Rquadtree was. The small depth (of the order of that of the Rtree and the R+tree, and twice as small as for the Rquadtree) unfortunately does not lead to a better performance. Indeed for example for N=256, the number of page requests is still around 8 or 9 (slightly more than for the Rquadtree, see Figure 10) while the depth is only 2 (N=256) or 3 (N=64). The larger N, the smaller the depth, the better the expected search performance. Here the value of N has a small influence on the search performance, since the number of page requests does not change much with N. Line nr in Figures 10, 13 and 14 give the number of tuples in the leaf. Note that nr is slightly larger with the Nquadtree than with the Rquadtree. This confirms that the expected leaf space utilisation is better with the Nquadtree than with the Rquadtree. Indeed, an Rquadtree quadrant is split into four quadrants while an Nquadtree leaf is split first into four leaves but then in two leaves only (as for the k-d-tree). Hence, scanning a leaf should eventually be slower with the Nquadtree for very large leaf sizes.

Point 3

Threshold t	15	30	80	130	200
CPU	20	20	40	40	30
Total time	330	330	320	350	380
nr	13	13	48	123	177
path depth	3	3	3	2	2
page requests	9	9	8	9	10

Point 4

Threshold t	15	30	80	130	200
CPU	30	30	30	20	20
Total time	280	340	340	270	300
nr	14	17	69	118	118
path depth	3	3	2	2	2
page requests	7	8	8	7	7

Point 5

Threshold t	15	30	80	130	200
CPU	30	30	30	30	40
Total time	135	100	170	190	280
nr	13	20	53	53	141
path depth	3	2	2	2	2
page requests	3	2	4	4	5

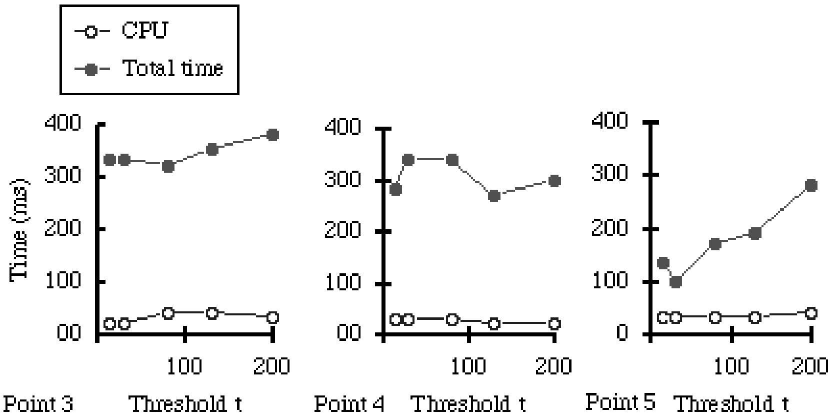


Figure 14: *Nquadtree N=256, search access performance, ACPH dataset, CPU and total time versus threshold t*

Figure 15 gives the performance of the Nquadtree (N=256) with 256ACPH and a threshold t= 130.

Points	Pt A	Pt B	Pt C	Pt D	Pt E	Pt F
CPU	20	20	50	50	40	30
Total time	280	420	440	360	320	300
nr	91	130	118	110	95	46
path depth	3	3	3	3	3	3
page requests	7	10	10	8	7	7

Figure 15: *Nquadtree,N=256, search access performance, 256ACPH dataset, CPU and total time*

Compared to ACPH, the depth is 3 while it was only two for ACPH, t= 130 (Figure 14). This leads to a larger number of page requests (from 7 to 10) and therefore a slightly larger total search time. Compared to the Rquadtree performance on the same points and the same dataset (Figure 11), the Nquadtree performance is slightly worse for the same reasons: on the average there are two more page requests with the Nquadtree.

But if the Nquadtree nodes were mapped onto pages, this index (as Hierarchical Excell) would definitely outperform the Rquadtree. In our case (Figure 15), an upperbound on the number of page faults would be 3, while with the Rquadtree, there might be in the worst case eight page faults if no clever paging algorithm were designed. Another advantage of such a structure is its good behavior in the presence of a skewed polygon distribution. The price to pay is an increased complexity.

5 Rtree and R+tree performance

Figure 16 lists the results from the construction of Rtree and R+tree indices.

