A Fun Application of Compact Data Structures to Indexing Geographic Data[*]

Nieves R. Brisaboa[1], Miguel R. Luaces[1], Gonzalo Navarro[2], and Diego Seco[1]

[1] Database Laboratory, University of A Coruña
Campus de Elviña, 15071, A Coruña, Spain
{brisaboa,luaces,dseco}@udc.es
[2] Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. The way memory hierarchy has evolved in recent decades has opened new challenges in the development of indexing structures in general and spatial access methods in particular. In this paper we propose an original approach to represent geographic data based on compact data structures used in other fields such as text or image compression. A *wavelet tree*-based structure allows us to represent minimum bounding rectangles solving geographic range queries in logarithmic time. A comparison with classical spatial indexes, such as the R-tree, shows that our structure can be considered as a fun, yet seriously competitive, alternative to these classical approaches.

Key words: geographic data, MBR, range query, wavelet tree.

1 Introduction

The ever-increasing demand for services that allow users to find the geographic location of some resources in a map has emphasized the interest in the field of Geographic Information Systems (GIS). The huge size of geographic databases has made the development of spatial access methods one of the most important topics of interest in this field. Even though many classical spatial indexes [7] provide an excellent performance, the way the memory hierarchy has evolved in recent decades has opened new opportunities in this topic. New levels have been added (e.g., flash storage) and the sizes at all levels have been considerably increased. In addition, access times in upper levels of the hierarchy have decreased much faster than in lower levels. Thus, reducing the size of spatial indexes is a topic of interest because placing these indexes in upper levels of the memory hierarchy reduces access times considerably, in some cases by several orders of magnitude. Nowadays it is feasible to place complete spatial indexes in

[*] This work has been partially supported by "Ministerio de Educación y Ciencia" (PGE y FEDER) ref. TIN2009-14560-C03-02, by "Xunta de Galicia" ref. 08SIN009CT, and by Fondecyt Grant 1-080019, Chile.

main memory. Note that spatial indexes do not contain the real geographic objects but a simplification of them. The most common simplification is the MBR (*Minimum Bounding Rectangle*).

In this paper we aim at the development of compact spatial indexes that can be placed in upper levels of the memory hierarchy. We build on previous solutions for two-dimensional points using a structure called a *wavelet tree* [9], and generalize them to an index able of answering range queries on rectangle data. Wavelet trees are interesting because they offer a compact-space solution to various point indexing problems. In previous work [3] we presented a spatial index for two-dimensional points based on wavelet trees. The generalization to support queries over MBRs, which we present here, turns out to be a rather challenging problem not arising in other domains where wavelet trees have been used. Our experiments, featuring GIS-like scenarios, show that our index is a relevant and funnier alternative to classical spatial indexes, such as the R-tree [10], and that it can take advantage of the fashionable research in compressed data structures.

2 Related Work

A great variety of spatial indexes have been proposed supporting the different kinds of queries that can be applied to spatial databases (*exact match, adjacency, nearest neighbor, etc.*). In this paper we focus on a very common kind of query, named *range query*, on collections of two-dimensional geographic objects. The problem is formalized as follows. In the 2-dimensional Euclidean space E^2, we define the MBR of a geographic object o, $MBR(o) = I_1(o) \times I_2(o)$ where $I_i(o) = [l_i, u_i](l_i, u_i \in E^1)$ is the minimum interval describing the extent of o along the dimension i. In the same way, we define a rectangle query $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$. Finally, the range query to find all the objects o having at least one point in common with q is defined as $RQ(q) = \{o \mid q \cap MBR(o) \neq \emptyset\}$.

The R-tree [10] is one of the most popular multidimensional access methods used to solve range queries in GIS. It consists of a balanced tree "derived from the B-tree" that decomposes the space into hierarchically nested, possibly overlapping, MBRs. Object MBRs are associated with the leaf nodes, and each internal node stores the MBR that contains all the nodes in its subtree. The algorithm to solve range queries using this structure goes down the tree from the root visiting those nodes whose MBR intersects the query window. Most of the numerous variants [13] of the original Guttman's proposal aim at improving the performance of the R-tree both in the general case and in particular applications (static collections). Two of these variants (the R*-tree [2] and the STR R-tree [12]) are used in Section 4 to compare the performance of our proposal.

