The xBR$^+$-tree: an efficient access method for points

George Roumelis[1,⋆], Michael Vassilakopoulos[2,⋆], Thanasis Loukopoulos[3,⋆], Antonio Corral[4,⋆], and Yannis Manolopoulos[1,⋆]

[1] Dept. of Informatics, Aristotle University of Thessaloniki, Greece. {groumeli, manolopo}@csd.auth.gr
[2] Dept. of Electrical and Computer Engineering, University of Thessaly, Volos, Greece. mvasilako@inf.uth.gr
[3] Dept. of Computer Science and Biomedical Informatics, University of Thessaly, Lamia, Greece. luke@dib.uth.gr
[4] Dept. of Informatics, University of Almeria, Spain. acorral@ual.es

Abstract. Spatial indexes, such as those based on Quadtree, are important in spatial databases for efficient execution of queries involving spatial constraints. In this paper, we present improvements of the xBR-tree (a member of the Quadtree family) with modified internal node structure and tree building process, called xBR$^+$-tree. We highlight the differences of the algorithms for processing single dataset queries between the xBR and xBR$^+$-trees and we demonstrate performance results (I/O efficiency and execution time) of extensive experimentation (based on real and synthetic datasets) on tree building process and processing of single dataset queries, using the two structures. These results show that the two trees are comparable, regarding their building performance, however, the xBR$^+$-tree is an overall winner, regarding spatial query processing.

Keywords: Spatial Access Methods, Quadtrees, xBR-trees, Query Processing.

1 Introduction

Hierarchical index structures are useful because of their ability to focus on the interesting subsets of the data [8]. This focusing results in an efficient representation and improved execution times on query processing and is thus particularly useful for performing spatial operations [13]. Important advantages of these structures are their conceptual clarity and their great capability for query processing. The Quadtree is a well known hierarchical index structure, which has been applied successfully on Geographical Information Systems (GISs), image processing, spatial information analysis, computer graphics, digital databases,

⋆ Work funded by the GENCENG project (SYNERGASIA 2011 action, supported by the European Regional Development Fund and Greek National Funds); project number 11SYN_ 8_1213.

etc. [11, 12]. It was introduced in the early 1970s [2], and it is based on the principle of recursive decomposition of space, and have become an important access method for spatial data [3].

The External Balanced Regular (xBR)-tree [15] belongs to the family of Quadtrees and it has been shown to be competitive to the R*-tree [1] for spatial queries involving a single dataset [10]. In this paper, we present an improved version of the xBR-tree [15], called xBR$^+$-tree, which is also a secondary memory structure that belongs to the Quadtree family. The xBR$^+$-tree improves the xBR-tree node structure and tree building process. The node structure of the xBR$^+$-tree stores information related to the quadrant subregions that contain point data. On the contrary, information related to the position of the quadrants is not stored explicitly in xBR$^+$-tree, but is computed when needed. These, make query processing more efficient.

Apart from the presentation of the xBR$^+$-tree, other contributions of this paper are the conclusions arising from an extensive experimental comparison (based on real and synthetic datasets) of xBR and xBR$^+$-trees, regarding I/O performance and execution time for

- Tree building,
- Point Location Queries (*PLQs*),
- Window Queries (*WQs*) and Distance Range Queries (*DRQs*),
- K-Nearest Neighbor Queries (*K-NNQs*) and Constrained K-Nearest Neighbor Queries (*CK-NNQs*).

To improve readabilty, in Table 1 we present the above and other abbreviations used in this paper.

Table 1. List abbreviations.

Abbreviation	**Term**
PLQ	Point Location Query
WQ	Window Query
DRQ	Distance Range Query
K-NNQ	K-Nearest Neighbor Query
CK-NNQ	Constrained K-Nearest Neighbor Query
K-DJQ	K-Distance Join Query
MBR	Minimum Bounding Rectangle
DBR	Data Bounding Rectangle
Address	Directional digits that specify position and size of a node
REG	Size and position of Quadrant
qside	Size of Quadrant
NAclN	North America Cultural Landmarks Dataset
NAppN	North America Populated Places Dataset
NArrN	North America Rail Roads Centers Dataset
NArrND	North America Rail Roads MBR Coordinates Dataset
NArdN	North America Roads Centers Dataset
NArdND	North America Roads MBR Coordinates Dataset
`XXX`KCN	`XXX` thousands of Clustered Normalized Points

This paper is organized as follows. In Section 2 we review Related Work on Quadtrees and comparable access methods, regarding query processing and provide the motivation for this paper. In Section 3, we review the xBR-tree and present the improvements of the xBR$^+$-tree, paying special attention to the node structure and the tree building process. In section 4, we present the algorithms for processing single dataset spatial queries over xBR and xBR$^+$-trees. In Section 5, we present representative results of the extensive experimentation performed, using real and synthetic datasets, for comparing the performance of the two Quadtree-based structures. Finally, in Section 6 we provide the conclusions arising from our work and discuss related future work directions.

