The Journal of Systems and Software 132 (2017) 165–185

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Efficient query processing on large spatial databases: A performance study

George Roumelis[a], Michael Vassilakopoulos[b,*], Antonio Corral[c], Yannis Manolopoulos[d]

[a] Department of Informatics, Aristotle University, Thessaloniki, Greece
[b] Department of Electrical & Computer Engineering, University of Thessaly, Volos, Greece
[c] Department on Informatics, University of Almeria, Almeria, Spain
[d] Department of Informatics, Aristotle University, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 5 October 2016
Revised 6 January 2017
Accepted 5 July 2017
Available online 6 July 2017

Keywords:
Spatial databases
Spatial access methods
Quadtrees
xBR-trees
R-trees
Query processing
Performance evaluation

ABSTRACT

Processing of spatial queries has been studied extensively in the literature. In most cases, it is accomplished by indexing spatial data using spatial access methods. Spatial indexes, such as those based on the Quadtree, are important in spatial databases for efficient execution of queries involving spatial constraints and objects. In this paper, we study a recent balanced disk-based index structure for point data, called xBR$^+$-tree, that belongs to the Quadtree family and hierarchically decomposes space in a regular manner. For the most common spatial queries, like *Point Location, Window, Distance Range, Nearest Neighbor* and *Distance-based Join*, the R-tree family is a very popular choice of spatial index, due to its excellent query performance. For this reason, we compare the performance of the xBR$^+$-tree with respect to the R*-tree and the R$^+$-tree for tree building and processing the most studied spatial queries. To perform this comparison, we utilize existing algorithms and present new ones. We demonstrate through extensive experimental performance results (I/O efficiency and execution time), based on medium and large real and synthetic datasets, that the xBR$^+$-tree is a big winner in execution time in all cases and a winner in I/O in most cases.

1. Introduction

Due to the demanding need for efficient spatial access methods in many spatial database applications (Rigaux et al., 2000; Shekhar and Chawla, 2003), significant research effort has been devoted to the development of new spatial index structures (Samet, 1990b; Manolopoulos et al., 2006; Samet, 2007). However, as shown in several previous comparative studies (Hoel and Samet, 1992; 1995; Kim and Patel, 2010; Kanth et al., 2002), there is no unique index structure that works efficiently, in all cases. These performance studies were executed taking into account a great variety of modern applications, where a variety of Spatial Queries arise.

The most common spatial queries where points are involved are Point Location, Window, Distance Range, Nearest Neighbor and Distance-based Join Queries. Moreover, such queries have been also used as the basis of many complex operations in advanced applications (e.g. multimedia databases (Faloutsos et al., 1994), medical images databases (Korn et al., 1996), geometric databases (Mehrotra and Gary, 1993), CAD (Jagadish, 1991), Geographical Information Systems (GIS) (Samet, 1990a), etc).

Hierarchical index structures are useful because of their ability to focus on the interesting subsets of data (Samet, 1990b; Manolopoulos et al., 2006). This focusing results in an efficient representation and improved execution times on query processing and is, thus, particularly useful for performing spatial operations (Samet, 2007). 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 GIS, image processing, spatial information analysis, computer graphics, digital databases, etc. (Samet, 1990a; 2007). It was introduced in the early 1970s (Finkel and Bentley, 1974), it is based on the principle of recursive decomposition of space and has become an important access method for spatial data (Gaede and Günther, 1998).

The External Balanced Regular (xBR)-tree (Vassilakopoulos and Manolopoulos, 2000) is a secondary memory structure that belongs to the Quadtree family (widely used in GIS applications (Samet, 1990a)), which is suitable for storing and indexing multidimensional points (and, in extended versions, line segments, or other spatial objects). We utilize an improved version of xBR-tree, called xBR$^+$-tree (Roumelis et al., 2015), which is also a disk-

* Corresponding author.
E-mail addresses: groumeli@csd.auth.gr (G. Roumelis), mvasilako@uth.gr (M. Vassilakopoulos), acorral@ual.es (A. Corral), manolopo@csd.auth.gr (Y. Manolopoulos).

http://dx.doi.org/10.1016/j.jss.2017.07.005
0164-1212/

resident structure. The xBR$^+$-tree improves the xBR-tree in the node structure and in the splitting process. The node structure of the xBR$^+$-tree stores information which makes query processing more efficient.

In this paper, we compare the xBR$^+$-tree with popular R-tree indexes, regarding storage requirements, time needed for the tree construction and spatial query performances. The family of R-trees has been populated with lots of assorted variations. Each variation tries to optimize a particular aspect (splitting, deletion, etc). However, we concentrate on the R*-tree (Beckmann et al., 1990), because *it is the most commonly employed spatial indexing structure in the spatial database community* (Manolopoulos et al., 2006; Shekhar and Chawla, 2003), and to the R$^+$-tree, because *it is an index structure based on disjoint decomposition of space like the xBR$^+$-tree.*

This paper substantially extends our previous work Roumelis et al. (2011b) (where xBR-trees were compared to R*-trees using single dataset queries and datasets of medium size and it was shown that the two structures are comparable) and Roumelis et al. (2015) (where a new tree, the xBR$^+$-tree, was presented and compared to the xBR-tree using single dataset queries and datasets of medium size and it was shown that the two structures are comparable in building, while the xBR$^+$-tree is a winner in query processing) and its contributions include the following:

- The presentation of a new alternative Depth-First (DF) algorithm for Distance Range Queries (*DRQs*), *K* Nearest Neighbor Queries (*KNNQs*) and Constrained *K* Nearest Neighbor Queries (*CKNNQs*), utilizing a minimum binary heap (*minHeap*) instead of sorting on the xBR$^+$-tree, R*-tree and R$^+$-tree,
- The presentation of the first algorithms for *K* Closest Pair Queries (*KCPQs*), εDistance Join Queries (ε*DJQs*) on the xBR$^+$-tree, and presentation of new alternative DF algorithms for *KCPQs* and ε*DJQs*, utilizing a *minHeap* instead of sorting, on R*-trees and R$^+$-trees,
- A detailed performance comparison (I/O and execution time) of xBR$^+$-trees (non-overlapping trees of the quadtree family) against R$^+$-trees (non-overlapping trees of the R-tree family) and R*-trees (industry standard belonging to the R-tree family) on tree building, single dataset queries (Point Location Queries -*PLQs*-, Window Queries -*WQs*-, *DRQs, KNNQs* and *CKNNQs*) and dual dataset (distance-based join) queries (*KCPQs*, ε*DJQs*). Note that the performance study was conducted on medium and large spatial (real and synthetic) datasets.

Note that, in this paper we utilize large spatial datasets (where the quantifier "large" designates several tens of millions of spatial objects) since we believe that such datasets can be effectively processed in centralized systems, if efficient methods are used. Even larger (huge) datasets would require the utilization of methods on parallel and distributed environments (e.g. http://spatialhadoop.cs.umn.edu).

This paper is organized as follows. In Section 2 we review related work on comparing spatial access methods, regarding spatial query processing and provide the motivation for this work. In Section 3, we briefly review the main characteristics of the R-trees (highlighting the R*-tree and R$^+$-tree). In Section 4, we describe the xBR$^+$-tree. In Section 5, we present the algorithms for processing spatial queries, where one or two datasets are involved, over R-trees and the xBR$^+$-tree. In Section 6, we show results of the extensive experimentation performed, using real and synthetic datasets, for comparing the performance of the two R-trees index structures (R*-tree and R$^+$-tree) and the xBR$^+$-tree. Finally, in Section 7 we provide the conclusions arising from this research work and discuss related future work directions.

2. Related work and motivation

The Quadtree belongs to a class of hierarchical data structures whose common property is that they are based on the principle of *recursive regular decomposition of space*. These structures are characterized as *space-driven access methods* according to Rigaux et al. (2000). 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 Samet (1984), 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 approaches: the *pointer-based* and *pointerless* approaches. 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 process specific operations, an extra pointer from a node to its father could also be included. The xBR$^+$-tree belongs to the category of *pointer-based Quadtrees*. On the other hand, the *pointerless representation of a Quadtree* defines each node of the tree as a unique locational code (Samet, 1990a). 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 Samet (1990a, 1990b, 2007) and Yin et al. (2011) for further details.

The xBR$^+$-tree (Roumelis et al., 2015) belongs to the category of *pointer-based Quadtrees* and it is an extension of the xBR-tree (Vassilakopoulos and Manolopoulos, 2000; Roumelis et al., 2011b). The xBR$^+$-tree has similarities with other well-known multidimensional access methods (Gaede and Günther, 1998). For example, the form of nodes in xBR$^+$-trees has similarities to the form of nodes of Generalized BD-trees (GBD-trees) (Ohsawa and Sakauchi, 1990). GBD-trees are based on kd-tree-like decomposition of space, while xBR$^+$-trees on Quadtree-like decomposition. Moreover, the splitting of internal nodes in xBR$^+$-trees is handled in a more sophisticated way than in GBD-trees. The xBR$^+$-tree has also similarities to the hB-tree (Lomet and Salzberg, 1990), where space is also partitioned according to kd-trees (unlike the xBR$^+$-tree, where partitioning follows the *data space hierarchy principle*) and *holey brick*-like regions are created. Unlike the hB-tree, in the xBR$^+$-tree, each internal node has only one pointer to a child node and the entries of an internal node are region-pointer pairs and not tree structures (kd-trees), as in the hB-tree. Finally, we refer the reader to Samet (1990a, 1990b), Gaede and Günther (1998), Samet (2007) and Yin et al. (2011) for further details on multidimensional access methods.

Regarding 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 Hoel and Samet (1992) a qualitative comparative study was performed taking into account three popular spatial indexes (R*-tree (Beckmann et al., 1990), R$^+$-tree (Sellis et al., 1987) and PMR Quadtree (Nelson and Samet, 1986)), in the context of processing spatial queries (point query, nearest line segment, window query, etc.) in large line segment databases. The conclusion reached was that the R$^+$-tree and PMR Quadtree were the best when the operations involve search, since they result in a disjoint decomposition of space. On the other hand, the R*-tree was more compact than the R$^+$-tree (and the PMR Quadtree) but its performance was not as good as the R$^+$-tree, due to the non-disjointness of the decomposition induced by it.

In Hoel and Samet (1995), various R-tree variants (R-tree (Guttman, 1984), R*-tree and R$^+$-tree) and the PMR Quadtree have been compared for the traditional spatial *overlap join* operation.

They showed that the R+-tree and PMR Quadtree outperform the R-tree and R*-tree using 2d GIS spatial data. That is, with respect to the overlap join, the spatial data structures based on a disjoint decomposition of space (like the R+-tree and PMR Quadtree) outperformed spatial data structures based on a non-disjoint decomposition such as the numerous variants of the R-tree including the R*-tree. Moreover, as the size of the output of the spatial join increases with respect to the larger of the two inputs, methods based on a disjoint regular decomposition (like the PMR Quadtree) performed significantly better. Due to the good performance results of the R+-tree for overlap join, in this research work, we have compared this structure to the xBR+-tree for spatial queries.

Another interesting comparison was presented in Kanth et al. (2002), where the R-tree and the Quadtree have been contrasted in the context of Oracle Spatial, using a variety of range and Nearest Neighbor (NN) queries on spatial data arising in 2d Geographical Information Systems (GISs). It was shown that, in general, the R-tree outperforms the Quadtree. From this experimental comparison, Oracle, in general, recommends using R-trees over Quadtrees, due to higher tiling levels in the Quadtree that cause very expensive preprocessing and storage costs.

In Chen and Patel (2007), the R*-tree and a Quadtree index enhanced with Minimum Bounding Rectangle (MBR) keys for the internal nodes (MBRQuadtree) have been compared with respect to the All-Nearest Neighbor (ANN) query. The ANN query takes as input two datasets of multidimensional points and computes for each point in the first dataset the NN in the second one. Experimentally, the authors showed that for ANN queries, the MBRQuadtree is a much more efficient indexing structure than the R*-tree index.