ACPH (2 MO)
polygons nb : 6865

Threshold t	15	30	80	130	200
Nb of stored objects	6865	6865	6865	6865	6865
Duplication Rate R	1	1	1	1	1
Depth d	4	3	3	2	2
Nb of Nodes	770	360	122	75	52
Nb of leaves	698	344	119	74	51
Nb of objects/leaf	10	20	58	93	164

Table 1 Rtree Construction for ACPH (different thresholds)

ACPH (2 MO)
polygons nb : 6865

Threshold t	15	30	80	130	200
Nb of stored objects	20250	13061	9368	9148	8435
Duplication Rate R	2,95	1,9	1,36	1,33	1,23
Depth d	5	4	3	2	2
Nb of Nodes	2240	585	132	113	65
Nb of leaves	1652	514	128	112	64
Nb of objects/leaf	12	25	73	82	132

256ACPH (512 MO)
polygons nb : 1 757 440

130
2 407 680
1,38
4
31 154
30 865
78

Table 2 R+tree Construction for ACPH (different thresholds) and 256ACPH

Figure 16: *Rtree and R+tree construction*

Each table corresponds to a given dataset (ACPH or 256ACPH). As for the quadtree-like structures, indices were constructed for various thresholds t in the case of ACPH and only for t=130 with 256ACPH. We give in each table, the tree depth d, the number of nodes, the number of leaf nodes , the duplication rate R (R+tree) and the average number of objects per leaf.

From a first look at these tables, one may note that the tree depth is rather low. The depth ranges from 2 ($t = 200$) to 4 ($t = 15$) for ACPH. When t is small, the duplication of objects in the R+tree is large (R=3 with t=15) and has some impact on the depth: the R+tree is one level deeper than the Rtree. With t=130, the duplication rate is small (R= 1.33). Increasing the dataset size by a factor of 256 only doubles the R+tree depth: with 256ACPH and t=130, d=4.

We now discuss the indices' access performance. Measures were done on the same set of 6 points (Figure 5). Point query includes two phases: search and selection. We only consider index search time, i.e., the time to go from the tree root to the leaves (Figures 17 and 18).

	Point 3					Point 4					Point 5				
Threshold t	15	30	80	130	200	15	30	80	130	200	15	30	80	130	200
CPU	40	120	50	40	50	30	30	100	40	60	30	30	60	50	50
Total time	340	780	200	220	250	450	460	560	220	290	330	320	140	120	120
Total nr	63	168	148	196	265	66	78	220	257	326	83	87	322	299	322
nb of nodes n	8	8	4	3	3	8	4	4	3	3	8	4	6	3	3
Tree depth	4	3	3	2	2	4	3	3	2	2	4	3	3	2	2
page requests p	8	18	4	5	6	12	12	13	5	7	7	8	2	2	2

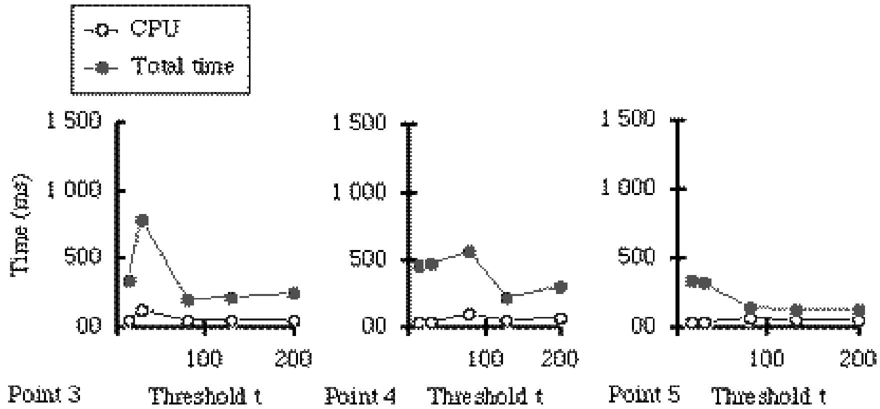


Figure 17: *Rtree search access performance, ACPH dataset, CPU and total time versus threshold t*

Recall that the selection phase consists in checking whether each geometric object whose MBR is in the result of the search phase actually contains the point (PointInPolygon). This time is independent from the index: it depends only on the point and was already given for the quadtree-like structures (Figure 12).

