Point and window queries with linear spatial indices : an evaluation with O_2

Michel Scholl[*], Gilles Grangeret[†], Xavier Réhsé[‡]

[*] Cedric/CNAM, 292 rue St Martin 75141, Paris Cedex 03, France
and INRIA, 78153 Le Chesnay, Cedex, France
[†] AGF-SI, 92040 La Défense, Cedex, France
[‡] ANABEL, Groupe LGS, 92247 Malakoff, Cedex, France
Michel.Scholl@inria.fr, grangeg@agf.fr, xrehse@lgs.ca

Spatial query optimization : a challenge for GIS design

Geographic Information Systems (GIS) utilize Database Management Systems (DBMS) for storing and retrieving geo-referenced data. Such data is described by means of attributes defined on domains such as Integer, String, etc., as well as spatial attributes that represent the data geometry and location on earth. As conventional DBMS, GIS have to *efficiently* answer user queries, related to the objects location (geometry) and expressed with a *spatial query language*. Then one crucial challenge of GIS design is spatial query optimization.

Compared to conventional DBMS query optimization, spatial query optimization is more difficult because it has to take into account the semantics of specific spatial operators, which semantics is heavily dependent on the application.

Efficient Spatial Access Methods have to be designed

Whatever the spatial query semantics, one necessary step toward spatial query optimization, is a fast access to geometric data, the volume of which is extremely large. This implies the use of efficient spatial access methods (SAM) to speed up data retrieval in spatial queries such as point selection, windowing, spatial joins, etc. *Point selection* computes the set of geo-referenced objects which contain a point argument. The semantics of a *window query* or *windowing* is the set of objects that overlap a window argument, usually of rectangular form. Point and window queries can be used as parts of more complex queries such as nearest neighbor queries, clipping, distance queries, *spatial joins*. Map overlay is an example of spatial join with join predicate "intersect". Other examples of spatial join predicates are "adjacent to", "north-west of". Conventional indices such as the B+tree provide a fast access to objects indexed on a key for which there exists a total order. Since there is no total order among spatial objects which preserves spatial proximity, simple extensions of conventional indices are inefficient for spatial queries. This explains that a considerable attention has been devoted to the design of SAMs which allow to narrow down the search space to objects satisfying the spatial search predicate.

Hashed-based and Tree-based Spatial Access Methods

SAMs allow to index points, lines, polygons, 3-dimensional polyhedra, etc. We restrict our attention to the 2-dimensional space. Among the numerous ways of representing the objects in the SAM's data structure, a common solution, chosen in the sequel, is to represent the object to be indexed by its Minimum Bounding Rectangle (MBR). Using rectangles instead of complex polygonal shapes allows a simple and fast filtering step through the index. As conventional indices, SAM's can be divided into two categories : hashed-based structures and tree-based structures. For a description, comparative studies, references on the structures quoted below, see [Sam90, GG96, SVP+96].

Hashed-based structures decompose the 2-dimensional space into elementary rectangular cells. With each cell is associated the overlapping object MBRs. Data retrieval consists in finding first the cells satisfying the query (containing a point, overlapping a window, etc.), and then checking the objects associated with the cell against the query. The cell size is such that its MBRs can be held in a single disk page. The *grid* of cells may be regular or not. A grid directory may be associated with the grid of cells. The various hash-based structures differ on the grid(+ directory) structure. Since the initial well known *Fixed Grid* [BF79] and *Grid File* [NHS84], numerous variants have been proposed , the most complex of which resemble the tree-based structures descibed below. Examples of hash-based variants are the Twin Grid File [HSW88], PLOP Hashing [KS88], Excell, the Two-Level Grid File, the Bang File, the Multi-Layer Grid File, the R-File, MOLHPE, the Buddy tree.

Tree-based structures recursively decompose the initial space into smaller regions (one per tree node). The internal nodes allow to guide the search. The objects overlapping with a leaf region must be contained in a single disk page. Otherwise the leaf region is decomposed into smaller regions and the leaf becomes an internal node. Data retrieval consists in traversing the tree from root to one or several leaves which contain the objects satisfying the query. The various tree structures differ on the decomposition process and the mapping node-regions. The most famous tree-structures are the quadtree (the space is recursively decomposed into four quadrants) [FB74], the R-tree (a height-balanced structure) [Gut84], the K-D-B tree and the Cell Tree. Except for the Cell Tree, the region associated with a tree node is of rectangular shape. The hB-tree, the LSD-tree, the R+tree, the R*tree, the P-tree, the BV-tree, the KD-tree, the SKD-tree, the GBD-tree are some variants or extensions of the four above structures.

Should the Spatial Access Method be implemented in the DBMS kernel?

Following are some questions that motivated this work : should the SAM be implemented at the system level, or is it sufficient to develop it with the DBMS application programming interface? Which facilities are then required? How does data volume influence the performance? In this paper, we address the problem of spatial index design with the object-oriented DBMS (ODBMS) O_2 [BDK92]. One advantage of ODBMS is their easy extension with application dependent functionalities. It is often advocated that the spatial index be integrated in the kernel of the system for performance reasons : ease of data clustering, use of the system concurrency control and recovery mechanisms, etc. However, its design requires a deep knowledge of the DBMS kernel, and once implemented, such an index cannot easily be modified. A more flexible solution consists in developing the index code on top of the ODBMS : it is easy to implement and sufficient for fast prototyping or for applications which do not require concurrency control, but it might result in a poor performance. In a companion paper [PRS95], we report a point selection comparison of spatial indices developed on top of the O_2 system with the O_2C language [BDK92]. The structures were a quadtree, a variant called NQuadtree [PRS95], a R-tree, and a R+tree [SRF87].

An intermediate solution, chosen in this paper, consists in :

1. constructing a spatial index from the collection of objects; the initial 2D space is divided into cells; with each cell is associated a disk page containing a small subset of the initial collection;
2. ordering the cells;
3. using the B+tree implemented in the DBMS for indexing the ordered list of cells, with key, the cell rank in the list.

One takes advantage of the data clustering provided by the B+tree, as well as of the concurrency and recovery mechanisms already existing through the B+tree. Such schemes can rapidly be developed on top of the DBMS system without any particular knowledge of the DBMS kernel; furthermore one may more easily change the initial spatial index (step 1 above) than if it was implemented inside the kernel. Numerous papers propose to map spatial data structures onto a B+tree, see for example [Gar82, AS83, Sam90, HS92]. The resulting SAMs are termed *linear*. Oracle 7 Spatial Data Option ^{TM} [Her96] provides a linear quadtree for spatial indexing.