In Kim and Patel (2010), the authors have compared the performance of R-trees and Quadtrees index structures for evaluating the *K*NN and the *K* Distance Join (using the algorithms described in Hjaltason and Samet (1998)) query operations and the index construction methods (dynamic insertion for the R*-tree and bucket Quadtree) and bulk-loading algorithm (Sort-Tile-Recursive, STR, for the R-tree (Leutenegger et al., 1997) and bulk-loading for the Quadtree). It was shown that the query processing performance of the R*-tree was significantly affected by the index construction methods, while the Quadtree was 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 *K*NN and *K* Distance Join queries. The Quadtree-based index structure could be a better choice than the widely used R*-tree for spatial queries when indices are constructed dynamically. Moreover, it was shown that when data are static (i.e. when a bulk-loading algorithm is used for an index construction) and *KNNQs* / *K* Distance Join Queries are executed, the STR built R-tree showed the best performance. However, when data are dynamic (i.e. there are frequent updates), a bucket Quadtree begins to outperform the R*-tree. This is due to overlap among MBRs that increases with increasing dataset sizes (once the dynamic R*-tree algorithm is used), and the R*-tree performance is degraded.

In the context of performance studies, in Corral and Almendros-Jiménez (2007), an interesting performance comparison (with respect to number of disk read accesses, response time and memory requirements) of distance-based query (Distance Range, *K*-Nearest Neighbors, *K*-Closest Pairs and εDistance Join) algorithms (Depth-First Search -DFS-, Best-First Search -BFS- and Recursive Best-First Search -RBFS-) on R*-trees was presented. The main conclusion was that BFS was the best for all studied distance-based queries, but it may consume many main memory resources. DFS was slightly worse than BFS (except for the case where an LRU-buffer is included), but it consumed less memory resources, since it needs linear space with respect to the height of the R*-trees. RBFS was the worst alternative (although it uses recursion and it needs linear space) since it revisits internal nodes to follow a best-first order.

In Roumelis et al. (2011b), the performance of R*-trees and xBR-trees was compared for the most usual spatial queries, like *Point Location, Window, Distance Range, K Nearest Neighbor* and *Constraint KNN* queries. The conclusions arising from this comparison showed that the two indexes were competitive. The xBR-tree is more compact and it is built faster than the R*-tree. The performance of the xBR-tree was higher for *PLQs, DRQs* and *WQs*, while the R*-tree was slightly better for *KNNQs* and needed less disk accesses for *CKNNQs*.

Finally, in Roumelis et al. (2015) improvements of the xBR-tree (modified internal node structure and tree building process) were presented, leading to the xBR+-tree. An extensive performance studio (I/O efficiency and execution time) based on real and synthetic datasets was also presented, taking into account the tree building process and the processing of single dataset queries, using the two Quadtrees-based structures. These results showed that the two trees are comparable regarding their building performance, however, the xBR+-tree was an overall winner regarding spatial query processing.

The main objective of this paper is to compare the xBR+-tree performance (Roumelis et al., 2015) (the best index structure of the *xBR-tree family*) against the performance of the most popular spatial access method of the *R-tree family*, the R*-tree and a non-overlapping member of this family, the R+-tree, considering the most representative spatial queries, where one or two indexes are involved and to highlight the performance winner, considering the characteristics of each query. Our contribution differs from Kim and Patel (2010) in the following aspects:

- We utilized a new dynamic, disk-resident, balanced Quadtree-based index structure (called xBR+-tree). In Kim and Patel (2010), a simple bucket Quadtree, a partially RAM-resident, unbalanced structure was utilized.
- The performance comparison is carried out for more spatial queries when one dataset is involved (*PLQ, WQ, DRQ* and *CKNNQ*) and when two datasets are involved (ε*DJQ*), not only for the *KNNQ* and *KCPQ* (called *K* Distance Join Query in Kim and Patel (2010)).
- We have compared the xBR+-tree with the R+-tree also (an R-tree index based on disjoint decomposition of space), not only with the R*-tree as in Kim and Patel (2010).
- We have used in our experiments two large *real* datasets from OpenStreetMap[1] with 5.8 and 11.5 million of 2d points, whereas in Kim and Patel (2010), the authors used *artificial* data from Palomar Observatory Sky Survey[2], choosing for their experiments just the first 2 millions of records from the original data (from around 90 millions) and for creating 2d points, the first two attributes of the 39 stored.

3. The R-tree family

R-trees (Guttman, 1984) are hierarchical, height balanced data structures, derived from B-trees (Comer, 1979) and designed to be used in secondary storage. R-trees are considered as excellent choices for indexing various kinds of spatial data (points, rectangles, line-segments, polygons, etc.) and have been adopted in known commercial systems (e.g. Informix (Brown, 2001), Oracle Spatial (Kothuri et al., 2007; Greener and Ravada, 2013), MySQL (Schwartz et al., 2012), PostGIS (Obe and Hsu, 2015; Corti et al., 2014), etc.). They are used for the dynamic organization of a set of

[1] http://spatialhadoop.cs.umn.edu/datasets.html.
[2] http://astronomy.mnstate.edu/cabanela/MAPS_Database/.

spatial objects represented by their Minimum Bounding Rectangles (MBRs). The MBR represents the smallest axes-aligned rectangle in which the spatial objects are contained. A 2d MBR is determined by two 2d points that belong to its faces, one that has the minimum and one that has the maximum coordinates (these are the endpoints of one of the diagonals of the MBR). Using the MBR instead of the exact geometrical representation of the object, its representational complexity is reduced to two points where the most important features of the spatial object (position and extension) are maintained. Consequently, the MBR is an approximation widely employed, and the R-trees belong to the category of *data-driven access methods* (Rigaux et al., 2000), since their structure adapts itself to the MBRs distribution in the space (the partitioning adapts to the object distribution in the embedding space).

The rules obeyed by the R-tree are as follows.

1. All leaves reside on the same level.
2. Each leaf node contains entries, *E*, of the form *(MBR, Oid)*, such that *MBR* is the minimum bounding rectangle that encloses the object determined by the identifier *Oid*.
3. Internal nodes contain entries, *E*, of the form *(MBR, Addr)*, where *Addr* is the address of the child node and *MBR* is the minimum bounding rectangle that encloses MBRs of all entries in that child node (it is also called *directory MBR*).
4. Nodes (except possibly for the root) of an R-tree of class *(m, M)* contain between *m* and *M* entries, where $m \leq \lceil M/2 \rceil$ (*M* and *m* are called maximum and minimum branching factor, or fan-out).
5. The root contains at least two entries, if it is not a leaf.

For more details about the R-tree structure, see Manolopoulos et al. (2006).

Like other spatial tree-like index structures, an R-tree partitions the multidimensional space by grouping objects in a hierarchical manner. A subspace occupied by a tree node in an R-tree is always contained in the subspace of its parent node, i.e. the *MBR enclosure property*. According to this property, an MBR of an R-tree node (at any level, except at the leaf level) always encloses the MBRs of its descendant R-tree nodes. This property of spatial containment between MBRs stored in R-tree nodes is commonly used by spatial queries as the *WQ* and spatial join. Another important property of the R-trees that store spatial objects in a spatial database is the *MBR face property*. This property says that every face of any MBR of an R-tree node (at any level) touches at least one point of some spatial object in the spatial database. Distance-based queries, like the *KNNQ, DRQ, KCPQ* and *εDJQ*, use this last property.

3.1. The R*-tree

Many variations of R-trees have appeared in the literature (exhaustive surveys can be found in Gaede and Günther, 1998; Manolopoulos et al., 2006). One of the most popular and efficient variations is the R*-tree (Beckmann et al., 1990). The R*-tree is a variant of the R-tree that provides several improvements to the insertion algorithm. Essentially, these improvements aim at optimizing the following parameters: (1) node overlapping, (2) area covered by a node, and (3) perimeter of an MBR of internal node. The latter is representative of the shape of the rectangles because, given a fixed area, the shape that minimizes the rectangle perimeter is the square.

The R*-tree added two major enhancements to the R-tree, in case a node overflows. First, rather than just considering the area, the *node-splitting* algorithm in the R*-tree also minimized the perimeter and overlap enlargement of the MBRs. It tends to reduce the number of subtrees to follow for search operations. Second, the R*-tree introduced the notion of *forced reinsertion* to make the tree

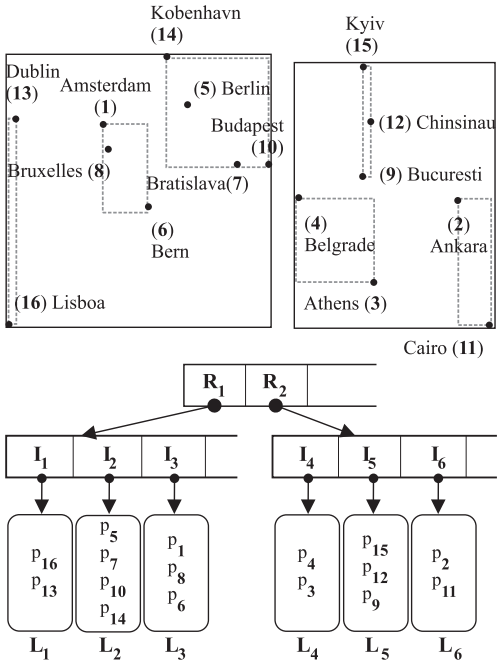

Fig. 1. A collection of points representing 16 capital cities, the corresponding grouping to R*-tree nodes and the R*-tree structure.

shape less dependent to the insertion order. When a node overflows, it is not split immediately, but a portion of entries of the node is reinserted from the tree root. The forced reinsertion provides two important improvements. First, it may reduce the number of splits and, second it is a dynamic technique for tree reorganization. With these two enhancements, the improved split algorithm and the reinsertion strategy, the R*-tree results in a much better organization with respect to the original R-tree.

It is worth remembering that the data structures for the R-tree and R*-tree are the same. Hence, the data retrieval operations defined for the R-tree remain valid for the R*-tree. Due to the better organization of the R*-tree, the performance of the spatial queries is likely to be significantly improved. For this reason, the R*-tree generally outperforms R-tree and it is commonly accepted that the R*-tree is one of the most efficient R-tree variants (Rigaux et al., 2000).

Fig. 1 depicts a collection of points representing 16 *capital cities* and the corresponding R*-tree (assuming $M = 4$ and $m = 2$), where the tree nodes correspond to disk pages. Observe that the index structure, while keeping the tree balanced, adapts to the skewness of data distribution. Solid lines denote the MBRs of the subtrees that are rooted in inner nodes (dotted rectangles). In this figure, the leaves are represented by L_i $(1 \leq i \leq 6)$, the MBRs enclosing points are denoted as I_i $(1 \leq i \leq 6)$ and R_i $(1 \leq i \leq 2)$ correspond to the MBRs enclosing I_i MBRs.