The tables in Figure 17 give for the Rtree, for each node size t and for each point query, the cpu search time, the total time (cpu time +I/O time), the total number nr of rectangles scanned[6], the number of nodes n visited during

[6]As opposed to the Rquadtree, nr is equal to the number of tuples visited in internal

	Point 3					Point 4					Point 5				
Threshold t	15	30	80	130	200	15	30	80	130	200	15	30	80	130	200
CPU	20	20	40	30	50	30	30	70	70	40	30	30	20	30	20
Total time	500	610	800	690	670	550	700	1200	1450	980	280	320	60	70	100
Total nr	32	44	126	114	157	35	47	104	152	181	28	37	106	91	144
path depth	5	4	3	2	2	5	4	3	2	2	5	4	3	2	2
page requests	14	17	24	19	15	16	20	37	44	27	7	9	1	1	2

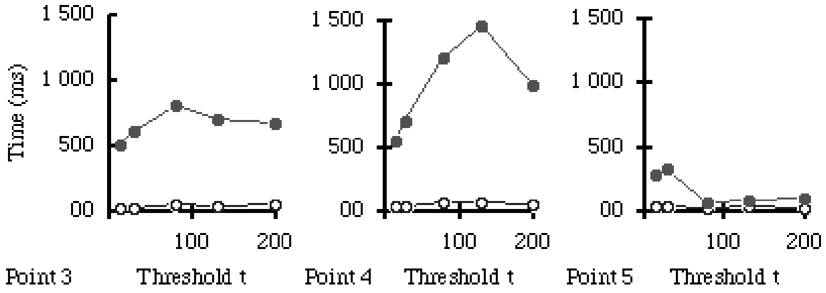


Figure 18: *R+tree search access performance, ACPH dataset, CPU and total time versus threshold t*

the search, the tree depth, and the number of page requests. The cpu and total time are plotted versus the node size t for each point.

The first observation that can be made is that as for the quadtree-like structures, the number of page requests p is much larger than the number of nodes visited n. For point 3 (t= 30) and point 4 (t=80) p is more than three times as large as n! This is due to the fact that we were unable to map nodes to pages. Therefore scanning a list of rectangles inside a node results in several page requests. This of course is unacceptable for an actual Rtree implementation. For a real life implementation a better performance criterion is the number of visited nodes n which is an upperbound (high load assumption) on the number of page faults. The second observation relates to the fact that in an Rtree search, several subtrees might be traversed (several paths from root to leaves are possibly followed). This has for a consequence that the number of visited nodes is possibly larger than the depth. As can be seen from the tables, the smaller t, the larger the redundancy rate (ratio number of visited nodes n over the depth d). Finally observe that the query on Point 5 is faster than those on points 3 and 4, as for the Rquadtree and the Nquadtree: the number of page requests is smaller since some of the pages have already been loaded.

We now turn our attention to the R+tree.The lack of clustering is even worse than with the Rtree: the number of page requests is of the order of 20 for point 3 and reaches 44 for point 4 (t=130)! It should be less or equal to

nodes + the number of tuples visited in leaf nodes, in a path from root to a leaf (possibly to several leaves).

the depth (d=2 for t=130) if the nodes were correctly mapped onto pages. As expected, the number of visited nodes (equal to the tree depth) is smaller with the R+tree than with the Rtree: for points 3 and 4, 1 visit is saved when t=130, 4 visits are saved when t=15 (see Figures 17 and 18). For a given threshold, the total number of scanned rectangles nr is significantly smaller with the R+tree than with the Rtree. This is not only due to the fact that the number of visited nodes is larger with the Rtree, but also because with the Rtree, all rectangles of a node are scanned while in the case of the R+tree, only half of them on the average are tested. The saving in cpu time has no impact on the performance here, since the performance is I/O bound. This might have some impact if, compared to the I/O time, the cpu time was more important.

Figure 19 gives the R+tree performance with 256ACPH (t=130) for the same points as for the quadtree-like structures (Figure 5).

Points	Pt A	Pt B	Pt C	Pt D	Pt E	Pt F
CPU	90	120	80	90	140	120
Total time	570	3 360	1 600	1800	2650	2480
Total nr	163	193	128	135	193	179
path depth	4	4	4	4	4	4
page requests	15	105	55	60	111	102

Figure 19: *R+tree, search access performance, 256ACPH dataset, CPU and total time*

The number of page requests is catastrophic. But, as mentioned above, multiplying the size by a factor of 256 only doubles the depth. Therefore, if the nodes were mapped onto pages, the upperbound on the number of page faults would only be doubled. Note that for the same value of the number of scanned rectangles nr, the cpu time is higher than with ACPH (Figure 18). This illustrates that part of the CPU time is accounted for processing page requests (which are more numerous with 256ACPH than with ACPH).

In conclusion, on the ACPH dataset, the R+tree clearly outperforms the Rtree: as predicted in the literature [FSR87], the disjoint decomposition provides a faster tree traversal. The duplication incurred, at least with this dataset, had no influence on the performance. Furthermore, the observed performance discrepancy when upsizing the dataset by two orders of magnitude should be negligible (given that nodes are mapped onto pages!).