Our objective is to prove the feasibility of this approach with the O_2 DBMS. We compare the performance of point and window queries provided by three spatial indices implemented on top of O_2 : a regular quadtree implemented with O_2C as in [PRS95], a linear grid file and a linear quadtree. The following section describes the three SAMs. We then report an empirical point and window query performance evaluation based on measures with three data sets.

Three Spatial Access Methods implemented and evaluated with O_2

Recall objects to be indexed are represented in the index by their MBR. Then, accessing an object comprises two phases : (i) index traversing, restricting the initial collection of MBRs to a small list of object identifiers (*oids*) of objects whose MBR satisfies the search criteria; (ii) checking whether the object itself satisfies the criteria; the computational cost of the latter test may be high because of the geometric algorithm complexity, but the test is limited to a very small number of objects. We shall not consider the latter phase which is the same whatever index is used.

Regular quadtree

Let us sketch the structure of the variant of quadtree evaluated in this paper, and thoroughly described in [FB74, PRS95, Sam90]. The 2D space is recursively decomposed into quadrants until the number of MBRs overlapping each quadrant is $\leq$ `c` tuples `[mbr, oid]`, where `oid` identifies the object with MBR, `mbr`. An MBR appears in as many leaf quadrants as it overlaps. Besides this redundancy that characterizes all structures studied in this paper, a major limit of the quadtree is that if no attention is paid to the internal nodes clustering, a performance degradation might be observed : in the worst case, if `d` is the quadtree depth, `d` page faults might occur when accessing a leaf node. In the quadtree studied here, an internal node is stored in the same page as its father node, as much as possible. Point selection traverses a single path from root to a leaf, choosing at each level the quadrant containing the point. Windowing keeps at each level of the tree, the quadrants overlapping the query window.

Linear grid

The first linear index is based on a simple *Fixed Grid* [BF79]. The 2D initial rectangular space is divided into equal size rectangular *cells* . With each cell is associated one or several disk pages with capacity `c` tuples `[mbr, oid]`. We store in each cell, the oid objects whose MBR *overlaps*, i.e. intersects, or includes or is included in the cell. An MBR may appear in several neighbor cells.

Traversing the index necessitates two phases : (i) computing the set `L` of cells relevant to the query (e.g. intersecting a rectangle in the case of window queries), (ii) sequentially scanning the cells in `L`, looking for the objects whose MBR satisfies the search criteria. This simple structure is well suited to uniform distributions of the spatial data in the 2D space. The number of cells and therefore the cell size is chosen so that the tuples in a given cell can be held in a single page. However, if the distribution is skewed, the list associated with a cell might be long, requiring several disk pages to be stored, which results in a performance degradation of the second phase. Note also that the redundancy rate (number of objects duplicated in neighbor cells over total number of objects) might be high depending on the data distribution but also on the cell size.

The linear data structure associated with the grid is extremely simple. Choose a total order of the grid cells. [Sam90] gives several possible orders, e.g. *row, row prime, Z-order* [OM84], *Peano-Hilbert*, among which we chose, as in most proposals for linear SAMs, the Z-order, also called N-order or Morton code. The initial space is assumed to be an N x N array of cells where $N = 2^d$. The grid can be seen as a complete quadtree of depth `d`. We associate with each node of the (complete) tree a label taken among strings over the alphabet (1,2,3,4) as follows : the root has for a label the empty string; the North-West (respectively North-East, South West and South-East) son of any internal node with label `k` has for a label `k.1`, (respectively `k.2`, `k.3`, `k.4`) where "`.`" denotes string concatenation. Then the grid cells are labelled by strings of size `d`. Let `f` be the above function GRID $\rightarrow 4^d$ which indexes cells by a string of size `d` over (1,2,3,4). Let `ListCells` be the list of grid cells sorted according to the Z-order. Then `f(ListCells)` is a list of labels in lexicographic order.

We assume that we index in the B+tree, a collection of tuples `[key, cellid]`, with key `key = f(c)`, the label of cell `c` identified by `cellid`. Then retrieving oids of objects satisfying a spatial query, necessitates three steps (valid for any linear index) :

1. key (label) computation : find the list `L` of candidate cells keys associated with the query ; the keys in `L` are in lexicographic order.
2. B+tree traversal for retrieving the cells given their keys; the result is a list `RES` of cell oids.
3. cells' access and scan. The result is a list of oids of objects whose MBR satisfies the query.

Point query with the linear grid : Algorithm A

Step 1 computes the key of cell `c` containing the search point `p`: the algorithm is from [OM84]; it is based on bit interleaving (`BitInterleaving(p)`). Step 2 traverses the B+tree and gives the cellid of cell `c` with key `BitInterleaving(p)`. Step 3 scans the tuples `[mbr, oid]` of cell `c`, and for each of them, keeps `oid` if `mbr` contains `p`. If the list of tuples in `c` requires more than a single disk page, the O_2 system automatically stores it in a B+tree whose leaves are sequentially scanned. This is transparent to the O_2C programmer who only declares a list of tuples and scans it.

Window query with the linear grid

The answer to a window query is the set of MBRs that *overlap* window `w`. Steps 1 and 2 are given below. We omit step 3, similar to point query step 3 (Algorithm A) : instead of a single cell, in general several cells are scanned.

Step 1 : the key of all cells overlapping `w` is computed. A naive algorithm computes for each overlapping cell in `w` scanned in row order, its key with function `BitInterleaving`. Another top-down algorithm, referred to in the sequel as Algorithm B, decomposes `w` into a set of blocks which are not necessarily *minimal* size grid cells. It looks at the grid as a complete quadtree and recursively checks at each level of the tree which of the four quadrants overlaps `w`. For each quadrant, the algorithm stops,

- either, if it is a leaf (a grid cell) : then the cell key is added to the list `L` of keys to be searched for in the B+tree;
- or if the quadrant is included in the window; such a quadrant, at level `l` in the quadtree, called "maximal quadtree block" in [AS93], has for a key, a string of size `l`, which is the key of the corresponding elementary cell in a grid of depth `l`.