3.2. The R+-tree

To overcome the problems associated with overlapping of regions in the R-trees, in Sellis et al. (1987) an access method called the R+-tree was introduced. The main motivation for the R+-tree is to avoid overlap among the MBRs. Unlike the R-tree, the R+-tree uses *clipping*; that is, there is no overlap between MBRs at the same tree level. MBRs that intersect more than one MBRs have to be stored on several different nodes. The result of this data structure is that there may be several paths, starting at the root to the same rectangle. As a result of this policy and taking into account its structure, the R+-tree will lead to an increase of the height of

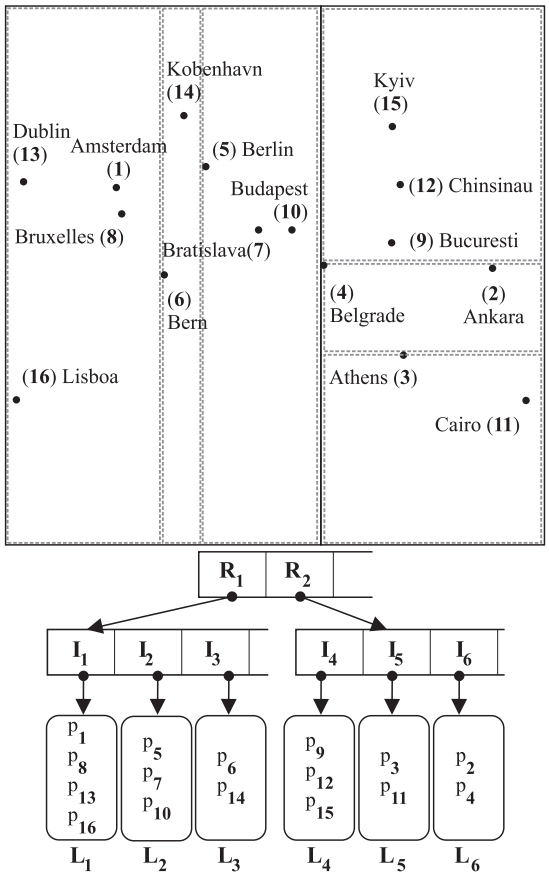

Fig. 2. The same collection of points, the corresponding grouping to R⁺-tree nodes and the R⁺-tree structure.

the tree. However, considering the retrieval time, point searches in R⁺-trees correspond to single-path tree traversals from the root to one of the leaves (and therefore, it tends to be faster than the corresponding R-tree operation). On the other hand, range searches usually lead to the traversal of multiple paths in both index structures.

The R⁺-tree can be characterized as follows (Rigaux et al., 2000):

1. The root has at least two entries, except when it is a leaf.
2. The MBRs of two internal nodes at the same level cannot overlap.
3. If node *N* is not a leaf (internal node), its MBR contains all rectangles in the subtree rooted at *N*.
4. A rectangle of the collection to be indexed is assigned to all leaf nodes the MBRs of which it overlaps. A rectangle assigned to a leaf node *N* is either overlapping *N.MBR* or is fully contained in *N.MBR*. This duplication of objects into several neighbor leaves is similar to what we encountered earlier in other *space-driven structures* (they are based on partitioning the embedding space into rectangular cells, independently of the distribution of the spatial objects).

Fig. 2 presents an R⁺-tree for the same collection of points. Note also that both at the leaf level and at internal levels, node MBRs are not overlapping (different organization of the nodes with respect to Fig. 1). The notation of internal nodes and leaves are the same as in the R*-tree of Fig. 1.

The structure of an R⁺-tree node is the same as that of the R-tree. However, because we cannot guarantee a minimal storage utilization *m* (as for the R-tree), and because rectangles are duplicated, an R⁺-tree can be significantly larger (in terms of height) than the R-tree built for the same dataset. The construction and maintenance of the R⁺-tree are rather more complex than with the other variants of the R-tree.

As examples of spatial query processing using R⁺-trees, the *point location query* performance benefits from the non-overlapping of nodes. As for *space-driven structures* (Rigaux et al., 2000), a single path down the tree is followed, and fewer nodes are visited than with the R-tree. The gain for *window query* is less obviously assessed. Object duplication not only increases the tree size, but potentially leads to expensive post-processing of the result (sorting for duplication removal).

4. The xBR⁺-tree

The xBR⁺-tree (Roumelis et al., 2015) (an extension of the xBR-tree (Vassilakopoulos and Manolopoulos, 2000; Roumelis et al., 2011b)) is a hierarchical, pointer-based, disk-resident index structure, built utilizing a regular decomposition of space (space-driven access method), suitable for indexing multidimensional points. For 2d the hierarchical decomposition of space in the xBR⁺-tree is that of Quadtrees (the space is recursively subdivided in 4 equal subquadrants) and the space indexed is a *square*. The nodes of the tree are disk pages of two kinds: *leaves*, which store the actual multidimensional data themselves and *internal nodes*, which provide a multiway indexing mechanism.

4.1. Internal nodes

Internal node entries in xBR⁺-trees contain entries of the form (*Shape, qside, DBR, Pointer*). Each entry corresponds to a child-node pointed by *Pointer*. The region of this child-node is related to a subquadrant of the original space. *Shape* is a flag that determines if the region of the child-node is a complete or non-complete square (the area remaining, after one or more splits; explained later in this subsection). This field is heavily used in queries. *DBR* (Data Bounding Rectangle) stores the coordinates of the rectangular subregion of the child-node region that contains point data (at least two points must reside on the sides of the *DBR*), while *qside* (not stored in xBR-tree internal node entries) is the side length of the subquadrant of the original space that corresponds to the child-node.

The subquadrant of the original space related to the child-node is expressed by an *Address*. This *Address* (which has a variable size) is not explicitly stored in the xBR⁺-tree (unlike the xBR-tree), although it is uniquely determined and can be easily calculated using *qside* and *DBR* (in fact, the coordinates of the subquadrant expressed by *Address* are calculated by query processing algorithms using *qside* and *DBR*). Each *Address* represents a subquadrant which has been produced by Quadtree-like hierarchical subdivision of the current space (of the subquadrant of the original space related to the current node). 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 2d space, we use two directional bits, one for each dimension. The lower bit represents the subdivision on the horizontal (*X*-axis) dimension, while the higher bit represents the subdivision on the vertical (*Y*-axis) dimension (Vassilakopoulos and Manolopoulos, 2000; Roumelis et al., 2011b). It is easy to extend this representation to three or more dimensions by using a number of directional bits equal to the number of dimensions. 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.

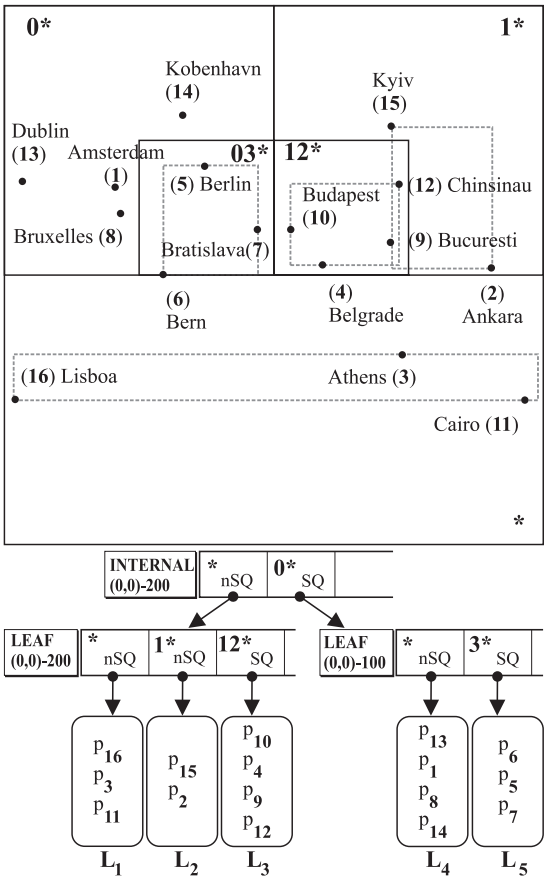

Fig. 3. The same collection of points, the corresponding grouping to xBR$^+$-tree nodes and the xBR$^+$-tree structure.

However, the actual region of the child-node is, in general, the subquadrant of its *Address* minus a number of smaller subquadrants, 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 Fig. 3 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 child is the NW quadrant of the original space. The region of the left child is the whole space minus the region of the NW quadrant, a non-complete square. 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 this child refers to the remaining space. 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 Fig. 3, the *Address* 3* is the SE subquadrant of the NW subquadrant of whole space (absolute *Address* 03*). 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.

Note that all the fields of an xBR$^+$-tree internal node entry have a fixed size. By avoiding storing the variable-sized field *Address* (unlike the xBR-tree), processing of internal nodes is simplified, since their capacity is fixed. Moreover, the use of the *DBR* field (not stored in xBR-tree internal node entries) makes processing of spatial queries more efficient, since it signifies the subregion 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.

4.2. Leaf nodes

External nodes (leaves) of the xBR$^+$-tree simply contain the data elements and have a predetermined capacity *C*. When *C* is exceeded, due to an insertion in a leaf, the region of this leaf is partitioned in two subregions. The one (new) of these subregions is a subquadrant of the region of the leaf which is created by partitioning the region of the leaf according to hierarchical (Quadtree like) decomposition, as many times as needed so that the most populated subquadrant (that corresponds to this new subregion) has a cardinality that is smaller than or equal to *C*. The other one (old) of these subregions is the region of the leaf minus the new subregion. In Roumelis et al. (2015), the criterion of choosing the new subregion was the cardinality of this subregion to be smaller or equal to xC, where $0.5 < x < 1$, however the criterion we use in this paper was more effective and simple. Note also that in the xBR$^+$-tree, data elements are stored in leaves in *X*-order (the elements are sorted in ascending order of their *X*-axis coordinate). This order 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.

4.3. Splitting process of nodes

When an internal node of the xBR$^+$-tree 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.

4.3.1. Splitting of internal nodes

The split in the xBR$^+$-tree 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 (Vassilakopoulos and Manolopoulos, 2000). 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 Vassilakopoulos and Manolopoulos (2000), or entries (a simpler and equally effective criterion, according to experimentation).

4.3.2. Splitting of leaves

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. More details appear in Vassilakopoulos and Manolopoulos (2000).

5. Spatial query processing

In this section, we outline algorithms for processing *PLQs, WQs, DRQs, KNNQs, CKNNQs, KCPQs* and *εDJQs* on the R-tree family (query processing in R*-trees, R$^+$-trees and R-trees, in general, is similar) and xBR$^+$-trees. In general terms, the definitions of these spatial queries are as follows:

- Given an index I_P and a query point q, the **PLQ** returns true if q belongs to I_P, false otherwise.
- Given an index I_P and a query rectangle r, the result of the **WQ** is the set of all points in I_P that are completely inside r.
- Given an index I_P, a query point q and a distance threshold $\varepsilon \geq 0$, the **DRQ** returns all points of I_P, that are within the specified distance ε from q (according to a distance function).

- Given an index I_P, a query point q, and a value $K > 0$, the ***K*NNQ** returns K points of I_P which are closest to q based on a distance function.
- Given an index I_P, a query point q, a value $K > 0$ and a distance threshold $\varepsilon \geq 0$, the ***CK*NNQ** returns K closest points of I_P which are within the distance ε from q.
- Given two indexes I_P and I_Q, and an integer value $K > 0$, the ***K*CPQ** (Corral et al., 2000; 2004) returns a set of K different pairs of points $(p_i, p_j) \in I_P \times I_Q$, such that $p_i \in I_P$, $p_i \in I_Q$, with the K smallest distances between all possible pairs of points that can be formed by choosing one point of I_P and one point of I_Q, based on a distance function.
- Given two indexes I_P and I_Q, and a real value $\varepsilon \geq 0$, ***ε*DJQ** (Corral and Almendros-Jiménez, 2007) returns all the possible different pairs of points $(p_i, p_j) \in I_P \times I_Q$ that can be formed by choosing one point $p_i \in I_P$ and one point $p_j \in I_P$, having a distance smaller than or equal to ε of each other, based on a distance function.

To answer the aforementioned spatial queries using members of the R-tree family, or xBR$^+$-trees a two-step procedure is followed (Brinkhoff et al., 1993). *Filter step*: the collection of all spatial objects whose MBRs/DBRs satisfy the given spatial query is found. These spatial objects constitute the candidate set. *Refinement step*: the actual exact geometry of each member of the candidate set is examined to eliminate false hits and find the final answer to the spatial query. In the following, we will describe in more detail the query processing techniques that have developed for each spatial query type. Since the *Refinement step* is orthogonal to the *Filtering step*, the developed techniques have mainly focused on the latter.

