Range Queries over a Compact Representation of Minimum Bounding Rectangles ⋆

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

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

Abstract. In this paper we present a compact structure to index semi-static collections of MBRs that solves range queries while keeping a good trade-off between the space needed to store the index and its search efficiency. This is very relevant considering the current sizes and gaps in the memory hierarchy. Our index is based on the *wavelet tree*, a structure used to represent sequences, permutations, and other discrete functions in stringology. The comparison with the R*-tree and the STR R-tree (the most relevant dynamic and static versions of the R-tree) shows that our proposal needs less space to store the index while keeping competitive search performance, especially when the queries are not too selective.

Key words: MBR, range query, wavelet tree, compact structures.

1 Introduction

The age of online digital availability has forced changes in the goals of many classical research fields. For example, the ever-increasing demand for *geographic information services*, which allow users to find the geographic location of some resources previously located in a map (e.g. public administration services, places of interest, etc.), has emphasized the interest in the classical *range queries* (or *window queries*). These define a rectangular query window and retrieve all the geographic objects having at least one point in common with it. Other queries, like *region queries* and *point queries*, can be well-approached by them.

A survey of spatial queries and Spatial Access Methods (SAMs) designed to solve them is that of Gaede and Gunther [1] which also details the special characteristics of spatial data that determine a set of requirements that spatial indexes should meet (e.g. secondary storage management, dynamism, etc.). All

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

these have been design principles of spatial indexes along the years but recent improvements in hardware mean that some of these requirements need to be carefully reevaluated.

On the one hand, although geographic databases tend to be very large, it is feasible nowadays to place complete spatial indexes in main memory because spatial indexes do not contain the real geographic objects but a simplification of them. The most common simplification is the MBR or bounding box, which, in the 2-dimensional Euclidean space E^2, needs four floating-point values for each geographic object. Note that the most common spatial data are two-dimensional, hence the interest in E^2 (though our structure represents d-dimensional MBRs). Indeed, considering the current sizes of main memories, the *space efficiency* requirement can replace that of *secondary storage management.* Furthermore, the design of compact indexes is a topic of interest because of the way memory hierarchy has evolved in recent decades. New levels have been added (e.g. flash storage) and the sizes at all levels have been considerably increased. In addition, access times in upper levels of the hierarchy have decreased much faster than in lower levels. Therefore, reducing the size of indexes is a timeless topic of interest because placing these indexes in upper levels of the memory hierarchy considerably reduces access times, by several orders of magnitude in some cases.

On the other hand, the data is semi-static in many applications. This is usual in Geographic Information Retrieval [2] systems, which have arisen from the combination of GIS and Information Retrieval systems. In the field of GIS, many public organizations are sharing their geographic information using Spatial Data Infrastructure [3]. The spatial information in these systems does not require frequent updates. When a spatial index is integrated in this kind of system, *dynamism* is not a very important requirement compared to *time efficiency* solving queries. Therefore, the design of static spatial indexes that take advantage of the knowledge of the data distribution is also a topic of interest.

In the previous edition of this workshop [4] we presented a spatial index for two-dimensional points based on wavelet trees. The generalization to support d-dimensional range queries over MBRs, which we present here, turns out to be a rather challenging problem not arising in other domains where wavelet trees have been used. As a reward, the index we obtain achieves a good trade-off between space and time efficiency. In particular, the index turns out to be quite insensitive to the size of the query window, and as a result it becomes most competitive on not too selective queries. Our experiments, featuring GIS-like scenarios, show that our index is a relevant alternative to classical spatial indexes such as the R-tree [5]. In addition, the strategy used to decompose the problem allows the use of other solutions in the different steps achieving different trade-offs.

