bottom-up from the optimal bucket set B_{opt} by level-oriented recursive calls of the optimization procedure. The second strategy creates the directory on the fly, i.e. during the bucket optimization process. The best strategy and its efficiency is still an open problem.

References

1. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 47–57, Boston, 1984.
2. Henrich, A., Six, H.-W., and Widmayer, P.: The LSD-tree: spatial access to multidimensional point- and non-point objects. In: 15th Int. Conf. on VLDB, pp. 45–53, Amsterdam, 1989.
3. Kamel, I. and Faloutsos, C.: On packing R-trees. In: Proc. 2nd Int. Conf. on Information and Knowledge Management, pp. 490–499, Washington D.C., 1993.
4. Lawler, E.L.: Combinatorial Optimization: Networks and Matroids. Holt, Rhinehart and Winston, 1976.
5. Meyer, B. Eiffel: The Language. Prentice Hall, 1991.
6. Pagel, B.-U. and Six, H.-W.: On optimal static spatial data structures. Technical report, University of Hagen, in preparation.
7. Pagel, B.-U., Six, H.-W., and Toben, H.: The transformation technique for spatial objects revisited. In: Abel, D., Ooi, B.-C. (eds.): Proc. 3rd Int. Symposium on Large Spatial Databases (SSD), pp. 73–88, Singapore, June 1993. Lecture Notes in Computer Science No. 692, Springer Verlag.
8. Pagel, B.-U., Six, H.-W., Toben, H., and Widmayer, P.: Towards an analysis of range query performance in spatial data structures. In: Proc. ACM 12th Symposium on Principles of Database Systems (PODS), pp. 214–221, Washington, D.C., May 1993.

A Hybrid Pointerless Representation of Quadtrees for Efficient Processing of Window Queries

Enrico Nardelli, Guido Proietti

Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Via Vetoio, Località Coppito, 67100, L'Aquila, Italy

and

Istituto di Analisi dei Sistemi ed Informatica, C.N.R., V.le Manzoni 30, 00185 Roma, Italy

E-Mail: nardelli@iasi.rm.cnr.it, guido@iasi.rm.cnr.it

Abstract: Efficient management of spatial data is becoming more and more important and for very large sets of 2-dimensional data, secondary memory data representations are required. An important class of queries for spatial data are those that extract a subset of the data: they are called *window queries* (also *region* or *range queries*). In this paper we propose and analyze the hybrid linear quadtree for the efficient secondary memory processing of three kinds of window queries, namely the *exist*, the *report* and the *select* query. In particular we show that it is possible to answer to all the above queries for multiple non-overlapping features with a number of accesses to secondary memory never greater than the number of pixels inside the window. More precisely, we prove that for a window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ (e.g., pixel elements) using the hybrid linear quadtree the exist and report query can be answered with $O(n \log_r T)$ accesses to secondary storage, while the select query can be answered with $O(n \log_r T + n^2/r)$ accesses to secondary storage. This is an improvement in worst-case time complexity over previous results [Nar93] and shows that multiple non-overlapping features (i.e., coloured images) can be treated with the same I/O complexity as single features (i.e., black and white images).

1 Introduction

Systems able to manage efficiently spatial data have now become an important research topic [Gun90], [Ege93]. Spatial access methods are therefore required to provide efficient retrieval of information in these systems. But, more in general, it is the whole field of efficient external processing of 2-dimensional data that is a hot research topic in itself (see, for example, [Kan93] where one of the most important issues in object oriented programming, namely *indexing classes*, and one of the most important issues in constraint logic programming, namely *indexing constraints*, are reduced to external 2-dimensional interval management).

Many different approaches can be used to represent spatial data: array representation (raster-based), runlength codes, polygons (vector-based), bounding boxes, mapping to higher or lower dimensional spaces and so on. Many data structures have been defined for specific classes of spatial data (i.e., point, lines, rectangles. . .): quadtrees, R-trees, cell-tree, grid-files and so on. See [Sam89] for a survey.