The problem of solving two-dimensional range queries on points has also been tackled in other research fields. The seminal computational geometry work by Chazelle [4] offers several space-time tradeoffs, including one that in two dimensions requires $O(N \log U)$ bits of space and answers range queries in time $O(\log N + k \log^{\epsilon} N)$, where N is the total number of points in $[1, U] \times [1, U]$, k

is the output size, and $0 < \epsilon < 1$ is a constant affecting memory consumption. The *wavelet tree* [9] can be regarded as a compact version of Chazelle's data structure, which requires exactly $N \log_2 U + o(N \log U)$ bits to index N points in the range $[1, U]$. Recently [3], we adapted the basic approach where the points form a permutation to handle an arbitrary set of points in a continuous space, following Gabow's arguments [6].

A basic tool in compact data structures is the *rank* operation: given a sequence S of length N, drawn from an alphabet Σ of size σ, $rank_a$ counts the occurrences of symbol $a \in \Sigma$ in $S[1, i]$. The dual operation, $select_a(S, i)$, finds the i-th occurrence of a symbol $a \in \Sigma$ in S. For the special case $\Sigma = \{0, 1\}$ (S is a bit-vector B), both rank and select operations can be implemented in constant time and using little additional space on top of B ($o(n)$ in theory [14,8]). For example, given a bitmap $B = 1000110$, $rank_0(B, 5) = 3$ and $select_1(B, 3) = 6$. In addition, the symbol a can be extended to a finite number of sequences with similar techniques. For instance, given two bitmaps $B = 1\underline{0}0011\underline{0}$ and $C = 0\underline{0}1101\underline{0}$, $rank_{00}(B, C, 7) = 2$ and $select_{00}(B, C, 1) = 2$ (00 represents occurrences of the symbol 0 in both bitmaps simultaneously).

3 Our Fun Structure

In this section we introduce our technique for range queries on MBRs. Recall our formal definition of the problem from the previous section. The following, easy to verify, observation provides a basis for our next developments. It says, essentially, that an intersection between a query q and an object o occurs when, across each dimension, the query finishes not before the object starts, and starts not after the object finishes.

Observation 1. $o \in RQ(q)$ iff $\forall i,\ u_i^q \geq l_i \ \wedge \ l_i^q \leq u_i$.

3.1 Index Construction

In the upcoming discussion, we assume that the first dimension represents the rows of the grid (y-axis or latitudes) and the second represents the columns (x-axis or longitudes). Assume now the set of MBRs $g = \{m_1, \ldots, m_N\}$ does not contain any MBR m_i whose projection in the x-axis is within the projection over the x-axis of other MBR m_j in the set (i.e., $\forall i, j$ if $l_2^i < l_2^j$ then $u_2^i \leq u_2^j$). We name g a *maximal set* and describe now a structure to represent a maximal set of MBRs. If the set of MBRs is not a maximal set, the problem can be decomposed into k independent maximal sets (see Section 3.4).

Then, let N be the number of MBRs in a maximal set, each one described by two pairs $\{(l_1, l_2), (u_1, u_2)\}$ (the coordinates of two opposite vertices). These MBRs can be represented in a $2N \times 2N$ grid with only one point in each row and column. Gabow et al. [6] proved that the orthogonal nature of the problem makes possible to work with the ranks of the coordinates instead of working with the coordinates themselves.

A wavelet tree with $\lceil \log_2 2N \rceil$ can be used to store this matrix (the permutation from the order of the MBRs in the x-axis to their order in the y-axis) with little storage cost (Figure 1). This is a binary tree where each node covers a range of positions in the Y^lY^u array that represents the first half of the range covered by its parent, in the case of a left child, and the second half in the case of a right child. The range covered by the root node is $[1{,}2N]$.