2 Related Work

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

The R-tree and its dynamic (e.g. R*-tree) and static (e.g. STR R-tree) variants [5] are the most popular SAMs used to solve range queries in GIS. In Gaede and Gunther [1] these structures are presented in the group of SAMs based on the technique of *overlapping regions*. An alternative technique consists in the *transformation* of the MBRs into a different representation such as higher-dimensional points or one-dimensional intervals. Our representation is based on this technique as it decomposes the problem into its d-dimensions, solves the d subproblems, and intersects the results. Each subproblem consists in finding the one-dimensional intervals intersecting the query, which is obtained applying the same decomposition to the original query.

Although many structures can be used to solve these subproblems, we propose the use of wavelet trees because they provide a good trade-off between space and time efficiency. Alternatively, classical interval data structures [6] like interval trees, segment trees, and priority trees can solve each subproblem in at least $\Omega(\log N + k)$ (considering the *output sensitivity complexity* where k is output size). Obviously, these classical structures using pointers need much more space than the compact wavelet tree. One-dimensional intervals can be interpreted as two-dimensional points (further details about this transformation are given in the next section) and thus all the data structures supporting two-dimensional range queries over points provide alternatives to solve intersection queries in sets of intervals. Some of them [7,8] theoretically improve the performance of the wavelet tree but they come with a significant implementation overhead. Recently, Schmidt [9] presented a simple structure solving the *Interval Intersection Problem* in asymptotic optimal space ($\Omega(n)$) and time ($\Omega(1 + k)$). However, this structure also needs much more space than the wavelet tree. A compact version of this structure is a promising line of future work because it could provide also an interesting trade-off.

A basic tool in compact data structures is the *rank* operation: given a sequence S of length N, drawn from an alphabet Σ of size σ, $rank_a(S, i)$ counts the occurrences of symbol $a \in \Sigma$ in $S[1, i]$. For the special case $\Sigma = \{0, 1\}$ (S is a bit-vector B), the rank operation can be implemented in constant time and using little additional space on top of B ($o(n)$ in theory [10]). For example, given a bitmap $B = 1000110$, $rank_0(B, 5) = 3$ and $rank_1(B, 7) = 3$.

3 Our Compact Representation

Our structure is based on the decomposition of the problem in its d dimensions. This decomposition produces d *interval intersection subproblems* that can be tackled with different structures. The one we propose uses a wavelet tree to solve each subproblem. The idea is to interpret every interval a as a point (l_a,

r_a) in the rank-space grid $N \times N$. Gabow et al. [11] proved that the orthogonal nature of the problem makes it possible to work in the rank space.

3.1 Translation to the Rank Space

In each dimension there are N intervals, each described by two float numbers (its endpoints). A common technique to perform the translation of these intervals to the rank space stores two ordered arrays, one with the left endpoints and the other one with the right endpoints. Then, the endpoints of an interval in the rank space correspond with the positions of its real endpoints in these arrays.

Although the real coordinates of the MBRs are floating point numbers, in GIS these numbers can be represented with four bytes each. Note that geographic coordinates can be represented in degrees or meters and in most cases it is possible to round them to integer values, after appropriate scaling, without losing any precision. We make use of this assumption, as it holds in most practical applications. To store these integer coordinates without losing precision we use a compressed storage scheme. An ordered array $X = x_1 x_2 \dots x_N$ is represented as a sequence of nonnegative differences between consecutive values $y_{i+1} = x_{i+1} - x_i$ and $y_1 = x_1$. Let $Y = y_1 y_2 \dots y_N$ be this sequence, so that $x_i = \sum_{1 \leq j \leq i} y_j$. Array Y is a representation of X that can be compressed by exploiting the fact that consecutive differences are smaller numbers. These small numbers can be encoded with different coding algorithms. We performed experiments (omitted due to lack of space) with four different well-known coding algorithms [12] (Elias-Gamma, Elias-Delta, Rice, and Vbytes) and conclude that Rice encoding outperforms the other in most synthetic and real scenarios. In addition, a vector stores the accumulated sum at regularly sampled positions (say every hth position, thus the vector stores all values $x_{i \cdot h}$) to efficiently solve the operations $rightSearch$ (given a coordinate v, find the largest $x_i \leq v$) and $leftSearch$ (the lowest $x_i \geq v$). The algorithm to map a real coordinate to the rank space first performs a binary search in the vector of sampled sums and then carries out a sequential scan in the resulting interval of Y.