2 Related Work and Motivation

A Quadtree is a class of hierarchical data structures whose common property is that they are based on the principle of *recursive decomposition of space* and it contains two types of nodes: non-leaf (internal) nodes and leaf (external) nodes. It is most often used to partition a 2d space by recursively subdividing it into four quadrants or regions: NW (North West), NE (North East), SW (South West) and SE (South East). According to [14], types of Quadtrees can be classified by following three principles: (1) the type of data that they are used to represent (points, regions, curves, surfaces and volumes), (2) the principle guiding the decomposition process, and (3) the resolution (variable or not).

In order to represent Quadtrees, there are two major approaches: *pointer-based Quadtree* and *pointerless Quadtree*. In general, the *pointer-based Quadtree* representation is one of the most natural ways to represent a Quadtree structure. In this method, every node of the Quadtree will be represented as a record with pointers to its four sons. Sometimes, in order to achieve special operations, an extra pointer from the node to its father could also be included. This representation should be taken into account when considering space requirements for recording the pointers and internal nodes. The xBR-tree belongs to the category of *pointer-based Quadtree*. On the other hand, the *pointerless representation of a Quadtree* defines each node of the tree as a unique locational code [12]. By using the regular subdivision of space, it is possible to compute the locational code of each node in the tree. The linear Quadtree is an example of pointerless Quadtree. We refer the reader to [11–13, 16] for further details.

Regarding to the performance comparison of spatial query algorithms using the most cited spatial access methods (R-trees and Quadtrees), several previous research efforts have been published.

- In [5] a qualitative comparative study is performed taking into account three popular spatial indexes (R*-tree, R$^+$-tree and PMR Quadtree), in large line segment databases. The main conclusion reached was that the R$^+$-tree and PMR Quadtree have the best performance when the operations involve search, since they result in a disjoint decomposition of space.
- In [6], various R-tree variants (R-tree, R*-tree and R$^+$-tree) and the PMR Quadtree have been compared for the traditional overlap spatial join operation. They showed that the R$^+$-tree and PMR Quadtree outperform the

R-tree and R*-tree using 2d spatial data for overlap join, since they are spatial data structures based on a disjoint decomposition of space.

- In [7], the authors have compared the performance of the R*-tree and the Quadtree for evaluating the *K-NN* and the *K* Distance Join (*K-DJ*) query operations and the index construction methods (dynamic insertion and bulk-loading algorithm). It was shown that the query processing performance of R*-tree is significantly affected by the index construction methods, while the Quadtree is relatively less affected by the index construction method. The regular and disjoint partitioning method used by the Quadtree has an inherent structural advantage over the R*-tree in performing *KNN* and distance join queries. Finally, the Quadtree-based index structure can be a better choice than the widely used R*-tree for studied spatial queries when indices are constructed dynamically.
- In [10], the performance of R*-trees and xBR-trees is compared for the most usual spatial queries, like *PLQs*, *WQs*, *DRQs*, *K-NNQs* and *CK-NNQs*. The conclusions arising from this comparison show that the two indexes are competitive. The xBR-tree is more compact and it is built faster than the R*-tree. The performance of the xBR-tree is higher for *PLQs*, *DRQs* and *WQs*, while the R*-tree is slightly better for *K-NNQs* and needs less disk access for *CK-NNQs*.

Finally, xBR-trees have been presented in [15] and results related to the analysis of their performance have been presented in [4]. Using xBR-trees for processing *PLQs*, *WQs* or *DRQs* is rather straightforward [10], due to the organization of the xBR-tree. However, algorithms for processing *K-NNQs* and *CK-NNQs* by using these trees have only recently been developed and tested with real datasets [9, 10], with excellent performance. Therefore, the main objective of this paper is to improve the xBR-tree, obtaining the xBR$^+$-tree, and compare its performance against the performance of the xBR-tree, considering the most representative spatial queries where a single index is involved.

3 The xBR-tree Family

In this section we describe the xBR-tree ([10, 15]) and illustrate the extension of it, the xBR$^+$-tree, which is the primary contribution of this paper. When a characteristic is common for both trees, we will refer to the *xBR-tree family*. Otherwise, we will refer explicitly to the tree type that has this characteristic.

For 2d the hierarchical decomposition of space in the xBR-tree family is that of Quadtrees (the space is recursively subdivided in 4 equal subquadrants). The space indexed by a member of the xBR-tree family is a *square*. The nodes of members of the xBR-tree family are disk pages of two kinds: *leaves*, which store the actual multidimensional data themselves and *internal nodes*, which provide a multiway indexing mechanism.

3.1 Internal Nodes

As described in [10], *internal* nodes of xBR-trees contain entries of the form (*Shape*, *Address*, *REG*, *Pointer*). An *Address* is used to determine the region of a child node and is accompanied by the *Pointer* to this child. Since *Addresses* are of variable size, the number of entries fitting in each node is not predefined. Apparently, the space occupied by all entries within a node must not exceed the size of this node. The maximum size of an *Address* is only limited by the node size and in practice it never reaches this limit. *Shape* is a flag that determines if the region of the child is a complete or non-complete square (the area remaining, after one or more splits; explained later in this subsection). This field will be used widely in queries. Finally, *REG* stores the coordinates of the region referenced by *Address*. We measured the execution time for queries and we found that it is more expensive if we do not save this field, but calculate its value when needed.