6 A qualitative comparison

The purpose of this section is not to decide from our experiment which of the four data structures performs best on point queries but to enlighten their differences in performance. Because of the lack of clustering primitives in the O_2C language that was chosen for implementation and measurement, the four data structures exhibit a time discrepancy due to a too large number of page requests. Therefore we decided to choose, for all data structures but the Rquadtree, the number of visited nodes n times 35 milliseconds (i.e. 35xn ms) as a predicted measure of the I/O time : the number of page requests in our measurement was replaced by the the number of visited nodes actually measured. This allowed us to forecast an upperbound on the number of page faults in the case where the programmer is provided with facilities for mapping nodes onto pages. To obtain the total search time, we add to this "forecast" bound on I/O time the cpu time as measured and compare the four data structures on their total search time (i) with ACPH, for various thresholds t and points 3, 4, 5 (Figure 20) ; (ii) with 256ACPH, for points A to F, t= 130 (Figure 21, except for the Rtree).

For the Rquadtree, we chose the performance as measured and reported in Section 4, since it was difficult to forecast its performance in the case where the programmer would be provided with clustering facilities. In that case a nice paging algorithm should be designed. Of course, because of this choice, the Rquadtree suffers from the comparison in Figure 20: in spite of the post packing attempted with O_2, the measured search time of the Rquadtree is by far larger than the time forecast for the three other data structures. The necessity for a clever paging algorithm increasing the design complexity (one of the advocated advantages of the Rquadtree is its design simplicity) and its possible unbalance for skewed data distribution are expected limitations of the Rquadtree under very high load conditions. We put ourselves however in worst case conditions where each page request results into a page fault. There are of course common situations where the Rquadtree provides a simple and fast access with no discrepancy when the dataset size increases (see the discussion in Section 4).

Looking at Figure 20, clearly The R+tree outperforms the Rtree (see the discussion of the previous section). But the interesting outcome of the comparison is that the Nquadtree (Hierarchical Excell) and the R+tree are comparable. The possible unbalance of the Nquadtree had no impact on the performance in this experiment. The Nquadtree (as all quadtree-like structures) should be better adapted to spatial join operations because of their space decomposition relatively data independent. But the R+trees have been advocated to be well suited for point queries and range (window) queries as well.

The Nquadtree, the R+tree and the Rquadtree are compared in Figure 21 for 256ACPH. The Nquadtree has a slight edge over the R+tree. Indeed not only the Nquadtree is less deep (by one level) but also its number nr of tuples scanned is significantly smaller (see Figures 15 and 19).

	Point 3			Point 4			Point 5		
Threshold t	80	130	200	80	130	200	80	130	200
Nodes visited Rtree	4	3	3	4	3	3	6	3	3
Nodes visited R+tree	3	2	2	3	2	2	3	2	2
Nodes visited Rquadtree	6	5	4	5	4	4	5	5	4
Nodes visited Nquadtree	3	2	2	2	2	2	2	2	2

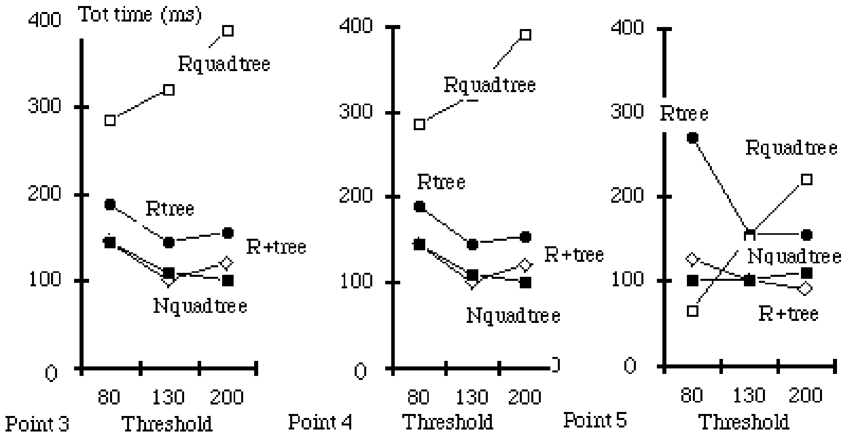


Figure 20: *ACPH, comparison between Rquadtree, Nquadtree, Rtree and R+tree*

Points	Pt A	Pt B	Pt C	Pt D	Pt E	Pt F
Nodes visited R+tree	4	4	4	4	4	4
Nodes visited Rquadtree	8	8	8	8	8	8
Nodes visited Nquadtree	3	3	3	3	3	3