3.2 Solving Queries in the Wavelet Trees

In the two-dimensional rank space, N MBRs can be represented in a $2N \times 2N$ grid (Figure 1 left) and an alternative representation for the N intervals in each dimension is an $N \times N$ grid where rows represent the intervals ordered according to their left endpoints and columns represent them ordered according to their right endpoints. Figure 1 (right) represents the grids resulting from the transformation of the vertical (up) and horizontal (down) intervals.

The following, easy to verify, observation provides a basis for our next developments. It says, essentially, that an intersection between a query q and an object o occurs when, across each dimension, the query finishes not before the object starts, and starts not after the object finishes.

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

In two dimensions, the four conditions of Observation 1 for query $q = [l_1^q, u_1^q] \times [l_2^q, u_2^q]$ are now split into $u_1^q \geq l_1 \ \wedge \ l_1^q \leq u_1$ and $u_2^q \geq l_2 \ \wedge \ l_2^q \leq u_2$. These are

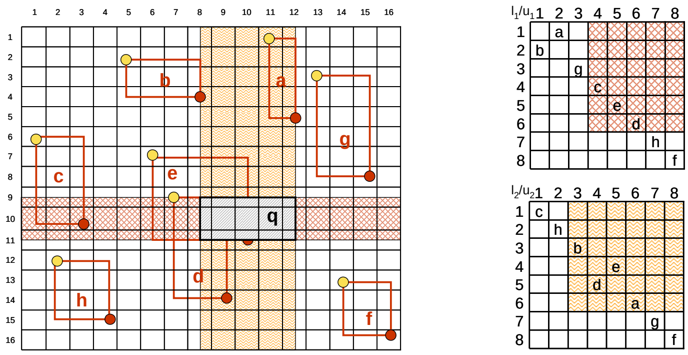


Fig. 1. Two dimensional example in the rank space.

illustrated in Figure 1 (right). In the original space, this partition results in two bands of $(l_2 - l_1) \times N$ and $(u_2 - u_1) \times N$, illustrated in Figure 1 (left). Objects intersecting each band are the candidates to be part of the query result. Finally, those objects in the intersection of the bands are the actual results of the query. Note that this intersection can be performed *on-line* because the candidates of the first dimension turn on their bits on a bitmap and the candidates of the second one report the MBR identifier only if their associated bits are turned on in the bitmap (an array of counters allows the generalization to d dimensions).

In both transformed grids there is only one point in each row and in each column, therefore we can build two wavelet trees as described in [4]. Figure 2 shows the wavelet trees corresponding to the first (left) and second (right) dimensions in the transformed space. The root node of each wavelet tree represents the permutation of the points in the order of the rows whereas the leaves represent the permutation of the points in the order of the columns. The wavelet tree is a perfect binary tree where each node handles an interval of the columns i, and thus knows only the points whose column falls in the interval. The root handles the interval of columns $[1,\ N]$ and the children of a node handling interval $[i,\ i']$ are associated to $[i, \lfloor (i+i')/2 \rfloor]$ and $[\lfloor (i+i')/2 \rfloor + 1, i']$. The leaves handle intervals for the form $[i,i]$. Each node v contains a bitmap B_v so that $B_v[r] = 0$ iff the r-th point handled by node v (in the order of the rows) belongs to the left child and $B_v[r] = 1$ iff it belongs to the right child.

In each wavelet tree we perform a query derived from the translation of the original query q to 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 two-sided 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 first intervals of each wavelet tree query represent rows in the transformed grid, and thus they indicate valid positions in the bitmaps of the wavelet trees, whereas the second intervals represent columns, and thus they indicate valid nodes in the tree traversal (and are used to prune these traversals). The algorithm solving the queries in each dimension is a simplification of the point case [4] because these are two-sided queries, whereas in the general case queries are four-sided.