Each *Address* represents a subquadrant which has been produced by Quadtree-like hierarchical subdivision of the current space. It consists of a number of directional digits that make up this subdivision. The NW, NE, SW and SE subquadrants of a quadrant are distinguished by the directional digits 0, 1, 2 and 3, respectively. For example, the *Address* 1 represents the NE quadrant of the current space, while the *Address* 10 the NW subquadrant of the NE quadrant of the current space. The address of the left child is * (has zero digits), since the region of the left child is the whole space minus the region of the right child.

However, the region of a child is, in general, the subquadrant of the related *Address* minus a number of smaller subquadrants. The region of this child is the subquadrant determined by the *Address* in its entry, minus the subquadrants corresponding to the next entries of the internal node (the entries in an internal node are saved sequentially, in preorder traversal of the Quadtree that corresponds to the internal node). For example, in Figure 1 an internal node (a root) that points to 2 internal nodes that point to 7 leaves is depicted. The region of the root is the original space, which is assumed to have a quadrangular shape. The region of the right (left) child is the NW quadrant of the original space (the whole space minus the region of the NW quadrant - a non complete square), depicted by the union of the black regions of the leaves of this child. The * symbol is used to denote the end of a variable size address. The *Address* of the right child is 0*, since the region of this child is the NW quadrant of the original space. The *Address* of the left child is * (has zero directional digits), since the region of the left child is the whole space minus the region of the right child. Each of these *Addresses* is expressed relatively to the minimal quadrant that covers the internal node (each *Address* determines a subquadrant of this minimal quadrant). For example, in Figure 1, the *Address* 2* is the SW subquadrant of the whole space (the minimal quadrant that covers the left right child of the root). During a search, or an insertion of a data element with specified coordinates, the appropriate leaf and its region is determined by descending the tree from the root. Although, for the sake of presentation, Figure 1 depicts a tree with nodes having up to four children, note that nodes are disk pages and they are likely to have a significant number of children (xBR-trees are multiway trees).

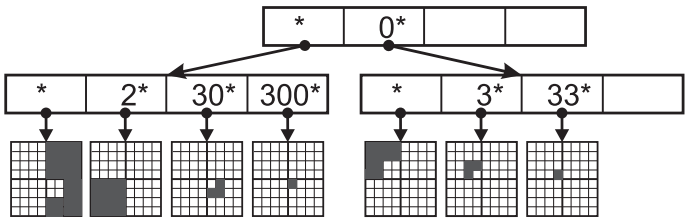

Fig. 1. An XBR$^+$-tree with two levels of internal nodes.

Internal node entries in xBR$^+$-trees contain entries of the form (*Shape*, *qside*, *DBR*, *Pointer*). The fields *Shape* and *Pointer* play the same role as in xBR-trees, *DBR* (Data Bounding Rectangle) stores the coordinates of the subregion that contains point data (at least one point must reside on each side of the *DBR*), while *qside* is the side length of the quadrant that corresponds to the child-node pointed by *Pointer*. In xBR$^+$-trees, the *Address* and the corresponding *REG* of the region of this child node are not explicitly stored. However, only *REG* may be needed by the query processing algorithms and it can be easily calculated using *qside* and *DBR* (in most steps of the query processing algorithms, the use of the values of *qside* and *DBR* is enough and further calculations are avoided). Note that the fields *qside* and *DBR* are represented using *double* precision floating point numbers (like *REG*) which have a fixed size defined by the implementation of the programming language. By avoiding to store the variable-sized field *Address* in xBR$^+$-trees, processing of internal nodes is simplified, since their capacity is fixed. Moreover, the use of *DBR* makes processing of several queries more efficient, since it signifies the subarea of the child node that actually contains data, which is (in general) different to and smaller than the region of this child node, leading to higher selectivity of the paths that have to be followed downwards when descending the tree and deciding the parts of the tree that may contain (part of) the query answer.

In summary, the basic structural differences between the xBR-tree and the xBR$^+$-tree are:

- Internal node entries of xBR$^+$-trees do not contain the *Address* and *REG* fields, but contain the *DBR* and *qside* fields.
- The *Address* and *REG* fields are calculated only when needed.
- Since variable fields are not stored in internal node entries of xBR$^+$-trees, internal nodes have a fixed capacity.

3.2 Leaf Nodes

External nodes (leaves) of members of the xBR-tree family simply contain the data elements and have a predetermined capacity C. When C is exceeded, due to an insertion, the leaf is partitioned according to hierarchical (Quadtree like) decomposition, until each of the resulting two regions (the one region corresponds to the most populated subquadrant and the other region to the rest of the node area) contains data elements with cardinalities $\leq xC$, $0.5 < x < 1$. The choice of

x affects the number of necessary subdivisions of an overflowed node and the size of addresses that result from a node split. A value closer to 0.5, in general, results in more subdivisions and larger addresses, since it is more difficult to partition the region of the leaf in subregions with almost equal numbers of elements. Of course, such a choice provides a better guarantee for the space occupancy of leaves. We used $x = 0.75$, which leads to a good compromise between size of addresses and leaf occupancy. Splitting of a leaf creates a new entry that must be hosted by an internal node of the parent level. This can cause backtracking to the upper levels of the tree and may even cause an increase of its height. Note also that in the xBR-tree family, data elements are stored in leaves in X-order. This ordering permits us to use the *plane sweep* technique (when appropriate) during processing of the data elements of a leaf, in the process of answering certain query types.