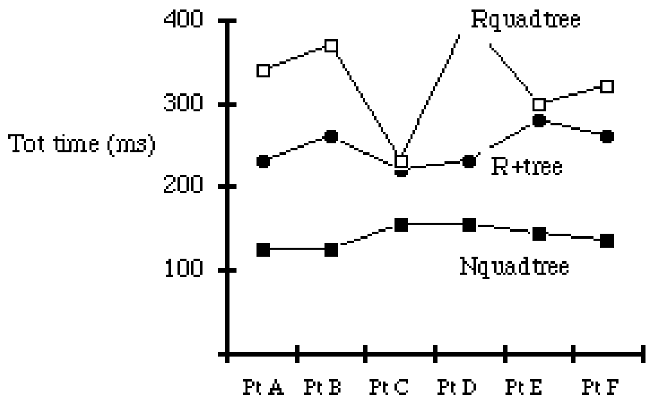


Figure 21: *256ACPH, comparison between Rquadtree, Nquadtree and R+tree*

7 Related work

Among the numerous recent performance analyses of spatial data structures [Tam83, Sam90, FSR87, Ooi90, Smi90, BPW93, GB91, Gre89, HS91, HS92] only a few [Smi90, GB91, HS91, HS92] address implementation issues on an actual DBMS. The study reported in [HS92] is closely related to our performance study. It is a qualitative comparison of implementations of a quadtree, an R*tree and a R+tree structures, with a large number of detailed empirical results. The size (in bytes) of their data sets is smaller than ACPH. On the other hand, the number of objects in their case is higher: 50 000 objects (line segments). Compared to our study, it is much more complete in terms of spatial operations evaluated (window queries, finding the nearest line segment, etc.). Another important difference is that the study has been restricted to line segments (while ours was to rectangles). Last, interestingly, the quadtree structure, a PMR quadtree [Sam90] is implemented as a linear quadtree: each line segment is identified by a tuple [L,oid] where L is a Morton code [Sam90] of the PMR quadtree leaf and oid points to the line segment. This tuple is stored on a B-tree indexed on L. Besides natural clustering through the B-tree, this implementation provides the advantage of using system facilities for query optimization, concurrency control and recovery already provided for the B-tree. However the authors report that the building complexity of their quadtree implementation was higher than that of the R+tree in terms of execution time. In terms of the spatial operations performance, the results do not seem to favor one SAM, although the PMR quadtree and the R+tree have a slight edge over the Rtree because of the disjoint decomposition of space the former induce.

8 Conclusion

In this paper we reported a measurement experiment for evaluating the access performance of point queries with several spatial data structures implemented on top of the object-oriented DBMS O_2 [O291] with the development language O_2C. Four data structures were tested, called Rquadtree, Nquadtree, R*tree and PBR R+tree. To our knowledge, the first two data structures are novel, although they are variants of known structures [Sam90, Tam83]. The third and the fourth data structures are well known.

Our objective was twofold:

1. contribute to the choice of Spatial Access Methods (SAM) to be implemented in geographic DBMS in order to accelerate spatial queries among which point and window queries are the most common.

2. understand where the performance bottlenecks are when implementing such structures with commercial DBMS.

To fulfill the first objective, the performance and design simplicity comparison between two types of structures (e.g. quadtree-like structures versus Rtree-like structures) was only one issue; among other fundamental issues were (i) to understand the relative importance of CPU time versus I/O time when traversing a spatial index, (ii) to measure the impact of the tree balance on the performance, etc...

The second goal implied the choice of a database technology and that of a specific system. We chose the object-oriented DBMS O_2 because of the ease it provides in developing and integrating new code. An important question to be answered was whether such SAM should be implemented for performance reasons at a low level inside the server or whether it was sufficient to implement it at the application programming interface (API) level in the client process.

As a first comment, the selection phase (testing which among a list of polygons contains the point), independent from the index turned out to be extremely expensive compared to the index traversal phase, not because of the geometric computation itself (as we first expected), but because we had no facility to cluster the geometric components of the polygons in a single page. As a consequence, I/O time in the selection phase was prominent compared to CPU time.

As a second remark, we have no claim on the superiority of any of the four data structures. Of course, we experimentally checked that the R+tree design was extremely complex, compared to that of the Rquadtree, or even that of the Nquadtree. But as far as response time is concerned even for our limited experiment (test of a point query on a single data set) we cannot decide which is best. As mentioned in [HS92], differing implementations of a given data structure might bias the comparison in favor of a given data structure: we are comparing implementations rather than data structures. In our implementations, because of the lack of clustering facilities at the API level, we were unable to correctly map index nodes onto pages (and thus to correctly implement the Nquatree, the Rtree and the R+tree). We improved the Quadtree structures clustering by post-packing the index. Nothing was done for the structures of the Rtree family but one possibility would be to use the packing techniques described in [RL85, KF93].