[AS93] gives another algorithm in O(n loglogN) for computing the set of maximal quadtree blocks associated with a window of size n x n. Such an algorithm is used in [AS95] for window queries with another variant of linear quadtree. Compared to Algorithm B, its design is more intricate and it would require (for our step 2) a post-sorting in lexicographic order of the list of keys. Although its time complexity might be lower than for Algorithm B for small windows (the sorting time of a short list might be negligible), we preferred implementing Algorithm B for design simplicity.

```
Algorithm B
Window query with the linear grid : step 1

Input  : window w
Output : list (key) L

begin
  quad = space /* initial rectangle space/*
  l = 1        /* l <= d level in the tree of quad/*
  r = e        /*empty string/*
  L = {}
  KEY (quad,r,l)
end
```

where `KEY` is defined as :

```
KEY(quad,r,l)
  if quad in w, then L +={r}
  else, for j in {1,2,3,4}
      if w inter quad(j) /* w overlaps  jth quadrant of quad/*
         then if l=d, then L +={r.j}
              else KEY(quad(j),r.j,l+1)
```

Step 2: a naive algorithm accesses the B+tree for each key in `L`. If two successive keys in `L` correspond to neighbor cells in the B+tree, they should be gathered in a range query. Algorithm C below, issues as many range queries as posssible, reducing the number of accesses to the B+tree. Besides, computing maximal blocks (step 1) decreases the time complexity of Step 2. Indeed, if instead of a maximal block key `K`, we had `m` keys (one per elementary cell included in the maximal block, where `m` is a power of 4), then we would have to scan those `m` strings to prepare a range query.

In the algorithm below, `suc(i)`, where `i` is a key of size `d`, denotes the immediate successor of `i` in the list of grid cell keys. `suc(i)` is of size `d`. `Pref(i)` denotes the set of keys including `i` and all its prefixes, e.g. `Pref(143)` = {`1,14,143`}. `suf(i)` is a grid cell key defined as follows : if `i` is a grid cell key, then `suf(i)` = `i`, else (`i`'s size is `l` < `d`), `suf(i)` is the largest suffix of `i`, i.e. the key obtained by concatenating to `i`, `d-l 4`'s. `req(R)` is the result of a query with range `R` over the B+tree: a list of cellids.

```
Algorithm C
Window query with the linear grid : step 2

Input  : list(key) L
Output : list(cellid) RES

begin
  init = 1
  RES = {}
  While init <= max /* max : L's cardinality/*
      begin
        i = L[init]
        while ( i < max and L[i+1] in Pref(suc(suf(L[i])))), i = i+1
        RES += req([L[init], suc(suf(L[i]))[)
        init = i+1
      end
end
```

Correctness of Algorithm C (sketch): if `L[i]` belongs to the range query, then the grid cell key `suf(L[i])` belongs to the result. Then, if `L[i+1] in Pref(suc(suf(L[i])))`, then the key in the B+tree `suc(suf(L[i]))`, neighbor to `suf(L[i])`, corresponds to the block with key `L[i+1]` or is included in that block. Then `L[i+1]` can be added to the same range query. The smallest cell key larger than the last `L[i]` in the range query (its upperbound) is by definition, `suc(suf(L[i]))`.

Linear quadtree

A regular quadtree is constructed as above from the initial collection of MBRs. The leaves are sorted in Z-order : they are given a key according to the same labelling function as for the grid. While with the grid file, all (elementary) cells keys have same size `d`, each leaf node of the quadtree at depth `l` ≤ `d` has for a key, a string of size `l`, resulting in a collection of variable size keys.

As for the linear grid, we index in the B+tree, a collection of tuples `[key, cellid]`, where `cellid` identifies a *single*[1] disk page containing tuples `[mbr, oid]` of objects whose MBR `mbr` overlaps the quadrant with key `key`.

[1]However, if the linear quadtree has a predetermined maximal depth, when the latter has been reached, then an overflow might occur. Transparently to the O2C programmer, as for the grid, the cell tuples are stored in a B+tree.

Point query with the linear quadtree

Step 1 : we compute as in Algorithm A, the key of the (elementary) cell of the grid corresponding to the *complete* quadtree of depth `d` containing the query point `p`: `k=Bitinterleaving(p)`.

Step 2: let `K` be the key of the quadtree leaf containing `p`. Then either `K=k`, the quadtree leaf is at depth `d`, or `K` has a size less than `k`'s size: the leaf is at depth `l` < `d`. It contains the grid cell (of the complete quadtree) including `p`. More precisely, `K` is the maximal key in the B+tree (according to lexicographic order) among the keys $\leq$ `k`. Then looking for the leaf containing `p`, implies looking in the B+tree for the leaf whose key = $max(key \mid [key, cellid] \in \text{B+tree} \wedge key \leq$ `k`$)$. Such a B+tree point query , denoted `MAXINF(k)` requires a slight modification of the B+tree access procedures. Since the latter was not available in O_2 at the implementation time, we simulated it by successive point queries : let `k` be the key associated with point query `i`. Then point query `i+1` will be issued on key `pref(k)` $\in$ `Pref(k)` obtained by deleting the last digit of `k`. The process ends up when, eventually, the search key is `K`. This of course provides with the current implementation, a discrepancy in step 2 performance. Algorithm D below summarizes step 2 of point query with a linear quadtree.

```
Algorithm D
Point query with the linear quadtree (Step 2)

Input  : key = Bitinterleaving (p)
Output : list(oid) RES

cellid = TraverseBtree(key) */MAXINF(key)/*
while cellid = nil, begin
                      key = pref(key)
                      cellid = TraverseBtree(key)
                    end
```

Window query with the linear quadtree

Exactly as for the linear grid, a list of maximal quadtree block keys in lexicographic order, associated with query window `w` is computed in step 1. Step 2 (Algorithm E below) also minimizes the number of B+tree accesses, by issuing as many range queries as possible. The algorithm however, is slightly more complex than with the linear grid. Indeed, in the presence of a window block key, three cases may occur : (i) the window block b is included in a quadtree leaf, (ii) b exists in the quadtree, (iii) only smaller leaves included in b are stored in the quadtree.

```
Algorithm E
Window query with linear quadtree, Step 2

Input  : list(key) L
Output : list(cellid) RES

begin
  init = 1
  i = 1
  RES = {}
  while init <= max
      begin
        init = i
        while (i < max and (L[i+1] in Pref(suc(suf(L[i]))))), i=i+1
        RES' = req ([MAXINF(L[init]), suc(suf(L[i]))[)
        if (RES'[1] < L[init] and RES'[1] not in Pref(L[i]))
           then RES'= pop(RES')
        RES += RES'
```

```
      i=i+1
      while (i<max and RES[last] in Pref(L[i])), i=i+1
    end
end
```

