Space-Efficient Representations of Rectangle Datasets Supporting Orthogonal Range Querying

Nieves R. Brisaboa[a], Miguel R. Luaces[a], Gonzalo Navarro[b], Diego Seco[a,*]

[a] *Database Lab., University of A Coruña, Campus de Elviña, 15071, A Coruña, Spain*
[b] *Dept. of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile*

Abstract

The increasing use of geographic search engines manifests the interest of Internet users in geo-located resources and, in general, in geographic information. This has emphasized the importance of the development of efficient indexes over large geographic databases. The most common simplification of geographic objects used for indexing purposes is a *two-dimensional rectangle*. Furthermore, one of the primitive operations that must be supported by every geographic index structure is the *orthogonal range query*, which retrieves all the geographic objects that have at least one point in common with a rectangular query region. In this work, we study several space-efficient representations of rectangle datasets that can be used in the development of geographic indexes supporting orthogonal range queries.

Keywords: GIS, data structure, space-efficient, orthogonal range query

1. Introduction

In the age of on-line digital availability, an ever-increasing demand for *geo-located information* has emphasized the importance of geographic search engines, such as Google Local or Yahoo! Local, which allow users to find the geographic location of some resources on a map (e.g., business, public administration services, places of interest, pictures, etc.). In this context, research on orthogonal range queries on geographic databases has become a

*Corresponding author. Tel.: +34 981 167 000 - 1306; fax: +34 981 167 160
Email addresses: brisaboa@udc.es (Nieves R. Brisaboa), luaces@udc.es (Miguel R. Luaces), gnavarro@dcc.uchile.cl (Gonzalo Navarro), dseco@udc.es (Diego Seco)

hot topic. These queries define a rectangular query window and retrieve all the geographic objects having at least one point in common with it. Other types of queries, such as *region queries* (which permit arbitrary orientations and shapes for the query regions) and *point queries* (which retrieve all the objects overlapping a query point), are also relevant for geographic information services. However, orthogonal range queries are the most interesting by far as they allow for more efficient solutions while providing good approximations to solve the others.

Orthogonal range querying on point datasets can be thought of as a restricted variant of the general problem. This variant has arisen in many research areas (e.g., computational geometry, text indexing, databases, etc.). For example, in conventional databases a typical application of range queries on point datasets are queries of the form *find all employees between* a_1 *and* a_2 *years old, and with a salary in the range* $[s_1, s_2]$. As a result, this problem has been tackled from several points of view, achieving many different trade-offs [1, 2, 3, 4].

The more general problem (i.e., geographic datasets) has been a topic of interest in the research field of Geographic Information Systems (GIS). A good survey of spatial queries and index structures designed to solve them is that of Gaede and Günther [5]. Although these multidimensional structures generalize our two-dimensional problem, the two-dimensional case models the most common problems in GIS.

In order to achieve a good time performance, both the two-dimensional and the multidimensional problem have been tackled using a filter and refinement strategy [5]. Complex geographic objects are simplified at the indexing stage. Queries in these indexes first filter a set of candidate objects (those whose simplification satisfy the query) and then refine the result using the real geographic objects. The most common simplification of geographic objects is the Minimum Bounding Rectangle (MBR), or Minimum Bounding Box in the multidimensional case. Figure 1 shows the MBRs of five geographic objects consisting of European countries. Query q_2 illustrates why the refinement step is necessary: both the MBRs of Spain and Italy intersect with q_2 but none of these countries actually intersect with such query.

Definition 1. *Let o be a geographic object. The* Minimum Bounding Rectangle *of o, $MBR(o) = I_1(o) \times I_2(o)$ where $I_i(o) = [l_i, u_i](l_i, u_i \in \mathbb{R})$ is the minimum interval containing all the points of o along the dimension i.*

Definition 2. *Let q be an orthogonal rectangle $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$. An*

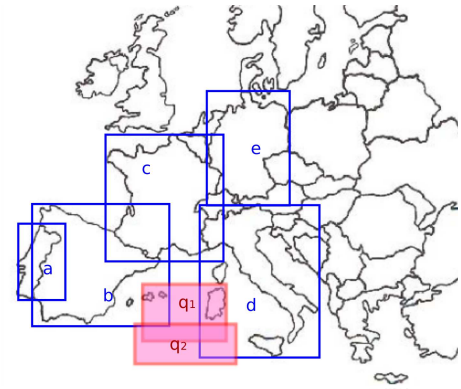


Figure 1: The problem modeled as MBRs intersecting with a query rectangle.

orthogonal range query (RQ) to find all the objects o having at least one point in common with q is defined as $RQ(q) = \{o \mid q \cap MBR(o) \neq \emptyset\}$.

In this paper we propose and compare several space-efficient representations for static collections of MBRs (or rectangles in general) supporting orthogonal range querying. These representations can be used as indexes in GIS, but applications in many other domains are feasible (CAD, VLSI, etc.).

Space-efficient data structures are a timeless topic of interest because of the memory hierarchy. Access times in upper levels of the hierarchy have decreased much faster than in lower levels. Thus, placing these data structures in upper levels of the memory hierarchy considerably reduces access times, by several orders of magnitude in some cases. In Figure 2 (Section 2), we show this with the example of an R-tree that can perform both in RAM and disk with minor changes (i.e., we just need to adapt the node size). We observe that the RAM variant performs up to 1,000 times faster. Note that we simulate a limited memory scenario using a virtual machine, and thus, times are not directly comparable with those shown in Section 8. In addition, dropping the space may be key not just to avoid using secondary memory (or lower levels of the memory hierarchy in general), but also to achieve feasible solutions when the memory is limited such as in mobile devices.

Finally, static indexes that take advantage of the knowledge of the data distribution are interesting in many applications where the data is semi-static (e.g., Geographic Information Retrieval systems [6]).

Preliminary partial versions of this work appeared in [7, 8]. Specifically, in [7] we introduce some of the ideas of the SE-PQ structure described in

Section 6. Similarly, [8] introduces the main idea of the WTR (Section 7). In this paper we analyze both structures more carefully and also perform a more exhaustive empirical comparison. In addition, we generalize the idea of the SE-PQ to a general strategy to design indexes on rectangle datasets (see Section 4). As a reward, we obtain new structures offering different space-time trade-offs. Those structures are also described in this paper. Moreover, we present several improvements on the CR-tree [9], and use this optimized structure in our experiments.

This paper is organized as follows. First, some basic concepts and related work are summarized in Sections 2 and 3. Then, in Section 4, we present a general strategy to develop indexes on rectangle datasets based on projecting to one-dimensional problems, and derive from it three space-efficient structures in Sections 5 and 6. In Section 7 we present a solution that does not fit in this general strategy, and that is adaptive to the complexity of the problem. Each structure presents some features that make it suitable for some scenarios. An experimental comparison of all the structures (and other ones in the state of the art) is shown in Section 8, which concludes with a discussion about the pros and cons of each structure. Finally, some conclusions and future lines of work are presented in Section 9.

2. Related Work

Orthogonal range search, both on point and general geographic object datasets, has received significant attention in the literature and many data structures have been proposed that achieve different space-time trade-offs. In this section we review the main related work.

Many different point access methods supporting orthogonal range search have been proposed along the years. The Kd-tree [1] is one of the most prominent because it achieves $O(\sqrt{n} + k)$[1] worst-case search time, which is optimal in k [10], and also performs well in practice. Non-linear-space structures, such as Range trees [11], can beat this worst-case search time. However, as we aim at space-efficient data structures these non linear-space solutions are out of the scope of this work.

The seminal computational geometry work by Chazelle [2] offers several space-time trade-offs, including one that in two dimensions requires

[1] k is the number of elements in the output. We use this *output-sensitive* analysis throughout the paper because otherwise range search is $\Theta(n)$ worst-case time.

$O(n \log U)$ 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 $[1, U] \times [1, U]$ and $0 < \epsilon < 1$ is a constant affecting memory consumption. The *wavelet tree* [12] can be regarded as a compact version of this structure that requires exactly $n \log U + o(n \log U)$ bits to index n points in the range $[1, U]$ (we assume binary logarithm). This was made explicit by Mäkinen and Navarro [13]. As the structures presented in this paper are closely related with this technique, we give further details on the technique in Section 3. Recent results [3, 4, 14], improve this structure in time and extend it to a general set of points in a discrete grid. However, they come with a significant implementation overhead.

Besides point access methods, many different spatial access methods have been also proposed along the years. A good survey of these structures is the one by Gaede and Günther [5]. Many of the methods initially proposed for point datasets have been generalized to support complex geographic objects. For example, the Extended Kd-tree and Skd-tree are extensions of the Kd-tree to support rectangles. In addition, other techniques allow the *transformation* of the original objects into a different representation that can be managed by point access methods or one-dimensional access methods (e.g., interval structures).

One of the most popular spatial access method and a paradigmatic example is the R-tree [15]. Several variations of the original R-tree have been proposed to improve its efficiency (e.g., the R^+-tree or the R^*-tree) and to take into account some specific problems (e.g., the STR R-tree for static data). Most of these proposals have been summarized by Manolopoulos et al. [16]. This structure does not provide worst-case guarantees (it could be forced to examine the entire tree in $O(n)$ time, even when the output is empty). However, it is simple to implement, uses linear space, and performs very well in practice. Although it has been original designed for secondary memory, it can be easily adapted to perform in main memory by reducing the size of the nodes. In Figure 2 we show that the difference between both alternatives is of several orders of magnitude.

There are some variants of the R-tree that use compression techniques achieving lower storage requirements than the STR R-tree. However, these structures produce precision loss, and thus, false positives. The CR-tree [9] stands out in this new setting. This structure is based on two key ideas. First, the coordinates of the MBRs are represented relatively to their parent node, and second, all the coordinates in a node are quantized with a fixed number of bits. The granularity of this quantization determines the precision

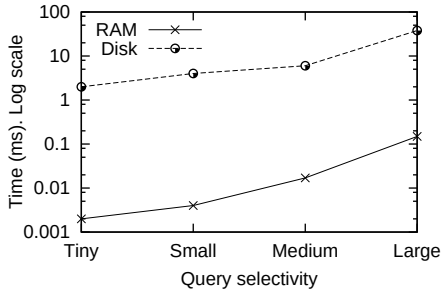


Figure 2: Performance comparison of a RAM-resident vs. a disk-resident R-tree.

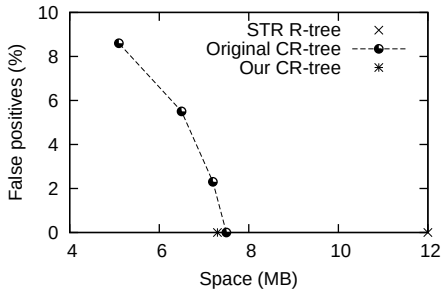


Figure 3: Influence in the number of false positives of a lossy-compression strategy.

of the structure, and thus, the number of false positives. In this paper, we assume a finite precision for the geographic coordinates, which is a realistic assumption in GIS. Under this assumption, the first contribution of this paper is a variant of the CR-tree that does not produce false positives. In addition, this structure takes advantage of the knowledge of the data distribution improving the compression of the original CR-tree for the static case (it can be devised as an STR CR-tree). In Figure 3 we motivate the interest of this structure with the example of the EIEL dataset (see Section 8 for more details). Both the classical and our variant of the CR-tree require around 60% the space of an STR R-tree without producing false positives. The CR-tree can save up to a 20% additional space by producing false positives. The decision of when this penalty is interesting depends on the complexity of the indexed objects (thus, it is completely application dependent). For example, if we consider some of the layers available in the TIGER dataset (see Section 8), on average, *primary roads* are composed of 200 points, *countys* of 1,800 points, and *states* of 15,000 points. Recall that each false positive requires to retrieve a complex object from disk and perform an intersection comparison, which complexity depends on the complexity of the object. Thus, we argue that, although interesting, lossy compression of spatial indexes is a different and non-comparable problem.

In our variant of the CR-tree , the MBRs stored at each node are ordered according to the projection of their left corner onto the x-axis. Therefore, the lower x-coordinate of an MBR can be differentially represented with respect to the lower x-coordinate of its left sibling (except for the first MBR of each node that is represented relatively to the lower x-coordinate of its parent). The representation of the lower y-coordinates is the same as in the original

CR-tree (i.e., relative to the lower y-coordinate of the parent). Finally, for each MBR the upper coordinates in both dimensions are differentially represented with respect to the lower coordinates of such MBR. As we assume that the original coordinates can be scaled to integers without losing any precision, this scheme produces a sequence of small integers that can be encoded using a variable length encoding. Note that, at each node, this sequence has to be sequentially decoded from the beginning, which does not add any time overhead because in a range query all the MBRs stored in a node visited during the traversal of the tree need to be compared with the query. In our experiments (Section 8) we refer to this variant as CR-tree (in spite of the aforementioned differences with the original CR-tree).

3. The Virtues of the Wavelet Tree

The wavelet tree [12] is a data structure used to store and index data in a compact way. Since Grossi et al. presented it in 2003, its many virtues [17] have been demonstrated and it has been widely used in many fields. For example, they have been used to index sequences [12, 18, 19], documents [20], and images [21]. They are efficiently implementable [22].