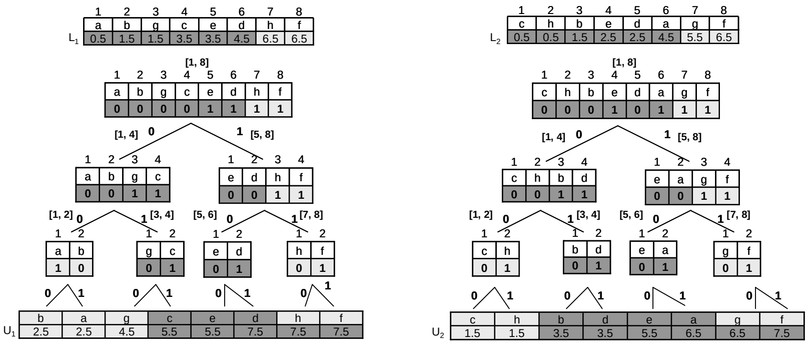


Fig. 2. Representing MBRs using wavelet trees.

The left wavelet tree in Figure 2 highlights nodes visited to solve the query decomposed in the ranges $[1, 6]$ (valid positions in the root bitmap of the first wavelet tree) and $[4, 8]$ (interval to prune the first wavelet tree traversal). The first range is projected in the child nodes of the root node as $[1, rank_0(B, 6)] = [1, 4]$ and $[1, rank_1(B, 6)] = [1, 2]$. In the same way the range $[1, 4]$ of the left child is projected in its children as $[1, rank_0(B, 4)] = [1, 2]$ and $[1, rank_1(B, 4)] = [1, 2]$, but the first one is not accessed because it covers the range [1,2] which does not intersect the query range [4,8]. If we repeat this process until the leaves, we obtain the set of candidates of the first wavelet tree $\{c, e, d\}$. Analogously we obtain the result $\{b, d, e, a\}$ in the second wavelet tree. Therefore, the solution of the query is $\{c, e, d\} \cap \{b, d, e, a\} = \{d, e\}$.

4 Experiments

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 gcc version 4.3.2 and option -O9. In these experimentes we used three synthetic collections with one million MBRs each and uniform, Zipf (world size $= 1,000 \times 1,000$, $\rho = 1$) and Gauss distribution (world size $= 1,000 \times 1,000$, $\mu = 500$, $\sigma = 200$). We created four query sets for each dataset with different selectivities that represent 0.001%, 0.01%, 0.1%, and 1% of the area of the space where the MBRs are located. They contain 1,000 queries with the same distribution of the original datasets and the ratio between the horizontal and vertical extensions varies uniformly between 0.25 and 2.25. Experiments using two real datasets are also presented. The first dataset, named Tiger, contains 2,249,727 MBRs from California roads and it is available at the U.S. Census Bureau[3]. Six smaller real collections available at the same place were used as query sets: Block (groups of buildings), BG (block groups), AIANNH (American Indian/Alaska Native/Native Hawaiian Areas), SD (elementary, secondary, and unified school districts), COUSUB (country subdivisions), and SLDL (state legislative districts). The second real collection,

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

named EIEL dataset, contains 569,534 MBRs from buildings in the province of A Coruña, Spain[4]. Five smaller collections available at the same place were used as query sets: URBRU (urbanized rural places), URBRE (urbanized residential places), CENT (population centers), PAR (parishes), and MUN (municipalities).

4.1 Space Comparison

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

On the other hand, our structure stores the coordinates of the N MBRs (four arrays of N 4-byte numbers without encoding), the MBR identifiers (two arrays, one per wavelet tree, of N 4-byte numbers to perform the intersection) and the two wavelet tree bitmaps (see grayed data in Figure 2). The wavelet tree needs only $N \times \lceil \log_2 N \rceil$ bits (1 bit per MBR per level, that is, N bits per level, and there are $\lceil \log_2 N \rceil$ levels). Moreover, in order to perform *rank* operations in constant time, some auxiliary structures are needed that use an additional space of around 37.5% of the wavelet tree size [10]. Therefore, the complete structure requires $4 \times 4 \times N + 2 \times 4 \times N + 2 \times (N \times \lceil \log_2 N \rceil \times 1.375)/8$ bytes, which is in addition subject to coordinate compression.