Correctness of Algorithm E (sketch) : as for the grid, the second while instruction allows to create range queries with neighbor keys in list `L`. The range query lowerbound is not `L[init]` but `MAXINF(L[init])`. Three cases may happen :

1. either the block corresponding to `L[init]` in window `w` is contained in a quadtree leaf, then `MAXINF(L[init])` $<$ `L[init]` and must be searched,
2. or it is equal to a leaf, then `MAXINF(L[init])` $=$ `L[init]` and must also be searched,
3. or it is a maximal block containing a set of quadtree leaves whose key `k` is such that `k` $>$ `L[init]` $>$ `MAXINF(L[init])`. In that case, `MAXINF(L[init])` should not have been searched for. The corresponding cellid must be deleted from the result. It occurs when the first block of the range query result has a key smaller than `L[init]` but is not a prefix of `L[init]`. `pop(RES')` deletes the first element of list `RES'`.

Finally, the last while instruction allows to save unnecessary accesses to the B+tree. Indeed, if the last fetched block `b` has for a key `k`, a prefix of the current block to be fetched `b'`, then either `b'` is contained in `b` which is a contradiction by construction of the window blocks, or both `b` and `b'` are contained in the quadtree leaf with key `k` already fetched.

Measurement environment

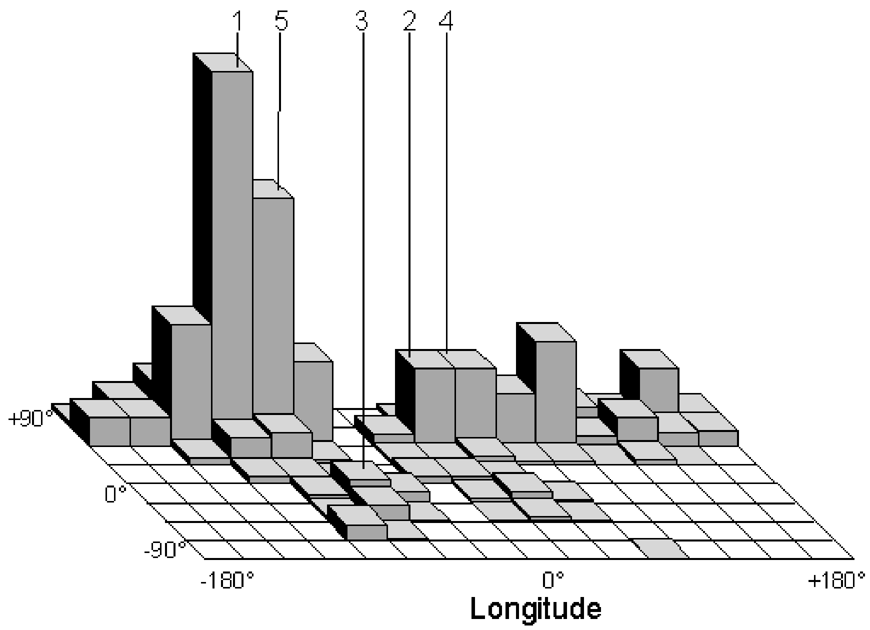

Figure 1: Dataset A: lakes density representation.

The three SAMs were implemented in O2C with version 4.5 of the ODBMS O_2, on a Sun SPARCstation 5 (33 MHz, 32 Mbytes RAM). 3 data sets were used :

1. the lakes of the world : this dataset of size 37 Megabytes referred to as dataset A contains 233 868 MBR's (one per lake). This "real life" set was chosen because of the non uniform distribution of lakes positions on the earth (figure 1).

2. Another (uniform) dataset referred to as dataset B was generated; a linear grid and a linear quadtree were created on this dataset in order to evaluate the influence of the dataset size on the response time, and on the rate of duplication of MBRs in neighbor cells (leaf quadrants). The size of the dataset was either 100 000, or 500 000 or 1 000 000 objects. The upper-left corner of each MBR in dataset B was uniformly chosen in a 40 000 x 40 000 space and the rectangle size was uniformly distributed in { [100,100], [200,100], [500,100], [300,200], [300,100]}.

3. To strengthen the evaluation, the algorithms have been run against another dataset (dataset C) generated according to a non uniform distribution drawn from a benchmark constructed for evaluating spatial joins [GOP+96]. Because of space limitations we only chose one dataset in [GOP+96], generated according to the model called "Continents". This model has 100 000 rectangles obtained by the nesting of two submodels. In a first step, ten large rectangle are generated with the following parameters : lower-left rectangle corner location uniformly distributed, diagonal slope gaussian ($\pi/4$, σ), rectangle area gaussian ($3\text{x}10^{-2}$, σ). Each of these ten continents is filled with 10 000 rectangles each generated with the following distribution and scaled down to fit the continent size : lower-left corner location uniform, slope gaussian ($\pi/4$, σ), area gaussian (10^{-3}, σ).

Datasets A and C	quadtree	linear quadtree		linear grid	
	Set A	Set A	Set C	Set A	Set C
Size (Kbytes)	50 300	33 752	25 230	26 704	20 288
Number of quadrants or cells	8 608	8 608	4 666	16 384	4 096
Number of non empty quadrants or cells	7 943	7 943	4 455	1 802	1 305
Number of overflowing cells	-	-	-	640	949
Maximal depth	11	11	8	7	6
Average depth	8,84	8,84	6,94	7	6
Duplication ratio	1,13	1,13	1,5	1,03	1,24
Average storage factor	40,83	40,83	44,75	16,76	28,62

Table 1: Datasets A and C: index construction.