The basic tool used in the wavelet tree is the bit-vector *rank* operation: given a bit vector $B[1, n]$, the query $rank(B, i) = rank_1(B, i)$ returns the number of bits set to 1 in the prefix $B[1, i]$ of B. Symmetrically, $rank_0(B, i) = i - rank_1(B, i)$. The dual query to $rank_1$ is $select_1(B, j)$, which returns the position of the j-th bit set to 1 in B. The definition of $select_0(B, j)$ is analogous. For example, given a bitmap $B = 1000110$, $rank_0(B, 5) = 3$, $rank_1(B, 5) = 2$, $select_0(B, 4) = 7$, and $select_1(B, 3) = 6$. Both *rank* and *select* operations can be implemented in constant time and using little additional space ($o(n)$ in theory) on top of B [23, 24]. This is achieved by storing precomputed results of the operations at regular positions. Many practical implementations of these structures sacrifice some theoretical guarantees to perform better in practice [25]. The construction of the structures requires linear time and little extra space [26, 27].

In [28] we adapt this structure to index two-dimensional points in GIS. The experimental evaluation in that paper shows that the wavelet tree keeps a good trade-off between the space needed to store the index and its search efficiency. Figure 4 illustrates how a grid of n points with one point in each row and column is represented using a wavelet tree. Let P_{X_i} be the i-th point sorted along the x-axis (longitudes) and P_{Y_i} the i-th point along the y-axis

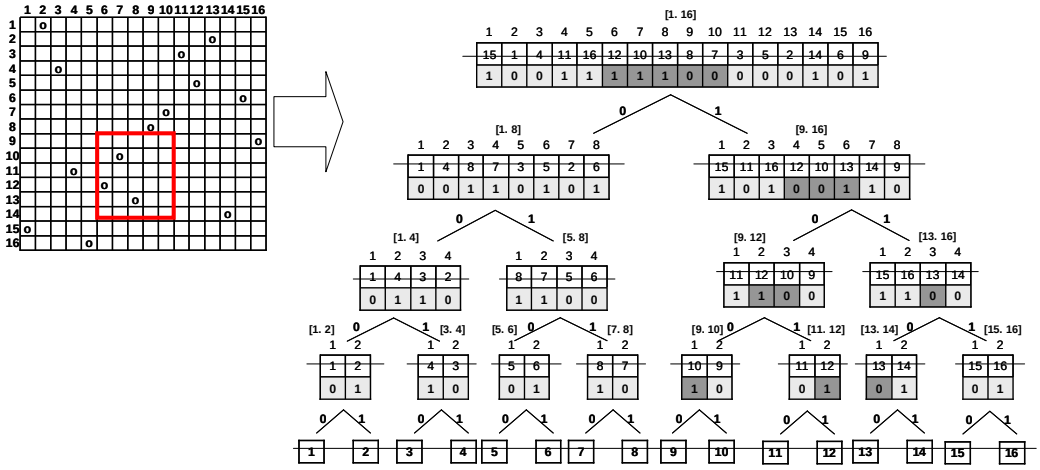


Figure 4: Representing an $n \times n$ grid of points using a wavelet tree. We highlight the areas corresponding to the query $[9, 14] \times [6, 10]$.

(latitudes). The root of our structure is a bitmap $B = b_1 \dots b_n$ where each position i represents the i-th point assuming them ordered along the x-axis. We set $b_i = 0$ if $P_{X_i} \in \{P_{Y_1} \dots P_{Y_{n/2}}\}$, and $b_i = 1$ if $P_{X_i} \in \{P_{Y_{n/2+1}} \dots P_{Y_n}\}$. The sequence of points given a 0 in this vector is recursively represented in the left child of the node, and those marked 1 are represented in the right child of the node. Hence, each node indexes half the points indexed by its parent node. This process is repeated recursively in each node until the leaves, where the sequence of indexed symbols corresponds to the permutation sorting the points along the y-axis. Therefore, the wavelet tree can be regarded as a device for progressively shuffling the points, initially sorted along the x-axis (tree root), into sorted along the y-axis (tree leaves). Only n bits are actually stored at each tree level, for a total of $n\lceil \log n \rceil$ bits, plus $o(n \log n)$ for the *rank* and *select* data structures.

Given an orthogonal range query $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$ and being $B_{1,n}$ the bitmap at the root of the tree, we can project the range $[l_2^q, u_2^q]$ onto the second level using *rank* operations. Note that we assume that the first interval of q are rows (latitudes) and the second one are columns (longitudes). Thus, the interval of columns ($[l_2^q, u_2^q]$) determines valid ranges inside the nodes of the wavelet tree and the interval of rows ($[l_1^q, u_1^q]$) determines nodes that can be pruned. We use $rank_0$ to project the range onto the left child as $[l_2^L, u_2^L] = [rank_0(B, l_2^q - 1) + 1, rank_0(B, u_2^q)]$ and $rank_1$ to project onto the right child as $[l_2^R, u_2^R] = [rank_1(B, l_2^q - 1) + 1, rank_1(B, u_2^q)]$. This process is

repeated recursively in each node until the leaves are reached. Nonetheless, paths are abandoned when a node v covering an interval $[l_1^v, u_1^v]$ does not intersect with the interval $[l_1^q, u_1^q]$, or when $[l_2^v, u_2^v]$ is empty. If we reach a leaf $[l_2, l_2]$ without discarding it, then the point with row value l_2 is part of the answer. In the worst case, each element is reported in $O(\log n)$ time.

Lemma 1. *An $n \times n$ grid with one point in each row and column can be stored in $n\lceil \log n \rceil + o(n \log n)$ bits, supporting orthogonal range queries in time $O((k+1)\log(n/(k+1)))$, where k is the size of the output. The structure is built in $O(n \log n)$ time and extra bit space.*

Proof. The space follows from the definition of wavelet tree: a perfect binary tree where each point is represented exactly once per level using one bit. The term $o(n \log n)$ allows *rank* operations in constant time over the aforementioned bitmaps [23]. Therefore, we can project a range in a node onto their two children in constant time by means of *rank* operations at the endpoints of the range. If there are k elements in the output, these force us to visit at most k nodes per level, but in the first levels they cannot be all different. In the worst case we visit 2^i nodes up to the level i where $2^i = k$, and then we visit k nodes per level. For $k > 0$ this adds up to $\sum_{i=0}^{\log k - 1} 2^i + \sum_{i=\log k}^{\log n - 1} k = 2^{\log k} + k(\log n - \log k) = O(k + k \log(n/k))$ nodes. The other nodes visited are those the search abandons because either the mapped $[l_2^q, u_2^q]$ becomes empty, or because the range $[l_1^q, u_1^q]$ does not intersect the node range. The first ones are amortized because their sibling cannot also be empty if the parent interval is nonempty. The second ones are at most two per level, and hence $O(\log n)$ in total, as they limit a contiguous interval $[l_1^q, u_1^q]$. As $k \log(n/k)$ increases with k, it is dominated by $O(\log n)$ only if $k = 0$. The given time formula matches the result in either case.

Finally, it is very easy to build the wavelet tree node by node, within the stated time and space. □

Figure 4 highlights the nodes visited to solve a range query $q = [9, 14] \times [6, 10]$. In the first step we compute the projection onto the left child as $[l_2^L, u_2^L] = [rank_0(B, 6-1)+1, rank_0(B, 10)] = [3, 4]$, and similarly onto the right child as $[l_2^R, u_2^R] = [rank_1(B, 6-1)+1, rank_1(B, 10)] = [4, 6]$. However, it is not necessary to access the left child because it covers the range $[1, 8]$, which does not intersect with the query range $[l_1^q, u_1^q] = [9, 14]$. We repeat this process in the right child to access its children nodes. The range

of valid positions in its left child is computed as $[l_2^L, u_2^L] = [rank_0(B, 4-1)+1, rank_0(B, 6)] = [2, 3]$, and similarly in its right child as $[l_2^R, u_2^R] = [rank_1(B, 4-1)+1, rank_1(B, 6)] = [3, 3]$. Both ranges belong to valid nodes because both $[9, 12]$ and $[13, 16]$ intersect with the query range $[l_1^q, u_1^q] = [9, 14]$. If we repeat this process until the leaves are reached, we obtain the result composed of the points in the rows 10, 12, and 13.

Although we show this example on a discrete grid, the real coordinates of the n points are stored in ordered arrays to translate the real queries to rank space (see Section 4.1). In addition, the identifiers of the n points are stored in the same order of the leaves, to return those identifiers as the result of the query.

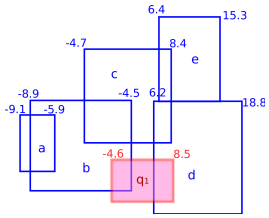
4. Reducing to One-Dimensional Problems

In this section we introduce a general strategy to design indexes on rectangle datasets, based on reducing the d-dimensional problem to d one-dimensional ones, which is a better studied case. The strategy consists of three steps: translation to rank space, projection of the problem into one-dimensional ones, and integration of the results. Although this general strategy is well known, we introduce at each step some ideas that will be used in our space-efficient variants. These will be developed in Sections 5 and 6.

4.1. Translation to rank space

Gabow et al. [29] proved that the orthogonal nature of the problem makes it possible to work with the rank of the coordinates (i.e., their position in sorted order) instead of the coordinates themselves. De Berg et al. [30] also proved that two-dimensional points sharing one coordinate can be ordered in both dimensions in a total order using the composite-numbers of their coordinates.

In practical terms this implies that we can store the real coordinates of the MBRs in sorted arrays and then use the ranks as the coordinates. Then, binary searches allow the mapping of queries in the original space to rank space (thus all the queries need $\Omega(\log n)$ query time). Depending on the way the data structure is organized, these arrays store different coordinates. In the simplest case, we can assume a sorted array for the x-axis coordinates and another one for the y-axis coordinates. This method to translate to rank space is particularly convenient for the design of space-efficient structures:



Rank	1	2	3	4	5	6	7	8	9	10
Original	-9.1	-8.9	-5.9	-4.7	-4.5	6.2	6.4	8.4	15.3	18.8
Scaled	1	3	33	45	47	154	156	176	245	280
Differences		2	30	12	2	107	2	20	69	35
Encoding		$\phi(2)$	$\phi(30)$	$\phi(12)$	$\phi(2)$	$\phi(107)$	$\phi(2)$	$\phi(20)$	$\phi(69)$	$\phi(35)$

Figure 5: Coordinate encoding.

storing the coordinates as sequences of non-decreasing values makes them easily compressible.

We present now a coordinate encoding scheme that takes advantage of this property. Let us assume these coordinates are scaled to integers in the range $[1, U]$. Geographic coordinates have finite precision: they can be represented in degrees or meters and in most cases they can be rounded to integer values, after appropriate scaling, without losing any precision.

Let $A = a_1 a_2 \dots a_n$ 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 $b_1 = a_1$, so that $a_i = \sum_{1 \le j \le i} b_j$. The array $B = b_1 b_2 \dots b_n$ is a representation of A that can be compressed by exploiting the fact that consecutive differences can be small. Several coding algorithms for integers can be used on these small numbers. In Section 8.3 we empirically compare four different well-known coding algorithms [31]: Elias-Gamma, Elias-Delta, Rice, and VBytes. Figure 5 shows the preprocessing steps used to transform the original floating-point coordinates to encoded values ($\phi(i)$ represents the integer value i encoded using a certain function ϕ).

In order to translate queries in the original space to rank space, this scheme must support the search for the endpoints of the original queries. Because duplicate coordinates are allowed, the result of those searches must always include in the resulting range the point coordinates that are equal to the query coordinates. Given a lower or upper coordinate v we are interested in finding, respectively, the leftmost $a_i \ge v$ or the the rightmost $a_i \le v$. We name these operations *leftSearch* and *rightSearch*, respectively. In order to solve them efficiently we store in an array the accumulated sum at regularly sampled positions (say every hth position, thus the array 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. The following, easy to prove, lemma summarizes the properties

of this storage scheme.

Lemma 2. *Let $\phi : \mathbb{N} \to \{0,1\}^*$ be an encoding function. Then a sorted array $A = a_1 a_2 \ldots a_n$ of coordinates in $[1, U]$ can be stored in $cs = \lceil n/h \rceil \log U + \sum_{i=2}^{n} |\phi(a_i - a_{i-1})|$ bits, supporting rightSearch and leftSearch in $O(\log(n/h) + d \cdot h)$ time, for any $1 \leq h \leq n$, where d is the time to decode a ϕ code.*

A simple formulation is obtained by using $h = \log n$ and Rice codes [31] with $\log(U/n)$ bits in the binary part of the representation. This is easy to modify to be decodable in constant time.

Corollary 1. *A sorted array $A = a_1 a_2 \ldots a_n$ of coordinates in $[1, U]$, $n = o(U)$, can be stored in $n \log(U)/\log(n) + n \log(U/n) + O(n) = n \log(U/n)(1 + o(1))$ bits, supporting leftSearch and rightSearch in $O(\log n)$ time. If $U = O(n)$ the space is $O(n)$ bits.*

The case of small U is obtained by just marking in a bitmap of length $n + U$ the number of points with each coordinate value in unary. That is, if there are n_i coordinates equal to i, the bitmap is $0^{n_1}10^{n_2}1 \ldots 0^{n_U}1$. Then the mapping is easily done with *rank*/*select* queries. In any case, the construction time of the representation is bounded by the $O(n \log n)$ time needed to sort the coordinates.