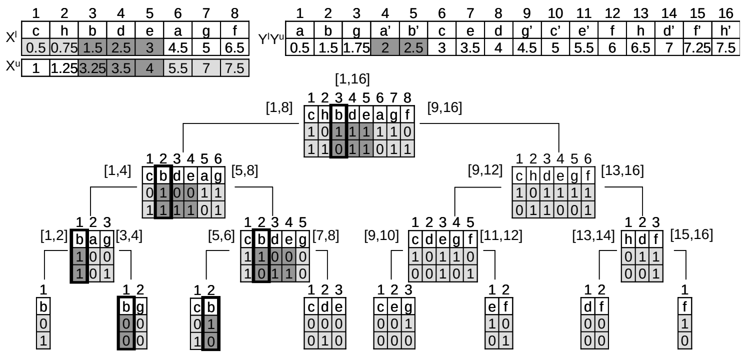


Fig. 1. Representing N MBRs using a wavelet tree.

Each node in the tree stores two bitmaps B_1 and B_2 of the same length, and each position in these bitmaps corresponds with a MBR (in the figure, these positions have been annotated with the identifier of the corresponding MBR). The MBRs in each node are ordered by the x-axis. Let MBR_i be the MBR stored at the position i of a node, $b_i^{B_1}$ the bit i in the bitmap B_1, and $b_i^{B_2}$ the bit i in the bitmap B_2. Then, $b_i^{B_1} = 1$ if the MBR_i is processed in the left child and $b_i^{B_2} = 1$ if MBR_i is processed in the right child. A MBR is processed in a node if, in the y-axis, it finishes not before the range covered by the node starts, and starts not after the range covered by the node finishes. Let lB and uB be the lower and upper bounds of the range covered by a node in Y^lY^u, then Equations 1 and 2 define the value of the bit i of this node in the first and second bitmap respectively. Note that a MBR can be processed in both the left and right child of a node and thus both $b_i^{B_1}$ and $b_i^{B_2}$ can store the value 1 simultaneously.

$$b_i^{B_1} = \begin{cases} 1 & \text{if } l_1^{MBR_i} \leq \frac{lB+uB}{2} \\ 0 & \text{otherwise} \end{cases} \quad (1) \qquad b_i^{B_2} = \begin{cases} 1 & \text{if } u_1^{MBR_i} > \frac{lB+uB}{2} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

We also need to store the real coordinates of the MBRs to perform the translation from the geographic space to the rank space. The order of the lower (X^l) and upper (X^u) coordinates in the x-axis is the same because we assume the matrix represents a maximal set. Thus, we use two sorted arrays with the lower (X^l) and upper (X^u) x-coordinates and an array storing the identifiers of the MBRs in the same order. Y-coordinates are stored also in an ordered array, Y^lY^u, containing both lower (Y^l) and upper (Y^u) y-coordinates. Each position in the Y^lY^u array of the figure has been annotated with the identifier of the corresponding MBR for clarity, but these identifiers are not stored.

As such, this structure may require quadratic space, however. The reason is that a MBR with a large extent in y can be represented in a linear number of nodes at the same level. In order to solve this problem Equation 3 presents a slight modification in the way the structure is created. When a MBR_i completely contains the range covered by the node both bitmaps store a 0 in the position i, and thus, this MBR is not stored in the nodes of this subtree. Then each MBR can be stored at most four times per level and we can guarantee logarithmic bit-space per MBR.

$$b_i^{B_1} = b_i^{B_2} = \begin{cases} 0 & \text{if } (l_1^{MBR_i} \leq lB) \text{ and } (u_1^{MBR_i} \geq uB) \\ \text{use (1) and (2) otherwise} \end{cases} \tag{3}$$