The effectiveness of this compression varies across datasets, so we show the results for each dataset in Figure 3 (we use Rice codes with sampling $h = 500$ and maintain this configuration in the time comparison). These results show that our structure, named *SW-tree* (from *spatial wavelet tree*) in the graphs, can index collections of MBRs in less space than the two variants of the R-tree due to the compressed encoding of the coordinates and the little space required by the wavelet trees. Some datasets are more compressible than others. The best results were obtained with the real Tiger dataset where we save more than 46% and 22% of the R*-tree and STR R-tree respectively. Compression rates are not so good with the EIEL datasets because geographic coordinates in this collection are in centimeters, and thus distances between consecutive coordinates are quite large. However, even in this case the space needed to represent our structure is considerably less than the space needed to represent an STR R-tree.

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

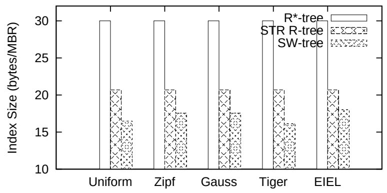
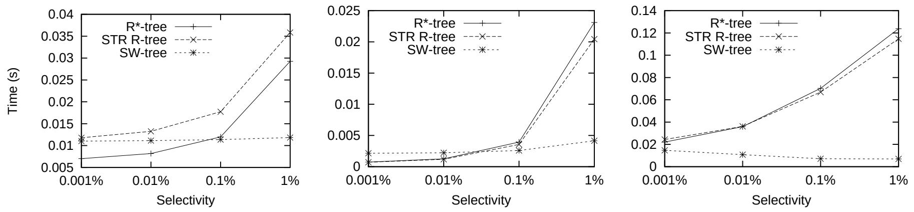


Fig. 3. Space comparison.

4.2 Time Comparison

To perform the time comparison we implemented our structure as described in Section 3 and use the R-tree implementation provided by the *Spatial index library*[5]. This library provides several implementations of R-tree variants such as the R*-tree and the STR packing algorithm to perform bulk loading. In addition, all these variants can run in main memory. In our experiments we run both the R*-tree and the STR R-tree in main memory with a load factor $M = 30$.

We first perform experiments with the three synthetic collections (Figure 4). The main conclusion that can be extracted from these results is that our structure is competitive with respect to query time efficiency when the queries are very selective (0.001% and 0.01%) and in less selective queries (from 0.1%) our structure is significantly better than the others.



(a) Uniform

(b) Gauss

(c) Zipf

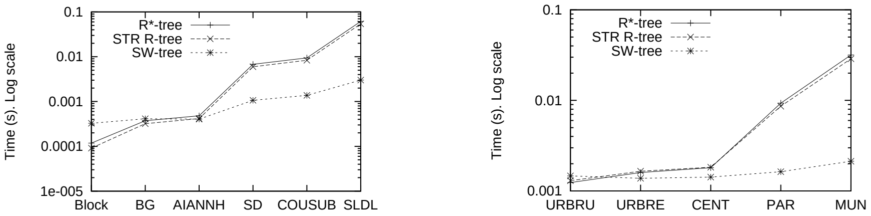
Fig. 4. Time comparison in three synthetic datasets with different distributions.