3.3 Splitting of Internal Nodes

When an internal node of a member of the xBR-tree family overflows, it is split in two. The goal of this split is to achieve the best possible balance between the space use in the two nodes. The split in the xBR-tree family is either based on existing quadrants or in ancestors of existing quadrants. First, a Quadtree is built that has as nodes the quadrants specified in the internal node [15]. This tree is used for determining the best possible split of the internal node in two nodes that have almost equal number of bits, as proposed in [15], or entries (a simpler and equally effective criterion, according to experimentation).

3.4 Tree Building

Building of a member of the xBR-tree family consists in repetitive insertion of the data elements in the tree by descending the tree for the root, seeking for the appropriate leaf to host each new element. The leaf found may be split, which may cause further splits of internal nodes in the path up to the root. In case of the xBR$^+$-tree, the possibility to avoid creating a new node (leaf, or internal node), but to merge it with another node is considered. The nodes examined for merging are either direct descendants, or direct ancestors of the new node, when we consider the Quadtree that corresponds to the entries of the parent internal node of the new node and candidate merging-nodes.

4 Query Processing Algorithms on the xBR-tree Family

In this section, extending material that appears in [9, 10], we present algorithms for processing *PLQs*, *WQs*, *DRQs*, *K-NNQs* and *CK-NNQs* on the xBR-tree family, highlighting the differences between the xBR and xBR$^+$-trees.

PLQs can be processed in a top-down manner on the xBR-tree family. During a *PLQ* for a point with specified coordinates, the appropriate leaf and its region is determined by descending the tree from the root. Initially, the region

under consideration is the whole space (the region of the root). As noted in Subsection 3.1, the entries in an internal node are saved in preorder traversal of the Quadtree that corresponds to the internal node and are examined in reverse sequential order (this means that we examine first a subregion, before examining an enclosing region of this subregion and in this way we avoid to examine multiple times overlapping regions). So first we examine the last node of the Quadtree. If its subquadrant specified by the *Address* field of the entry, in case of the xBR-tree, or its *DBR*, in case of the xBR$^+$-tree, does not contain the query point, we continue with the next entry in reverse sequential order. The first subquadrant / *DBR* that hosts the query point determines the smallest region that hosts this point. Then we follow the *Pointer* field to the related child at the next lower level, until we reach the leaf level. This way, we reach the unique leaf that may contain the query point. Note that, it is possible the query point to fall within a subquadrant without falling inside the *DBR* of the corresponding a child node. This means that, in case of an unsuccessful search, the search is likely to stop at a middle level of the xBR$^+$-tree, while it will always continue until the leaf level of the xBR-tree.

Processing of *WQs* follows the same strategy to *PLQs*, regarding the way we examine regions/entries of an internal node. The decision about whether we are at a entry with a region likely to contain points inside the query window is the answer to the question: do the subquadrant / *DBR* of the current entry (specified by the *Address* / *DBR* field of the entry) and the query window intersect? If yes, then we follow the pointer to the related child at the next lower level. We repeat until we have examined all entries of the internal node, or until the query window is completely inscribed inside the region (subquadrant) of the entry that we examine (because none of the other, not examined, regions of the tree overlaps with this region). Note that, the use of *DBRs* in xBR$^+$-trees eliminates the possibility to visit a subtree storing data outside the query window.

DRQ follows the same strategy as *WQ*. At first, the querying circle is replaced from its MBR (the calculations are faster in this way) and if the answer about the intersection of the subquadrant / *DBR* of the current entry and the query MBR is positive, then we follow the pointer to the related child at the next lower level. If we reach a leaf with a region that intersects the query MBR, we select the points that are inside the query circle.

For *K-NNQs*, the algorithmic approach for both trees is similar. The search algorithm traverses recursively the tree in a DF (Depth-First) manner (in each node, entries are examined according to *mindist* from the query point and recursively child nodes are visited), or in a BF (Best-First) manner (among all nodes visited so far, entries are examined, and respective children are visited, according to *mindist* from the query point). Both algorithmic approaches utilize a *max K-heap* that stores the K points found so far with the shortest distances to the query point. In the case of the DF (BF) algorithm, an additional local to every internal node (global) *min heap* is kept, storing node entries according to their *mindist* from the query point, calculated using the *Address* field, in case of the xBR-tree, or the DBR field, in case of the xBR$^+$-tree. When the leaves are

reached, the set of entries is sorted in ascending order on X-axis, next this set is split into two subsets, taking as reference the *query point*. Then, the algorithm scans both subsets (from left to right and from right to left) while the distance on X-axis is smaller than the distance value of the *Kth-NN* that has been found so far, inserting those points in *max K-heap*. The *CK-NNQ* algorithm inserts entries in *min heaps* / points in the *max heap*, in case *mindist* of entries / distance of points is larger than a distance threshold. More details about these algorithms for the case of xBR-trees appear in [9, 10].