3.2 Solving Queries

This structure can be used to solve range queries in the rank space derived from the translation of the original queries in the geographic space using the ordered arrays of coordinates (X^l, X^u, and Y^lY^u). A $leftSearch(S, t_i)$ finds the lowest $s_i \geq t_i$ in an ordered array S by means of a binary search. In a similar way, a $rightSearch(S, t_i)$ returns the largest $s_i \leq t_i$. Thus, a query in the geographic space $q = [y_l, y_u] \times [x_l, x_u]$ is translated into the equivalent query $q' = [y'_l, y'_u] \times [x'_l, x'_u]$ ($y'_l = leftSearch(Y^lY^u, y_l)$, $y'_u = rightSearch(Y^lY^u, y_u)$, $x'_l = leftSearch(X^u, x_l)$, and $x'_u = rightSearch(X^l, x_u)$) in the rank space (yes, the upper x coordinates of the MBRs are searched for the lower x coordinate of the query, and vice versa). For example, the query $q = [2.0, 2.75] \times [2.0, 3.5]$ translates into $q' = [4, 5] \times [3, 5]$ ($leftSearch(Y^lY^u, 2.0) = 4$, $rightSearch(Y^lY^u, 2.75) = 5$, $leftSearch(X^u, 2.0) = 3$, and $rightSearch(X^l, 3.5) = 5$).

Algorithm 1 shows the recursive method to solve range queries once they have been translated into the rank space. The interval $[x'_l, x'_u]$ determines the valid range inside the root node of the wavelet tree and the interval $[y'_l, y'_u]$ determines nodes that can be pruned (because the wavelet tree maps from the order in the x-axis to the order in the y-axis). This algorithm recursively projects a range, $[x'_l, x'_u]$ at the beginning, onto the child nodes using $rank_1$ operations over the two different bitmaps. The first bitmap B_1 is used to project onto the left child and the second bitmap B_2 is used to project onto the right child. The recursive traversal stops when the result of the two child nodes has been computed. Note that the same MBR can be reported by both child nodes but no repeated results should be reported by their parent node. Thus, the results of both siblings are merged to compute the result of their parent node. In addition, there can be local results in a node corresponding with MBRs that completely contain the range covered by the node (i.e., all the MBRs in a position i where $b_i^{B_1} = b_i^{B_2} = 0$), which are added to the result in the merge stage.

Figure 1 highlights the nodes visited to solve the query of the example $q = [2.0, 2.75] \times [2.0, 3.5]$. As we noted before, this query is translated into the

Algorithm 1 Range query algorithm in the rank space.

```
Require: $cNode$,$p_{min}$,$p_{max}$,$lB$,$uB$; current node, valid node positions [$p_{min}$,$p_{max}$],
  query range [$lB$,$uB$]
  $result \leftarrow []$; $leftResult \leftarrow []$; $rightResult \leftarrow []$; $localResult \leftarrow []$
  if $cNode.range \subseteq [lB, uB]$ then
    for $i = p_{min}$ to $p_{max}$ do
      add $i$ to $localResult$
    end for
  else
    if $cNode.leftChild.range \cap [lB, uB] \neq \emptyset$ then
      $leftResult \leftarrow$ recursive call with:
        $p_{min} \leftarrow rank_1(cNode.B_1, p_{min} - 1) + 1$
        $p_{max} \leftarrow rank_1(cNode.B_1, p_{max})$
        $cNode \leftarrow cNode.leftChild$
    end if
    if $cNode.rightChild.range \cap [lB, uB] \neq \emptyset$ then
      $rightResult \leftarrow$ recursive call with:
        $p_{min} \leftarrow rank_1(cNode.B_2, p_{min} - 1) + 1$
        $p_{max} \leftarrow rank_1(cNode.B_2, p_{max})$
        $cNode \leftarrow cNode.rightChild$
    end if
    for $i = rank_{00}(cNode.B, p_{min} - 1) + 1$ to $rank_{00}(cNode.B, p_{max})$ do
      add $select_{00}(cNode.B, i)$ to $localResult$
    end for
  end if
  for all $lR \leftarrow leftResult.next()$,$j \leftarrow rightResult.next()$,$k \leftarrow localResult.next()$ do
    $merge(select_1(cNode.B_1, i), select_1(cNode.B_2, j), k)$
  end for
  return $result$
```