Therefore under high load conditions (the whole path from root to a searched leaf has to be loaded into main memory), the Nquadtree, the Rtree and the R+tree are disadvantaged in our design. However, measured performance criteria such as tree depth and number of visited nodes are good indicators of the actual performance. Their measurement confirms well known expectations on the behavior of the data structures. In particular the Nquadtree and the R+tree are comparable and should outperform the Rquadtree, especially in the case of skewed data distributions.

Under light load conditions (central memory indices), a complex paging algorithm is not worth. Then as expected the simple Rquadtree outperforms the other SAM's. Our experiment results show that, surprisingly with a large dataset (more than 1.5 million rectangles), its performance is still good (less than 800ms) in spite of its rough implementation. Its expected drawbacks (large depth, unbalance) have no influence on its overall performance. However the observed relative balance of the Rquadtree is due to the regular distribution of the MBR in the 2D plane.

Compared to the Rtree, the R+tree has a significantly better access performance. This is due to the large redundancy rate with the chosen data sets. This might be less significant with other data distributions. The fact that R+trees (as well as Rquadtrees) might have a large duplication rate, might lead to a performance degradation not to be expected with the Rtree.

The other results relate to the issues of a good implementation with an object oriented DBMS. Should one develop SAM's below the application programming interface (API)? Which system facilities do we require? A clear demonstration was made that clustering facilities are really needed at the API level. Then if the programmer is provided with clustering facilities, why should we write the index code at lower implementation levels? The only reasons for doing so are query optimization, concurrency control and recovery.

Other reasons for implementing the index at the system level would be to optimize the index code. This was not necessary for point queries. Indeed with our configuration, search (index tree traversal) as well as selection (PointInPolygon test) were I/O bound. Even with no page fault and a 500 megabytes data set, the index response time (cpu time) is only of the order of 200ms. Therefore clustering (especially among geometric components) is by far a priority to code optimization. However, more complex geometric operations (windowing, clipping, polygon intersection) might require code optimization. It is not clear anyway whether these geometric operations are part of the index code. In summary, for fast prototyping or in applications where there is no need for concurrency control, we claim that it is enough to implement indices at the programming level (provided there are clustering facilities at this level).

The partial results of this study should be strenghtened by other more complete experiments. Following are some challenging and necessary extensions to our work:

1. Only point queries were studied. How do these schemes behave in the presence of window queries, map overlay,... Rtree-like structures were designed for fast window queries. We expect, as mentioned in [HS92] that quadtree structures are better for complex operations such as map overlays because of their relative data independent decomposition of the 2D space.

2. The data used for this study was very specific. We already mentioned the possible impact of data type (line or region) on the performance (see in particular [HS92]). The space distribution and the size of the MBR's also have some impact on the index performance. It might then be interesting to simulate the load and therefore to design tools for generating data

with different MBR distributions and sizes. [Smi90] is an interesting performance study in that direction.

3. We restricted our choice of data structures to Quadtrees and Rtrees. However, there exist several other interesting structures to be looked at: among them let us quote the cell tree [Gün89], grid files [TS82, NHS84] (interesting when the objects distribution is uniform) or k-d-trees [Ben75, Rob81]. In particular k-d-B-trees are faster built than R+trees while point and window queries are comparable (see the discussion in [HS92]).

4. Last but not least, it is not clear to what extent the conclusions of this study are sensitive to the size of the data and to the DBMS architecture and more specifically to its memory management. With faster hardware, high memory loads, optimized algorithms, and very large datasets the index search might become even more I/O bound. However, as far as the dataset size is concerned, we have already checked on a sample data set that increasing the size by more than two orders of magnitude (i.e., increasing the indices depth by at most four for some structures) has only implied a small number of supplementary page faults which does not seriously change the results.

Acknowledgements
We are indebted to Benoit David and Laurent Raynal from IGN who have provided us with the prototype GeO2 from which we made our implementation and measurement experiments. Agop Gurunlian from O2Technology deserves special thanks for the numerous helpful discussions on the DBMS O_2 architecture. He also made first measures on the Rtree with an earlier version of GeO2 and O_2 [Gur90]. Thanks also go to O2Technology and in particular to L. Hyafil for their technical support in the experiment, to Bernd Amann and to the anonymous referees whose detailed comments significantly helped in improving the quality of the paper.

References

[AS83] D.J. Abel and J.L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing 24*, 1983.

[BDK92] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[BDQV90] K. Bennis, B. David, I. Quilio, and Y. Viemont. Géotropics : Database support alternatives for geographic applications. *Proceedings of the 4th International Symposium on Spatial Data Handling*, 1990.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of ACM*, 18(9):509–517, 1975.