4.2. Projection

The resulting one-dimensional subproblems after projection onto any dimension are *interval stabbing* problems, that is, find the intervals of a set that intersect a query interval (see Figure 6). This problem is well-known and many structures solving it can be found in the literature. For example, common structures like Interval trees and Priority trees (see de Berg et al. [30] for a good survey) solve the problem in time at least $\Omega(\log n + k)$.

However, one can do better when working in rank space. Schmidt [32] presented a structure solving several interval stabbing problems in rank space in optimal time $O(k + 1)$ and linear space. It can be built in $O(n)$ time and $O(n \log n)$ extra bits, given the sorted coordinates.

Schmidt defines the *parent* of an interval a, *parent*(a), as the one with rightmost starting point among all intervals that cover a. This *parent* relation defines a tree, where sibling intervals are sorted left to right (note that sibling intervals cannot contain each other). The root of the tree is a special node that acts as the parent of all the nodes not covered by any interval. In order

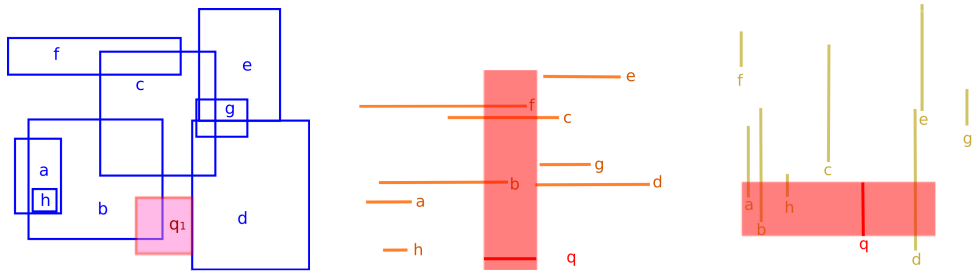


Figure 6: Decomposition of a two-dimensional query into two one-dimensional problems.

to solve interval stabbing queries, the structure stores for each point the rightmost-starting interval stabbed by the point (array *start*, see Figure 7).

Array *start* serves to find, at query time, the rightmost minimal interval that intersects the query $[l^q, u^q]$, $start(u^q)$. After reporting it, the algorithm traverses the tree using a recursive procedure that reports all the siblings to the left of a reported node, stopping at the first sibling that is not stabbed by the query interval. For each reported node, it also considers its descendants (starting recursively from the rightmost child), the ancestors of the first node reported, and their descendants to the left of the ancestor path. This procedure is easily seen to traverse $O(k+1)$ nodes in order to report k results [32].

It is possible, however, that the upper coordinate of the query interval does not stab any interval. A solution to this problem requires maintaining a second array[2]. However, in our case, where the ranked coordinates are not the original universe but come from endpoints of the original intervals, a better solution is possible.

Consider a query $Q = [L^q, U^q]$ in the original space. This is mapped via binary search to $q = [l^q, u^q] = [leftSearch(L^q), rightSearch(U^q)]$. Let us call $o(v)$ the original coordinate in the set of intervals that was mapped to v in rank space. If $U^q = o(u^q)$, then $start(u^q)$ is the rightmost minimal interval containing U^q. If, instead, it holds $o(u^q) < U^q < o(u^q + 1)$, the right answer is still $start(u^q)$ if $o(u^q)$ corresponds to a lower coordinate of an interval (as the intervals that cover $o(u^q)$ and U^q are exactly the same). Yet, if $o(u^q)$ is an upper coordinate, $start(u^q)$ may end at $o(u^q)$ and not cover point U^q.

To handle this case, note that if the query was slightly different, $Q' =$

[2] J. Schmidt, personal communication.

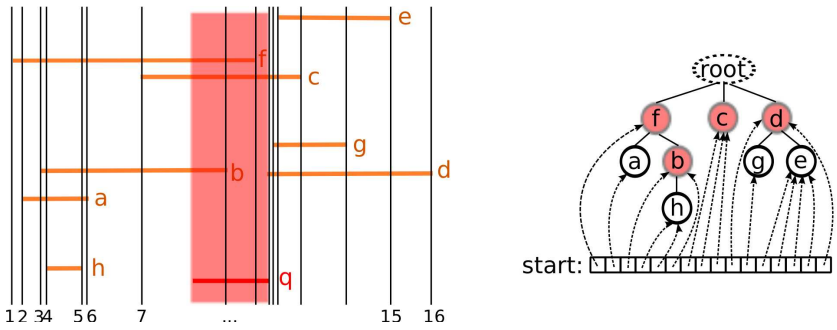


Figure 7: Representing x-axis intervals in Figure 6 using Schmidt's structure.

$[L^q, U^{q'}]$, with $U^{q'} = o(u^q + 1)$, then the correct starting point would have been $start(u^q + 1)$. If $o(u^q + 1)$ is an upper coordinate, then the intervals that contain $U^{q'}$ and those that contain U^q are exactly the same, and we can run the normal procedure starting from $start(u^q + 1)$. If $o(u^q + 1)$ is a lower coordinate, then $start(u^q + 1) = [o(u^q + 1), x]$ and the only difference between Q and Q' is that Q does not intersect this last interval. Hence we can run the normal procedure starting from $start(u^q + 1)$ and exclude the first value.

Although this structure requires linear space, this space is considerably high. Given n two-dimensional points, for each coordinate it stores a tree of n nodes (each storing 4 pointers: to its parent, rightmost child, next left sibling, and upper coordinate of the interval) and array *start* with $2n$ entries. This gives a total of $6n\lceil \log n \rceil$ bits per dimension, in addition to the storage of the real coordinates and the identifiers.

A way to decrease the space by $n\lceil \log n \rceil$ bits is to map only the lower coordinates of intervals, and build array *start* only for those coordinates. Then, if query $Q = [L^q, U^q]$ is mapped to $q = [l^q, u^q]$, we only know that there is an interval starting at $o(u^q)$, which may or may not contain U^q (we would only be sure it does if $U^q = o(u^q)$). What is sure is that $o(u^q + 1)$ starts after U^q, and then we can use $start(u^q + 1)$ to boot the recursive reporting algorithm, omitting the first element (see Figure 8).

The reason why this works is, again, that if the query had been $Q' = [L^q, o(u^q + 1)]$, then $start(u^q + 1)$ would have been the starting point yielding the correct output. We analyze now which intervals belong to the output of Q' but not of Q. Between U^q and $o(u^q + 1)$ there can only be upper coordinates, $U^q < u_1 < u_2 < \ldots < u_r < o(u^q + 1)$, but no lower coordinates. All those upper coordinates belong to intervals that also intersect Q, and

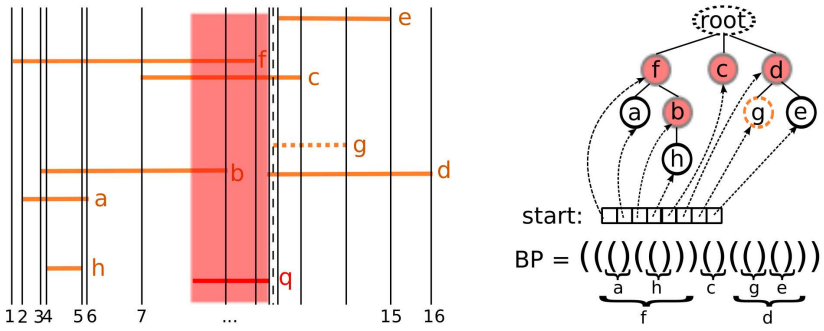


Figure 8: Representing x-axis intervals in Figure 6 using a variation of the Schmidt's structure and balanced parenthesis (BP) representation of the tree.

all other intervals belong to either none or both outputs as they have no endpoints between U^q and $o(u^q + 1)$. Thus the only difference between both outputs is the interval $start(u^q + 1)$ itself.

Since upper coordinates of intervals are not used for mapping, we store these upper coordinates in the tree in their original form. This way the tree stores only 3 values per node, apart from the coordinate data. This subtracts other $n\lceil \log n \rceil$ bits from the data structure.

Lemma 3. *Given a set of n intervals, there is a data structure using $4n\lceil \log n \rceil$ bits, plus the storage needed for the coordinates, able to report all the k intervals that intersect a query interval in time $O(\log n + k)$.*

4.3. Intersection

The final step is to intersect the results obtained at each projected dimension. This intersection can be performed on-line using an additional bit-vector of n bits (whose space is charged to that of the index itself). The algorithm solving the first subproblem marks the results obtained in this bit-vector and the algorithm solving the second subproblem reports the results obtained if they have been previously marked (the unmarking can be done by rerunning the first query to maintain the same complexities, or in practice, storing the results of the first query or sweeping the bitmap).

We do not assume that the identifiers are in the range $[1, n]$. Rather, we store intervals sorted according to their first coordinate and use $n\lceil \log n \rceil$ additional bits to store the permutation from the order in the second coordinate to the order in the first coordinate.

Although we focus on two dimensions, all the structures obtained with this general strategy are easily generalizable to d dimensions. Combining the techniques developed in this section (while still representing the coordinates in plain form) we have the following result. We call the structure *IS*, for "interval stabbing".

Theorem 1. *Given n rectangles with coordinates in $[1, U]$, there is a data structure (called* IS*) that requires $n(4 \log U + 8 \log n + O(1))$ bits and finds the rectangles that intersect a query rectangle in time $O(\log n + k_1 + k_2)$, where k_1 and k_2 are the intervals that intersect the query when projected along each dimension. The structure is built in $O(n \log n)$ time and extra bit space.*

Note this structure returns an identifier of the interval in the range $[1, n]$. The application may store an array mapping those positions to pointers or another kind of identifier.

5. Space-Efficient Interval Stabbing Structures

5.1. Fully-functional tree-based structure (IS-FF)

Succinct data structures represent data types using an amount of space close to the theoretic lower bounds, and still supporting common navigation and search operations. In the case of trees, the general pointer representation requires $\Theta(n \log n)$ bits whereas succinct representations require $2n + o(n)$ bits and support a broad range of operations in constant time. A subfamily of these succinct representations called BP, for balanced parentheses, represents the tree via a DFS preorder traversal, where a parenthesis is opened when arriving at a node and closed when leaving the node. A recent BP-based proposal, named Fully-Functional Succinct Trees (FF) [33], achieves constant time for the widest range of operations and can be built in $O(n)$ time. Arroyuelo et al. [34] experimentally compared various succinct representations for trees and concluded that FF offers an excellent space-time trade-off and wide functionality. In our space-efficient variant of the IS structure we use FF trees to represent the tree of intervals. An example of the generic BP representation is given in Figure 8.

For each dimension, we store the lower coordinates of the intervals in a sorted array (using the storage scheme proposed in Section 4.1).

Instead of representing array *start*, we will handle the queries using the mapping between coordinates and tree nodes. Since a preorder traversal of

the tree enumerates the intervals by increasing lower coordinate, it turns out that the preorder numbers of the tree nodes correspond to positions in the array sorted by lower coordinate. FF representation solves in constant time operation *preorderSelect*(p), which gives the tree node with preorder p.

To solve an interval intersection query we first map it via binary searches to $q = [l^q, u^q]$ in the array that contains the lower coordinates. Then we proceed as in the final strategy described in Section 4.2. We start the tree traversal from the node representing the interval with lower coordinate u^q+1 and recursively traverse the tree from that node, omitting the first interval from the output.

Starting from the node, we apply the recursive traversal of the tree using operations *parent*, *lastChild* (which gives the last child of a node), and *prevSibling*(i) (which gives the previous sibling of a node), all of which are computed in constant time. The final challenge is how to compare the upper coordinate of an interval in the tree with l^q, in order to determine whether the interval is to be reported and the traversal must continue. FF provides in constant time operation *preorderRank*, which gives the preorder of a tree node. We store the upper coordinates in preorder and then access that array at the position given by *preorderRank*. Algorithm 1 shows the pseudo-code to do this traversal.

Since the upper coordinates are not stored in increasing order of values, we cannot use the compact storage technique of Section 4.1. In practice, however, there is some correlation between consecutive MBRs, so we use a folklore technique for sequences that are not necessarily increasing. Instead of storing the differences $b_i = a_i - a_{i-1}$, we store the *xor*, $b_i = a_i$ xor a_{i-1}, and represent its significant bits using, for example, Rice codes. A regular sampling is added to provide direct access, just as in Corollary 1.

Theorem 2. *Given a set of n rectangles with coordinates in $[1, U]$, $n = o(U)$, there is a data structure (called* IS-FF*) that requires $n(4\log U - \log n + o(\log(U/n)))$ bits and finds the rectangles that intersect a query rectangle in time $O(\log n + k_1 + k_2)$, where k_1 and k_2 are the intervals that intersect the query when projected along each dimension. If $U = O(n)$, the space is $3n\log n + O(n)$ bits. The structure is built in $O(n\log n)$ time and extra bit space.*

Proof. Construction and time complexities have been already discussed. The space includes, for each dimension, $n\log(U/n)(1 + o(1))$ bits for storing the lower coordinates (Corollary 1), $n\lceil\log U\rceil$ for the upper coordinates, and

Algorithm 1 Tree traversal for IS-FF. *ffTree* is the fully-functional tree, $Q = [L^q, U^q]$ is the query interval, and L and U the storage of the lower and upper coordinates, respectively. S is an auxiliary stack.