ranges $[3, 5]$ (valid positions in the root node) and $[4, 5]$ (interval to prune the tree traversal). The first range is projected onto the child nodes of the root node as $[rank_1(B_1, 3 - 1) + 1, rank_1(B_1, 5)] = [2, 4]$ and $[rank_1(B_2, 3 - 1) + 1, rank_1(B_2, 5)] = [3, 4]$ but the second one is not accessed because it covers the range [9,16] which does not intersect the query range [4,5]. In the same way the range $[2, 4]$ of the left child is projected onto its children as $[rank_1(B_1, 2 - 1) + 1, rank_1(B_1, 4)] = [1, 1]$ and $[rank_1(B_2, 2 - 1) + 1, rank_1(B_2, 4)] = [2, 4]$. In the next level, the first node accessed is the second one that covers the range [3,4]. The result of this node comes from the local result that is computed in this way: there is one local result (because $rank_{00}(B, 1) = 1$) that is at the position 1 (because $select_{00}(B, 1) = 1$). When the recursive call returns the control to the parent of this node, its result is computed using the merge of the left child result (an empty set), the right child result ($select_1(B_2, 1) = 1$) and the local result (an empty set). In the parent of this node, there are no local results and the left result ([1]) and right result ([2]) reference the same MBR ($select_1(B_1, 1) = select_1(B_2, 2) = 2$). Finally, in the root node the result comes from the left child and it is computed as $select_1(B_1, 2) = 3$. Note that the MBR at position 3 is b, the result of the query.

3.3 Coordinate Encoding

We introduce a compressed storage scheme to store the ordered arrays of coordinates (X^l, X^u, and $Y^l Y^u$). We assume that these coordinates can be represented with four bytes, which is sufficient for the finite precision used in GIS. Geographic coordinates can be represented in degrees or meters and in most cases it is possible to round the coordinates to integer values, after appropriate scaling, without losing any precision. We make use of this assumption, as it holds in most practical applications.

Let $A = a_1 a_2 \ldots a_N$ be one of the arrays of integers to encode. Then, we encode A as a sequence of non-negative differences between consecutive values $b_{i+1} = a_{i+1} - a_i$ and $b_1 = a_1$. Let $B = b_1 b_2 \ldots b_N$ be this sequence, so that $a_i = \sum_{1 \le j \le i} b_j$. The array B is a representation of A that can be compressed by exploiting the fact that consecutive differences are smaller numbers. These small numbers can be encoded with different coding algorithms. We compare four different well-known coding algorithms [15]: Elias-Gamma, Elias-Delta, Rice, and VBytes.

Given a value v, we are interested in finding the largest $a_i \le v$ and the lowest $a_i \ge v$. These operations are the $rightSearch$ and $leftSearch$ described in Section 3.2. In order to solve them efficiently we store a vector that stores the accumulated sum at regularly sampled positions (say every hth position, thus the vector stores all values $x_{i \cdot h}$). The search algorithm first performs a binary search in the vector of sampled sums, and then it carries out a sequential scan in the resulting interval of B.

3.4 Decomposition into Maximal Sets

In the general case, a maximal set is not enough to properly encompass the dataset but k maximal sets are needed. Each such set must be queried separately. We use a single shared $Y^l Y^u$ array for all of them, to reduce the number of binary searches. Thus the query time complexity can be bounded by $O(k \log N)$. Therefore, minimizing the number of maximal sets k is a key factor to improve the performance of our structure.

We can in fact decompose a general set of MBRs into the optimal number k of maximal sets, at indexing time, within $O(N \log N)$ complexity, as follows. We first order the MBRs by the left x-axis value, and process them in that order. We start with an empty set of maximal sets, which is kept sorted by rightmost x value in the set. Each new segment can be inserted into any such maximal set whose rightmost value does not exceed the rightmost x value of the new segment. From those, we search the one with maximum rightmost value. If no candidate exists, the new segment creates its own new maximal set.

This solution is not new. It is well known to find the longest increasing subsequence in a stream of numbers, and is also related to the problem of decomposing a permutation Π over $\{1 \ldots N\}$ into the minimum number of Shuffled (i.e., not consecutive) UpSequences [1] (the rightmost values of the MBRs correspond to the permutation values). Our algorithm is equivalent to Fredman's [5] one to find the optimal number of Shuffled UpSequences.