From the storage point of view, we can roughly classify data structures either as suitable to main memory or as secondary storage oriented. Each approach has its pros and cons, and the choice depends on the kind of application to be developed. Anyway, very often the choice is forced by the size of the data, in the sense that if we process a very large amount of data that cannot fit into core memory, we have to use a disk-based representation. In this work we focus on secondary memory representations of 2-dimensional regions. Some data structures have been designed explicitly for this purpose: R-trees [Gut84], R*-trees [Bec90] and R^+-trees [Fal87] are for representation of rectangles, cell-tree [Gun89] allows to represent arbitrary convex polygons by decomposing them in cells. Others, like grid-files [Nie84], have been originally designed for secondary memory management of points but can be used for regions by means of a transformation process. A third class of structures, like region quadtrees [Sam84], have been designed for main memory processing but can be implemented for secondary memory by means of suitable techniques. We follow the third approach and investigate efficient secondary memory implementations of quadtrees.

The term quadtree is generally used to denote a hierarchical data structure developed by a regular decomposition of the space. The hierarchical decomposition is data-driven, but always proceeds according to a regular scheme, going to a deeper level only where represented features are irregularly distributed. In this way space is saved where the distribution is more regular. There exist two categories of quadtree representations oriented to secondary memory: the collection of leaf nodes in the quadtree [Gar82] and the string obtained by a preorder tree traversal of the nodes in the quadtree, also called DF-expression [Kaw83]. Our hybrid approach is a blend of the two and allows us to obtain an efficient processing of window queries.

In this paper we therefore consider the following class of spatial data: an item of the class is a set of non-overlapping 2-dimensional features. Each feature, that is an area homogeneous with respect to a certain value, is represented by a 2-dimensional region (not necessarily connected). A special case is when only one feature is present. In the latter case we can think to the item as a black and white image, while in the former and more general case we can think to it as a coloured image.

Spatial data can be queried in many possible ways. Window queries have a primary importance to efficiently process spatial data, since they allow to extract from the whole image only the parts of interest. Window-based queries we analyze are the following:

exist(f,w): Determine whether or not the feature *f* exists inside window *w*;
report(w): Report the identity of all the features that exist inside window *w*;
select(f,w): Determine the locations of all occurrences of the feature *f* inside window *w*.

A worst-case time efficient main memory implementation of these operations is given in [Are90]: Aref and Samet proved that the exist and report queries can be answered in $O(n\log\log T)$* worst-case time for a window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ (e.g., pixel elements), using always $\Theta(T^2)$ space for representing the image. Their computational model is a RAM with uniform costs, therefore the number of features does not appear in asymptotic bounds. So, though the window contains n^2 pixels, the worst case time complexity of their algorithms is almost linearly proportional (and not quadratic) to the window diameter. These results are achieved via the introduction of the incomplete pyramid data structure, a main memory representation of the image derived by the pyramid data structure. The incomplete pyramid representation, which uses $\Theta(T^2)$ space in all cases, independently from the amount of data really contained in it, becomes inadequate for large amounts of spatial data, when secondary memory needs to be used. Besides, when considering

* All log are base 2 unless differently stated

secondary memory data structures, the performance measure to be used is the number of random access to data, since main memory processing time is usually negligible compared with latency and seek times of disks. We make the standard assumption that each secondary memory access transmits one page of r units of data, and we count this as one I/O.

Aref and Samet [Are92] also gave a secondary memory approach for the efficient processing of the window queries, using a particular technique (the *active border* technique) requiring $O(n)$ additional space. Assuming the underlying spatial database stored in a disk-based structure, their algorithm executes all the above queries with M disk I/O requests, where M is the number of quadtree blocks in the underlying spatial database that overlap the window, and therefore is $M=O(n^2)$; in this way, window queries can be answered (in terms of I/O complexity) in $O(M\log_r T)$.