IS − FF($ffTree, Q = [L^q, U^q], L, U$)

```
1: cNodePreorder ← rightSearch(L, U^q) + 1
2: cNode ← ffTree.preorderSelect(cNodePreorder)
3: while ffTree.parent(cNode) ≠ ∅ do
4:   cNode ← ffTree.parent(cNode)
5:   S.push(cNode)
6:   cNodePreorder ← ffTree.preorderRank(cNode)
7:   Mark/report interval cNodePreorder
8: end while
9: while S ≠ ∅ do
10:  cNode ← ffTree.prevSibling(S.pop())
11:  while cNode ≠ ∅ do
12:    cNodePreorder ← ffTree.preorderRank(cNode)
13:    if U.access(cNodePreorder) < L^q then
14:      break
15:    end if
16:    S.push(cNode)
17:    Mark/report interval cNodePreorder
18:    cNode ← ffTree.lastChild(cNode)
19:  end while
20: end while
```

$2n + o(n)$ for structure FF. In addition we need $n\lceil \log n \rceil + n$ bits for the intersections (Section 4.3). □

5.2. Independent interval set-based structure (IS-IIS)

Although the space required by the IS-FF improves the other structures presented in this paper, the query time of that structure is not competitive (we shall show this in Section 8), and this is mainly due to the performance of the Fully-Functional tree. In this section we present an alternative representation for interval sets that does not require the usage of such trees. We name this structure IS-IIS because it is based on a decomposition into Independent Interval Sets.

Definition 3. *A set of intervals* $I = \{i_1, \ldots, i_n\}$ *is* independent *if no interval* $i_j \in I$ *is strictly contained in other interval* $i_k \in I$*. More precisely,* $\forall i_j = [l^j, u^j], i_k = [l^k, u^k], i_j, i_k \in I$*, if* $l^j < l^k$ *then* $u^j \leq u^k$*.*

Reporting the k intervals in an independent interval set that intersect a query $Q = [L^q, U^q]$ can be easily solved in $O(\log n + k)$ by storing the intervals in order. Note that by definition of independent set the order of the lower and upper coordinates is the same. We first locate the rightmost interval stabbed by U^q (using binary search), and then we walk through the set (from right to left) reporting all the intervals until we find one that does not intersect the query.

If the set of intervals is not independent, we can decompose it into m independent sets in $O(n \log m)$ time, once the intervals have been ordered by left endpoint. The algorithm is well-known to find the longest increasing subsequence in a stream of numbers, and it is also related to the problem of decomposing a permutation Π over $[1, n]$ into the minimum number of shuffled (i.e., not necessarily consecutive) upsequences [35] (the rightmost endpoints of the intervals correspond to the permutation values). Our algorithm is equivalent to Fredman's [36] to find the optimal number of shuffled upsequences.

In the general case, reporting the intervals that intersect a query Q can be solved in time $O(m \log n + k)$ by running the algorithm proposed above in each independent set . However, we can improve this query time to $O(\log n + m + k)$. The data structure stores the lower coordinates of all the intervals in ascending order using a single array (the storage scheme proposed in Section 4.1 is used to save space). Hence, the m binary searches to find the rightmost interval stabbed by the query in each group are replaced by a single binary search. Let $l^q = \mathit{rightSearch}(U^q) + 1$ be the result of such binary search. If we can compute in constant time the rank of rightmost interval of each group that starts before l^q (let us name this operation $\mathit{rankAll}(l^q)$), the algorithm performs in the claimed complexity (just by walking through the lower coordinates of each group starting from the position computed by *rankAll*). In parallel to the lower coordinates, we store a sequence that indicates for each coordinate the respective group to which it belongs. A wavelet tree can be used to store such sequence [12, 18, 19] in $n\lceil \log m \rceil + o(n \log m)$ bits supporting $\mathit{rankAll}(l^q)$ in $O(m)$ time (i.e., constant time per group). The algorithm traverses the whole wavelet tree from the root to the leaves. It starts at position l^q and maps this position into each

node until the leaves, where mapped positions are the result of the query. Thus, the algorithm performs $1 + 2 + 4 + \ldots + m = O(m)$ rank operations.

Lemma 4. *Given a set of n intervals, there is a data structure using $n\lceil \log m \rceil + o(n \log m)$ bits, plus the storage needed for the coordinates, able to report all the k intervals that intersect a query interval in time $O(\log n + m + k)$.*

In the algorithm we have just described, the lower coordinates of each maximal set cannot been stored using the general scheme proposed in Section 4.1 because they are decompressed from right to left. However, a symmetric traversal starting at the first interval that ends after L^q, and walking from left to right, computes the same result. This algorithm allows the use of the compact storage technique of Section 4.1.

Theorem 3. *Given a set of n rectangles with coordinates in $[1, U]$, $n = o(U)$, there is a data structure (called* IS-IIS*) that requires $n(4 \log(U/n) + 3 \log m + o(\log(U/n)))$ bits and finds the rectangles that intersect a query rectangle in time $O(\log n + m + k_1 + k_2)$, where k_1 and k_2 are the intervals that intersect the query when projected along each dimension, and m is the sum of the number of independent sets along both dimensions. If $U = O(n)$, the space is $3n \log m + O(n)$ bits. The structure is built in $O(n \log n)$ time and extra bit space.*

6. Space-efficient Point Query Structure (SE-PQ)

Instead of reducing the two-dimensional problem to one-dimensional interval stabbing problems, in this section we reduce it to two-dimensional range search problems, yet on points rather than on rectangles. As noted in Section 2, there are many structures solving orthogonal range searching on point datasets. We use wavelet trees to achieve a space-efficient structure, while keeping competitive query times.

The idea is to interpret an interval $a = [l^a, u^a]$ as a point (l^a, u^a) in an integer grid $n \times n$ and then transform the queries in the original space to orthogonal range queries in this grid. The following, easy to verify, observation provides a basis for this transformation. It says, essentially, that in the original space an intersection between a query q and an object o (an MBR) occurs when, across each dimension, the query does not finish before the MBR starts, and the query does not start after the MBR finishes.

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

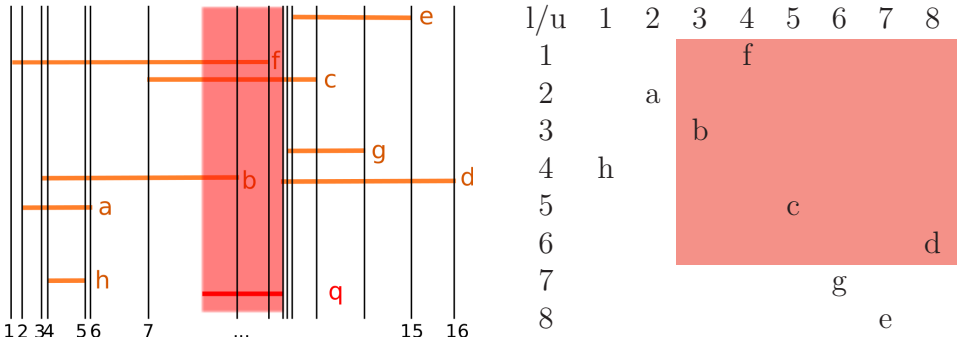


Figure 9: Transforming x-axis intervals in Figure 6 into a two-dimensional grid.

Let us define the function $lorder_i(o)$, which computes the order of the object o in the dimension i according to the lower coordinate l_i^o, and the analogous $uorder_i(o)$ defined according to the upper coordinate u_i^o. Figure 9 shows the transformation of the x-axis intervals in the previous example to a two-dimensional grid. In this example, $lorder_1(f) = 1$, $lorder_1(a) = 2$, $lorder_1(b) = 3$, $uorder_1(f) = 4$, $uorder_1(a) = 2$, and $uorder_1(b) = 3$. These functions provide a mechanism to compute the cell of an object o in the first grid as $c_{g_1}(o) = (lorder_1(o), uorder_1(o))$ and its cell in the second grid as $c_{g_2}(o) = (lorder_2(o), uorder_2(o))$. In the example, $c_{g_1}(f) = (lorder_1(f), uorder_1(f)) = (1, 4)$, $c_{g_1}(a) = (2, 2)$, and $c_{g_1}(b) = (3, 3)$.

This transformation involves the construction of two-dimensional grids with one point in each row and in each column. Therefore we can represent each grid with a wavelet tree as described in Section 3. In each wavelet tree (i.e., in each dimension) we perform a query derived from the transformation of the original query q into the new space. The query $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$ is decomposed according to its two dimensions, resulting in a query to each wavelet tree: $q_{wt_1} = [1, u_1^q] \times [l_1^q, n]$ and $q_{wt_2} = [1, u_2^q] \times [l_2^q, n]$. The conceptual idea behind this formula is to retrieve those MBRs with lower coordinates that are less or equal than the upper coordinate of the query and with upper coordinates that are greater or equal than the lower coordinate of the query (in each dimension). These queries are two-sided, and they are a particular case of the four-sided queries that the aforementioned wavelet tree handles. Hence, an optimization of the algorithm to solve the queries is possible.

Algorithm 2 shows the pseudo-code to solve these particular two-sided queries. This algorithm is very similar to the point case but there are two important differences. First, just one *rank* operation (lines 7 and 9) has to

Algorithm 2 Wavelet tree traversal for SE-PQ. *cNode* is the wavelet tree node, $maxR$ the maximum row, and $minC$ the minimum column. The initial call is $\mathbf{PQ}(root, n, 1)$. It outputs points ranked by their y-axis.

$\mathbf{PQ}(cNode, maxR, minC)$

1: $maxRange \leftarrow maxR$
2: **if** $maxRange \geq 1$ **then**
3: **if** $(maxRange = cNode.size)$ and $(cNode.range \subseteq [minC, n])$ **then**
4: **output** all $ids \in cNode.range$
5: **else**
6: **if** $cNode.leftChild.range \cap [minC, n] \neq \emptyset$ **then**
7: $\mathbf{PQ}(cNode.leftChild, rank_0(cNode, maxRange))$
8: **end if**
9: $\mathbf{PQ}(cNode.rightChild, rank_1(cNode, maxRange))$
10: **end if**
11: **end if**

be performed to compute the valid range in each node, because these valid ranges always start in the first position of the node. Second, the recursive call to the right child (line 9) is always performed because the range used to prune the tree traversal only prunes left branches.

We store the lower and upper coordinates separately, each in increasing order using the compact representation of Section 4.1. Then binary searches convert the query coordinates into rank space along each dimension. The resulting structure is called *SE-PQ* (for "point queries") and offers the following trade-off.

Theorem 4. *Given a set of n rectangles with coordinates in $[1, U]$, $n = o(U)$, there is a data structure (called* SE-PQ*) that requires $n(4 \log U - \log n + o(\log U))$ bits and finds the rectangles that intersect a query rectangle in time $O((k_1 + k_2 + 1) \log(n/(k_1 + k_2 + 1)))$, where k_1 and k_2 are the intervals that intersect the query when projected along each dimension. If $U = O(n)$, the space is $3n \log n + o(n \log n)$ bits. The structure is built in $O(n \log n)$ time and extra bit space.*

Proof. The coordinates are stored in $4n \log(U/n)(1 + o(1))$ bits according to Corollary 1. From Lemma 1 we can represent each of the two grids in $n\lceil \log n \rceil + o(n \log n)$ bits. Additionally, $n\lceil \log n \rceil + n$ bits are necessary for the intersection, recall Section 4.3. The $O(\log n)$ time to translate to rank

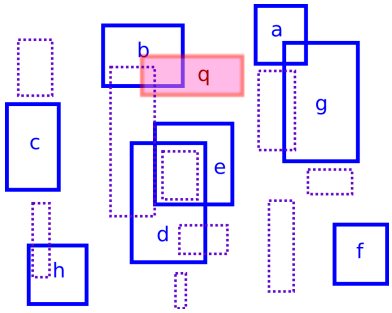


Figure 10: Base example for the WTR structure.

space is absorbed by that to report the k_1 and k_2 points from the wavelet trees, $O((k_i + 1)\log(n/(k_i + 1))$ (Lemma 1). As this function increasing sublinearly with k_i, the costs for k_1 and k_2 can be added as stated in the theorem. Construction is dominated by sorting the coordinates and building the wavelet trees. □

7. A Wavelet Tree on Rectangles (WTR)

We propose a new structure for the rectangle intersection problem that considerably deviates from the framework used in the previous sections. It also uses a translation to rank space and a decomposition of the problem, but this decomposition is not on the dimensions. Instead we define a new decomposition, into *independent sets*, where each resulting subproblem (each independent set) is also a two-dimensional rectangle intersection problem where rectangles do not contain each other along the x-axis (i.e., their x-projections form a set of independent intervals, see Definition 3). This decomposition results in a structure that is adaptive in the number of independent sets required to decompose the space. Figure 10 shows a new example of the orthogonal range query problem that we use as a base to describe our structure.

7.1. Data structure

Recall we assume that the first dimension represents rows (y-axis or latitudes) and the second represents columns (x-axis or longitudes). We will describe a structure able to handle *independent sets* of rectangles.