5.1. Algorithmic techniques used

All the single dataset queries above can be processed in a top-down manner beginning from the root of the tree. There are two, well known, basic techniques that can be applied.

The first one is processing the nodes of the tree in Depth-First (DF) mode: By examining the relation of an entry of the current internal node to the query object, point or area, we decide on recursively applying the same procedure on the child node pointed by this entry. When this recursive call returns, another entry of this internal node may be examined, depending on the query being processed and the result calculated so far. When a recursive call reaches a leaf node, the *Refinement step* is applied.

The second one is processing the nodes of the tree in Best-First (BF) mode: By examining the relation of each entry of the current internal node to the query object, point or area, we decide about inserting this entry in a global priority queue, where there may already exist entries inserted during earlier stages of the algorithm. After all entries of the current node have been examined, the entry with top priority is extracted from the queue and processing continues with the node pointed by this entry.

We applied four versions of DF algorithms.

- The first one, named Normal Depth First (N-DF) algorithm, is the simplest of all. The query object is tested first against each entry of the current node, in the order that the entries are stored. The criterion for such a test depends on the query being processed and the result calculated so far and its result is boolean (true / false). If the result for the entry tested is true, then the algorithm is applied recursively on the child node pointed by this entry.
- The second one is named Depth First (DF) algorithm. For each entry of the current node (in the order that the entries are stored), the minimum distance, *mindist*, between the query object and the region of the entry is calculated. If the (non-boolean) value of this metric for the entry examined satisfies the criterion corresponding to the query being processed and the result calculated so far, then the algorithm is applied recursively on the node pointed by this entry.
- The third one, named Sorted Depth First (S-DF) algorithm, is a fairly used and efficient DF technique. There is an initial step that must be implemented when an internal node is visited, so as to select the entry of this node that best satisfies the criterion corresponding to the query being processed and the result calculated so far. In this step, for all entries of the current node, *mindist(q, M)* values are calculated, inserted in an array and sorted. Then the algorithm is applied recursively on the node pointed by the entry corresponding to the lowest *mindist* value. When this recursive call returns, recursion is possibly (depending on the query being processed and the result calculated so far) applied on the entry with the next *mindist* value.
- The fourth one, named Heap Depth First (H-DF) algorithm, is a new technique that utilizes one local (to the current node) minimum Heap (*minHeap*) prioritized by the *mindist* metric. The *minHeap* replaces the sorted array of S-DF and this is expected to speed up the selection process of the next best entry for applying recursion. In fact, the fewer the entries of the current node that will be eventually used for recursive calls, the more the algorithm will accelerate (since extracting from *minHeap* part and not all of its entries corresponds to a partial application of HeapSort, in contrast to always completely sorting the respective array of S-DF).

We also applied one BF algorithm.

- In the following, this is called BF algorithm and it is iterative. It keeps a global (to the whole tree) minimum binary heap, *minHeap*, holding (part of) the entries of the nodes visited so far, prioritized by their *mindist* to the query object. Initially, *minHeap* contains the tree root. Iteratively, the entry at the root of *minHeap* is removed from the heap and the node pointed by this entry is visited; its entries are potentially added to the heap, according to the relation of *mindist* of each entry to the criterion of the query being processed and the result calculated so far. The algorithm continues by visiting the node pointed by the entry with the minimum *mindist* in *minHeap* until the heap becomes empty or the *mindist* value of the entry located in the root of the heap does not satisfy the criterion corresponding to the query being processed and the result calculated so far. When the algorithm reaches a leaf node, the *Refinement step* is applied.

All the dual dataset queries above can be processed in a top-down manner by synchronous tree traversals, beginning from the roots of the two trees. Again, the basic ideas of processing the nodes of the trees in DF and BF mode are utilized.

We applied three versions of DF algorithms for dual dataset queries. N-DF cannot be applied, due to its boolean criterion. We did not apply a version analogous to DF, because the number of combinations that should be examined is large when the entries of two nodes (one from each tree representing a different dataset) are combined, unless a technique that reduces the number these combinations is applied. Thus, we applied versions analogous to S-DF and H-DF, only.

- The first one is named Sorted Depth First for 2 datasets (S-DF-2) algorithm. We start at the roots of the two trees (current pair of nodes). For each pair of entries formed by combining the entries of the current pair of nodes, the minimum distances, *mindist* values, between the regions of the elements of the pair are calculated (these are distances between rectangular regions), inserted in an array and sorted. Then the order of this array is used for recursive application of the algorithm. If

mindist of the next array entry (a pair of nodes) satisfies the criterion corresponding to the query being processed and the result calculated so far, then the algorithm is applied recursively on the nodes pointed by the elements of this pair. In case a recursive call reaches a pair of nodes where one of its elements is a leaf, then the pairs of entries are formed by the region of this leaf and the entries of the node pointed by the other element of the pair (which is an internal node). In case a recursive call reaches a pair of nodes where both of its elements are leafs, the *Refinement step* is applied.

- The second one, named Heap Depth First for 2 datasets (H-DF-2) algorithm, is a new technique that utilizes one local (to the current pair of nodes) minimum Heap (*minHeap*) prioritized by the *mindist* metric that replaces the sorted array of S-DF-2. For the reasons described above, this algorithm is expected to speed up the selection process of the next best entry for applying recursion.
- The third one, is named Classic Plane Sweep Depth First for 2 datasets (C-DF-2) algorithm. In this algorithm, when a pair of nodes is visited, for each node of this pair, the starting coordinate of one of the axes, w.l.o.g. let's assume this is *x*-axis, of the rectangular regions of this node entries are sorted and Classic Place Sweep (Roumelis et al., 2016) is applied between the two sorted coordinate sequences. If *x*-distance of the pair of entries under examination is smaller than the current threshold corresponding to the query being processed and the result calculated so far, then the actual *mindist* is calculated for this pair of entries and, if the calculated value satisfies the criterion corresponding to the query being processed and the result calculated so far, the algorithm is applied recursively on the nodes pointed by the elements of this pair. Unlike S-DF-2 and H-DF-2, this algorithm avoids unnecessary calculations of *mindist* values. Note that in C-DF-2, when a recursive call reaches a pair of nodes where one of its elements is a leaf, plane sweep is not applied (plane sweep makes sense when two sets of rectangular regions are combined), but the region of this leaf is combined with all the entries of the other node.

We also applied one BF algorithm for dual dataset queries.

- This is called Classic Plane Sweep Best First for 2 datasets (C-BF-2) algorithm and also utilizes Classic Place Sweep (Roumelis et al., 2016). This algorithm is iterative. It keeps a global (to the whole pair of trees) minimum binary heap, *minHeap*, holding (part of) the pairs of entries of the pairs of nodes visited so far, prioritized by their *mindist*. Initially, *minHeap* contains the two tree roots. Iteratively, the entry at the root of *minHeap* is removed from the heap and the pair of nodes pointed by this entry is visited. For the pairs of entries formed from this pair of nodes plane sweep is applied, like in C-DF-2 and, each pair of entries that satisfies the criterion corresponding to the query being processed and the result calculated so far is inserted in *minHeap*. The algorithm continues by visiting the pair of nodes pointed by the pair of entries with the minimum *mindist* in *minHeap* until the heap becomes empty or the *mindist* value of the pair of entries located in the root of the heap does not satisfy the criterion corresponding to the query being processed and the result calculated so far. In case the algorithm visits a pair of nodes where one of its elements is a leaf, then the pairs of entries are formed by the region of leaf and the entries of the node pointed by the other element of the pair (which is an internal node) and plane sweep is not applied, but the region of this leaf is combined with all the entries of the other node. In case the algorithm visits a pair of nodes where both of its elements are leafs, the *Refinement step* is applied.

5.2. Point location and window queries

PLQs and *WQs* can be processed using N-DF algorithm on both the R-tree and xBR$^+$-tree families. The query object in the case of *PLQs* is the query point and the testing criterion is whether there is overlapping between the query point and the *MBR*/*DBR* of the current entry of the R-tree/xBR$^+$-tree. The query object in the case of *WQs* is the query window (rectangle) and the testing criterion is whether there is intersection between the query window and the *MBR*/*DBR* of the current entry. Since the criterion can only get one of two possible values TRUE/FALSE, there is no way or reason the values of the criterion to be compared between entries. When the node pointed by the *Addr* is a leaf then *Refinement step* is applied. For *PLQs*, the query point is searched between the points of the leaf and if it is found the result is returned in the calling procedure in order the searching process to be terminated. For *WQs*, the set of points of the current leaf within the query window are found and this set of points is returned. The searching process will be terminated when all entries of the root node have been tested.

Especially for the xBR$^+$-tree, as noted in Section 4.1, 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 first a subregion is examined before any enclosing region of this subregion, and in this way, multiple examinations of overlapping regions are avoided). So the last entry of the current internal node is examined first. Moreover, for the xBR$^+$-tree, in *WQs* a termination condition can be applied and the searching process can be terminated before all entries of the current node have been tested: whenever the query window is contained within the *REG* of the current entry of the node processing stops. This is due to no overlapping between regions of the nodes.

5.3. Distance range queries

DRQs can be performed with all variants of DF and BF algorithms that were described above in Section 5.1 on both the R-tree and xBR$^+$-tree families. The query object is a circle centered on the query point with radius a given value ε. Since in N-DF algorithm the testing criterion is whether there is intersection between the query circle and the *MBR*/*DBR* of the current entry, in order to simplify processing, the query circle is replaced by its MBR in the *Filter step*, while in the *Refinement step* the points inside the actual query circle are selected. Especially for the xBR$^+$-tree, the same termination condition noted in Section 5.2 can be applied in the N-DF algorithm.

For the other four algorithms (DF, H-DF, S-DF and BF) the query object is the circle described above and *mindist* is the distance between the center of the query circle and the *MBR*/*DBR* of the current entry. The testing criterion is whether this *mindist* value is smaller than or equal to ε. The special termination condition of the xBR$^+$-tree for the DF algorithm can be applied just like the N-DF one, while for the other three algorithms (S-DF, H-DF and BF) it must be partially changed, since the examination of the entries is not in the reverse order in which they are saved in the node. Thus, if the query circle is contained in the *REG* and the region of the entry is a complete square then the termination condition is applied.

5.4. Nearest neighbor queries

Based on the branch-and-bound paradigm, the *K* Nearest Neighbor Query algorithms use several metrics to prune the search space (Roussopoulos et al., 1995). The most important metric is *mindist*, the minimum distance between the query object and the region of the entry under examination. Another metric, *minmaxdist*, refers to

the minimum distance from the query object within which a point in the region of the entry under examination is guaranteed to be found. Finally, *maxdist* is the maximum distance between the query object and any point in the region of the entry under examination.

The first Nearest Neighbor Query (*NNQ*) algorithm for R-trees, proposed in Roussopoulos et al. (1995), traverses the tree recursively in a DF manner. Starting from the root, all entries are sorted according to their *mindist* from the query object, and the entry with the smallest *mindist* is visited first. The process is repeated recursively until the leaf level is reached, where a potential NN is found. During backtracking to the upper levels, the algorithm only visits entries whose *mindist* is smaller than or equal to the distance of the NN found so far. This algorithm was enhanced in Cheung and Fu (1998), proving that any node can be pruned by using *minmaxdist* (Roussopoulos et al., 1995) distance function. A BF algorithm for *NNQ* was proposed in Hjaltason and Samet (1995) for Quadtrees and in (Hjaltason and Samet, 1999) for R-trees. BF keeps a minimum binary heap, *minHeap*, with the entries of the nodes visited so far, prioritized according to their *mindist*. Initially, *minHeap* contains the entries of the tree root. When the root of *minHeap* is chosen for processing, it is removed from the heap and the entries of its tree node are added to the heap. The algorithm continues visiting the entry with the minimum *mindist* in *minHeap*, until the heap becomes empty or the *mindist* value of the node entry located in the root of heap is larger than the distance value of the nearest neighbor that has been found so far (i.e. the pruning distance). BF is I/O optimal because it only visits the nodes necessary for obtaining the NN. The generalization to find the *K* Nearest Neighbor (*K*NN) is straightforward. An additional data structure is just needed, a maximum binary heap, *maxKHeap* (prioritized by the distance from the query point), holding the *K* nearest points encountered so far.