Another important conclusion that can be extracted from these results is the little dependence of our structure to changes in the selectivity. This is due to the space transformation. We divide the problem into two subproblems, each one concerning one dimension. This decomposition makes queries in the two wavelet trees only marginally dependent on the query size (i.e. selectivity). Note that indexed MBRs correspond to points near the main diagonal of the transformed grids, so that larger MBRs translate into points farther from the main diagonal (above it). The query translates into a 2-sided range query. The point where this transformed query starts is below the main diagonal, farther from it as the query

[5] http://research.att.com/∼marioh/spatialindex/

size is larger, and thus returning more points (more precisely, the returned set size increases linearly with the width of the query across each coordinate). The impact of the size of the query is more clearly reflected in the intersection of the results of both wavelet trees, as an increase by a factor of s^2 in the query area (that is, s per coordinate), translates into a factor of just s (i.e. $s \times N + s \times N$) in the amount of data to intersect. This "square root" impact of the query size in the performance of the algorithm explains its resilience to the query selectivity. Of course, they also explain why our technique does not perform so well when queries are very selective, as we work $O(s)$ time in order to retrieve a result of size $O((s/N)^2 N)$ (taking s as the asymptotic variable).

The surprising time decrease with the increase of the query size in Figure 4(c) is understood 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. Therefore, 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.



(a) Tiger

(b) EIEL

Fig. 5. Time comparison in two real datasets.

Finally, we present the results with the two real datasets. Figures 5(a) and 5(b) present the results with the Tiger and EIEL datasets respectively. In these graphs the real query sets have been sorted accordingly with their selectivity (from left to right the query selectivity is fewer). Even though both R-tree variants improve the performance of our structure when queries are too selective (Block and BG in the Tiger dataset, and URBRU in the EIEL dataset) these results are very promising because our structure improves the performance of both R-tree variants in a broad range of real query sets. Note that all of them are meaningful queries. For example, in the EIEL dataset the query set CENT contains queries of the form *which buildings are contained in the population center X*. The results in the less selective queries (SD, COUSUB and SLDL in the Tiger dataset, and PAR and MUN in the EIEL dataset) are particularly good because the differences with the classical solutions are impressive.

5 Future Work

There are other versions of the R-tree (e.g. the CR-tree [13]) that use compression techniques achieving lower storage requirements than the STR R-tree. However, these structures can produce precision loss, and thus, false positives. We could reduce the precision of the coordinates in order to achieve higher compression rates yet producing false positives too. Note that each false positive involves a huge penalty because a real geographic object has to be retrieved 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. We are also working on allowing the insertion and removal of points once the structure has been constructed. Some recent works [14] describe dynamic versions of *rank* structures that can be used in the design of wavelet trees with insertion and deletion capabilities. Finally, algorithms to solve other queries like k-nearest neighbor or spatial joins are in our plans too.

References

1. Gaede, V., Gnther, O.: Multidimensional access methods. ACM Computing Surveys **30**(2) (1998) 170–231
2. Jones, C.B., Purves, R.S.: Geographical information retrieval. International Journal of Geographical Information Science **22**(3) (2008) 219–228
3. Global Spatial Data Infrastructure Association: Online documentation. Retrieved March 2008 from http://www.gsdi.org/.
4. Brisaboa, N.R., Luaces, M.R., Navarro, G., Seco, D.: A new point access method based on wavelet trees. In: Proc. SeCoGIS'09. ER 2009 Workshops. (2009) 297–306
5. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer-Verlag (2005)
6. Samet, H.: Multidimensional and Metric Data Structures. M. Kaufmann (2006)
7. Alstrup, S., Brodal, G.S., Rauhe, T.: New data structures for orthogonal range searching. Proc. 41st FOCS (2000) 198–207
8. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Proc. 11th WADS. (2009) 98–109
9. Schmidt, J.M.: Interval stabbing problems in small integer ranges. In: Proc. 20th ISAAC. (2009) 163–172
10. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th WEA (Poster). (2005) 27–38
11. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th STOC. (1984) 135–143
12. Salomon, D.: Data Compression: The Complete Reference. Springer-Verlag (2004)
13. Kim, K., Cha, S.K., Kwon, K.: Optimizing multidimensional index trees for main memory access. SIGMOD Record **30**(2) (2001) 139–150
14. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science **410** (2008) 4414–4422