Definition 4. *A set of MBRs* $g = \{m_1,\ldots,m_n\}$ *is* independent *if no projection over the x-axis of an MBR* $m_i \in g$ *is strictly contained in the projection*

over the x-axis of another MBR $m_j \in g$, that is, if $\{[l_2^1, u_2^1], \ldots, [l_2^n, u_2^n]\}$ is an independent set of intervals.

If the set of MBRs to index is not independent, we will decompose it into m independent sets using the same algorithm described in Section 5.2. All the solid rectangles (identified by letters) in Figure 10 form a single independent set, whereas the dotted rectangles must form other independent sets.

Let n be the number of MBRs in an independent set, each one described by two pairs $\{(x_l,y_l),(x_u,y_u)\}$ (the real coordinates of two opposite vertices). These MBRs can be represented in rank space in a $2n \times 2n$ grid with only one point in each row and column (see Section 4.1). Let X^l, X^u, and Y be ordered arrays storing the real coordinates of the MBRs. X^l stores all the x-axis lower coordinates (x_l), X^u stores the x-axis upper coordinates (x_u), and Y stores y-axis coordinates (both lower, y_l, and upper, y_u, in the same array). In Figure 11 each position in the Y array has been annotated with the identifier of the corresponding MBR for clarity, but these identifiers are not stored. In addition, in this figure we use the original identifier when the coordinate corresponds to a lower coordinate and append a quote (') when it corresponds to an upper coordinate (for example, the MBR a starts at position 1 and ends at position 4). Note that the order of the lower (X^l) and upper (X^u) coordinates in the x-axis is the same because the set is independent. In Section 7.2 we show how these arrays are used to translate real queries to rank space. The space-efficient storage scheme presented in Section 4.1 considerably reduces the space necessary to store them.

A wavelet tree-like structure (Figure 11) with $\lceil \log 2n \rceil$ levels is used to represent the independent set of MBRs. Akin to the basic wavelet trees (Section 3), this is a binary tree where each node covers a range of positions in the Y array that represents the first half of the array 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 $Y[1, 2n]$.

Each node in the tree stores two bitmaps, B_1 and B_2, of the same length, and each position in these bitmaps corresponds to an MBR *represented* in the node, ordered by their x-axis (in the figure, these positions have been annotated with the identifier of the corresponding MBR). Let $Y[lB, uB]$ be the range of Y values covered by the node. Then an MBR is *represented* in that node if its y-axis extension intersects $Y[lB, uB]$, that is, it does not finish before $Y[lB]$ and it does not start after $Y[uB]$. Let $\{m_1, m_2, \ldots\}$ be the rectangles represented in the node. Then B_1 and B_2 tell whether the

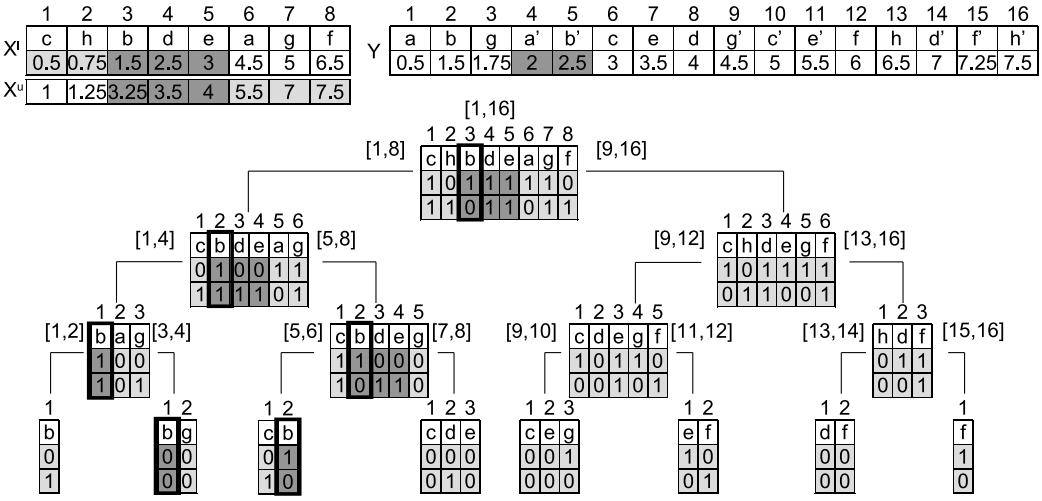


Figure 11: Representing the independent set in Figure 10 using a wavelet tree-like structure. We highlight the traversal for query $Q = [2.0, 2.75] \times [2.0, 3.5]$.

nodes are represented in the left and right children, respectively, according to Eqs. (1) and (2). Note that an MBR can be represented in both the left and right child of the node and thus both $B_1[i]$ and $B_2[i]$ can be 1 simultaneously, but not 0 simultaneously.

$$B_1[i] = \begin{cases} 1 & \text{if } l_1^{m_i} \leq Y[\lfloor \frac{lB+uB}{2} \rfloor] \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$$B_2[i] = \begin{cases} 1 & \text{if } u_1^{m_i} \geq Y[\lfloor \frac{lB+uB}{2} \rfloor + 1] \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

As such, this structure might require quadratic space. The reason is that an MBR with a large extent in the y-axis can be represented in $\Theta(n)$ nodes at the same level. In order to solve this problem Eq. (3) slightly modifies the way the structure is created. When an MBR m_i completely contains the range covered by the node, this will be signaled with the assignment $B_1[i] = B_2[i] = 0$, and m_i will not be stored in the descendants of this node (a similar technique is used in segment trees [30]). Then each MBR can be stored at most four times per level and we can guarantee logarithmic bit-space per MBR (Lemma 5).

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

Lemma 5. *Given an independent set of n rectangles with coordinates in $[1, U]$, $n = o(U)$, structure* WTR *requires at most $n(4 \log U + 4 \log n + o(\log U))$ bits. If $U = O(n)$, the space is at most $8n \log n + o(n \log n)$ bits.*

Proof. Each of the n MBRs has two endpoints. Each appears in a tree node at each level. The MBR spans that node, plus possibly its sibling that is completely covered by the MBR. In each of those 4 nodes per level the MBR induces 2 bits, in B_1 and B_2, for a total of $8n \log(2n)$ bits, plus $o(n \log n)$ bits for *rank* queries. Added to the arrays X^l and X^u, each requiring $n \log(U/n)(1 + o(1))$ and Y requiring $2n \log(U/(2n))(1 + o(1))$ (Corollary 1), we have the result. □

7.2. Solving queries

Our wavelet tree-like structure can be used to solve range queries in rank space derived from the translation of the original queries in geographic space using the ordered arrays of coordinates (X^l, X^u, and Y). The scheme presented in Section 4.1 stores these arrays space-efficiently. We make use of operations *leftSearch* and *rightSearch* introduced in that section.

A query in geographic space $Q = [Y_l, Y_u] \times [X_l, X_u]$ is translated into $q = [y_l, y_u] \times [x_l, x_u]$ in rank space as follows:

$$\begin{aligned} y_l &= \mathit{leftSearch}(Y, Y_l), & x_l &= \mathit{leftSearch}(X^u, X_l), \\ y_u &= \mathit{rightSearch}(Y, Y_u), & x_u &= \mathit{rightSearch}(X^l, X_u). \end{aligned}$$

Note the upper x-axis coordinates of the MBRs are searched for the lower x-axis coordinate of the query, and vice versa. In this way, $[x_l, x_r]$ will cover the ranks in bitmaps B_1 and B_2 of the MBRs that intersect the query: We will exclude those MBRs $o = m_i$ such $X^l[i] > X_u$ or $X^u[i] < X_l$. In the nomenclature of Observation 1, this is $l_2^o > u_2^q$ or $u_2^o < l_2^q$. For example, the query $Q = [2.0, 2.75] \times [2.0, 3.5]$ translates into $q = [4, 5] \times [3, 5]$ ($\mathit{leftSearch}(Y, 2.0) = 4$, $\mathit{rightSearch}(Y, 2.75) = 5$, $\mathit{leftSearch}(X^u, 2.0) = 3$, $\mathit{rightSearch}(X^l, 3.5) = 5$).

Algorithm 3 shows the recursive method to solve range queries once they have been translated into 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. The algorithm recursively projects a range, $[x_l, x_u]$ at the beginning, onto the left child using $rank_1$ operations over bitmap B_1, and onto the right child using $rank_1$ over B_2. 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., MBRs at positions i where $B_1[i] = B_2[i] = 0$), which are added to the result in the merging stage. Those local results can be obtained using an extension of the *rank* and *select* operations introduced in Section 3. Given two bitmaps B_1 and B_2, $rank_{00}(B_1, B_2, p)$ counts the number of positions $i \leq p$ where $B_1[i] = B_2[i] = 0$. For example, given $B_1 = 1\underline{0}0011\underline{0}$ and $B_2 = 0\underline{0}1101\underline{0}$, $rank_{00}(B_1, B_2, 7) = 2$. In a similar way, $select_{00}(B_1, B_2, i)$ returns the position of the i-th occurrence of the symbol 00 (representing occurrences of the symbol 0 in both bitmaps simultaneously). For instance, in the previous example, $select_{00}(B_1, B_2, 1) = 2$. Then $rank_{00}(B_1, B_2, u) - rank_{00}(B_1, B_2, l-1)$ returns the number of local results in $[l, u]$ and $select_{00}$ is used to find the position of each single local result in such range.

It is easy to solve $rank_{00}$ and $select_{00}$ in constant time and $o(n)$ extra space. One can create the sublinear-space data structures for a virtual bitmap B such that $B[i] = 1$ iff $B_1[i] = B_2[i] = 0$, and then translate those queries into $rank_1$ and $select_1$ queries on B. Any desired portion of B that needs be consulted to complete the queries is created on the fly in constant time from the corresponding areas of B_1 and B_2.[3]

Lemma 6. *Given an independent set of n rectangles, structure* WTR *finds the k rectangles that intersect a query rectangle in time $O((k+1)\log n)$.*

Proof. The process is similar as for the point wavelet tree (Lemma 1), where each result may require a full root-to-leaf traversal. In addition, each unique

[3]More precisely, to achieve constant time in the RAM model we need to be able to retrieve portions of $\Theta(\log n)$ bits of B in constant time. This is easily obtained from B_1 and B_2 since $B[i] = \sim B_1[i] \wedge \sim B_2[i]$, using machine-word bitwise operations, or universal tables in theory.

Algorithm 3 Range query algorithm on structure WTR. *cNode* is the tree node, $[lB, uB]$ is the valid Y range, and $[x_l, x_u]$ is the valid bitmap range. It returns MBRs ranked by their x-axis. Note *cNode.B* is a virtual bitmap.

WTR$(cNode, lB, uB, x_l, x_u)$
  if $cNode.range \subseteq [lB, uB]$ **then**
   return $[x_l, x_u]$
  end if
  $leftResult \leftarrow \langle\rangle$; $rightResult \leftarrow \langle\rangle$; $localResult \leftarrow \langle\rangle$
  if $cNode.leftChild.range \cap [lB, uB] \neq \emptyset$ **then**
   $leftResult \leftarrow$ **WTR**$(cNode.leftChild, lB, \lfloor \frac{lB+uB}{2} \rfloor],$
                $rank_1(cNode.B_1, x_l - 1) + 1, rank_1(cNode.B_1, x_u))$
  end if
  if $cNode.rightChild.range \cap [lB, uB] \neq \emptyset$ **then**
   $rightResult \leftarrow$ **WTR**$(cNode.rightChild, \lfloor \frac{lB+uB}{2} \rfloor] + 1, uB,$
                $rank_1(cNode.B_2, x_l - 1) + 1, rank_1(cNode.B_2, x_u))$
  end if
  for $i = rank_1(cNode.B, x_l - 1) + 1$ to $rank_1(cNode.B, x_u)$ **do**
   add $select_1(cNode.B, i)$ to *localResult*
  end for
  for $i = 1$ **to** $|leftResult|$ **do**
   $leftResult[i] \leftarrow select_1(cNode.B_1, leftResult[i])$
  end for
  for $i = 1$ **to** $|rightResult|$ **do**
   $rightResult[i] \leftarrow select_1(cNode.B_2, rightResult[i])$
  end for
  return $merge(leftResult, rightResult, localResult)$

result is merged upwards, costing $O(1)$ time per node, until the root. Repeated results add a cost that is equal to the number of times the same result appears duplicated in a merge and copies are deleted. The number of times this can occur for a unique result is limited by the number of times any MBR is represented in the tree. This is at most $4\log(2n)$, see the proof of Lemma 5. □

Figure 11 highlights the nodes visited to solve the query $Q = [2.0, 2.75] \times [2.0, 3.5]$. As noted before, this query is translated into the ranges $[x_l, x_u] = [3, 5]$ (valid positions in the root node) and $[y_l, y_u] = [4, 5]$ (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 $rank_{00}(B_1, B_2, 1) = 1$) that is at position 1 (because $select_{00}(B_1, B_2, 1) = 1$). 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 $\langle 1 \rangle$ and right result $\langle 2 \rangle$ 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.

7.3. Generalization to m independent sets

In the general case, the dataset can be decomposed into m independent sets using the same algorithm mentioned in Section 5.2. Note that we define independent set over the x-axis projection of the MBRs, which form a set of intervals. In the same way, we propose two techniques in order to reduce the number of binary searches.