Dataset B	linear quadtree			linear grid		
Number of indexed MBRs	100 000	500 000	1 000 000	100 000	500 000	1 000 000
Size (Kbytes)	20 229	257 209	794 002	17 328	178 136	205 792
Number of quadrants or cells	4 018	54 697	173 920	4 096	16 384	16 384
Number of non empty quadrants or cells	4 018	54 694	173 920	4 068	16 289	16 315
Number of overflowing cells	-	-	-	0	12 860	16 028
Maximal depth	6	8	9	6	7	7
Average depth	6	7,94	8,83	6	7	7
Duplication ratio	1,72	4,55	9,05	1,72	2,62	2,62
Average storage factor	56,82	55,24	69,36	55,76	46,07	80,73

Table 2: Dataset B: index construction.

Table 1 depicts the result of the construction of the 3 data structures with dataset A and the linear ones with dataset C. Note that the unbalance of the quadtree is limited and that less than 10 % of the leaves are non empty with set A in the linear grid (1802 out of the 16 384 grid cells are non empty : only 1802 disk pages have been created). In table 2, the indices statistics are displayed for dataset B for the various data sizes. Since the datasets are uniform, there is no unbalance of the quadtree and more important, almost no cell is empty. However, the interesting point is the dramatic increase of the duplication rate with the dataset size, for the linear quadtree. With one million objects, 9 050 000 objects are created : the duplication rate is 9,05 while with the linear grid it is more limited (2.62).

For each point or window query, the average was taken on 4 successive measures, where for each measure, the O_2 server process is connected, the O_2 client is connected, the measures associated with the query are collected, the client and the server are disconnected. The (constant) time and number of page faults for code loading is included.

We therefore measure the SAMs performance in worst case conditions where to each first access page corresponds a page fault. The measured time is the sum of the client cpu time plus the time to deliver objects when there is a page fault. The number of tuples `[mbr, oid]` in a single O_2 disk page is 75.

Performance evaluation

We give in this section, performance results for (i) point queries on dataset A (5 points have been chosen so as to fall into regions with variable lakes density, see figures 1 and 2), (ii) window queries on dataset A with 5 windows shown in figure 2, (iii) window queries on dataset A with a set W of generated windows, and (iv) window queries on datasets B and C with the same set W of windows. The set W contains 63 square windows of uniform location in the plane and whose size was uniformly chosen among 6 different sizes taken as powers of 4 of the grid cells size corresponding to each dataset.

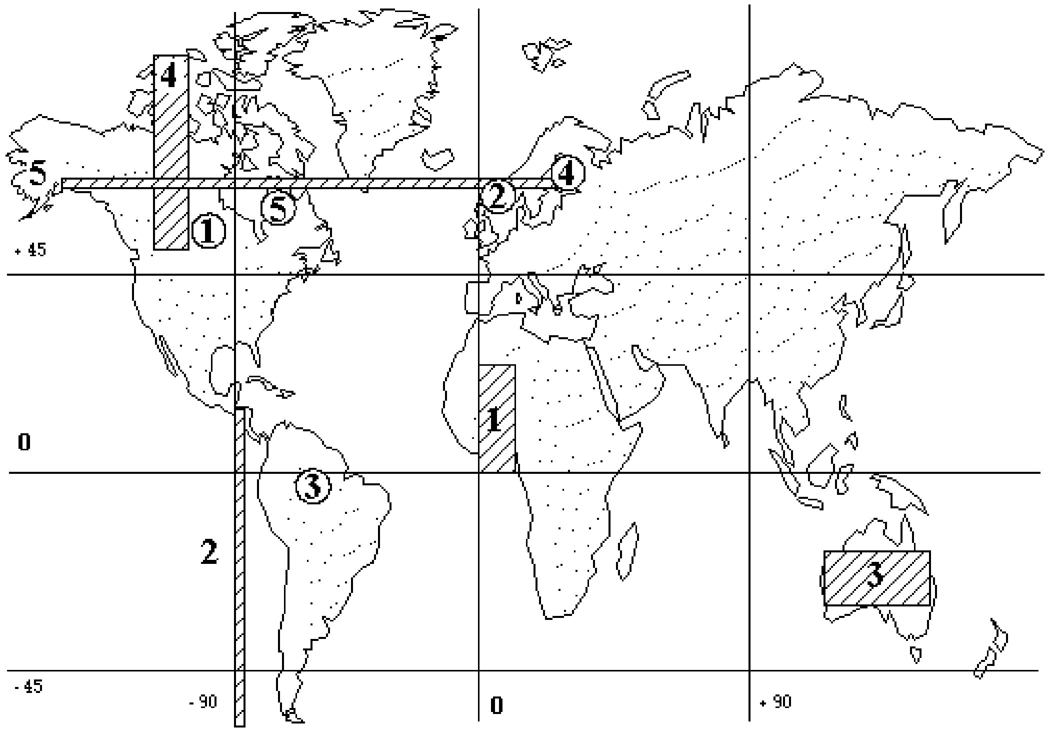

Figure 2: Dataset A: points and windows position.

Point query evaluation

Figure 3 displays, for dataset A, time and page faults number versus (query) point. The response time is < 0.1 seconds, and the number of page faults < 8 except, as expected, with the linear grid, for points 1 and 5 which lie in a high density region where the grid cells overflow and lead respectively to 29 page faults (1 for crossing the B+tree and 28 for scanning the pages associated with the cell) and 35 page faults. This explains the time degradation : 0,38 seconds for point 1 and 0,49 seconds for point 5.

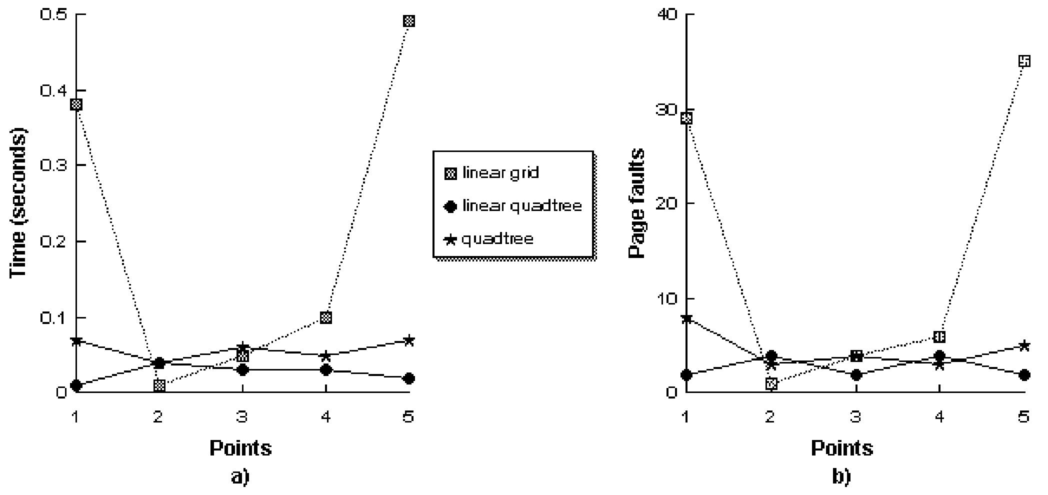

Figure 3: Point selection on dataset A: response time and page faults.