In this paper we refine the approach defined in [Nar93]. This new approach allows us to improve worst-case time complexity for exist and report queries from $O(n\log_r T+n^2/r)$ to $O(n\log_r T)$, maintaining the same $O(T^2)$ worst-case space complexity in the RAM with uniform costs model. We show that for all the above queries the number of accesses is never greater than the number of pixels inside the window. In particular, we prove that for a window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ (e.g., pixel elements) the exist and report queries can be answered with $O(n\log_r T)$ accesses on secondary storage, while the select query can be answered with $O(n\log_r T+n^2/r)$ accesses on secondary storage.

If the solution proposed in [Are90] is put on secondary memory using B^+-trees, the following bounds can be obtained: $\Theta(T^2/r)$ for space, $O(n\log_r T)$ for exist and report, $O(n\log_r T+n^2/r)$ for select (in [Are90] no dependency from the n^2 term is claimed for the select query, using a technique called *feature-dependencies*, but it is not clear how it is possible to give as answer a list of $O(n^2)$ elements in less than $O(n^2)$). Therefore our solution has the same worst case bounds as a secondary memory implementation of [Are90], but uses less space in the average since do not store always the complete quadtree as it is done in [Are90].

The paper proceeds as follows: in Section 2 we recall the quadtree structure for single and multiple non-overlapping features, also giving an overview of pointerless representations. In Section 3 we present the algorithms to perform the window operations with an efficient number of accesses on secondary storage. Finally, section 4 contains considerations for further work and concluding remarks.

2 The data representation

In this section we describe how the quadtree data structure is used to represent efficiently spatial data for multiple non-overlapping features.

2.1 The quadtree data structure

The quadtree is a progressive refinement of an image that saves storage being based on regularity of features distribution. Assuming to have at disposal an image of size $T \times T$ (e.g., pixel elements), where T is such that it exists an integer m such that $2^m=T$, we proceed in the following way: at level 0 there is the whole image, of side length T. At the first level of decomposition the image consists of four *quadrants* of side length $T/2$. We shall use also the term *block* to denote a quadrant. At the second level each quadrant is then subdivided into four quadrants of side length $T/2^2$ and so on. The decomposition process carried out by the quadtree goes on until each quadrant is covered by only one feature. The decomposition can go on until the pixel level, with quadrants of side length $T/2^m$, where we assume that a quadrant is always covered by only one feature. The decomposition can be represented as a tree of outdegree 4, with the root (at level 0) corresponding to the whole image and each node (at level d) to a quadrant of side length $T/2^d$. The sons of a node are, in preorder, labelled NW, NE, SW and SE. For a given image, nodes are then *homogeneous* (leaf nodes) or *heterogeneous* (intermediate nodes). Correspondingly, we speak of homogeneous and heterogeneous blocks. Using a drawing as an example, ideas will be clearer:

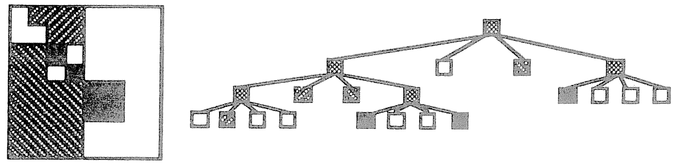

Fig. 1. Multiple non-overlapping features and their quadtree

The quadtree can be implemented as a tree (pointer-based representation) or as a list (pointerless representation). In the former, random access is privileged, but a substantial amount of overhead in terms of space is associated to it. In the latter, random access is not supported, but the pointerless nature simplifies disk-based representations, absolutely needed for large amounts of spatial data.

2.2 Secondary memory implementation

There exist substantially two categories of list-based representation of a quadtree: the collection of the leaf nodes and the linear list resulting from a preorder traversal of the quadtree.

One of the most attractive approach in the first category is the *FL linear quadtree* [Gar82]. Gargantini gave a representation starting from the assumption to have a binary image, but extending the FL linear quadtree to multiple non-overlapping features is immediate. Also here the collection of leaf nodes is stored as a sorted linear list, but now each node has two fields:

1. the *locational key*, whose digits reflect successive quadrant subdivisions;
2. the *value field*, that contains the index of the feature associated with the node.