[BF90] K. Bennis and M. Freville. Méthode de construction d'index spatial R+tree pour l'organisation des données géométriques. *6th journées de BD avancées*, July 1990.

[BKS93] T. Brinkhof, H.P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, May 1993. Washington DC.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*tree : An efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 322–331, 1990.

[BM72] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, 1972.

[BPW93] H. Toben B. Pagel, H. Six and P. Widmayer. Towards an analysis of range query performance. In *Proc. PODS 93*, pages 214–221, Wahington, D.C., May 1993.

[Cha91] D. Charlet. Prototype de système d'informations géographiques orienté objet, la couche géométrique. Technical report, Conservatoire National des Arts et Métiers Paris, 292, rue Saint-Martin 75141 PARIS Cedex 03, July 1991. Mémoire ingénieur.

[DRSM93] B. David, L. Raynal, G. Schorter, and V. Mansart. GEO2 : Why objects in a geographical DBMS. In *Proc. of the 3rd International Symposium on Large Spatial Databases, SSD'93, Singapour*, 1993.

[FB74] R.A. Finkel and J.L. Bentley. Quad-trees : A data structure for retrieval on composite keys. *Acta informatica*, 4:1–9, 1974.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extensible hashing, a fast access method for dynamic files. *ACM Trans. on database sys.*, 4(3):315–344, 1979.

[Fra82] A. Frank. Mapquery : data base query language for retrieval of geometric data and their graphical representation. *Computer Graphics*, 16, 1982.

[Fre87] M. Freeston. The BANG file : A new kind of Grid File. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 260–269, 1987. San Francisco, California.

[FSR87] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 426–439, 1987.

[GB91] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: implementation and performance evaluation. *IEEE Trans. on knowledge and Data Engineering*, 3(3):342–356, October 1991.

[Gre89] D. Greene. A implementation and performance analysis of spatial data access methods. *Proc. of the Fifth IEEE Int. Conf. on Data Engineering*, pages 606–615, February 1989. Los Angeles, California.

[Gün88] O. Günther. Efficient structures for geometric data management. *Lecture Notes in Computer Science No. 337*, 1988.

[Gün89] O. Günther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proc. of the Fifth IEEE Int. Conference on Data Engineering*, pages 598–605, February 1989. Los Angeles, California.

[Gün93] O. Günther. Efficient computations of spatial joins. 1993. Vienna, Austria.

[Gur90] A. Gurunlian. Etude comparée des performances de différents index spatiaux par rapport au r-tree. Master's thesis, Université de Paris VI, 4 Place Jussieu 75252 PARIS Cedx 05, 1990. Rapport de Stage de D.E.A.

[Gut84] A. Guttman. R-trees : A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, May 1984.

[Güt88] R.H. Güting. Geo-relational algebra : A model and query language for geometric database systems. *Conference on Extending Database Technology*, pages 506–527, 1988.

[HS91] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In O. Gunther and H. J. Schek, editors, *Advances in Spatial Databases, 2nd Symp., SSD'91*, pages 237–256. Lecture Notes in Computer Science 525, Springer-Verlag, Berlin, 1991.

[HS92] E. G. Hoel and H. Samet. A qualitative comparaison study of data structures for large line segment databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–214, June 1992. San Diego, California.

[HSW89] A. Henrich, H.W. Six, and P. Widmayer. The LSD-tree : A spatial access to multidimensional point and non point objects. *Proc. 15th International Conference on Very Large Data Bases*, pages 45–53, 1989.

[KBS92] H.P. Kriegel, T. Brinkhoff, and R. Schneider. A efficient map overlay algorithm based on spatial access methods and computational geometry. In *Proc. Int. Workshop on DBMS for Geographical Applications,Capri, Italy, in Geographic DBMS, G. Gambosi, M. Scholl, H. Six*. Springer Verlag, 1992.

[Ked82] G. Kedem. The Quad-Cif-tree : A data structure for hierarchical on-line algorithms. *Proc. 19th Design Automation Conference*, pages 352–357, 1982.

[KF93] I. Kamel and C. Faloutsos. On packing r-trees. *Second Int. Conf. on Information and Knowledge Management*, November 1993.

[LS90] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[LT93] R. Laurini and D. Thompson. *Fundamentals of Spatial Information System.* London : Academic Press, March 1993. 680 pages.

[NHS84] J. Nievergelt, H. Hinterger, and K.C. Sevcik. The Grid File : An adaptable symmetric multikey file structure. *Proc. ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[NP94] E. Nardelli and G. Proietti. A hybrid pointerless representation of quadtrees for efficient processing of window queries. *presented to IGIS'94, Int. Workshop on Advanced Research in Geographic Information Systems*, March 1994. Ascona, Switzerland.