In summary, the basic algorithmic differences for spatial query processing between the xBR-tree and the xBR$^+$-tree are:

- In *PLQs*, due to the use of *DBRs* an unsuccessful search is likely to stop without descending the whole height of the tree.
- In *WQs*, due to the use of *DBRs* we avoid visiting subtrees that either do not store any data, or store data outside the query window.
- In *DRQs*, due to the use of *DBRs* we avoid visiting subtrees that either do not store any data, or store data outside the circumscribed square of the query circle.
- In *K-NNQs*/*CK-NNQs*, the precedence of *min heap* entries follows *mindist* of *DBRs* from the query point which gives better estimates of the actual distances of the data from the query point.

5 Experimentation

We designed and run a large set of experiments to compare xBR and xBR$^+$-trees and not R-tree variants ([10] documents comparable performance of xBR and R*-trees). We used 6 real spatial datasets of North America, representing cultural landmarks (NAclN with 9203 points) and populated places (NAppN with 24493 points), roads (NArdN with 569120 line-segments) and rail-roads (NArrN with 191637 line-segments). To create sets of 2d points, we have transformed the MBRs of line-segments from NArdN and NArrN into points by taking the center of each MBR (i.e. |NArdN| = 569120 points, |NArrN| = 191637 points). Moreover, in order to get the double amount of points from NArrN and NArdN, we chose the two points with *min* and *max* coordinates of the MBR of each line-segment (i.e. |NArdND| = 1138240 points, |NArrND| = 383274 points). The data of these 6 files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 125000, 250000, 500000 and 1000000 points, with 125 clusters in each dataset (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. The experiments were run on a Linux machine, with Intel core duo 2x2.8 GHz processor and 4 GB of RAM. We run experiments for tree building, counting tree characteristics and creation time and experiments for *PLQs*, *WQs*, *K-NNQs* and *CK-NNQs*, counting disk-page accesses (I/O) and total execution time.

In the first experiment, we built the xBR and xBR$^+$-trees. We stored point coordinates as *double numbers*[5] and constructed each tree for the following node (page) sizes: 512B, 1KB, 2KB, 4KB, 8KB and 16KB. Results for the construction characteristics indicate that, the xBR$^+$-tree in comparison to the xBR-tree has larger height (by 1, or more rarely by 2 levels) in 1/3 of the cases and creates more (54/60) or equal (6/60) internal nodes, uses less space in the most cases (46/60) (i.e. it is more compact) and the creation time varies, in some cases (22/60) is faster having maximum relative difference 18.75% while in some cases (34/60) is slower having worst relative difference -27.18%. Thus, the xBR$^+$-tree is slightly higher and has more internal nodes but needs less space in disk, which means that it has (in all cases) a smaller number of leafs (this is due to the use of *merging leafs or internal nodes* that improves searching efficiency). Results for the construction characteristics (Table 2) are depicted only for one node size for each dataset, due to the limited space (other results were analogous).

Table 2. Tree construction characteristics.

Dataset	**Node size**	**Tree height**		**Tree size (KBytes)**		**Creation time (secs)**	
		xBR	xBR$^+$	xBR	xBR$^+$	xBR	xBR$^+$
NAclN	512B	4	5	412	408	0.16	0.13
NAppN	1KB	4	4	1034	1002	0.31	0.35
NArrN	2KB	4	4	7596	7252	3.19	3.33
NArrND	4KB	3	3	14932	14140	8.60	8.81
NArdN	8KB	3	3	22160	20668	20.12	19.40
NArdND	16KB	3	3	44272	41280	71.30	62.29
125KCN	2KB	3	4	4582	4632	1.97	2.04
250KCN	4KB	3	3	9112	9100	5.36	5.13
500KCN	8KB	3	3	18168	18104	17.22	15.64
1000KCN	16KB	3	3	36224	36000	60.32	52.53

For each dataset, we created rectangular query windows (and their inscribed circles) for studying *WQs* (*DRQs*) by splitting the whole space into 2^4, 2^6, $\cdots$, 2^{16} windows, in a row-order mapping manner. The centroids of these windows were also used as query inputs for all the other queries (*PLQs*, *DRQs*, *K-NNQs* and *CK-NNQs*). Especially, for *K-NNQs* and *CK-NNQs* we used the following set of K values: 1, 10, 100 and 1000. Since the number of experiments performed was vast, we show only representative results, since results were analogous for each query category.

For *PLQs* we executed two sets of experiments. In the first set we used as query input the original datasets and in the second one we used as query input the centroids of the query windows. The results showed that the xBR$^+$-tree needs the same number of disk accesses as its height for every query point, if

[5] Note that we used double numbers for coordinates, instead of float numbers used in [10], to be able to represent large number of points in the normalized square space. Due to this change of representation, results for specific datasets that appear in [10] for the xBR-tree differ slightly from the respective results in this paper.