The locational key (*l-key* in the following) for a node of level d in a $2^m \times 2^m$ space is recursively defined as it follows. Define the key for the root as an all-zero string of length m. Given a node n and its father node n' in the quadtree, let n' have l-key k'. Then, the l-key k of n is:

$$k=k'+s5^{m-d}$$

where s=1, 2, 3 or 4 if n is respectively the NW, NE, SW, or SE son of n':

The l-key is then a base 5 code with m digits. So, for example, the locational keys over a $2^2 \times 2^2$ space are the following:

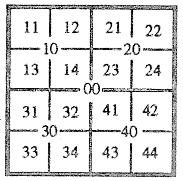

Fig. 2. Locational keys in a 4x4 space

So, for example, for the image in figure 1, we generate the following list:

(111,0),(112,2),(113,0),(114,0),(120,2),(130,3),(141,1),(142,0),(143,0),
(144,1),(200,0),(300,3),(410,1),(420,0),(430,0),(440,0)

corresponding to the following features table:

feature	index
□	0
■	1
▨	2
▩	3

Fig. 3. Features index table

Note that in this way also the background can be treated as a particular feature. We could save space excluding the background nodes from the list, but for the sake of generality it is preferable to treat the background like all other features.

Representing a quadtree as an ordered list of the homogeneous nodes is efficient because reduces space occupancy and improves performances of sequential operations (and we shall see how much this can be important). Also, when the l-keys are sorted in increasing order, the sequence corresponds to the preorder traversal of the leaves in the quadtree.

Concerning representations in the form of a linear list resulting from a preorder traversal of the quadtree, the *DF-expression* [Kaw83] is surely one the most used. The DF-expression for multiple non-overlapping features can be viewed, treating the background as a feature, as a string containing two symbols: 'H' denoting heterogeneous nodes and 'F' followed by the index of the feature associated for homogeneous nodes. As an example, the following string is the DF-expression for the image in figure 1:

HHHF0F2F0F0F2F3HF1F0F0F1F0F3HF1F0F0F0

Representing a quadtree as a DF-expression is efficient because of the data compression, but accessing leaf blocks is rather time-consuming and this represents an obstacle to its use for window queries processing. Furthermore, when the image size forces us to use secondary memory, a balanced tree structure to improve searching performances has to be used, but this can be done only for an indexed list. Thus, for example, an implementation based on B$^+$-trees may be used for the FL linear representation [Abe83], but not for the DF-expression.

2.3 A hybrid representation

In this paragraph, an improved representation, called *hybrid linear quadtree* is proposed. The idea is to represent also the internal nodes of the quadtree (like in the DF approach) but coding them with an l-key (like in the FL approach). The result is a list as in the FL approach, but containing all the nodes in the quadtree. This allows the use of a balanced tree structure as the B$^+$-tree to efficiently represent the quadtree and simultaneously supports random access to every node. We have a storage overhead because of the need to store intermediate nodes, but these are no more than 1/3 of the leaf nodes and thus the asymptotic space occupancy remains the same.

We assume that each page of the B$^+$-tree has a capacity of r items, whether it contains leaf nodes or internal nodes, and it is sorted in increasing order.

Each node of the hybrid linear quadtree (*HL-quadtree* in the following) has two fields:

1. the locational key, whose digits reflect successive quadrant subdivision;
2. the value field, that is a string of a fixed number s of features slots.

The l-key is defined as in the FL approach. Concerning the value field, the bit slot is set to 1 if and only if the associated feature is contained in the spatial region corresponding to the node. As an example, the following list is the HL-quadtree for the image in figure 1:

(000,1111),(100,1111),(110,1010),(111,1000),(112,0010),(113,1000),(114,1000),
(120,0010),(130,0001),(140,1100),(141,0100),(142,1000),(143,1000),(144,0100),
(200,1000),(300,0001),(400,1100),(410,0100),(420,1000),(430,1000),(440,1000).