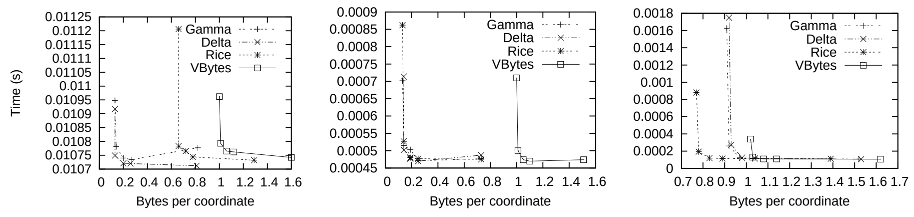
4 Experiments

Our machine is an Intel Core2Duo with two processors Intel Pentium 4 CPU 3.00GHz, with 4GB of RAM. It runs GNU/Linux (kernel 2.6.27). We compiled with gcc version 4.3.2 and option -O9. Both synthetic and real datasets were used in our experiments. The three synthetic collections have one million MBRs each, the first one with a uniform distribution, the second one with a Zipf distribution (world size = 1000×1000, $\rho = 1$), and the third one with a Gauss distribution (world size = 1000×1000, $\mu = 500$, $\sigma = 200$). We created four query sets for each dataset, with different selectivities that represent 0.001%, 0.01%, 0.1%, and 1% of the area of the space where the MBRs are located. They contain 1,000 queries with the same distribution of the original datasets and the ratio between the horizontal and vertical extensions varies uniformly between 0.25 and 2.25. The algorithm generating these query sets is based on the one used in the evaluation of the R*-tree [2]. The first real collection, named Tiger dataset, contains 2,249,727 MBRs from California roads and it is available at the U.S. Census Bureau[3]. In addition, six smaller real collections available at the same place were used as query sets: Block (groups of buildings), BG (block groups), AIANNH (American Indian/Alaska Native/Native Hawaiian Areas), SD (elementary, secondary, and unified school districts), COUSUB (country subdivisions), and SLDL (state legislative districts). The second real collection, named EIEL dataset, contains 569,534 MBRs from buildings in the province of A Coruña, Spain[4]. Five smaller collections available at the same place were used as query sets: URBRU (urbanized rural places), URBRE (urbanized residential places), CENT (population centers), PAR (parishes), and MUN (municipalities).

4.1 Coordinate Encoding

Coordinate encoding does not have a key influence in search time performance (these arrays are only used to translate the queries from the geographic space to the rank space). Thus we can tolerate a small loss in performance in exchange for better compression. We performed experiments with four coding algorithms (Elias-Gamma, Elias-Delta, Rice, and VBytes) and five sampling rates h. Figure 2 shows the results of these experiments in the Zipf, Tiger, and EIEL datasets respectively. Query sets contained 1,000 uniformly distributed queries in the surface covered by each dataset with a selectivity that represents the 0.01% of the area. The four lines correspond to the coding algorithms and each point in these lines represents a different sampling rate (10, 50, 100, 1,000 and 10,000 are the different h values from left to right).

[3] http://www.census.gov/geo/www/tiger
[4] http://www.dicoruna.es/webeiel



(a) Zipf (b) Tiger (c) EIEL

Fig. 2. Influence of the coordinate encoding.

All the coding algorithms provide a good compression rate (the size is significantly lower than the 4 bytes per coordinate necessary without encoding). Elias-Gamma and Elias-Delta provide the best performance when the differences are very small (e.g., Zipf dataset), but their performance is quite worse in the EIEL dataset where the differences are larger. VBytes coding provides better time performance than the rest of the algorithms but its compression rate is not competitive. Note that VBytes works at the byte level whereas the rest work at the bit level. Hence, Rice coding can be identified as the algorithm that offers a better space/time trade-off in the majority of the situations. In addition, an interval of sampling rates providing an optimal space/time trade-off can be identified around 500. In the rest of the experiments we use a sampling rate $h = 500$ and a preprocessing stage to choose the best coding algorithm.