On the one hand, we use a single shared Y array for the whole dataset. In this way, the m wavelet tree-like structures are built in the same range $[1, 2n]$ (i.e., the nodes of the m trees at the same position and level cover the

same range). Hence, the y-axis range of the query is the same for all the trees and it can be obtained using two single binary searches on the endpoints of the query interval.

On the other hand, the x-axis range of the query is not the same for all the trees and thus we use the same technique described in Section 5.2. We store x-axis lower coordinates (the same holds for the upper coordinates) in a single array and an additional wavelet tree over a sequence that indicates for each coordinate the corresponding independent set.

Although these two techniques do not affect the asymptotic query time complexity (dominated by the m traversals in the trees), they improve the practical performance of the structure.

Theorem 5. *Given a set of n rectangles with coordinates in $[1, U]$, $n = o(U)$, that can be partitioned into m independent sets, there is a data structure (called* WTR*) that requires at most $n(4 \log U + 4 \log n + 2 \log m + o(\log U))$ bits and finds the rectangles that intersect a query rectangle in time $O((m + k) \log n)$. If $U = O(n)$, the space is at most $8n \log n + 2 \log m + o(n \log n)$ bits. The structure is built in $O(n \log n)$ time and extra bit space.*

Proof. The space follows from Lemma 5, considering that each MBR is represented in just one independent set, and the two additional wavelet trees used to improve the translation to rank space. The time follows from Lemma 6 considering that we carry out m searches and collect k results overall. Construction time is dominated by the decomposition into m independent sets (analyzed in Section 5.2) and the construction of the wavelet tree-like structures (see Section 3). □

7.4. Additional considerations

The minimum number m of independent sets that cover the MBRs can be thought of as the difficulty of the problem, thus our $O((m+k) \log n)$ time query algorithm is adaptive to this difficulty. Yet, the situation is indeed more complex. As a simple example, the number of independent sets can be different if we rotate the data. For example, in the TIGER dataset (see Section 8), we obtain 19 independent 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 almost doubles when using the y-axis decomposition). A finer analysis is as follows (considering a version of the structure where each independent set has its own Y array). Assume

$n_1, n_2, \ldots, n_m$ are the sizes of the m independent sets. Then, the space necessary to store the wavelet tree-like structures is at most $8 \sum n_i \lceil \log n_i \rceil$, and they solve the queries in time $O(\sum \log n_i)$ plus $O(\log n_i)$ for each result reported by the i-th tree. 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 independent sets improves the size of the structure whereas the opposite improves query time performance. Hence, we can design heuristics to create the independent sets based on this trade-off. For example, the algorithm to create the independent sets decomposition can choose the set that, without violating the constraints, contains fewest/most elements, or 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 m as the number of independent sets accessed to solve a query (and not all the independent sets necessary to represent the dataset). In this case, heuristics that minimize the overlap between independent 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 [16].

Table 1 summarizes all the theoretical space-time trade-offs achieved along the article, together with the R-tree variants. The way we compute the space for the STR R-tree is explained in Section 8. For the CR-tree we have given a space lower bound considering that, if a good spatial packing of rectangles is achieved, the differences between consecutive coordinates could be as small as if we ordered them all. The next section is devoted to a practical comparison among all these data structures.

8. Experimental Results

8.1. Datasets

Both synthetic and real datasets were used in our experiments. The two synthetic collections have one million MBRs each, the first one with a Zipf distribution (world size = 1000×1000, $\rho = 1$) and the second one with a Gauss distribution (world size = 1000×1000, $\mu = 500$, $\sigma = 200$). We created two query sets for each dataset. In the first one, queries are uniformly distributed over the area of the space where the MBRs are located, whereas in the second one queries follow the same distribution of the dataset. The second variant emulates the typical user behavior of performing more queries in areas containing more information. Each of these four

Structure	Space per rectangle in bits	Same space if $U = O(n)$	Query time (order, worst-case)
IS	$4\log U + 8\log n + O(1)$	$12\log n + O(1)$	$\log n + k_1 + k_2$
IS-FF	$4\log U - \log n + o(\log(U/n))$	$3\log n + O(1)$	$\log n + k_1 + k_2$
IS-IIS	$4\log U - 4\log n + 3\log m + o(\log(U/n))$	$3\log m + O(1)$	$\log n + m + k_1 + k_2$
SE-PQ	$4\log U - \log n + o(\log U)$	$3\log n + o(\log n)$	$(k_1 + k_2 + 1)\log(n/(k_1 + k_2 + 1))$
WTR	$4\log U + 4\log n + 2\log m + o(\log U)$	$8\log n + 2\log m + o(\log n)$	$(k + \min(m_1, m_2))\log n$
STR R-tree	$(4\log U + \log n)M/(M-1)$	$5M/(M-1)\log n$	—
CR-tree	$\geq (4\log(U/n) + \log n)M/(M-1)$	$\geq M/(M-1)(\log n + O(1))$	—

Table 1: Space/time complexities of the described data structures for orthogonal range queries on rectangles. Here n is the number of rectangles, U the coordinate universe size, k the query output size, k_1 and k_2 the output sizes of the projected one-dimensional queries ($k \ll \min(k_1, k_2)$), m_1 and m_2 the number of independent intervals when projecting the data rectangles to either dimension, $m = m_1 + m_2$, and M the R-tree arity. The first five structures are described in Theorems 1 to 5, respectively, whereas the last two come from the literature (and we have adapted them slightly).

					Tiny			Medium			Broad		
	n	m	m_1	m_2	k	k_1	k_2	k	k_1	k_2	k	k_1	k_2
GAUSS_U	1,000,000	20	10	10	187	14,396	13,695	1,808	47,137	43,662	12,763	123,512	125,122
GAUSS_G	1,000,000	20	10	10	446	20,918	21,371	3,711	60,180	62,022	25,313	159,505	158,258
ZIPF_U	1,000,000	20	10	10	121	12,214	14,496	707	27,982	29,290	3,501	60,818	56,978
ZIPF_Z	1,000,000	20	10	10	23,876	153,649	157,757	81,307	268,849	288,758	145,124	380,945	377,871
EIEL	569,534	69	35	34	41	1,814	1,715	136	2,986	2,749	10,375	66467	58,578
TIGER	2,249,727	55	19	36	14	2,903	2,236	412	10,703	20,395	10,739	128,998	88,401

Table 2: Quantitative analysis of the parameters that may have an influence on the performance of the data structures (see Table 1 for a description of these parameters). We study these parameters in all the four different datasets. GAUSS and ZIPF dataset are studied with the two kinds of query sets (uniform distribution, GAUSS_U and ZIPF_U, and same distribution of the dataset, GAUSS_G and ZIPF_Z). For each query set, three different selectivities are shown: Tiny: 0.001%/URBRU/Block; Medium: 0.1%/CENT/AIANNH; and Broad: 1%/MUN/COUSUB.

query sets contains four subsets of queries grouped by selectivity that represents 0.001%, 0.01%, 0.1%, and 1% of the space. These subsets contain 1,000 queries each with a ratio between the horizontal and vertical extensions of the queries varying 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 [37]. The first 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). The second real collection, named TIGER dataset, contains 2,249,727 MBRs from California roads and is available at the U.S. Census Bureau[5]. 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).

In Table 2 we study various parameters of the collections (data and queries) that may affect the result. We refer to these results later in the discussion. For each dataset we show the average number of results, k, and the average number of results in each dimension, k_1 and k_2, for three query sets representing different selectivities. For example, *Tiny* contains queries that represent 0.001% of the space for Gauss and Zipf datasets, URBRU queries for the EIEL dataset, and Block queries for the TIGER dataset.

8.2. Implementation details

Our experiments evaluate the space and time performance of our data structures, and compare them with a static variant of the R-tree (by far the most popular index for MBRs) named STR R-tree. All the structures were implemented by ourselves[6], and the source code is available at

[4]http://www.dicoruna.es/webeiel

[5]http://www.census.gov/geo/www/tiger

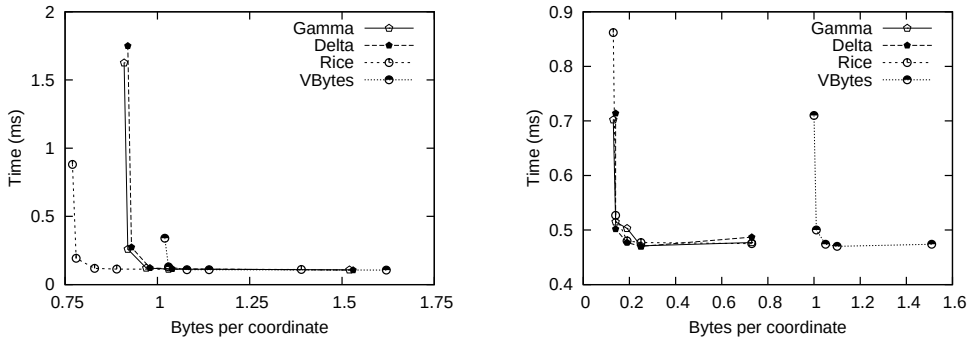
[6]In previous work [7, 8] we used the STR R-tree implementation due to Marios Hadjieleftheriou (`http://libspatialindex.github.com`). That implementation is general and uses the Standard Library, which makes it not suitable to measure the time performance of the structure. Our implementation uses its construction algorithm but its time performance has been drastically optimized (by several orders of magnitude in some cases).

`http://lbd.udc.es/research/serangequerying`. The computer used features an Intel Pentium 4 processor at 3.00GHz with 4GB of RAM. It runs GNU/Linux (kernel 2.6.27). We compiled with `gnu/g++` version 4.3.2 and options `-m32 -O9`. Our time measures always show average user time per query.

In our analyses we have assumed for simplicity that we can return any injective function from the MBRs onto $[1, n]$ as a valid MBR identifier. This is hardly the case in practical applications, where MBRs are usually simplifications of more complex objects, and thus, MBR identifiers are in most cases pointers to data in memory or disk. In the experiments we will assume these MBR identifiers are stored in an array of n 4-byte integers, and will consider this array as part of the space cost of the data structures.

Two of our structures, SE-PQ and WTR, are based on wavelet trees and thus their performance heavily relies on the implementation of *rank*/*select* for bitmaps. There exist several implementations offering different space-time trade-offs and we consider some of them in our experiments. Specifically, we use a practical one-level implementation proposed by González et al. [25]. The space-time trade-off offered by this implementation can be easily configured, and we use three different space overheads: 5%, 25%, and 33%. We also use a practical version of the classical two-levels solution [23], which was also described in the same paper [25] and requires 37.5% extra space. We noticed that the time performance of the latter is much better that the one-level solution with 33% of space overhead (see for example Figure 13(a)). Therefore, we emphasize the two endpoints of the trade-off (i.e., the one-level 5% solution when aiming at space efficiency, and the two-levels solution to improve the time performance). Also, when the *rank*/*select* operations are not very frequent, the most space-efficient solution is usually preferable. This is the case for the WTR structure that, besides the common *rank*/*select* operations for bitmaps, requires $rank_{00}$ and $select_{00}$ queries. As these two operations are not very frequent, the 5% implementation is much more suitable for them. We omit the experiments with other solutions because the impact in the time performance is minimal, whereas the space is considerably increased.

We compute the size of STR R-trees conservatively, assuming that the nodes are perfectly full (this is not far from real as this static variant can be built bottom-up). When using nodes of M entries, the R-tree will contain $\frac{n}{M-1}$ nodes. The size of an entry is the size of an MBR plus a pointer to the child (or the MBR identifier 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



(a) EIEL (b) TIGER

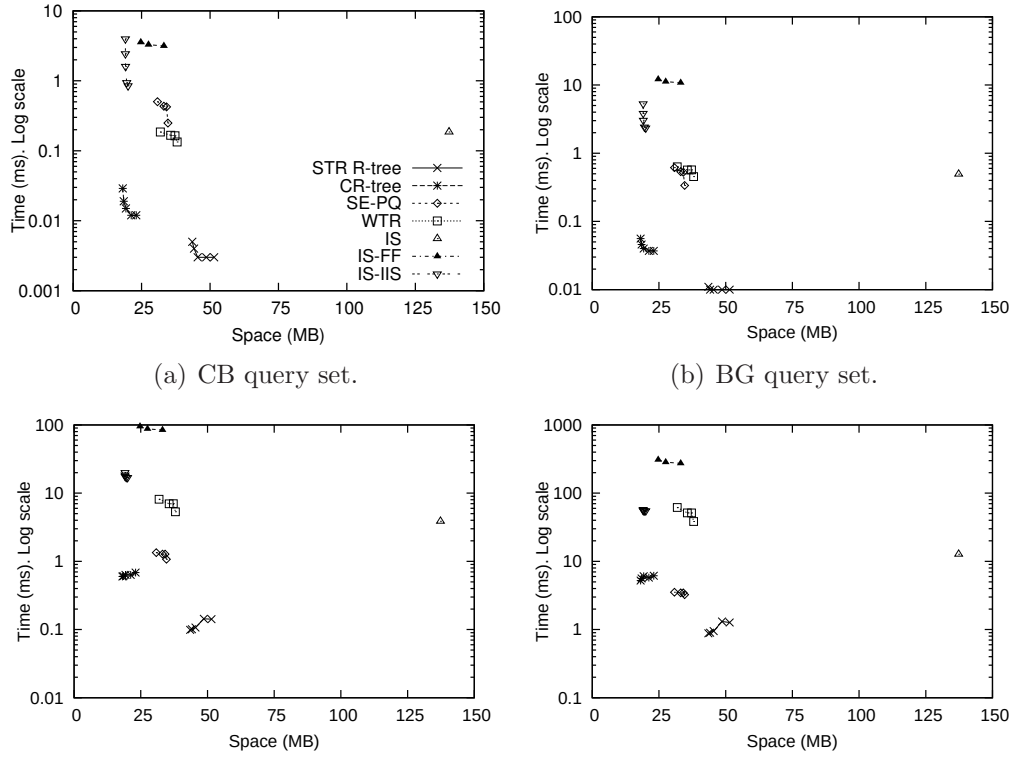
Figure 12: Influence of the coordinate encoding.

coordinates in 4-byte numbers) and the pointer in 4 bytes. Hence, the total size of an STR R-tree is $\frac{n}{M-1} \times 20 \times M$ bytes. Note that the coordinates stored by the STR R-tree are not sorted, thus it is not possible to apply differential encoding. Therefore, for sufficiently large M, the size of the STR R-tree is essentially the size of the raw data without compression: MBR coordinates and identifiers.