It is obvious that the four algorithms (DF, S-DF, H-DF and BF) described in Section 5.1 can be adapted to *KNNQs* on both the R-tree and xBR$^+$-tree families. The query object is the circle centered at the query point and having radius equal to the key of the root of the full *maxKHeap*; otherwise (not full *maxKHeap*) this circle covers the whole space. The testing criterion (*Filter step*) is whether there is an intersection between the query circle and the *MBR*/*DBR* of the current entry; in the *Refinement step* the points inside the actual query circle are selected.

Especially for the xBR$^+$-tree, the same termination condition noted in Section 5.3 can be applied in the algorithms for *KNNQs*; when the region of the current entry is square and contains the query circle then the process is terminated. More details about this algorithm appear in Roumelis et al. (2011b); 2011a).

The *CKNNQ* is a combination of the *KNNQ* and *DRQ*; for this uery, we can adapt the DF, S-DF, H-DF or BF algorithms for *NNQ* on both the R-tree and xBR$^+$-tree families. The query object is the circle with center the query point and radius the given maximum ε value for the case of not full *maxKHeap*, otherwise the radius is the key of the root of the full *maxKHeap*. The testing criterion (*Filter step*) is whether there is intersection between the query circle and the *MBR*/*DBR* of the current entry in the *Filter step*; in the *Refinement step* the points inside the actual query circle are selected. Especially for the xBR$^+$-tree, the same termination condition can be applied as in the *NNQ* algorithms.

5.5. Distance join queries

Be reminded that the *KCPQ* asks for the *K* closest pairs of spatial objects in the Cartesian Product of two datasets. If both datasets are indexed by R-trees, the concept of synchronous tree traversal and DF or BF traversal order can be combined for query processing (Hjaltason and Samet, 1998; Corral et al., 2000; Shin et al., 2003; Corral et al., 2004). Details on such DF and BF algorithms on two R*-trees, from the non-incremental point of view, using several optimization techniques (i.e. plane-sweep, distance functions like *minmaxdist* and *maxdist*) appear in Corral et al. (2004). In the following, we outline the distance join algorithms we applied on all the three trees.

- For *KCPQs*, we applied all the four algorithms S-DF-2, H-DF-2, C-DF-2 and C-BF-2 described in Section 5.1. The testing criterion is based on the distance threshold which is, either equal to the key of the root of the *maxKHeap*, in case of a full *maxKHeap*, or to an infinite length, in case of a non-full *maxKHeap*. The testing criterion is whether the distance of the pair objects (*MBR*/*DBR*) under examination is smaller than the distance threshold. In the *Refinement step* (when the algorithm visits a pair of leaves), Classic Plane Sweep is applied between the points of the two leaves. If a pair of points consists of points with a distance smaller than the distance threshold, this pair is inserted in *maxKHeap*.
- For the ε*DJQ* ($\varepsilon \geq 0$), the above DF or BF algorithms for *KCPQ* (for all trees) are adapted in a straightforward way. There is no *maxKHeap*, or limit on the cardinality of the result and the distance threshold always equals ε. Starting from the root nodes, tree nodes are traversed down to the leaves, depending on the result of whether *mindist* of the pair of entries under examination is less than or equal to ε. When the algorithm reaches a pair of leaves, Classic Plane Sweep is applied between the points of the two leaves. All the pairs of points with a distance smaller than or equal to ε are added to the answer set.

These algorithms (except for H-DF-2, which is new) have been proposed in the past for the R-tree family. However, algorithms for the *KCPQ* and the ε*DJQ* have not been presented for the xBR$^+$-tree before. In this work, we adapted the existing R-tree algorithms and applied the H-DF-2 technique on the specific structure of xBR$^+$-tree.

6. Experimentation

We designed and run a large set of experiments to compare xBR$^+$-trees with respect to R-tree variants (R*-tree and R$^+$-tree).

6.1. Experimental settings

We used 6 real spatial datasets of North America, representing cultural landmarks (NAclN with 9203 points) and populated places (NAppN with 24,491 points), roads (NArdN with 569,082 line-segments) and rail-roads (NArrN with 191,558 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| = 569,082 points, |NArrN| = 191,558 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| = 1,138,164 points, |NArrND| = 383,116 points). The data of these 6 files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 125,000, 250,000, 500,000 and 1,000,000 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. We have also used two large real spatial datasets (retrieved from http://spatialhadoop.cs.umn.edu/datasets.html (Eldawy and Mokbel, 2015)) to justify the use of spatial query algorithms on disk-resident data instead of using them in-memory. They represent water resources of North America (Water) consisting of 5,836,360 line-segments and parks or green areas of all world (Park) consisting of 11,504,035 polygons. To create sets of points, we have transformed the MBRs of line-segments

Table 1
Tree construction characteristics.

Dataset	Num Elem ($\times 10^3$)	Node size (KB)	Tree height			Tree size (MBytes)			Creation time (s)		
			xBR$^+$	R*	R$^+$	xBR$^+$	R*	R$^+$	xBR$^+$	R*	R$^+$
NAclN	9.203	1	4	4	4	0.615	0.669	0.652	0.0511	0.3119	0.1252
NAppN	24.491	1	4	4	4	1.600	0.820	1.718	0.2286	1.0239	0.2663
NArrN	191.558	2	4	4	4	11.61	13.47	12.39	2.5324	15.385	2.8480
NArrND	383.180	4	4	3	4	22.78	26.82	24.03	6.5064	86.771	8.0914
NArdN	569.082	8	3	3	3	34.73	40.67	34.97	12.112	461.78	19.558
NArdND	1138.19	16	3	3	3	69.31	82.10	67.95	39.606	3450.7	66.697
125KCN	125.000	2	4	4	4	7.578	7.984	7.242	1.0643	8.6246	1.5947
250KCN	250.000	4	3	3	3	15.09	15.90	14.23	2.9603	46.682	4.5291
500KCN	500.000	8	3	3	3	30.02	31.83	28.06	8.5643	339.39	15.969
1000KCN	1,000.00	16	3	3	3	59.33	63.49	56.19	28.882	2360.6	60.635
Water	5,836.36	2	5	5	11	359.2	438.1	395.4	114.97	584.23	286.98
Water	5,836.36	4	4	4	5	352.9	443.6	382.3	139.92	1638.6	262.23
Park	11,504.0	8	4	4	4	684.0	839.7	731.6	402.91	9460.3	947.12
Park	11,504.0	16	3	3	3	682.7	855.5	719.0	565.42	37,174	1240

from *Water* into points by taking the center of each MBR and we have considered the centroid of polygons from Park.

The experiments were run on a Ubuntu Linux v. 14.04 machine with Intel core duo 2x2.8 GHz processor, 4GB of RAM and a Seagate Barracuda 3TB SATA 3 hard disk, using the GNU C/C++ compiler (gcc).

For page (node) sizes of 1KB, 2KB, 4KB, 8KB and 16KB we run experiments for tree building, counting tree characteristics and creation time and experiments for all spatial queries studied here (*PLQ, WQ, DRQ, KNNQ, CKNNQ, KCPQ* and ε*DJQ*), counting disk read accesses (I/O) and total execution time (I/O and CPU).

6.2. Experiments for comparing index structures

In these experiments, we built the xBR$^+$-tree, the R*-tree and the R$^+$-tree. We constructed each tree, using LRU-buffer[3] of 1024 pages. For each dataset, the insertion order of the data was the same for all trees. In Table 1, construction characteristics of the three trees, for a representative set of dataset and node size combinations (for the sake of presentation length), are depicted.

Regarding tree heights, studying the complete set of construction characteristics of the three trees (for all dataset and node size combinations), we conclude that:

- The xBR$^+$-tree and R*-tree have similar tree heights.
- The R$^+$-tree for the large real spatial datasets and the smaller node sizes (1KB and 2KB) is significantly higher.

This is due to the fact that the R$^+$-tree, to avoid overlapping, in many cases, splits internal nodes and several of their descends at subsequent levels, creating nodes with limited occupancy. For a smaller node size, an internal node is more likely to be split unevenly and the new node with the smaller occupancy may not increase significantly its occupancy in the future, if there are not enough new data within its region. This shows the sensitivity of the R$^+$-tree to the order of insertion of the data.

Regarding tree sizes, the three trees have similar sizes, since the largest part of each tree consists of leaves and the leaves exhibit similar occupancy in all trees (average leaf occupancy of the xBR$^+$-tree, the R*-tree and the R$^+$-tree is 65.14%, 68.24% and 65.14%, respectively). In conclusion:

- For real datasets the xBR$^+$-tree needs less space in disk (i.e. it is more compact).
- For synthetic datasets the R$^+$-tree has the smallest disk size.

[3] The improvement of the creation times of the xBR$^+$-tree in relation to the respective creation times in Roumelis et al. (2015) is due to the use of the LRU-buffer.

Regarding creation times:

- The xBR$^+$-tree is always the fastest.

This is due to the regular way that the xBR$^+$-tree divides the space. Moreover, node splitting follows a single path, starting from the leaf level and ending, on the worst case, at the root level. On the contrary, in the R$^+$-tree splits may be propagated to parent, as well as, to children nodes (Sellis et al., 1987).

- The R$^+$-tree is always the slowest.

This is due to forced reinsertion and the multiple paths while searching for the appropriate leaf that will host the new point (Beckmann et al., 1990).

6.3. Creation of input for queries on single datasets

We split the whole space into 2^4, 2^6, $\cdots$, 2^{16} equal rectangular windows, in a row-order mapping manner. These windows were used as query windows for *WQs*. The centroids of these windows were used as query points *PLQs, K-NNQs* and *CK-NNQs*. The incircles of these windows were used as query ranges for *DRQs* (the centroid of each of these windows was used as the center of a query range and the extend of this range, ε, was equal to the half of the side length of this window). For *K-NNQs* and *CK-NNQs*, we used the set of *K* values {1, 10, 100, 1000}.

6.4. Experiments for non distance-based queries on single datasets (PLQs *and* WQs)

As the number of experiments performed was huge, we show only representative results, since they were analogous for each query category. For *PLQs* we executed two sets of experiments using the N-DF algorithm. In the first set we used as query points the original datasets and in the second one we used as query points the centroids of the query windows. Indicative results for the *Water* dataset are shown in Fig. 4a (I/O) and b (execution time) and for the 1000*KCN* dataset are shown in Fig. 4c (I/O) and d (execution time).

These figures show that the results are different for the two cases of experiments. For the case of the query shown in Fig. 4a and b (*Water*), when searching for an existing point into the spatial dataset, the number of disk read accesses needed by the R$^+$-tree and the xBR$^+$-tree is equal to the tree height. On the other hand, the number of disk read accesses for the R*-tree is a little larger than the height of the tree. The execution time of R*-tree is smaller than the one of the R$^+$-tree and a little larger than the one of the xBR$^+$-tree.