Window query evaluation

Figure 4 displays, for the five sample windows of figure 2 (dataset A), the response time and the page faults versus the number of MBRs scanned : to each window (1,2,3,4,5) according to its location and to its size, corresponds a different number of scanned MBRs : for example, for the linear quadtree, the number of scanned MBRs respectively are 174, 46, 0, 26909 and 15153. Observe the performance degradation of the regular quadtree compared to that of the linear structures, as the number of MBRs increases. This degradation is mainly due not to a significantly higher page fault number but to a larger cpu time when traversing the quadtree index. Indeed, even though at a level of the quadtree, a quadrant has been identified as included in the window, one still has to traverse the complete associated subtree.

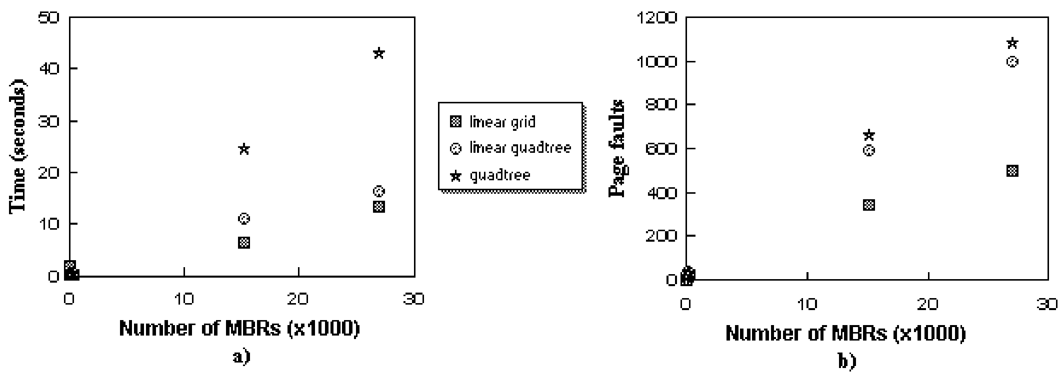

Figure 4: Dataset A: query on the windows of figure 2, response time and page faults

The window query performance is more systematically measured for datasets A, B and C with the set of windows W. Figures 5 and 6 display for dataset A, the response time and the number of page faults versus the number of scanned MBRs for respectively windows of size 64x64 cells (of the grid file) and 256x256 cells.

Observe first that, as already explained above (figure 4), the larger the number of scanned MBRs, the more the linear structures outperform the quadtree. Second, the grid performance is not anymore sensitive to the density of the lakes. This is to be expected, since the larger the number of scanned MBRs, the more important step 3 (scanning the cells) compared to steps 1 and 2 (key computation and index traversal) which step 3 is linear in the number of MBRs. Then, not only this structure is the simplest of the three to implement, but it also provides a slightly better window performance for large windows

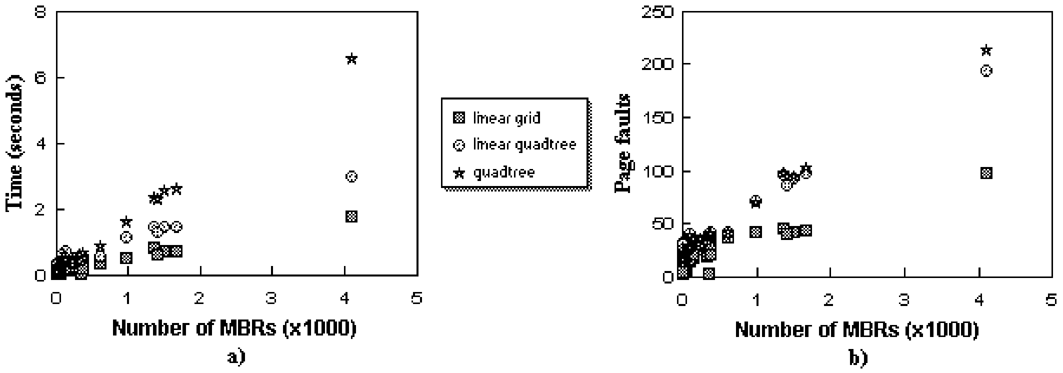

Figure 5: Dataset A: query on square windows (set W) of size 64x64 cells.

than the more complex linear quadtree. However recall the current implementation of the linear quadtree does not provide the primitive `MAXINF` and thus disadvantages the structure with respect to the linear grid. The asymptotic behavior of the three structures is depicted in figures 6.c and 6.d where the time per scanned MBR and the number of page faults per scanned MBR is displayed. The important result is that the asymptotic time per MBR is of the order of 0.5 ms with the linear structures (0.4 ms for the grid and 0.6 ms for the quadtree), and the asymptotic page faults number is 0.039 for the linear quadtree and 0.0175 for the linear grid, which is not far from the lower bound of 1/75=0.013 corresponding to a sequential scan of the MBRs. 0.0175-0.013=0.0045 is the asymptotic index access cost.