From the example it could appear that space for representing features dominates over the space for l-keys, but remember that: (i) from a theoretical point of view, given the RAM with uniform costs model, in asymptotic space bounds it is only the number of nodes that is important; (ii) from a practical point of view, for typical sets of spatial data, quadtree depth is in the range 10-16, which leaves space (given current 32-bits architectures) for at least 16 features (usually more than it is needed) in a 2-words per node implementation.

3 Techniques and algorithms

In this section we describe techniques and algorithms to perform efficiently (in term of secondary storage accesses) window operations for images containing multiple non-overlapping features and represented as HL-quadtrees implemented using a B$^+$-tree structure. First, we present how the query window can be decomposed in smaller subwindows to obtain a more efficient processing; then, algorithms are presented and discussed, and their worst-case complexity is analyzed.

3.1 Decomposing Windows

The basic approach to window query processing is to divide the window into subwindows and therefore to decompose the query into a sequence of smaller queries onto the subwindows. Subwindows are the blocks corresponding to the leaf nodes in the quadtree representing the inside of the window region in the image space. We name these subwindows *maximal blocks*. We are interested to the number of maximal blocks that are generated in the worst-case, since this number directly affects the number of accesses on secondary storage. Dyer [Dye82] and Shaffer [Sha88] proved the following:

Lemma 1: The number of maximal quadtree blocks inside an $n{\times}n$ window on a $T{\times}T$ image is in the worts-case K=3(2n-log$n$)-5.

◦

Maximal blocks can be generated in $O(n\log\log T)$ time [Are90]. Let L^m denote the set of all l-keys over a $2^m{\times}2^m$ space.

Definition 1: Let k and k' be in L^m. k is *ancestor* of k' if it exists an integer j ($0 \leq j < m$) such that the leftmost digits of k and k' are equal and the remaining m-j digits of k are equal to 0. Correspondingly, we say that k' is a *descendant* of k.

◦

Note that, given the correspondence between each block and its l-key, the l-keys sorted in increasing order correspond to the preorder traversal of the quadtree. Therefore the above defined ancestor and descendant relations correspond to the hierarchical relations between blocks in the quadtree. Moreover, due to the properties of a preorder visit of a tree, all l-keys relative to the descendants of a block with l-key k are the smallest of the l-keys greater than k. Finally, the following lemma can be derived:

Lemma 2: Let Q be a quadtree representing an image of size $2^m{\times}2^m$ and let l be its hybrid linear representation. Let k be in L^m and let r denote the spatial region with l-key k. Then r is homogeneous in Q iff l contains either k or an ancestor of k, with exactly one bit-slot set to 1.
◦

3.2 Window queries

For multiple non-overlapping features, window queries are formulated as follows:
exist(f,w): Determine whether or not the feature f exists inside window w;
report(w): Report the identity of all the features that exist inside window w;
select(f,w): Determine the locations of all occurrences of the feature f inside window w. This means to output all blocks homogeneous for feature f.

The basic approach to process all window queries is to decompose the query over a window into a sequence of smaller queries onto the maximal blocks contained inside the window. In the following we examine them separately.

3.2.1 The exist query

For each maximal block b, we search the B$^+$-tree the corresponding l-key l to know if b contains the features f. There are two possibilities:
(1) l appears in the B$^+$-tree and the bit-slot corresponding to f is set to 1. Then we output YES.
(2) l does not appear in the B$^+$-tree. Then we check the block having the l-key immediately smaller than l (with at most an additional access in case it resides on the previous page of the B$^+$-tree). This block necessarily corresponds to an homogeneous ancestor of b. If the bit-slot corresponding to f is set to 1, then we output YES. Otherwise we pass to the next maximal block.
If we check for all the maximal blocks inside the window without finding the feature f, then we output NO.