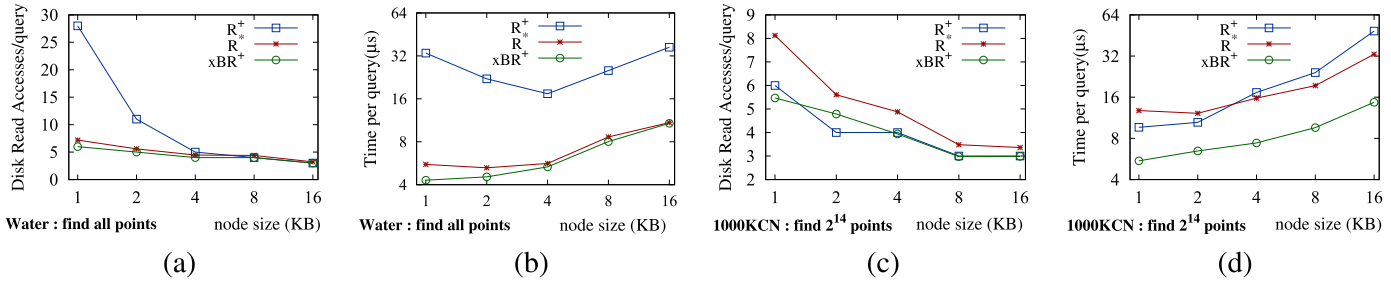

Fig. 4. *PLQ*: disk read accesses (a) and exec. time (b) vs. node size (*Water*) with query points all dataset points and disk read accesses (c) and exec. time (d) vs. node size (1000*KCN*) with query points the centroids of the query windows.

Studying the complete set of results of *PLQs* using as query points the original datasets, we find out that the same situation appears. Regarding I/O, we conclude that:

- For both the xBR$^+$-tree and the R$^+$-tree, the number of disk read accesses is equal to the height of the tree for every query point, if this point exists in the dataset, because of the single path that has to be followed until this point is found.
- For the R*-tree, the number of disk read accesses is a little larger than the height of the tree because of the multiple paths that are possibly needed to be followed until the point is found.

Summarizing the results for the execution time:

- The xBR$^+$-tree was faster than the R$^+$-tree in all cases (60/60) and faster than the R*-tree in most cases (56/60).
- The R*-tree was faster than the R$^+$-tree in all cases, for all datasets and node sizes.
- Especially, for the node size of 16KB, the xBR$^+$-tree needed fewer disk read accesses for all datasets, with an average relative difference of 5.75% to the R*-tree.
- Moreover, it was faster for all datasets (12/12) with an average relative difference of 70.9% to the R*-tree.

For the case of the experiment shown in Fig. 4c and d (1000*KCN* dataset), the number of disk read accesses needed by the R$^+$-tree when searching for a point non-existing in the spatial dataset is equal to the tree height. Note that for this dataset, the tree height of the R$^+$-tree, the R*-tree and the xBR$^+$-tree equals 6, 5, 6 for 1KB nodes, 4, 4, 5 for 2KB nodes, 4, 4, 4 for 4KB nodes and 3, 3, 3 for 8KB and 16KB nodes, respectively. In the case of the xBR$^+$-tree, the number of disk read accesses needed is less than the tree height for most query points. In the case of R*-tree, the number of disk read accesses depends on the size of the empty space in relation to the occupied space (inside MBRs) and is larger than the tree height for all node sizes. Studying the results for the execution time, we note that there is a fairly constant difference in favor of the xBR$^+$-tree against the other two trees. This query (*PLQ*) is related to the tree height and the size of MBRs that enclose the data points. So it seems easier for the R*-tree to decide that the query point does not exist in the dataset, than for the xBR$^+$-tree. But this fact is not enough to make the R*-tree faster than the xBR$^+$-tree, since CPU processing of the tree structure is lighter for the xBR$^+$-tree.

Studying the complete set of results of *PLQs* using as query points the centroids of the query windows, we find out that for R*-trees, the number of disk read accesses is smaller than the height of the tree for all real datasets, while for synthetic datasets this number is larger than the height of the tree. For R$^+$-trees, the number of disk read accesses is always equal to the height of the tree because there is no overlapping between its leafs. For xBR$^+$-trees, the number of disk read accesses is always smaller than the tree height. Summarizing I/O results, we find out that:

- The R*-tree needed the smallest number of disk read accesses in most cases (41/60).

Table 2
WQ: disk read accesses and exec. time per query on NArdND (2^{12} query windows) vs. node size.

	Disk Read Accesses			Execution Time (μs)		
Node Size	R$^+$	R*	xBR$^+$	R$^+$	R*	xBR$^+$
1	144.4	24.60	**21.81**	241.6	41.62	**19.72**
2	69.05	13.82	**11.71**	182.7	33.40	**14.39**
4	36.33	9.049	**7.039**	165.0	34.19	**13.62**
8	20.75	5.714	**4.113**	173.0	36.38	**14.30**
16	14.71	4.267	**3.139**	239.5	47.98	**18.31**

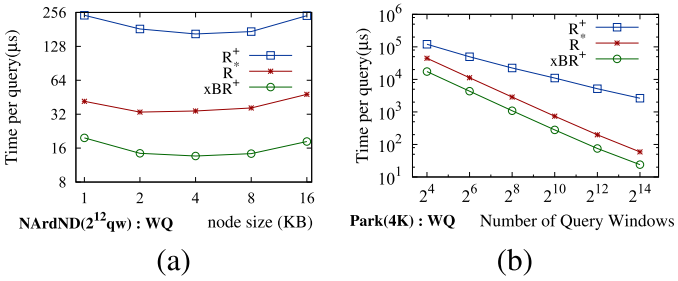

Fig. 5. *WQ*: exec. time vs. (a) node size (NArdND, 2^{12}, query windows) and (b) number of query windows (Park with node size=4K).

- The xBR$^+$-tree needed the smallest number of disk read accesses in 18/60 cases.
- Only in one case the R$^+$-tree needed the smallest number of disk read accesses.

The results for the execution time showed that:

- The xBR$^+$-tree is faster for all datasets and node sizes (60/60).

The *WQ* was executed using the N-DF algorithm, for all datasets (12) and all node sizes (5), searching for the points residing inside the query windows of various sizes (6). The results of the *WQ* for the NArdND dataset, using 2^{12} windows that cover the whole data space, regarding the number of disk read accesses per query (Table 2) and the execution time vs. the node size (Table 2 and Fig. 5a) are shown as a representative case. The use of a table is preferred due to the large difference of values between R$^+$-tree and the other two trees, especially for the cases of small node sizes (1KB, 2KB). Note that, in Tables 2–7, a value in bold is the best value of its line.

In Table 2, it is shown that the xBR$^+$-tree needed fewer disk read accesses (*Acc*) than the other two trees. As the node size increases, the absolute I/O difference between the trees becomes smaller, but the relative difference $(Acc_{R^+} - Acc_{R^*})/Acc_{R^+}$ has values (0.83, 0.80, 0.75, 0.72, 0.71) that are all in favor of R*-tree, while the relative difference $(Acc_{R^*} - Acc_{xBR^+})/Acc_{R^*}$ has values (0.11, 0.15, 0.22, 0.28, 0.26) that are all in favor of xBR$^+$-tree. Note the reduction of the difference from the smallest node size (1KB) to the

Table 3
WQ: disk read accesses and exec. time per query on Park (node size = 4KB) vs. number of Query Windows.

Number Query Wins	Disk Read Accesses R+ × 10^3	R* × 10^3	xBR+ × 10^3	Execution Time R+ μs	R* μs	xBR+ μs
2^4	26.18	11.38	**11.085**	120,510	44,651	**17,375**
2^6	10.71	2.865	**2.773**	49,653	11,272	**4,338**
2^8	4.869	0.726	**0.695**	22,350	2861	**1,088**
2^{10}	2.312	0.188	**0.175**	10,997	739.0	**280.6**
2^{12}	1.122	0.051	**0.045**	5170	198.4	**75.09**
2^{14}	0.555	0.015	**0.013**	2638	59.18	**24.91**

largest size (16KB). This is due to the reduction of tree height, as the node size increases. In Table 2 and in Fig. 5a, it is shown that the xBR+-tree is the fastest and the R*-tree is faster than the R+-tree, for all node sizes. All three trees needed less total execution time (I/O and CPU) for the node size equal to 4KB, even though larger node sizes needed fewer disk read accesses.

The results of the *WQ* on the large dataset *Park* indexed by trees with node size = 4KB, for windows with variable size, regarding the number of disk read accesses (Table 3) and the execution time (Table 3 and Fig. 5b) vs the number of query windows are shown as one representative case. The use of a table is preferred due to the large difference of values between R+-tree and the other two trees. For both metrics (disk read accesses and execution time), the xBR+-tree has the best performance and the R+-tree the worst.

Studying the complete set of results (360 experiments) of *WQs*, we validate the above performance behavior. Regarding I/O:

- The number of disk read accesses per query window for the xBR+-tree was the smallest for the most experiments (323/360).
- For the R*-tree, it was smallest for the remainder of the experiments (37/360).

Regarding the execution time metric:

- The xBR+-tree had the best performance in all cases (360/360).
- The average relative difference of execution time performance between the R*-tree and the xBR+-tree is between 49.6% and 64.7%, increasing with the enlargement of the node size.
- In all trees, the execution time is reduced as the node size is increased. It is minimized for node size equal to 4KB, or 8KB. This is due to a tradeoff between I/O cost and CPU processing.

This behavior holds for the experiments of all datasets and all query windows.

6.5. Experiments for distance-based queries on single datasets (DRQs)

The *DRQ* was executed for all datasets (12) and all node sizes (5), searching for the points inside the incircles of the query windows, for various radius sizes (6 ε-values). Five Algorithms, N-DF, DF, S-DF, H-DF and BF, were tested for all (360) experiments. The number of disk read accesses is the same for the algorithms DF, S-DF, H-DF and BF because they all use the *mindist* metric and the query object is fixed. The R+-tree responded best with the algorithms using the *mindist* in all cases (360/360) in disk read accesses and faster with the DF algorithm (216/360 in execution time). The R*-tree responded best with the algorithms using the *mindist* in all cases (360/360) and faster with the BF algorithm in most cases (350/360 in execution time). The xBR+-tree responded best with the algorithms using the *mindist* in most cases (327/360) and faster with the DF algorithm (191/360 in execution time). So the performance comparison for the *DRQ* was performed among the R+-tree with the DF, the R*-tree with the BF and the xBR+-tree with the DF algorithm. The results of the *DRQ* executed on the

Table 4
DRQ: disk read accesses per query on NArrND (2^{12} query ranges) vs. node size.

Node Size	Disk Read Accesses R+ (DF)	R* (BF)	xBR+ (DF)	Execution Time (μs) R+ (DF)	R* (BF)	xBR+ (DF)
1	46.95	8.606	**8.238**	78.70	15.92	**9.288**
2	24.47	5.441	**4.769**	65.47	14.72	**7.585**
4	11.72	**3.519**	3.845	53.90	15.74	**8.587**
8	10.07	2.825	**2.410**	86.92	20.17	**9.034**
16	5.554	2.500	**2.185**	93.70	31.52	**12.43**

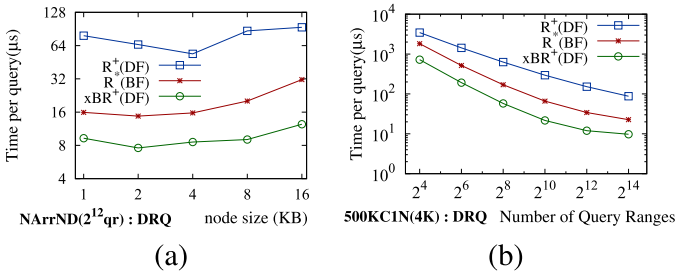

Fig. 6. *DRQ*: exec. time vs. (a) node size (NArrND, 2^{12} query ranges) and (b) number of query ranges (500KCN with node size=4K).

Table 5
DRQ: disk read accesses and exec. time per query on 500KC1N (node size = 4KB) vs. number of Query Ranges.