8.3. General overview of the results

The experiments in this section are aimed at giving a general overview of the main results. Next sections carry out more exhaustive and detailed empirical analyses.

First, we studied the influence of different coding algorithms and configuration parameters for coordinate storage. As described in Section 4.1, the sampling rate provides a space-time trade-off. We aim at reducing the space while keeping good search time performance. In general, coordinate encoding does not have a key influence in search time performance because these arrays are only used to translate the real coordinates of the query to rank space by means of binary searches. 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 (10, 50, 100, 1,000 and 10,000). Figure 12 shows the results of these experiments in the EIEL and TIGER datasets when these algorithms are plugged in the SE-PQ structure (similar graphs could be presented for the other structures). Note that this experiment is



(a) CB query set.

(b) BG query set.

(c) SD query set.

(d) SLDL query set.

Figure 13: Space-time trade-off, on TIGER dataset.

only meaningful in real scenarios because the compression achieved strongly depends on the precision and distribution of the data. Query sets contained 1,000 uniformly distributed queries in the surface covered by each dataset with a selectivity equivalent to 0.01% of the area. The four lines correspond to the coding algorithms and each point in these lines represents a different sampling rate.

Although all the coding algorithms provide a good compression rate (i.e., the space usage is much smaller than the 4 bytes per coordinate necessary when no coding algorithm is used), Rice coding achieves the best space-time trade-off in most situations. It can also be seen that, unless the sampling step h is too large, time performance is quite stable, so tuning is not hard. Due to this fact, in the rest of the experiments we will use Rice codes with sampling $h = 500$.

We now carry out a space-time comparison of all the structures presented in this paper, including a static R-tree (named STR R-tree) and our proposal of a lossless compressed R-tree (named CR-tree). The lines correspond to different structures and each point in the lines represents either a different node size (in the case of STR R-tree and CR-tree), a different space overhead for the *rank*/*select* structures (in the case of SE-PQ and WTR), or a different sampling rate for the upper coordinates (in the case of IS-FF and IS-IIS).

We use the TIGER dataset and four query sets of different selectivities. CB contains the most selective (i.e., smallest) queries, SLDL the least selective (i.e., biggest) queries, and BG and SD contain medium size queries.

The first conclusion of these experiments is that the theoretical complexities given in Table 1 are good predictors of practical performance, possibly with the exception of the space of WTR, where theory is a bit pessimistic, as explained later in this section.

Our space-efficient structures offer interesting space-time trade-offs, and many of them clearly beat the space of the classical STR R-tree (the space saving is around 20% for the SE-PQ and WTR structures, and more than 50% for the CR-tree and IS-IIS structures). The SE-PQ structure is quite insensitive to the selectivity of the queries, which makes it more competitive on broader queries. We will discuss further this behavior in Section 8.5.

Although we claimed at the beginning of this section that the coordinate encoding does not have a key influence in search times, this is not completely true for the IS-FF and IS-IIS structures. For both it is still true that the encoding scheme of the lower coordinates, which only take part in the translation to rank space, does not have a key influence in search times. However, the upper coordinates are used in a different way. As we explained in Section 5.1, the upper coordinates in the IS-FF structure require an xor-based compression and a dense sampling rate to provide direct access. In the case of IS-IIS, the upper coordinates are sequentially accessed from the first candidate to the first non-result. As the first candidate might not be a sampled value, some coordinates might be decompressed without being part of the result. The influence of the sampling rate is emphasized even more because this process must be completed for each independent interval set.

Finally, we found that a convenient trade-off for R-tree variants is achieved with $M = 30$, which we use for the rest of the experiments.

8.4. Detailed space analysis

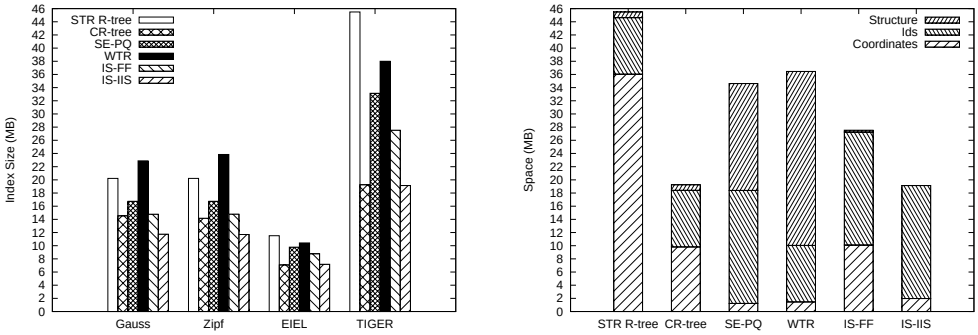
Figure 14(a) shows the space to store all the compact structures presented in this paper. The STR R-tree provides a comparison with the state of the art. As noted before, we are assuming that the nodes of the STR R-tree are completely full, and thus, we give the least space an R-tree can achieve.

As the classical IS structure requires about seven times more space than the most space-efficient variant (IS-IIS), we left IS out of the graph. The effectiveness of the compression scheme, and thus the space of the resulting structures, varies across datasets, so we show the results for each dataset.

These results show that the structures presented in this paper can index geographic data in less space than the STR R-tree. Some datasets are more compressible than others. The best results were obtained with the real TIGER dataset, where the CR-tree and IS-IIS structures save around 60% of the space of the STR R-tree. Compression rates are not so impressive for the synthetic and EIEL datasets. Geographic coordinates in these synthetic collections use four decimal positions, whereas in the EIEL dataset they are given in centimeters, and thus in all of these cases distances between consecutive coordinates are quite large. However, even in these cases the space needed to represent the IS-IIS structure is considerably less than the space needed to represent an STR R-tree (the IS-IIS saves around 40%). The SE-SQ and IS-FF structures do not save as much space as the CR-tree or IS-IIS structures, but the space-saving is still considerable (around 20% with the TIGER dataset) when compared with the STR R-tree. Finally, the WTR is competitive with the STR R-tree (especially in the TIGER dataset, where the encoding of the coordinates performs best).

Figure 14(b) shows a breakdown of the space into the three basic components of each structure: the coordinate storage, the MBR identifiers, and the main structure. We only show the results obtained with the TIGER dataset because the graphs obtained with the other collections just differ in the proportion used by the coordinate storage scheme. The differential scheme performs quite well in all of our structures, as the proportion of space required by this component is minimum when compared with the other two components. There is a noticeable difference in the case of IS-FF, because the upper coordinates are not stored differentially but using *xor*. Note that the space used to store the coordinates in this structure is comparable with that of the CR-tree, which uses a compression scheme local to nodes.

The structures using the projection into one-dimensional problems (i.e., SE-PQ, IS-FF, and IS-IIS) require twice the space of the others to store the



(a) Structures comparison. (b) Space decomposition. TIGER dataset.

Figure 14: Space usage.

identifiers. The reason is that for the R-tree variants and the WTR we are considering just the identifier data, whereas in the others we are considering in addition the extra data needed to connect both projected problems. If identifiers are unique numbers in $[1, n]$, they can be used themselves to connect the subproblems; otherwise we need to store a permutation as described earlier in the paper. In these experiments this makes no difference.

Finally, the main structure corresponds to the pointer-based tree for the R-tree variants, with wavelet trees in the case of SE-PQ and WTR, with the FF tree representation in the case of IS-FF, and with some additional pointers for each independent set in the case of IS-IIS. Note that in these last two structures the space of the main structure is almost negligible. In a sense, IS-FF trades main-structure data for coordinate data (as we could alternatively have sorted upper coordinates in increasing order and a permutation in the main structure, but this scheme would have been slower). With respect to SE-PQ and WTR, the two wavelet trees stored by SE-PQ use less space than the wavelet tree-like structure stored by WTR. The WTR space depends on the number of times an MBR is represented across the tree, and this turns out to be much less than the 4-per-level upper bound, as explained. This number is entirely data dependent and, for example, with the TIGER dataset, each MBR is represented around 2.35 times per level. Although the core of the WTR is not competitive with the SE-PQ wavelet trees, the resulting global structure (considering coordinates and identifiers) does not require much more space (in particular, we do not need to map among projections). In a sense, the WTR main structure contains the information that is lost in the

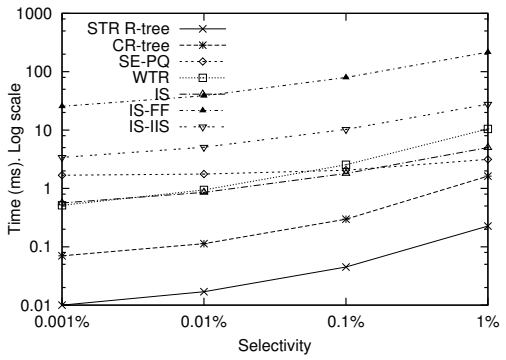
projected wavelet trees, and thus it trades main structure space for identifier space. It is also important to notice that m is extremely small in practice (see Table 2), and thus its influence on the space required by the data structures that depend on it (IS-IIS and WTR, see Table 1) is negligible.

As explained, the STR R-tree space (or, more precisely, the lower bound we use) is almost exactly the size of the raw data. Our data structures can be thought of as ways to represent these data in different form: coordinates are reorganized to make them compressible and data structures provide the linkage information lost. Our results show that this strategy not only reduces the raw data space, but it supports efficient searches.

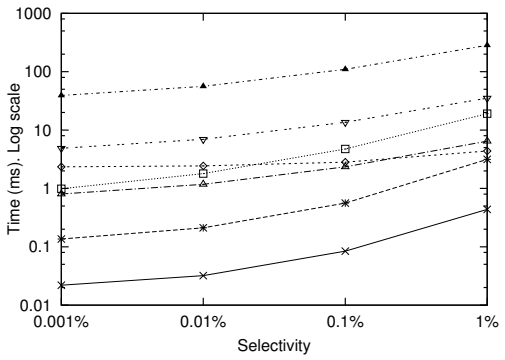
8.5. Detailed time analysis

We experiment both with synthetic and real collections. Recall that we use query sets with different distributions and selectivities for the synthetic datasets, whereas for the real datasets we use real query sets. These contain queries of different selectivities. In the graphs, real query sets have been sorted accordingly with their selectivity (from left to right queries are broader). Figure 15 shows the results.

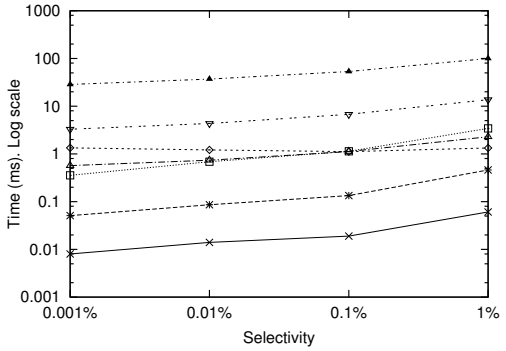
These experiments confirm the good performance of the STR R-tree, as it performs best in all the scenarios and with all the different selectivities. The good performance of its compressed version, our CR-tree improved lossless variant, is also remarkable. For the different selectivities, the CR-tree is consistently about 7 times slower than the STR R-tree. Although the number of node accesses is comparable for both structures, the cost of a node access is higher in the case of the CR-tree because of the compression. The query algorithm has to decompress the values in each node, which involves some bit-operations that are computationally expensive. It is interesting that the WTR, a structure completely different in nature to those R-tree variants, also performs very similarly to them for the different selectivities. The other structures, especially the SE-PQ, are much less sensitive to the selectivity of the query. The space-efficient structures based on intervals (i.e., IS-FF and IS-IIS) are not competitive in time. However, their space-inefficient variant is not far from the CR-tree or the SE-PQ. This is makes the IS-FF and IS-IIS still interesting, since new developments (for example, improvements in the implementation of succinct trees) may turn them just slightly worse than the IS and thus competitive with the others. The SE-PQ is competitive with the CR-tree and represents a good choice when queries are not too selective. Note that, in the real scenarios, even the broadest queries are meaningful.