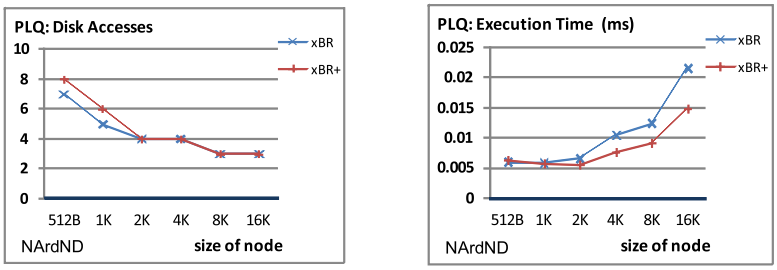

Fig. 2. Total disk accesses (left) and execution time (right) vs. node size for *PLQs* (NArdND). The query points were 1138240 points existing in the dataset.

that point exists in the dataset. In the other case, searching for not existing points in the dataset, the disk accesses may be less than the tree-height. The xBR-tree, in both cases, needs always the same number of disk accesses as its height. Nevertheless, *PLQ* execution time was faster in xBR$^+$-tree in almost all of the experiments. Results for the largest dataset (NArdND) are shown in Figures 2 and 3. Especially in Figure 2 the first set of *PLQs* experiments are shown, where the query points were the 1,138,240 original (existing) points of this dataset. In Figure 3 the second set of *PLQs* experiments are shown, where the query points were the centroids of the 2^{16} query windows. The results for the other datasets were analogous, and always in favor of the xBR$^+$-tree.

We noticed that the xBR-tree needs a number of disk accesses equal to its tree height, while the xBR$^+$-tree needs at most this number of accesses, especially for the case of non existing query points. This finding can easily be explained from the analysis of the algorithms discussed above. This is due to the structural difference of the two trees. Internal nodes of xBR$^+$-trees contain information about *DBRs* that only include data points. In this way, if the dataset has empty regions, the xBR$^+$-tree does not store *DBRs* for these, depending on the data distribution in the space of the node. However, the xBR-tree saves in internal nodes information about the regions in which the space is split, regardless of whether they contain data points. It is important to mention that the number of disk accesses becomes lower as the size of node increases, while execution time increases (for both trees). This may be surprising at first, but can be explained considering the fact that when the size of nodes increases, so does the time for main memory calculations and, consequently, the execution time.

In Figure 4, for the *WQ*, we depict the results for the third synthetic dataset (500KCN), as one representative example. It is shown that the xBR$^+$-tree needed fewer accesses than xBR-tree, to find the population of 4096 windows with which we scanned the whole space occupied from the 500K clustered data points of this dataset. As the size of node increases the relative I/O difference between the two trees becomes smaller. In both trees, a logarithmic dependence of the number of disk accesses to the size of the node appears. Note the reduction of the difference from the smallest node size (512B) to the largest size (16KB). This is due to the reduction of tree height as the size of node increases.

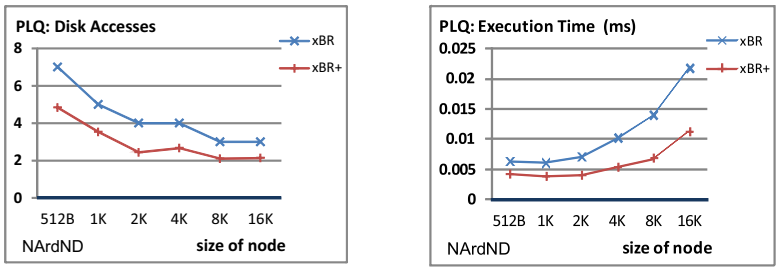

Fig. 3. Total disk accesses (left) and execution time (right) vs. node size for *PLQs* (NArdND). The query points were 16384 points non-existing in the dataset.

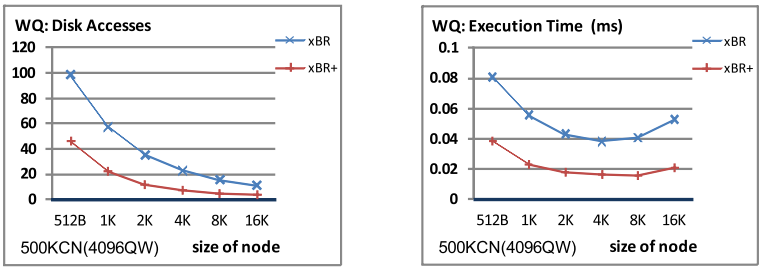

Fig. 4. Total disk accesses (left) and execution time (right) vs. node size for *WQs* (500KCN, 4096 query windows).

In the right part of Figure 4, for the execution time, it is shown that the xBR$^+$-tree is faster for all sizes of nodes. The explanation for this fact is again related to the structural difference between two trees. Two additional remarks can be made. First, in a more detailed observation we can see that the relative difference of performance between the two trees in execution time is a little smaller than in disk accesses. This may be explained by the fact that even though xBR$^+$-tree stores *DBRs* that improve time performance, it occasionally needs to spend time to calculate the coordinates of the region pointed by the *Address* of the subquadrant. This fact does not affect the number of disk accesses. The second remark is that the number of disk accesses appears to have monotonic dependence to the size of nodes. The execution time does not have the same characteristic. Both trees seem to exhibit an inflection point in the dependence of the execution time to the size of nodes. This point of optimal execution time performance appears when the size of nodes is equal to *4KB*. This optimization in execution time is stopped for size of nodes larger than 8KB. This behavior holds for the experiments of all datasets and all query windows.