4.2 Space Comparison

We compare now our structure with two variants of the R-tree in terms of space needed to store the structure. The space needed by an R-tree over a collection of N MBRs can be estimated considering a certain arity (M). Dynamic versions of this structure, such as the R*-tree, estimate that nodes are 70% full whereas static versions, such as the STR R-tree, assume that nodes are full. Therefore, an R*-tree needs $\frac{N}{0.7 \times M-1}$ nodes and an STR R-tree needs $\frac{N}{M-1}$ nodes. Each node needs $M \times sizeof(entry)$ bytes. The size of an entry is the size of an MBR plus a pointer to the child (or to the data if the node is a leaf). In order to compare these variants with our structure we assume that MBRs are stored in 16 bytes (4 coordinates with numbers of 4 bytes) and the pointer in 4 bytes. Hence, the total size of an R*-tree is $\frac{N}{0.7 \times M-1} \times 20 \times M$ whereas the size of an STR R-tree is $\frac{N}{M-1} \times 20 \times M$. In our experiments the best time performance of the R*-tree and STR R-tree is achieved with an effective M value of 30. Note that the coordinates stored by the R-tree are not sorted, thus it is not possible to apply our differential encoding.

On the other hand, our structure stores the encoded coordinates of the N MBRs, their identifiers (N 4-byte numbers) and the wavelet tree bitmaps (see grayed data in Figure 1). The wavelet tree needs $\lceil \log_2 2N \rceil$ levels but the number of times a MBR appears in each level is not constant (four times per level is a pessimistic upper bound). In addition, in order to perform *rank* operations in constant time, some auxiliary structures are needed that use an additional space. In our experiments we use the classical two-level solution to perform $rank_1$ and $select_1$ over the bitmaps B_1 and B_2 (37.5% in addition to the bitmaps) and a simpler one level solution to perform $rank_{00}$ and $select_{00}$ over the virtual double bitmap that is composed of B_1 and B_2 (an additional 5%). A description and empirical comparison of these solutions can be found in [8]. As well as the size of the wavelet tree the effectiveness of the coordinates compression also varies across datasets, so we show the results for each dataset in Figure 3.

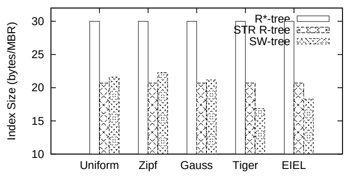


Fig. 3. Space comparison.

These results show that our structure, named *SW-tree* (from *spatial wavelet tree*) in the graphs, can index collections of MBRs in less space than the R*-tree in both synthetic and real scenarios, and it also needs less space than the STR R-tree in real scenarios and a comparable space in synthetic ones. This is due to the compressed encoding of the coordinates and the little space required by the wavelet tree.

4.3 Time Comparison

To perform the time comparison we implemented our structure as described in Section 3 and used the R-tree implementation provided by the *Spatial index library* [11]. This library provides several implementations of R-tree variants such as the R*-tree and the STR packing algorithm to perform bulk loading. In addition, all these variants can run in main memory. In our experiments we run both the R*-tree and the STR R-tree in main memory with a load factor $M = 30$.

We first perform experiments with the three synthetic collections. Figures 4(a), 4(b), and 4(c) show the results obtained with uniform data, Gauss distributed data, and Zipf distributed data, respectively. The main conclusion that can be extracted from these results is that our structure is competitive with respect to query time efficiency. It outperforms both variants of the R-tree with the uniform dataset. In the other two datasets the performance of the three structures is very similar. The R-tree variants outperform our structure when the queries are very selective and in less selective queries the results are the opposite.

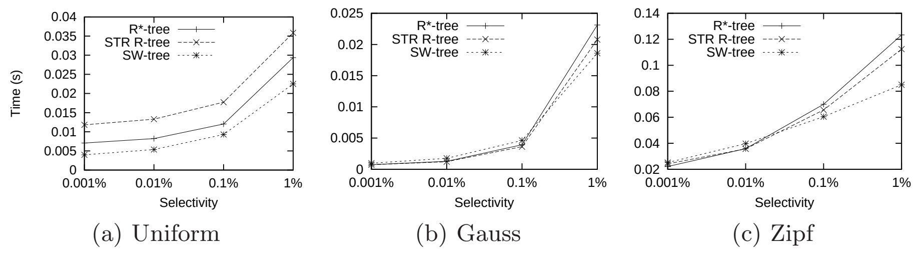


Fig. 4. Time comparison in three synthetic datasets with different distributions.