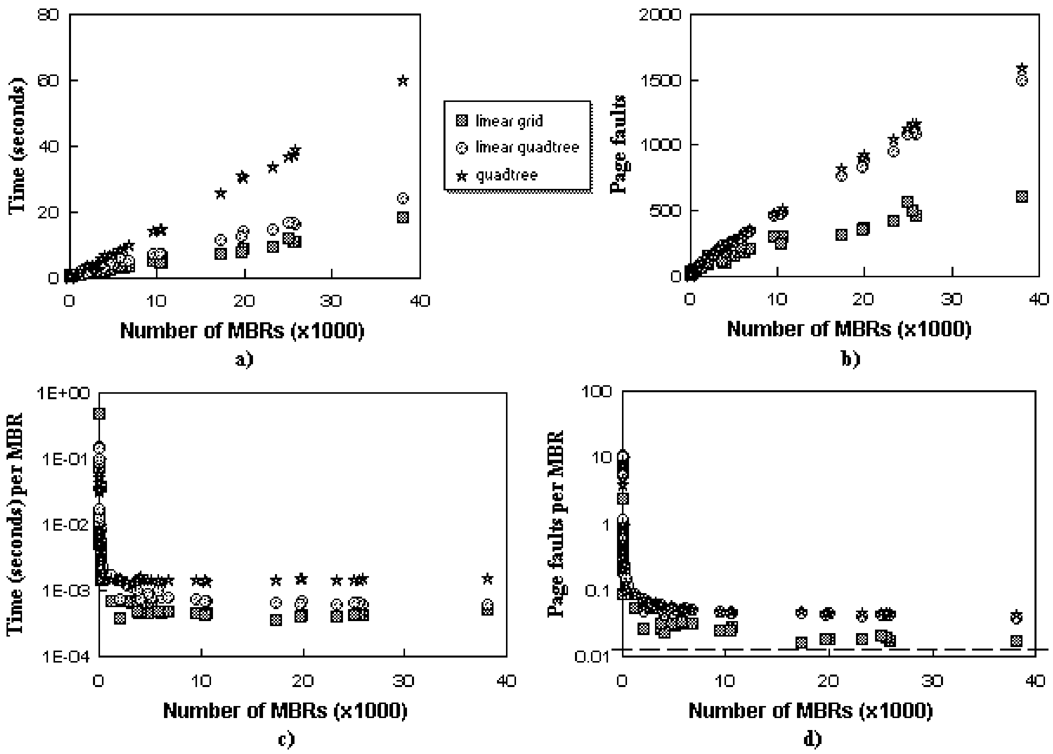

Figure 6: Dataset A: query on square windows (set W) of size 256x256 cells.

The relative importance of the three steps of windowing (key computation, B+tree traversal and cells or quadrants scan) is illustrated in figure 7, for windows of various size. Look first at the quadtree : the larger the window, the longer the traversal compared to the (leaves) access and scan. Indeed, as already mentionned above, this is due to the window query algorithm : even though the quadrant is included in the window, one still has to traverse the whole structure which takes a time exponential in the quadrant depth! Look now at the linear quadtree. The key computation time is less than 10% for small windows and negligible with larger windows. As expected, the larger the window, the more important the leaves scan compared to the B+tree traversal. The behavior of the linear grid is similar.

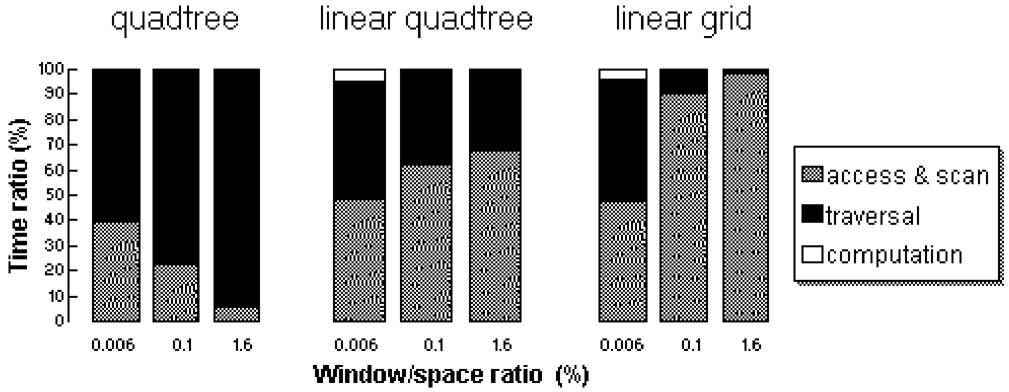

Figure 7: Dataset A, window query, relative importance of the three steps.

No conclusion can be drawn from the measures on dataset A, with respect to the outperformance of a linear structure compared to the other. An implementation providing the primitive MAXINF would decrease the number of page faults for the linear quadtree and might advantage this SAM compared to the linear grid in terms of response time.

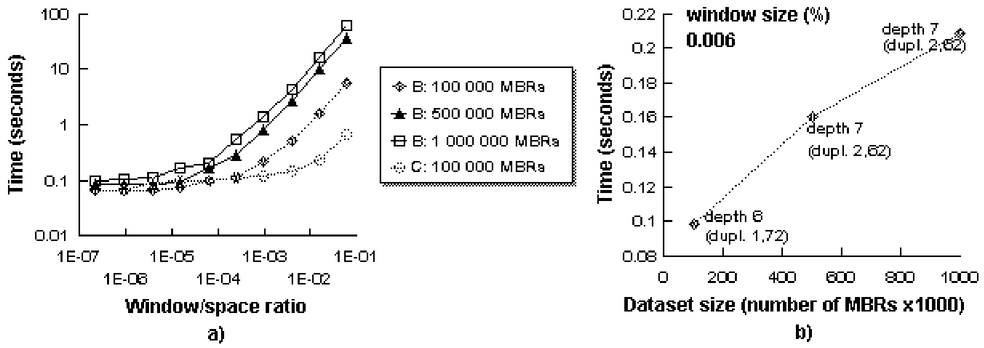

Figure 8: Datasets B and C, linear grid: response time versus window size and dataset B size.

The use of a simple uniformly generated dataset (dataset B) allowed to study the two linear structures performance when the dataset is larger, as well as the duplication rate impact on the index performance. The results are compared to the performance obtained with a non uniform dataset : dataset C. Figures 8.a and 9.a give for the three datasets B of sizes 100 000, 500 000, and 1 000 000 objects, as well as for dataset C (of size 100 000 objects), time versus window size with respectively the linear grid and the linear quadtree. The size of the window is given as a percentage of the total space. The time measure is an average over window queries with same size windows. The performance obtained with the non uniform dataset C is comparable to that of the uniform dataset B of same size. Figures 8.b and 9.b display for dataset B time versus dataset size, for a window of size 0.006 per cent of the space. Although the time should be logarithmic in the dataset size, it increases faster with the linear grid and even faster with the linear quadtree, since it is almost linear in the dataset size. Each point of the curves is labelled with the duplication rate, and with the tree depth. Look first at figure 8.b (linear grid). The larger the dataset, the smaller the cell size, the larger the depth [2]. Observe that between 500 000 and 1 000 000 objects, the cell size has not changed and that for 1 000 000 objects, the duplication rate is still reasonable: 2

[2]The cell size is chosen so that, on the average, the number of MBRs per cell is equal to the page capacity.