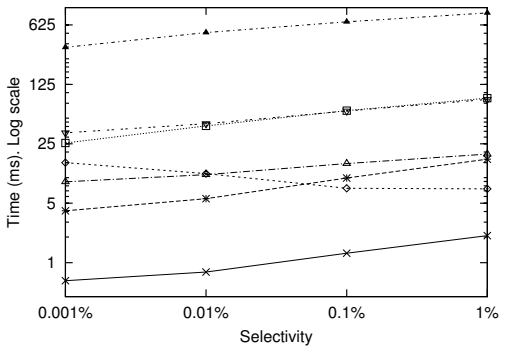
(a) Gauss dataset. Uniform query set.



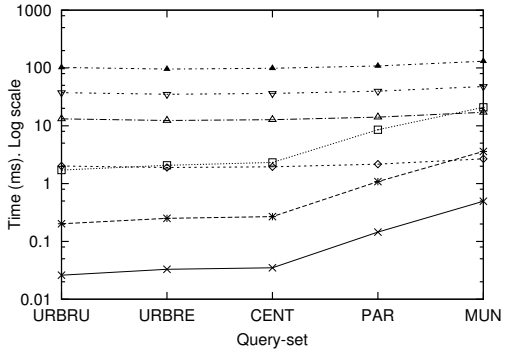
(b) Gauss dataset. Gauss query set.



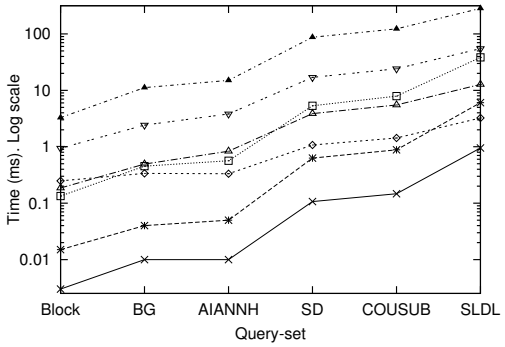
(c) Zipf dataset. Uniform query set.



(d) Zipf dataset. Zipf query set.



(e) EIEL dataset.



(f) TIGER dataset.

Figure 15: Time performance.

For example, in the EIEL dataset the query set MUN contains queries of the form *buildings contained in a municipality*. Actually, even broader queries, such as *buildings in a certain region or state*, may be also interesting in GIS.

As we mentioned above, another important conclusion is the little dependency of the SE-PQ to changes in selectivity. This is due to the space transformation. We divide the problem into two subproblems, each concerning one dimension. This decomposition makes queries in the two wavelet trees only marginally dependent on the query size (i.e., selectivity). For example, on uniformly distributed small MBRs, a query that selects a fraction α along each coordinate will retrieve $\alpha^2 n$ rectangles, but the projected wavelet trees will retrieve αn candidates to verify. This "square root" impact of the query size in the performance of the algorithm explains its resilience to the query selectivity. Of course, it also explains why our projection techniques do not perform so well when queries are very selective, as we work $O(\alpha n)$ time in order to retrieve a result of size $O(\alpha^2 n)$. A similar analysis applies to the interval based structures (IS, IS-FF, and IS-IIS), which query time depends on k_1 and k_2 (see Table 1). In table 2, we show that $k_1, k_2 \gg k$ for very selective queries, but they get closer for broader queries.

The surprising time decrease of SE-PQ with the increase of the query size in Figures 15(c) and 15(d) is explained because all the MBRs represented in a node are directly reported without reaching the leaves if the node range is completely contained in the query range and all the positions of the node are valid. Hence, while smaller queries prune the tree more than bigger ones, bigger queries report more elements without reaching the leaves. The Zipf dataset markedly increases the number of directly reported objects due to the high concentration of MBRs near the origin of coordinates (see Table 2).

8.6. Discussion

We performed experiments in various synthetic and real scenarios and we found that, in general, the new CR-tree variant stands out as an excellent space-time trade-off, because it is not much slower than the STR R-tree and it requires up to 60% less space. The SE-PQ also stands out as a good candidate when queries are not too selective, because it requires up to 20% less space than the STR R-tree and it is not much slower on very selective queries. For the more selective queries, the WTR is faster than the SE-PQ. While not faster than the R-tree variants, it takes less space than the STR R-tree in real datasets. Interval set-based structures require around the same space as the CR-tree (slightly less in some datasets), but they are slower.

However, their non space-efficient variant (named IS) performs similarly to the SE-PQ and provides a lower bound on the best performance we may expect from the space-efficient variants.

The structures that come from a reduction to one-dimensional problems (i.e., SE-PQ, IS-FF, and IS-IIS) have some common characteristics. The space is dominated by the storage of the identifiers. Recall that this strategy stores the identifiers for each dimension (or an equivalent representation to perform the intersection). Then, they use some extra space for structures that improve the query time performance (SE-PQ uses much more space than the interval set-based structures, but its time performance is also much better). The anomaly in this general behavior for the differences between the IS-FF and the IS-IIS (the latter outperforms the former both in space and query time) is explained by the poor performance of the succinct trees for our application. IS-FF stores an additional succinct tree to improve query times with respect to the IS-IIS, but a profiling of the code shows that this succinct tree is the time bottleneck.

In the case of the WTR, practical results are not very exciting. However, it is interesting to notice that, although completely different in nature, it performs similar to the R-tree variants.

Based on this empirical evaluation, we conclude that the STR R-tree is preferable when space is not an issue. However, space is actually an issue in many domains, and can mark the difference between operating in main memory or on disk. When this is the case, the CR-tree is preferable when the majority of the queries are very selective, whereas the SE-PQ is preferable for applications with broader queries.

9. Conclusions and Future Work

We have introduced four new space-efficient structures (named SE-PQ, WTR, IS-FF, and IS-IIS) for general sets of minimum bounding rectangles, and experimentally compared them with the best classical representation (a slightly space-optimistic STR R-tree). We have shown that the new data structures offer large space reductions, while retaining a time performance that is reasonable for many applications. As such, they constitute relevant alternative access structures for geographic data.

Our space-efficient representations make heavy use of well-known compact data structures such as *rank/select* structures, wavelet trees, succinct trees, differential encoding, and others. On top of those, we apply different

decompositions of the problem into simpler subproblems, whose results are then combined. The achieved complexities are adaptive to different measures of difficulty of the problem.

In addition, we have optimized the CR-tree, a static compressed version of the R-tree. In our GIS scenario, the CR-tree can be set not to produce any false positive. Note that each false positive involves a huge penalty in most applications because a real complex (e.g., geographic) object has to be retrieved (possibly from disk) and a complex comparison operation between this object and the query window has to be performed to check whether it is a true hit. This CR-tree also takes advantage of the knowledge of the data distribution, achieving an excellent compression rate on static data.

The experimental evaluation shows that two data structures excel in the case of little space available for the index: our optimized CR-tree and the SE-PQ. The former is more suitable for applications where queries are very selective, whereas the latter is suitable for applications with broader queries. We showed some real scenarios where both kind of queries arise. Interval set-based structures and the WTR have not been competitive, yet they are still promising because they are based on relatively new succinct data structures that may be improved in the next few years. See Section 8.6 for a detailed discussion about the empirical evaluation.

In our experiments all the structures were tested in the same level of the memory hierarchy (specifically, they were tested in RAM). However, it is important to notice that our space-efficient structures use less than half the space of a classical index for many datasets. Thus, it may be the case that in some large real datasets our structures fit into an upper level of the memory hierarchy, compared to classical indexes. When that is the case, the time performance of our space-efficient structures will overcome the performance of classical indexes. Halving the space may be also key to achieve feasible solutions when memory is limited, such as in mobile devices. An alternative scenario is that of maintaining the structure distributed across the main memories of as many servers as necessary, as in current large search engines. In this case, the main cost factor is the *energy* to power those servers. Reductions in memory usage impact directly and proportionally on the number of servers needed to hold the structure, and thus on the energy cost of the engine.

We are working on allowing the insertion and removal of MBRs once the structures have been built. Some recent works [38, 33] describe dynamic versions of bitmaps and trees that can be used in the design of data structures

with insertion and deletion capabilities. Finally, algorithms to solve other queries like k-nearest neighbor or spatial joins are in our plans too.

10. Acknowledgements

This work was supported in part by Ministerio de Ciencia e Innovación (PGE and FEDER) [grant TIN2009-14560-C03-02], Xunta de Galicia (FEDER) [grants 10SIN028E and 2010/17, and Ángeles Alvariño], and Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

References

[1] J. L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (9) (1975) 509–517.

[2] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, SIAM J. on Comp. 17 (3) (1988) 427–462.

[3] Y. Nekrich, Orthogonal range searching in linear and almost-linear space, Computational Geometry 42 (4) (2009) 342–351.

[4] P. Bose, M. He, A. Maheshwari, P. Morin, Succinct orthogonal range search structures on a grid with applications to text indexing, in: Proc. 11th WADS, 2009, pp. 98–109.

[5] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.

[6] C. B. Jones, R. S. Purves, Geographical information retrieval, Journal of Geographical Information Science 22 (3) (2008) 219–228.

[7] N. R. Brisaboa, M. R. Luaces, G. Navarro, D. Seco, Range queries over a compact representation of minimum bounding rectangles, in: Proc. 29th ER Work., 2009, pp. 33–42.

[8] N. R. Brisaboa, M. R. Luaces, G. Navarro, D. Seco, A fun application of compact data structures to indexing geographic data, in: Proc. 5th FUN, 2010, pp. 77–88.

[9] K. Kim, S. K. Cha, K. Kwon, Optimizing multidimensional index trees for main memory access, SIGMOD Record 30 (2) (2001) 139–150.

[10] K. V. R. Kanth, A. K. Singh, Optimal dynamic range searching in non-replicating index structures, in: Proc. 7th ICDT, 1999, pp. 257–276.

[11] G. S. Lueker, A data structure for orthogonal range queries, in: Proc. 19th FOCS, IEEE, 1978, pp. 28–34.

[12] R. Grossi, A. Gupta, J. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th SODA, 2003, pp. 841–850.

[13] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theoretical Computer Science 387 (3) (2007) 332–347.

[14] T. Chan, K. Larsen, M. Patrascu, Orthogonal range searching on the RAM, revisited, in: Proc. 27th SoCG, 2011, pp. 1–10.

[15] A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, in: Proc. SIGMOD, ACM Press, 1984, pp. 47–57.

[16] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, Y. Theodoridis, R-Trees: Theory and Applications, Springer-Verlag, 2005.

[17] G. Navarro, Wavelet trees for all, in: Proc. 23rd CPM, 2012, pp. 2–26.

[18] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2) (2007) article 20.

[19] Y.-F. Chien, W.-K. Hon, R. Shah, J. S. Vitter, Geometric Burrows-Wheeler transform: Linking range searching and text indexing, in: Proc. 18th DCC, 2008, pp. 252–261.

[20] N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, in: Proc. 18th CPM, Vol. 4580 of LNCS, 2007, pp. 205–215.

[21] V. Mäkinen, G. Navarro, On Self-Indexing Images - Image Compression with Added Value, in: Proc. 18th DCC, 2008, pp. 422–431.

[22] F. Claude, G. Navarro, Practical Rank/Select Queries over Arbitrary Sequences, in: Proc. 15th SPIRE, 2008, pp. 176–187.

[23] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th FOCS, 1989, pp. 549–554.

[24] I. Munro, Tables, in: Proc. 16th FSTTCS, 1996, pp. 37–42.

[25] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proc. 4th WEA, CTI Press and Ellinika Grammata, 2005, pp. 27–38.

[26] F. Claude, P. K. Nicholson, D. Seco, Space efficient wavelet tree construction, in: Proc. 18th SPIRE, 2011, pp. 185–196.

[27] G. Tischler, On wavelet tree construction, in: Proc. 22nd CPM, 2011, pp. 208–218.

[28] N. R. Brisaboa, M. R. Luaces, G. Navarro, D. Seco, A new point access method based on wavelet trees, in: Proc. 28th ER W, 2009, pp. 297–306.

[29] H. N. Gabow, J. L. Bentley, R. E. Tarjan, Scaling and related techniques for geometry problems, in: Proc. 16th STOC, ACM, 1984, pp. 135–143.

[30] M. de Berg, M. V. Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry – Algorithms and Applications, Springer-Verlag, 2000.

[31] D. Salomon, Data Compression: The Complete Reference, Springer-Verlag, 2004.

[32] J. M. Schmidt, Interval stabbing problems in small integer ranges, in: Proc. 20th ISAAC, 2009, pp. 163–172.

[33] K. Sadakane, G. Navarro, Fully-Functional Succinct Trees, in: Proc. 21st SODA, SIAM, 2010, pp. 134–149.

[34] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct Trees in Practice, in: Proc. 12th ALENEX, SIAM, 2010, pp. 84–97.

[35] J. Barbay, G. Navarro, Compressed representations of permutations, and applications, in: Proc. 26th STACS, 2009, pp. 111–122.

[36] M. L. Fredman, On computing the length of longest increasing subsequences, Discrete Mathematics 11 (1) (1975) 29 – 35.

[37] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, SIGMOD Record 19 (2) (1990) 322–331.

[38] R. González, G. Navarro, Rank/select on dynamic compressed sequences and applications, Theoretical Computer Science 410 (2008) 4414–4422.