[O291] O2. The O2 system. *Communications of the ACM*, 34(10):34–48, 1991.

[OMSD87] B.C. Ooi, K.J. McDonnel, and R. Sacks-Davis. Spatial K-Dtree, an indexing mechanism for spatial databases. *Proc. IEEE Comp. Software and Application Conf.*, 1987. Tokyo, Japan.

[Ooi90] B.C. Ooi. Efficient query processing in a geographic information system. *Springer-Verlag*, 1990.

[Pel92] J.P. Peloux. Indexation d'objets spatiaux. Evaluation de trois index implantés avec le SGBD O2. Master's thesis, Conservatoire National des Arts et Métiers Paris, 292, rue Saint-Martin 75141 PARIS Cedex 03, November 1992. Mémoire ingénieur.

[Ray90] L. Raynal. Etude et implémentation d'un index spatial sur O2, un système de gestion de base de données orienté-objet. Master's thesis, Université de Paris VII, 4 Place Jussieu 75252 PARIS Cedx 05, 1990. Rapport de Stage de D.E.A.

[Rey93] G. Reynal. Implantation d'un R+tree avec le SGBDOO O2. Comparaison à d'autres index. Master's thesis, Conservatoire National des Arts et Métiers Paris, 292, rue Saint-Martin 75141 PARIS Cedex 03, 1993. Mémoire ingénieur.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 17–26, 1985.

[Rob81] J.T. Robinson. The K-D-B-tree : A search structure for large multi-dimensional dynamic indexes. *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 10–18, 1981. Ann Arbor, Michigan (10-18).

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company Inc, 1990.

[SDMO87] R. Sack-Davis, K.J. McDonnel, and B.C. Ooi. GEOQL - a query language for geographic information systems. *Australian and New Zeland Association for the Advancement of Science Congress*, 1987.

[Sha86] C.A. Shaffer. Application of alternative Quadtree representations. Technical report, Computer Science Departement, University of Maryland, College Park, June 1986. Ph.D.dissertation, TR-1672.

[Smi90] T.R. Smith. Experimental performance evaluations on spatial access methods. *Proc. of the 4th International Symposium on Spatial Data Handling*, 2:991–1002, 1990.

[SPSW90] H.J. Schek, H.B. Paul, M.H. Scholl, and G. Weikum. The DAS-DBS project: Objectives, experiences, and future prospects. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):25–43, 1990.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+tree a dynamic index for multi-dimentional objects. *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518, 1987.

[SV89] M. Scholl and A. Voisard. Thematic map modeling. *Design and Implementation of Large Databases (SSD'89) Springer-Verlag*, 409:167–192, 1989. Lecture Notes in Computer Science.

[SW85] H. Samet and R.E. Weber. Storing a collection of polygons using Quadtrees. *ACM Transactions on graphics*, 4(3):182–222, july 1985.

[Tam83] M. Tamminen. Performance analysis of Cell based geometric file organizations. *International Journal Comp. Vision, Graphics and Image Processing*, 24:160–181, 1983.

[TS82] M. Tammimen and R. Sulonen. The Excell method for efficient geometric access to data. *Proc. 19th Design Automation Conference*, pages 345–351, 1982.

[TS94] Y. Theodoridis and T. Sellis. Optimization issues in r-tree construction. *presented to IGIS'94, Int. Workshop on Advanced Research in Geographic Information Systems*, March 1994. Ascona, Switzerland.

Michel Scholl, professeur des Universités au CNAM et conseiller scientifique à l'INRIA dirige l'équipe VERTIGO du laboratoire CEDRIC/CNAM. Ses centres d'intérêt actuels portent sur les systèmes de gestion de Bases de Données orientés objet et leur application aux Systèmes d'Information Géographique et à la gestion et l'interrogation des informations biologiques liées au génome, des hypertextes, des documents structurés et autres informations multimédia.

Jean-Paul Peloux termine une thèse en Informatique dans le cadre d'une bourse CIFFRE dans la société Fleximage en collaboration avec le laboratoire CEDRIC/CNAM. Sa thèse porte sur la conception et l'implantation d'un langage, GQL, pour l'interrogation de bases de données géographiques. Il s'est notamment attaché à l'étude d'index pour optimiser ce type d'interrogation.

Gérald Reynal de St Michel est actuellement Ingénieur consultant chez Software A.G. Il a participé à ce travail dans le cadre de som mémoire d'ingénieur CNAM au laboratoire CEDRIC.