Procedure *exist_multiple_feature* here below gives the pseudo-code (note that the **return**() instruction stops the execution of the procedure):

```
procedure exist_multiple_feature(f);
list_of_blocks listmb;          /listmb contains the list of maximal block inside
                                /the window
begin
  listmb=window_gen(w);         /generate the list of maximal blocks inside the
                                /window w;
for (each record r in listmb) do begin
  B:=search(r);  /search for the l-key of r in the B+-tree representing the HL-quadtree
                 /and returns the bucket that may contain r: if r exists in the B+-tree,
                 /it is contained in this bucket; each bucket contains also the pointer
                 /to the previous and next bucket;
  for (each record s in B) do
     if ((s is an ancestor of r) or (r=s)) then
        if (the the bit-slot associated to f in s is 1) then return (YES);
  A:=getpreviousbucket(B);      /load in main memory the bucket A preceding B;
                                /return nil if B is the first bucket;
  if (A is not nil) then
     if (the last record of A is an ancestor of r) then
        if (the the bit-slot associated to f in s is 1) then return (YES);
end;
```

```
return (NO);
end.
```

Theorem 1: Given an $n \times n$ window on a $T \times T$ image containing multiple non-overlapping features represented as an HL-quadtree stored on a B+-tree, the procedure *exist_multiple_feature* given above answers to the exist query with $O(n \log_r T)$ accesses to secondary storage.
Proof: Let the number of nodes in the quadtree be N. Knuth [Knu73] shown the depth of a B$^+$-tree of order r is:

$$h \leq 1+\log_r((N+1)/2)$$

and then, since is $N=O(T^2)$, it follows that for each maximal block we do, considering also the extra access to the contiguous bucket, $O((1+\log_r T^2)+1)=O(\log_r T)$ accesses on disk. From lemma 1, the thesis follows. ◦

3.2.2 The report query

We follow the same technique as in procedure *exist_multiple_feature*. The only difference is that here we report all features we find.

```
procedure report_multiple_feature();
boolean block_covered, outside_block;
list_of_blocks listmb;
list_of_features listf;      /listf contains the list of features in the window;
begin
listmb=window_gen($w$);
for (each record $r$ in listmb) do begin
    block_covered:=false;
    $B$:=search($r$);
    for (each record $s$ in $B$) do
        if (($s$ is an ancestor of $r$) or ($r=s$)) then begin
            add (listf, $f$);      /add feature $f$ associated to $s$ to listf;
            block_covered:=true;
        end;
    if (not block_covered) then begin
    $A$:=getpreviousbucket($B$);
    if ($A$ is not $nil$) then
        if (the last record $s$ of $A$ is an ancestor of $r$) then add (listf, $f$);
end;
end;
delete_duplicates (listf);      /eliminate duplicate features in listf;
return (listf);
end.
```

Theorem 2: Given an $n \times n$ window on a $T \times T$ image containing multiple non-overlapping features represented as an HL-quadtree stored on a B+-tree, the procedure *report_multiple_feature* given above answers to the report query with $O(n \log_r T)$ accesses to secondary storage.
Proof: Same as theorem 1. ◦

3.2.3 The select query

Following the same idea as developed above, for each maximal block b we search in the B$^+$-tree its locational key l. There are four possibilities:

(1) l appears in the list and the block is homogeneous with respect to the feature f: then we output it directly;
(2) l appears in the list and the block contains feature f but is not homogeneous with respect to it. Then we have to examine all the descendants of b until we reach the first node that is not a descendant of b.
(3) l appears in the list and the block does not contain the feature f: then we pass to the next maximal block;
(4) l does not appear in the list; in this case we have to check the l-key immediately smaller than l (with at most an additional access in case it resides on the previous page of the B$^+$-tree). This block necessarily corresponds to an homogeneous ancestor of b: if the block contains the feature f, then we output b directly, otherwise we pass to the next maximal block.