Number Query Points	Disk Read Accesses R+ (DF)	R* (BF)	xBR+ (DF)	Execution Time (μs) R+ (DF)	R* (BF)	xBR+ (DF)
2^4	725.5	**376.5**	413.0	3463	1822	**724.3**
2^6	303.9	**106.9**	108.5	1427	516.1	**191.6**
2^8	136.6	35.43	**31.30**	629.3	170.0	**57.80**
2^{10}	65.48	14.38	**10.48**	295.3	66.15	**21.62**
2^{12}	33.69	7.721	**5.090**	152.1	34.14	**12.00**
2^{14}	18.56	5.343	**4.187**	87.52	22.67	**9.738**

NArrND dataset for the 2^{12} ranges (with $\varepsilon \leq \frac{1}{2} \times \frac{1}{\sqrt{2^{12}}}$) scanning the data space are shown as a representative case. The number of disk read accesses per query (Table 4) and the execution time (Table 4 and Fig. 6a) vs node size, are depicted. The use of a table is preferred because of the large difference of values between R+-tree and the other two trees, especially for the cases of small node sizes (1KB, 2KB, 4KB).

In this table, it is shown that the xBR+-tree needed fewer disk read accesses (*Acc*) than the other two trees. As the node size increases, the I/O difference between the trees remains almost stable. The relative difference $(Acc_{R^+} - Acc_{R^*})/Acc_{R^+}$ has values (0.82, 0.78, 0.70, 0.72, 0.55) that are all in favor of the R*-tree, while the relative difference $(Acc_{R^*} - Acc_{xBR^+})/Acc_{R^*}$ has values (0.04, 0.12, −0.09, 0.15, 0.13) that are all (except the negative one) in favor of the xBR+-tree. Note the reduction of the difference from the smallest node size (1KB) to the largest one (16KB). This is due to the reduction of the tree height as the node size increases. In Table 4 and in Fig. 6a, it is shown that the xBR+-tree is the fastest and the R*-tree is faster than the R+-tree, for all node sizes. The R+-tree needed less total execution time (I/O and CPU) with node size equal to 4KB, while the other two trees needed less execution time with node size equal to 2KB, even though larger node sizes needed fewer disk read accesses.

The results of the *DRQ* on the synthetic dataset 500*KCN* indexed by trees with node size = 4KB, for various ε sizes, regarding the number of disk read accesses (Table 5) and the execution time (Table 5 and Fig. 6a) vs. the number of query ranges are shown. The xBR+-tree has the best performance regarding disk read accesses

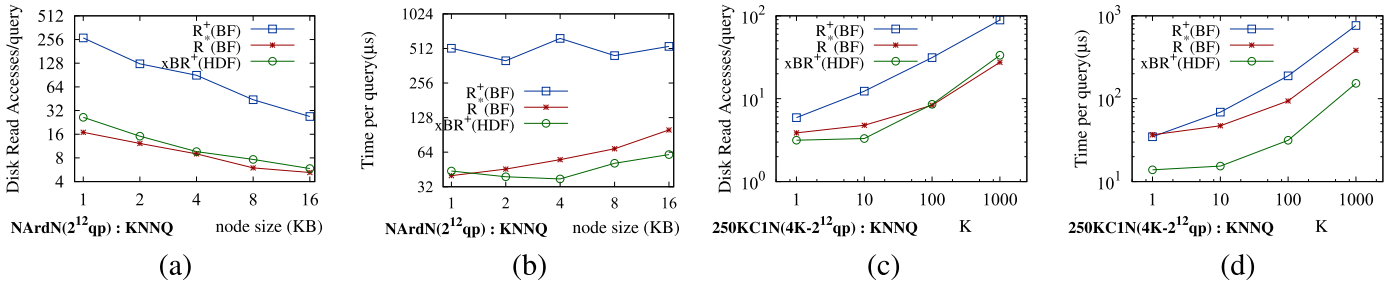

Fig. 7. *K*(=100) *NNQ*: disk read accesses (a) and exec. time (b) vs. node size (NArdN, 2^{12} query points) and disk read accesses (c) and exec. time (d) vs. *K* values of NN (250KCN with node size = 4K, 2^{12} query points).

in most of the cases (except the one for node size = 4KB) and the R+-tree has the worst, while the xBR+-tree has always the best execution time performance and the R+-tree the worst.

Studying the complete set of results (360 experiments) of *DRQs*, we validate the above performance behavior. Regarding I/O:

- The xBR+-tree had the best performance in most experiments (292/360).
- The R*-tree had the best performance in the remainder of the experiments (68/360).

Regarding the execution time metric:

- The xBR+-tree had the best performance in all cases (360/360).
- The average relative difference of execution time performance between the R*-tree and the xBR+-tree is between 49.3% and 68.4%, increasing with the enlargement of node size.

6.6. Experiments for neighboring queries on single datasets (K-NNQs and CK-NNQs)

The *KNNQ* was executed for all datasets (12) and all node sizes (5), searching for the points near the centroids of the (2^{12}) query windows, for various values of *K* (4). Four Algorithms, DF, S-DF, H-DF and BF, were tested for all (240) experiments. The R+-tree responded best with the BF algorithm (222/240 in disk read accesses and 201/240 in execution time). The R*-tree responded best with the BF algorithm (224/240 in disk read accesses and 239/240 in execution time). The xBR+-tree responded best with the BF algorithm in disk read accesses in most cases (166/240) and was faster with H-DF algorithm in 161/240 cases. We considered as most important criterion the execution time and selected the BF algorithm for both R-trees and the H-DF algorithm for the xBR+-tree to continue the performance comparison for *KNNQs*. In Fig. 7a and b, we can see the results of the *KNNQ* with *K*=100 executed on the NArdN dataset for the 2^{12} query points, distributed evenly in data space, as one representative case. The number of disk read accesses per query point and the execution time vs. the node size are shown. Because of the large difference of values between R+-tree and the other two trees, it is not easy to distinguish the differences between xBR+-tree and R*-tree. Therefore, Table 6 has been included.

In this table and in Fig. 7a, it is shown that the R*-tree needed fewer disk read accesses (*Acc*) than the other two trees. As the node size increases, the I/O difference between the trees decreases. The relative difference $(Acc_{R+} - Acc_{R*})/Acc_{R+}$ has values (0.94, 0.90, 0.90, 0.86, 0.81) that are all in favor of the R*-tree and the relative difference $(Acc_{R*} - Acc_{xBR+})/Acc_{R*}$ has values (−0.55, −0.24, −0.07, −0.28, −0.12) that are also all in favor of the R*-tree. As the node size is increased exponentially, the number disk read accesses is reduced but not with the same ratio. For the R+-tree the ratio of the numbers of disk read accesses between two consecutive node sizes varies (from 0.47 up to 0.72). For the R*-tree the ratio of the numbers of disk read accesses between two consecutive node sizes presents smaller variation (from 0.66 up to 0.87) while for the xBR+-tree this ratio presents an intermediate level of variation (from 0.58 up to 0.79). In Table 6 and in Fig. 7b, it is shown that the xBR+-tree is faster than R*-tree for node sizes larger than 1KB, and the R*-tree is faster than the R+-tree, for all node sizes. All three trees have different behavior in total execution time (I/O and CPU). For the R+-tree, the total execution time varies without any monotony. For the R*-tree the execution time has monotonous enlargement with node size. Contrary to the previous behaviors, the xBR+-tree needed less total execution time for a node size equal to 4KB, even though larger node sizes needed fewer disk read accesses.

Table 6
K(=100) *NNQ*: disk read accesses per query on NArdN (2^{12} query points) vs. node size.

Node Size	Disk Read Accesses			Execution Time (μs)		
	R+ (BF)	R* (BF)	xBR+ (H-DF)	R+ (BF)	R* (BF)	xBR+ (H-DF)
1	268.1	**16.93**	26.28	514.1	**40.00**	43.88
2	125.7	**12.23**	15.11	400.1	45.58	**39.12**
4	89.92	**9.022**	9.636	625.9	55.10	**37.41**
8	43.91	**5.979**	7.655	444.0	68.53	**51.27**
16	26.92	**5.227**	5.860	533.9	99.92	**60.99**

In Fig. 7c and d, we can see the results of the *KNNQ* on the synthetic dataset 250*KCN* indexed by trees with node size = 4KB, for 2^{12} query points, regarding the number of disk read accesses and the execution time vs. the value of *K*. The xBR+-tree has the best performance regarding disk read accesses looking for the 1 or 10 NNs, while R*-tree has best performance for the 10^2 or 10^3 NNs. The R+-tree needed the most disk read accesses for all values of *K*. The xBR+-tree has always the best execution time performance and the R+-tree has the worst.

Studying the complete set of results (240 experiments for each tree) of the *KNNQ* we validate the above performance behavior:

- The number of disk read accesses per query point was the smallest for the R*-tree in most experiments (173/240).
- It was the smallest for the xBR+-tree in the rest of the experiments (67/240).
- Regarding the execution time metric, the xBR+-tree had the best performance in most cases (209/240).
- The xBR+-tree has the minimum number of best performance cases in execution time with the smallest node size (26/48) and has the maximum number of best performance cases with the biggest node size (48/48).
- The average relative difference of execution time performance between the R*-tree and the xBR+-tree in the node size of 16KB is 49.2%.

The *CKNNQ* was executed for all datasets (12) and all node sizes (5), searching for the points inside the incircles of the (2^{12}) query

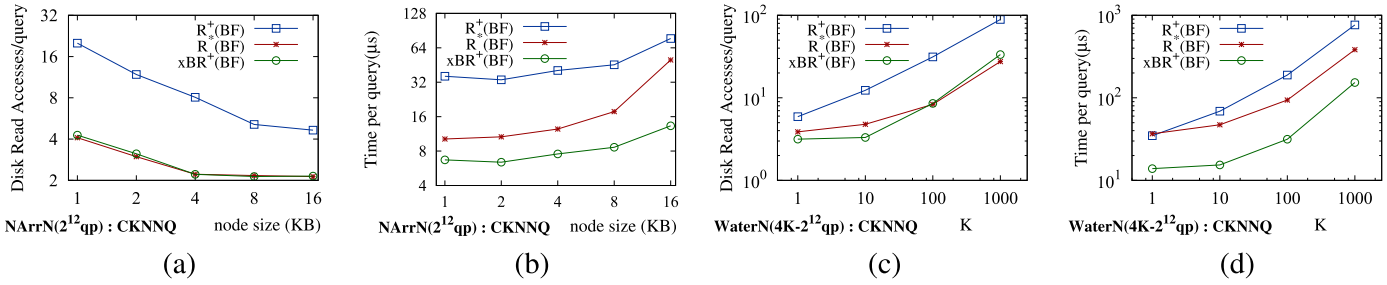

Fig. 8. *CK NNQ*: disk read accesses (a) and exec. time (b) vs. node size (NArrN, 2^{12} query points) and read disk read accesses (c) and exec. time (d) vs. *K* values of NN (*Water* with node size = 4K, 2^{12} query points).

Table 7
CK NNQ: disk read accesses per query on NArrN (2^{12} query points) vs. node size.

	Disk Read Accesses			Execution Time (μs)		
Node Size	R^+ (BF)	R^* (BF)	xBR^+ (BF)	R^+ (BF)	R^* (BF)	xBR^+ (BF)
1	20.04	**4.094**	4.273	36.06	10.21	**6.703**
2	11.83	**2.977**	3.128	33.58	10.65	**6.389**
4	8.067	2.215	**2.214**	40.53	12.47	**7.556**
8	5.113	2.169	**2.135**	45.50	17.72	**8.641**
16	4.648	**2.131**	2.147	77.13	50.07	**13.31**

windows (with $\varepsilon \leq \frac{1}{2} \times \frac{1}{\sqrt{2^{12}}}$), for various values of *K* (4). Four Algorithms, DF, S-DF, H-DF and BF, were tested for all (240) experiments. All the three structures responded best with the BF algorithm. In detail, the R^+-tree responded best in 197/240 experiments in disk read accesses and in 191/240 in execution time. The R^*-tree responded best 202/240 in disk read accesses and in 132/240 in execution time. Finally, xBR^+-tree responded best 144/240 in disk read accesses and in 190/240 in execution time. In Fig. 8a and b, we can see the results of the *CKNNQ* executed on the NArrN dataset for the 2^{12} query points, distributed evenly in data space, as one representative case. The number of disk read accesses per query point and the execution time vs. the node size are shown. Because of the large difference of values between R^+-tree and the other two trees, it is not easy to distinguish the differences between xBR^+-tree and R^*-tree. Therefore, Table 7 is depicted.