For *DRQs* (1024 query circles, inscribed into the respective rectangular windows, δ value equal to $1/(2\sqrt{1024})$=1/64 of the space side length), the xBR$^+$-tree needed less disk accesses and was faster than the xBR-tree, in all cases and for all datasets (Figure 5). The performance of the xBR$^+$-tree for all datasets and for all sets of query circles (2^4, 2^6, 2^8, 2^{10}, 2^{12} and 2^{14}) for every size of node (512B, 1KB, 2KB, 4KB, 8KB and 16KB) is 60-0 in all cases for the disk accesses

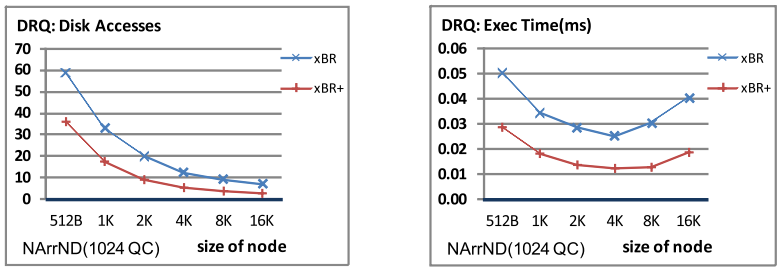

Fig. 5. Total disk accesses (left) and execution time (right) vs. node size for *DRQs* (NArrND, 1024 query circles).

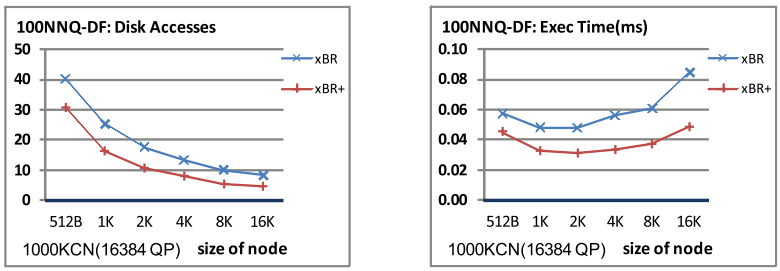

Fig. 6. Total disk accesses (left) and execution time (right) vs. node size for *KNNQs* using DF algorithm (1000KCN, 16384 query points, searching for K=100).

and (59-1, 57-3, 60-0, 60-0, 60-0 and 60-0) for the execution time, where a pair like 57-3 signifies that the xBR$^+$-tree wins 57 times and the xBR-tree wins 3 times, in whole set of experiments. The lower improvement in execution time can be explained just as in the previous paragraph.

For the *K-NNQ*, the xBR-tree showed similar behavior to the *WQ*. The xBR$^+$-tree needed fewer disk accesses for finding the nearest neighbors than the xBR-tree. Furthermore, the difference became larger when the size of node (for the same dataset) increased. Regarding the execution time, the xBR$^+$-tree showed improved performance, in relation to its I/O difference from the xBR-tree. In Figures 6, we show results for K=100 and the large synthetic dataset (1000KCN) using algorithms DF. At this point, the worse time performance of both trees, for larger node sizes (where the I/O cost is smaller) must be noted. This is due to the fact that as the node size increases, the trees become very wide and very short. In this case, a node holds many elements to be processed and branching during tree descend plays a smaller role in restricting the search space. This leads us to the conclusion that the increase of the node size leads to many more calculations in main memory, canceling the benefit of reducing I/O. For all experiments performed, the xBR$^+$-tree shows better performance than the xBR-tree in disk accesses and execution time for both DF and BF algorithms.

Finally, for the *CK-NNQs* we noticed that the xBR$^+$-tree was improved for both performance categories of our study. In Figure 7, we present the results of *CK-NNQs* for the largest real dataset (NArdN) for all (256 query points, setting

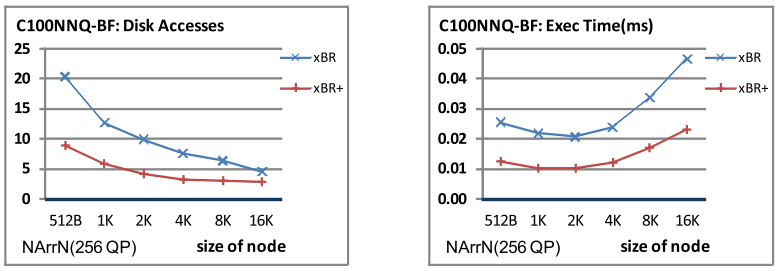

Fig. 7. Total disk accesses (left) and execution time (right) vs. node size for *CKNNQs* using BF algorithm (NArdN, 256 query points, searching for K=100 and δ=1/32).