Procedure *select_multiple_feature* here below gives the pseudo-code:

```
procedure select_multiple_feature($f$);
boolean block_present, outside_block;
list_of_blocks listmb, listbb;      /listbb contains the list of all occurrences of
                                    /feature $f$ in the window;
begin
listmb=window_gen($w$);
for (each record $r$ in listmb) do begin
    block_present:=false;
    $B$:=search($r$);
    for (each record $s$ in $B$) do      /remember that records appear in increasing order;
        if (($s$ is an ancestor of $r$) or ($r=s$)) then begin
            block_present:=true;
        if (only the bit-slot associated to $f$ in $s$ is 1) then      /$s$ is homogeneous
                                                                  /with respect to $f$
            add(listbb, $r$);
        if ((the bit-slot associated to $f$ in $s$ is 1) and (at least another different bit-slot
            in $s$ is 1)) then begin      /$s$ contains $f$ but is not homogeneous
                outside_block:=false;
                while (($B$ is not $nil$) and (not outside_block)) do begin
                    for (each record $s$ in the bucket $B$) do
                        if (the l-key of $s$ is greater than the l-key of $r$) then
                            if ($s$ is a descendant of $r$) then begin
                             if (only the bit-slot associated to $f$ in $s$ is 1) then
                                add(listbb, $s$);
                            end
                            else outside_block:=true;
                $B$:=getnextbucket($B$);
                end;
        end;
        /the third possible case is when $s$ does not contain $f$;
    end;
    if (not block_present) then begin
    $A$:=getpreviousbucket($B$);
    if ($A$ is not $nil$) then
        if (the last record $s$ of $A$ is an ancestor of $r$) then
if (the bit-slot associated to $f$ in $s$ is 1) then add(listbb, $r$);
```

```
            /if the bit-slot associated to f in s is 1 none of the other bit-slots in s is 1
        end;
    end;
    return (listbb);
    end.
```

Theorem 3: Given an $n{\times}n$ window on a $T{\times}T$ image containing multiple non-overlapping features represented as an HL-quadtree stored on a B$^+$-tree, the procedure *select_multiple_feature* given above answers to the select query with $O(n\log_r T+n^2/r)$ accesses to secondary storage.

Proof: Cases (1), (3) and (4) cost $O((1+\log_r T^2)+1)=O(\log_r T)$ accesses to disk. Concerning case (2), we note that each block has width $O(n)$ and then it has $O(n^2)$ sons in the worst case; then, $O(n^2/r)$ accesses to secondary storage for each maximal block in the window may be necessary. From lemma 1, this analysis would derive an upper bound on the number of accesses of $O(n(\log_r T+n^2/r))$. But since $O(n^2)$ is an upper bound on the number of homogeneous blocks in the window, at most we transfer in main memory $O(n^2/r)$ pages in the totality of the accesses. Therefore the bound becomes $O(n\log_r T+n^2/r)$. ∘

We conclude this section giving a comparison between the performances in terms of space occupancy and I/O time complexity of our structure and those provided by the previous approaches proposed in literature. Let us just recall that our notations are: T for the image size, n for the window size, r for the bucket size, Q for the total number of quadtree blocks in the underlying spatial database (and therefore is $Q=O(T^2)$), M for the total number of quadtree blocks in the underlying spatial database that overlap the window, (and therefore is $M=O(n^2)$).

	SPACE # buckets	TIME # I/O access		
		exist	*report*	*select*
HL-quadtree	$O(Q/r)$	$O(n\log_r Q)$	$O(n\log_r Q)$	$O(n\log_r Q+n^2/r)$
Linear Quadtree	$O(Q/r)$	$O(n\log_r Q+n^2/r)$	$O(n\log_r Q+n^2/r)$	$O(n\log_r Q+n^2/r)$
Active border	$O(Q/r)$	$O(M\log_r Q)$	$O(M\log_r Q)$	$O(M\log_r Q)$
Inc. Pyramid	$\Theta(T^2/r)$	$O(n\log_r T)$	$O(n\log_r T)$	$O(n^2\log_r T)$

4 Conclusions

In this paper we have proposed and analyzed worst-case complexity of a new technique for efficiently processing window queries on spatial data stored on secondary memory. We use as performance measure the number of secondary storage accesses and we show that for multiple non-overlapping features the number of accesses is proportional to the window width for the exist and the report queries and never greater than the number of pixels inside the window for the select query.