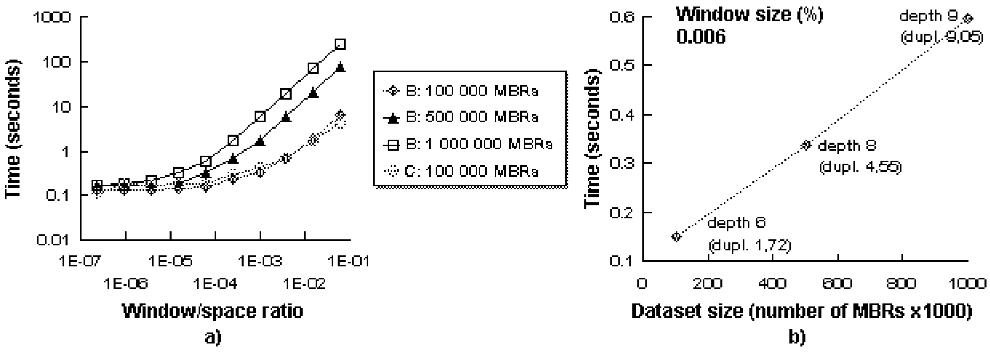

Figure 9: Datasets B and C, linear quadtree: response time versus window size and dataset B size.

620 000 objects are stored. The time increases not only because the duplication rate increases but also because the window contains more objects to be scanned. Finally note that even for 1 000 000 objects the response time is still less than 0.2 seconds, for a window corresponding to 644 MBRs scanned.

In contrast with the linear grid, the linear quadtree duplication rate is drastically more sensitive to the dataset size (figure 9.a). Storing 1 000 000 objects results in actually storing 9 050 000 objects[3]! The response time for 1 000 000 objects (0.6 seconds) is then 3 times the linear grid response time (0.2 seconds).

Summary

Algorithms for point and window queries were given for two linear SAMs and were implemented with the O_2 DBMS; their performance was measured and compared to that of a regular quadtree. A linear index takes advantage of the natural clustering and concurrency mechanisms provided by the system B+tree. Its implementation is extremely simple and flexible compared to that of a SAM inside the DBMS kernel. It turns out from this experiment, that in spite of the mapping of a two dimensional SAM onto the one dimensional B+tree, the point and window query performance is good even for large datasets. However a very high duplication ratio is observed with the SAMs chosen for linearization when the dataset is large. The use of these linear SAMs is currently experimented for the implementation and evaluation of spatial join strategies with O2.

Acknowledgements: This work was part of the CNAM engineer diploma thesis of the second and third authors. We are grateful to Jayant Sharma who mentioned to us the work by Oracle Spatial Products [Her96]. We are indebted to P.Rigaux who corrected some errors and to J.M. Saglio who carefully read the manuscript and generated dataset C, to M. Fiegel and D. Steve from O_2Technology for their technical support, to J.P. Peloux who provided us with dataset A and implemented the regular quadtree [PRS95], to V. Oria and P. Picouet for fruitful discussions and to the anonymous referrees for their constructive comments.

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*, 24:1–13, 1983.

[AS93] W.G. Aref and H. Samet. Decomposing a Window into Maximal Quadtree Blocks. *Acta Informatica*, 30:425–439, 1993.

[3]Observe, however that there exists other decomposition rules for quadtrees and other policies for assigning an MBR to a given quadrant, some of them resulting in no duplication. For example in [AS83] each MBR is assigned to the smallest quadrant that completely contains it. Mapping the resulting tree onto a B+tree leads to another linear structure. Further work should include a comparison between such a scheme and ours.

[AS95] W.G. Aref and H. Samet. A Window Retrieval Algorithm for Spatial Databases Using Quadtrees. In *Proc. 3rd Intl. Workshop on Advances in Geographic Information Systems (ACM-GIS)*, Baltimore, MD, USA, December 1995.

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

[BF79] J.L. Bentley and J.H. Friedman. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4), 1979.

[FB74] R.A. Finkel and J.L. Bentley. Quad-Trees : A Data Structure for Retrieval on Composite Keys. *Acta informatica*, 4:1–9, 1974.

[Gar82] I. Gargantini. An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25:905–910, 1982.

[GG96] V. Gaede and O. Günther. Multidimensional access methods. Technical Report TR-96-043, International Computer Science Institute, Berkeley, Calif., October 1996.

[GOP+96] O. Günther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. 'A la carte' GIS benchmark, 1996. URL: http://www-inf.enst.fr/~bdtest/.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Intl. Conf. on the Management of Data*, volume 4, pages 47–54, June 1984.

[Her96] J.R. Herring. Oracle 7 Spatial Data Option ^{TM}, Advances in Relational Technology for Spatial Data Management, September 1996.

[HS92] E.G. Hoel and H. Samet. A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. In *Proc. ACM SIGMOD Intl. Conf. on the Management of Data*, pages 205–214, June 1992.

[HSW88] A. Hutflesz, H.W. Six, and P. Widmayer. Twin Grid Files: Space Optimising Access Schemes. In *Proc. ACM SIGMOD Intl. Conf. on the Management of Data*, pages 183–190, June 1988.

[KS88] H.P. Kriegel and B. Seeger. PLOP Hashing: A Grid File without Directory. In *Proc. 4th Intl. Conf. on Data Engineering*, pages 369–376, 1988.

[NHS84] J. Nievergelt, H. Hinterger, and K.C. Sevcik. The Grid File: An Adaptable Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[OM84] J.A. Orenstein and T.H. Merett. A Class of Data Structures for Associative Searching. In *Proc. 3rd ACM SIGMOD Intl. Symp. on the Management of Data (PODS)*, pages 181–190, April 1984.

[PRS95] J.P. Peloux, G. Reynal, and M. Scholl. Evaluation of Spatial Indices Implemented with the DBMS O_2. *Ingénièrie des Systèmes d'Information*, 3(4):517–554, 1995.

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

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[SVP+96] M. Scholl, A. Voisard, J.P. Peloux, L. Raynal, and P. Rigaux. *SGBD géographiques, spécificités*. International Thomson Publishing, France, February 1996.