K = 100 and δ value equal to $1/(2\sqrt{256})$=1/32 of the space side length) using the BF algorithm. The behavior is similar to *K-NNQs*.

In summary, the experimental comparison showed that

- The xBR$^+$-tree needs less space and is built in a similar time, while its height is slightly larger than the xBR-tree.
- The time and I/O performance of the xBR$^+$-tree is better than the xBR-tree for *PLQs*, *WQs*, *DRQs*, *K-NNQs* and *CK-NNQs* for DF and BF algorithms.
- Main memory processing of xBR$^+$-trees is simpler and faster thanks to storing *DBRs*, though some execution efficiency is spent for the cases where the coordinates of *REGs* need to be calculated.

6 Conclusions and future work

In [10], the xBR-tree was compared to the R*-tree and, as future work, possible variations of the xBR-tree that might improve its performance were proposed. In this paper, elaborating on these variations, we developed an improved version of the xBR-tree, called xBR$^+$-tree, and compared it experimentally to the xBR-tree. Moreover, we presented the differences of the algorithms for processing PLQs, WQs, DRQs, KNNs and CKNNs in xBR$^+$-trees.

The presented extensive experimental comparison, based on real and synthetic data, showed that the xBR$^+$-tree is a global winner in I/O and execution time, considering the most representative spatial queries that involve a single index, while building of the two trees has comparable efficiency. More specifically, I/O is improved (on the average) by 23% in PLQs for non-existing points, by 45.4% in WQs, by 45.4% in DRQs, by 33.4% in 100NNs-DF, by 42% in 100NNs-BF, by 49.6% in C100NNs-DF, by 55% in C100NNs-BF and execution time is improved (on the average) by 37.4% in PLQs for non-existing points, by 42.2% in WQs, by 41.7% in DRQs, by 24.2% in 100NNs-DF, by 40.1% in 100NNs-BF, by 40.5% in C100NNs-DF, by 48.9% in C100NNs-BF. In fact, in 98% of the cases, the xBR$^+$-tree excels in I/O and time performance by at least 5%.

Due to its improved building process, the xBR$^+$-tree is smaller and taller than the xBR-tree, while the use of the *DBR* field in xBR$^+$-trees represents populated regions more accurately, gives better estimates of the actual distances

of the data from the query point, improves pruning of subtress and simplifies main memory processing.

Future work might include studying how the higher performance of the xBR$^+$-tree is achieved, in terms of Euclidean distance and X-axis distance calculations saved and heap operations performed. This insight might permit further optimizations of the new structure. Moreover, a detailed relative performance study of the xBR$^+$-tree against the R*-tree and/or R$^+$-tree for single dataset and multi-dataset queries is a target in process.

References

1. Beckmann N., Kriegel H.P., Schneider R. and Seeger B.: The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference, pp. 322-331, 1990.
2. Finkel R. and Bentley J.: Quad trees: A data structure for retrieval on composite keys. Acta Informatica 4, 1-9, 1974.
3. Gaede V. and Gunther O.: Multidimensional Access Methods. ACM Computing Surveys, 30(2): 170-231, 1998.
4. Gorawski M. and Bugdol M.: Cost Model for XBR-tree. Chapter in book New Trends in Data Warehousing and Data Analysis. Kozielski S. & Wrembel R. (Eds.), Springer, 2009.
5. Hoel E.G. and Samet H.: A Qualitative Comparison Study of Data Structures for Large Line Segment Databases. SIGMOD Conference, pp. 205-214, 1992.
6. Hoel E.G. and Samet H.: Benchmarking Spatial Join Operations with Spatial Output. VLDB Conference, pp. 606-618, 1995.
7. Kim Y. J. and Patel J.: Performance Comparison of the R*-tree and the Quadtree for kNN and Distance Join Queries. IEEE Transactions on Knowledge and Data Engineering 22(7): 1014-1027, 2010.
8. Manolopoulos Y., Nanopoulos A., Papadopoulos A. and Theodoridis Y.: R-Trees: Theory and Applications. Springer, 2006.
9. Roumelis G., Vassilakopoulos M. and Corral A.: Algorithms for processing Nearest Neighbor Queries using xBR-trees, Panhellenic Conference on Informatics, pp. 51-55, 2011.
10. Roumelis G., Vassilakopoulos M. and Corral A.: Performance Comparison of xBR-trees and R*-trees for Single Dataset Spatial Queries. ADBIS Conference, pp. 228-242, 2011.
11. Samet H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, 1990.
12. Samet H.: Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, 1990.
13. Samet H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers, 2006.
14. Samet H.: The Quadtree and related hierarchical data structure. Computing Surveys 16(2), pp. 187-260, 1984.
15. Vassilakopoulos M. and Manolopoulos Y.: External Balanced Regular (x-BR) Trees: New Structures for Very Large Spatial Databases. Panhellenic Conference on Informatics, pp. 324-333, 2000.
16. Yin X., Düntsch I. and Gediga G.: Quadtree Representation and Compression of Spatial Data. Transactions on Rough Sets, 13: 207-239, 2011.