Future work will be in the direction of an extension of this technique to the more general case of multiple overlapping features, including a definition of a secondary memory data structure supporting them.

References

[Abe83]: D.J. Abel, A B$^+$-tree Structure for Large Quadtrees, in *Computer Vision, Graphics and Image Processing 27*, 1 (July 1984), pp. 19-31.

[Are90]: W.G. Aref and H.Samet, Efficient processing of Window Queries in the Pyramid Data Structure, in *Proc. of the Ninth ACM-SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, Nashville, TN, April 1990, pp.265-272.

[Are92]: W.G. Aref and H.Samet, An Efficient Window Retrieval Algorithm for Spatial Query Processing, Technical Report n° 2866 of Computer Science Department, University of Maryland, March 1992.

[Bec90]: N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, The R*-tree: an Efficient and Robust Access Method for Points and Rectangles, in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, NJ, 1990, pp. 322-331.

[Dye82]: C.R.Dyer, The space efficiency of quadtrees, Computer Graphics and Image Processing, 19(4):335-348, August 1982.

[Ege93]: M.J. Egenhofer, What's special about Spatial: Database Requirements for Vehicle Navigation in Geographic Space, *Session on Database Challenges* in *Proc. of ACM Sigmod Int. Conf. on Management of Data*, Washington DC,1993,pp.398-402.

[Fal87]: C. Faloutsos, T. Sellis and N. Roussopoulos, Analysis of Object Oriented Spatial Access Methods, in *Proc. of the SIGMOD Conf.*, San Francisco, May 1987, pp. 426-439.

[Gar82]: I. Gargantini,.An Effective Way to Represent Quadtrees, *Comm. of the ACM*, Vol. 25, No. 12, 1982, pp. 905-910.

[Gun89]: O. Günther, The Design of the Cell Tree: An Object Oriented Index Structure for Geometric Databases, in *Proc. of the Fifth IEEE Int. Conf. on Data Engineering*, Los Angeles, CA, February 1989, pp. 598-605.

[Gun90]: O. Gunther, O. and A. Buchmann, Research Issues in Spatial Databases, SIGMOD RECORD, Vol. 19, No. 4, December 1990, pp. 61-68.

[Gut84]: A. Guttman, R-Trees: a Dynamic Index Structure for Spatial Searching, in *Proc. of the SIGMOD Conf.*, Boston, June 1984, pp. 47-57.

[Kan93]: P. Kanellakis, S. Ramaswamy, D. Vengroff and J. Vitter, Indexing for Data Models with Constraints and Classes, in *Proc. of the 12th ACM Symp. on Principles of Database Systems*, Washington, DC, May 1993, pp. 233-243

[Kaw83]: E. Kawaguchi, T. Endo and M Yokota, Depth-first Expression Viewed from Digital Picture Processing, in *IEEE Trans. on Pattern Analysis and Machine Intelligence*, July 1983, pp. 373-384.

[Knu73]: D.E.Knuth, *The art of computer programming, Vol.3: sorting and searching*, Addison-Wesley, Reading, MA, 1973.

[Nar93]: E. Nardelli and G. Proietti, Efficient Secondary Memory Processing of Window Queries on Spatial Data, in *Eigth Int. Symposium on Computer and Information Sciences*, Antalya, Turkey, November 1993.

[Nie84]: J. Nievergelt, H. Hinterberger and K.C. Sevcik, The Grid File: an Adaptable. Symmetric, Multikey File Structure, *ACM Trans. on Database Systems 9*, March 1984, pp. 38-71.

[Sam84]: H. Samet, The Quadtree and Related Hierarchical Data Structures, *in Computing Surveys*, Vol. 16, No. 2, June 1984, pp. 187-260.

[Sam89]: H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1989.

[Sha88]: C.A.Shaffer, A formula for computing the number of quadtree node fragments created by a shift, *Pattern Recognition Letters*, 7(1):45-49, January 1988.