Finally, we present the results with the two real datasets. Figures 5(a) and 5(b) present the results with the Tiger and EIEL datasets respectively. In these graphs the real query sets have been sorted accordingly with their selectivity (from left to right the query selectivity is looser). Note that all of them are meaningful queries. For example, in the EIEL dataset the query set CENT contains queries of the form *which buildings are contained in the population center X*. In the same way as Zipf and Gauss datasets the performance of the three structures is quite similar. Our structure outperforms both R-tree variants in less selective queries and it is less competitive in the more selective ones.

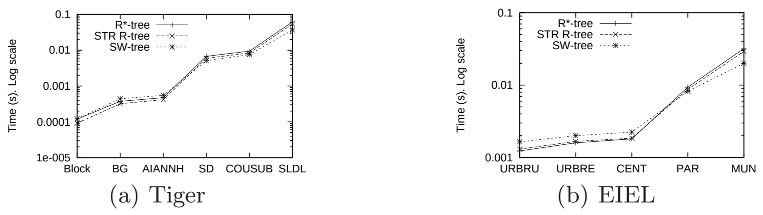


Fig. 5. Time comparison in two real datasets.

5 Further Fun

The minimum number k of maximal sets that cover the MBRs can be thought of the difficulty of the problem, thus our $O(k \log N)$ time query algorithm is adaptive to this difficulty. Yet, the situation is indeed more complex (and fun). As a simple example, the number could be different if we rotated the data. For example, in the TIGER data set, we obtain 19 maximal sets in the x-axis and 36 in the y-axis. This difference is also reflected in the query time performance (for example, using the *Block* query set, the time is almost the double in the second option). A finer consideration is as follows. Assume $N_1, N_2, \ldots, N_k$ are the sizes of the k maximal sets. Then, $\sum N_i \lceil \log N_i \rceil$ is the space necessary to store the wavelet tree that solves the queries in $\sum \lceil \log N_i \rceil$ time. This is interesting because the space is a convex function whereas the time is a concave function. Therefore, balancing the number of elements in the maximal sets improves the size of the structure whereas the opposite improves the query time performance. Hence, we can design heuristics to create the maximal sets based on this tradeoff. For example, the algorithm to create the maximal sets decomposition can choose the set that, without violating the constraints, contains fewer/more elements, minimizes $N_i \lceil \log N_i \rceil$, etc. Finally, the analysis of the query time performance can be refined by defining the complexity of the problem k as the number of maximal sets accessed to solve a query (and not all the maximal sets necessary to represent the dataset). In this case, heuristics that minimize the overlap between maximal sets can improve the query time performance. This leads us to a band-decomposition of the space very typical in some packing algorithms for spatial indexes.

References

1. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: Proc. 26th STACS 2009. pp. 111–122 (2009)
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD Record 19(2), 322–331 (1990)
3. Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: A new point access method based on wavelet trees. In: Proc. SeCoGIS'09. ER 2009 Workshops. pp. 297–306 (2009)
4. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)
5. Fredman, M.L.: On computing the length of longest increasing subsequences. Discrete Mathematics 11(1), 29 – 35 (1975)
6. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th STOC. pp. 135–143 (1984)
7. Gaede, V., Gnther, O.: Multidimensional access methods. ACM Computing Surveys 30(2), 170–231 (1998)
8. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th WEA (Poster). pp. 27–38 (2005)
9. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th ACM-SIAM SODA. pp. 841–850 (2003)
10. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proc. SIGMOD. pp. 47–57. ACM Press (1984)
11. Hadjieleftheriou, M.: Spatial index library., retrieved March 2009 from http://research.att.com/ marioh/spatialindex/
12. Leutenegger, S., Lopez, M., Edgington, J.: STR: A simple and efficient algorithm for R-tree packing. In: Proc. 13th ICDE. pp. 497–506 (1997)
13. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer-Verlag (2005)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1) (2007)
15. Salomon, D.: Data Compression: The Complete Reference. Springer-Verlag (2004)