In this table and in Fig. 8a, it is shown that the R^*-tree and xBR^+-tree needed an almost equal number of disk read accesses (*Acc*). As the node size increases, the I/O difference between the trees decreases. The relative difference $(Acc_{R^+} - Acc_{R^*})/Acc_{R^+}$ has values (0.80, 0.75, 0.73, 0.58, 0.54) that are all in favor of the R^*-tree and the relative difference $(Acc_{R^*} - Acc_{xBR^+})/Acc_{R^*}$ has values (−0.04, −0.05, 0.00, 0.02, −0.01), the 3 negative ones being in favor of the R^*-tree. For the R^+-tree the ratio of the numbers of disk read accesses between two consecutive node sizes varies widely (from 0.59 up to 0.91). For the R^*-tree this ratio presents a smaller variation (from 0.73 up to 0.98) and for the xBR^+-tree it presents a similar variation (from 0.73 up to 1.01). In Table 7 and in Fig. 8b, it is shown that the xBR^+-tree is the fastest and the R^*-tree is faster than the R^+-tree, for all node sizes. All three trees have similar behavior in total execution time (I/O and CPU). The total execution time has monotonous increment with node size. In Fig. 8c and d, we can see the results of the *CKNNQ* on the large real dataset *Water* indexed by the three trees with node size = 4KB, for 2^{12} query points, regarding the number of disk read accesses and execution time vs. the value of *K*. Because of the large number of nodes in this dataset the number of disk read accesses is quite stable for all trees. It is most stable for the R^+-tree while for the other two trees varies between 1.4 and 1.8 for R^*-tree and 1.8 and 2.2 for the xBR^+-tree. The R^*-tree has the best performance regarding disk read accesses and the R^+-tree the worst, while the xBR^+-tree has always the best execution time performance and the R^+-tree has the worst.

Studying the complete set of results (240 experiments for each tree) of the *CKNNQ*, we validate the above performance behavior.

- The number of disk accesses per query point for the xBR^+-tree was the smallest in most experiments (138/240) and for the R^*-tree it was the smallest in the rest of the experiments (102/240).
- Regarding the execution time metric, the xBR^+-tree has the best performance in all the experiments (240/240).
- The average relative difference of execution time performance between the R^*-tree and the xBR^+-tree is between 39.6% and 60.5%, increasing with the enlargement of the node size.

6.7. Creation of input for queries on dual datasets

In order to evaluate the performance of the trees in spatial queries where two indexes are involved (distance join queries), we have used ten combinations between real and synthetic spatial datasets. Four combinations between real datasets of North America (i.e. NAppN × NArrN, NAppN × NArdN, NArrN × NArdN and NArrND × NArdND), four combinations between two separate instances of synthetic clustered datasets (i.e. $250KC1N \times 250KC2N$, $500KC1N \times 500KC2N$, $500KC1N \times 500KC2N$, and $1000KC1N \times 1000KC2N$) and two combinations between the largest real datasets (i.e. NArdND × Water and Water × Park) for query processing of the *KCPQ* and *εDJQ*. For *KCPQs*, the number *K* of closest pairs gets values from the set {1, 10, 10^2, 10^3, 10^4} and for *εDJQs*, the maximum distance (ε) gets values from the set $\{0, 1.25 \times 10^{-5}, 2.5 \times 10^{-5}, 5 \times 10^{-5}, 10 \times 10^{-5}\}$.

6.8. Experiments for join (dual dataset) queries (KCPQs and εDJQs)

In the experiments performed, the effect of LRU-buffer has also been studied, because a node of the one tree can be paired with several nodes of the other tree, in successive or not time points. Moreover, both trees corresponding to a combination of datasets were of equal node size.

6.8.1. Selection of buffer size and algorithms for the KCPQ

All combinations of datasets (10) and all node sizes (5), for various values of *K* (5) and with several values of LRU-buffer size (5) were used. In this series of experiments the target was to find out for which buffer size and with which algorithm among S-DF-2, H-DF-2, C-DF-2 and C-BF-2 each tree responded better. It is obvious that as the size of the LRU-buffer increases, the number of disk read accesses decreases and the related results will be omitted. So for the above target, only the execution time per query will be studied.

In Fig. 9a, we can see the results of the *KCPQ* on the combination of synthetic datasets $1000KC1N \times 1000KC2N$, both indexed by

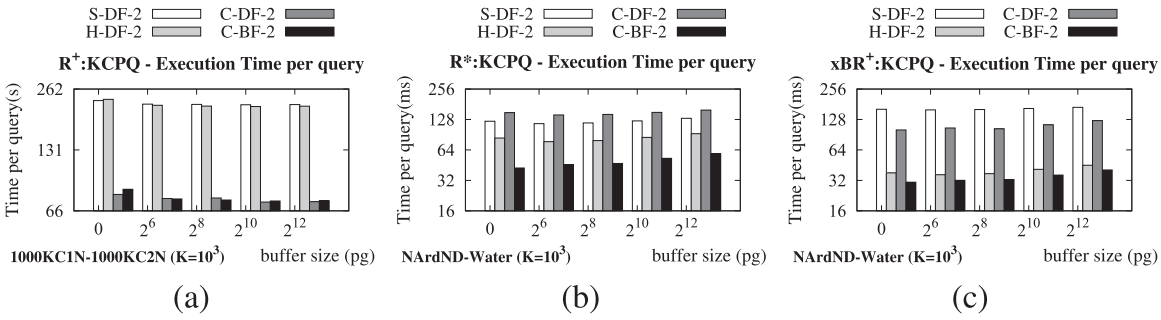

Fig. 9. *K CPQ* with (a) R⁺-tree on 1000KC1N × 1000KC2N; (b) R*-tree on NArdND × Water; (c) xBR⁺-tree on NArdND × Water: exec. time vs. buffer size per Algorithm with $K = 10^3$ and node size = 2KB.

Table 8
$K(= 10^3)$ *CPQ* with R⁺-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec time (s)	Node size (KB)	Buffer size (pages)	Algorithm
NAppN × NArrN	0.1725	1	2^8	C-BF-2
NAppN × NArdN	0.4091	1	0	C-DF-2
NArrN × NArdN	1.0496	8	0	C-BF-2
NArrND × NArdND	1.8803	8	0	C-BF-2
250KC1N × 250KC2N	1.7793	8	2^{10}	C-BF-2
500KC1N × 500KC2N	5.6271	16	2^8	C-BF-2
500KC2N × 1000KC1N	12.112	16	2^{12}	C-BF-2
1000KC1N × 1000KC2N	28.515	16	2^{12}	C-BF-2
NArdND × Water	342.64	16	2^{12}	C-BF-2

Table 9
$K(= 10^3)$ *CPQ* with R*-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec time (ms)	Node size (KB)	Buffer size (pages)	Algorithm
NAppN × NArrN	56.640	2	0	C-BF-2
NAppN × NArdN	136.46	4	0	C-BF-2
NArrN × NArdN	157.64	1	2^8	C-BF-2
NArrND × NArdND	310.96	1	2^8	C-BF-2
250KC1N × 250KC2N	206.93	4	2^6	C-BF-2
500KC1N × 500KC2N	405.03	4	2^6	C-BF-2
500KC2N × 1000KC1N	599.60	8	2^6	C-BF-2
1000KC1N × 1000KC2N	845.10	16	0	C-BF-2
NArdND × Water	30.255	1	0	C-BF-2
Water × Park	1,031.9	4	0	C-BF-2

Table 10
$K(= 10^3)$ *CPQ* with xBR⁺-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec time (ms)	Node size (KB)	Buffer size (pages)	Algorithm
NAppN × NArrN	17.302	16	0	C-DF-2
NAppN × NArdN	34.306	4	0	H-DF-2
NArrN × NArdN	57.432	8	0	C-BF-2
NArrND × NArdND	114.51	16	0	C-BF-2
250KC1N × 250KC2N	42.111	4	0	C-BF-2
500KC1N × 500KC2N	71.459	8	0	C-BF-2
500KC2N × 1000KC1N	99.164	8	0	C-BF-2
1000KC1N × 1000KC2N	124.28	8	0	C-BF-2
NArdND × Water	30.706	2	0	C-BF-2
Water × Park	473.56	4	0	C-BF-2

R⁺-trees with node size of 2KB, searching for the K=1000 closest pairs with all four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12} pages, as one representative case. It is shown that with the C-DF-2 and C-BF-2 algorithms the R⁺-tree is approximately 3 times faster than with the other algorithms for all buffer sizes. The lowest execution time value is with a buffer size of 2^{10} pages (nodes) for all algorithms and the minimum execution time value, 72,450 ms (72 s), with the C-DF-2 algorithm.

Considering the complete set of 45/50 experiments for the 5 buffer sizes (the biggest of the 10 dataset combinations, Water × Park, was not tested for all node sizes because of the big execution time values it required), we collected the minimum execution time values for each combination and these results are shown in Table 8. It is shown that there is not a single best buffer size, neither a single best node size. We conclude that for combinations between small real datasets it is better to have no buffering, while for combinations of small synthetic and larger real datasets it is better to have buffering larger than 2^8 pages (nodes). The best algorithm for R⁺-trees executing the *KCPQ* is the C-BF-2 (in 9/10 cases).

In Fig. 9b, we can see the results of the *KCPQ* on the combination of real datasets NArdND × Water, both indexed by R*-trees with node size = 2KB, searching for the K=1000 closest pairs with all four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12}, as one representative case. It is shown that with the C-BF-2 algorithm the R*-tree is from 1.6 up to 2 times faster than the best of the other tree algorithms for all buffer sizes. The smallest execution time value was achieved with buffer size of 0 pages (nodes) for all algorithms and the minimum execution time value, 42.418 ms, was achieved with the C-BF-2 algorithm.

Considering the complete set of all (50) experiments for the 10 dataset combinations and 5 buffer sizes, we collected the minimum execution time values for each combination. These results are shown in Table 9. It is shown that R*-tree responded best in half of the cases (5/10), including the combination between large real datasets, without buffering, while in the 4 combinations with synthetic datasets it responded best with 2^6 pages in LRU-buffer (in 3/4 cases). The best algorithm for R*-trees executing the *KCPQ* is the C-BF-2.

Finally, in Fig. 9c, we can see the results of the *KCPQ* on the combination of real datasets NArdND × Water, both indexed by xBR⁺-trees with node size = 2KB, searching for the K = 1000 closest pairs, with all four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12}, as one representative case. It is shown that with the C-BF-2 algorithm the xBR⁺-tree is from 3.0 up to 3.3 times faster than the best of the S-DF-2 and C-DF-2 algorithms and from 1.1 up to 1.2 faster than the H-DF-2 algorithm for all buffer sizes. The lowest execution time value was achieved with buffer size of 0 pages (nodes) for all algorithms and the minimum execution time value, 30.706 ms, was achieved with the C-BF-2 algorithm.

Considering the complete set of all (50) experiments for the 10 dataset combinations and 5 buffer sizes, we collected the minimum execution time values for each combination and these results are shown in Table 10. It is shown that xBR⁺-tree responded best in all the cases (10/10) without buffering. The best algorithm for xBR⁺-trees executing the *KCPQ* was the C-BF-2 (in 8/10 cases).

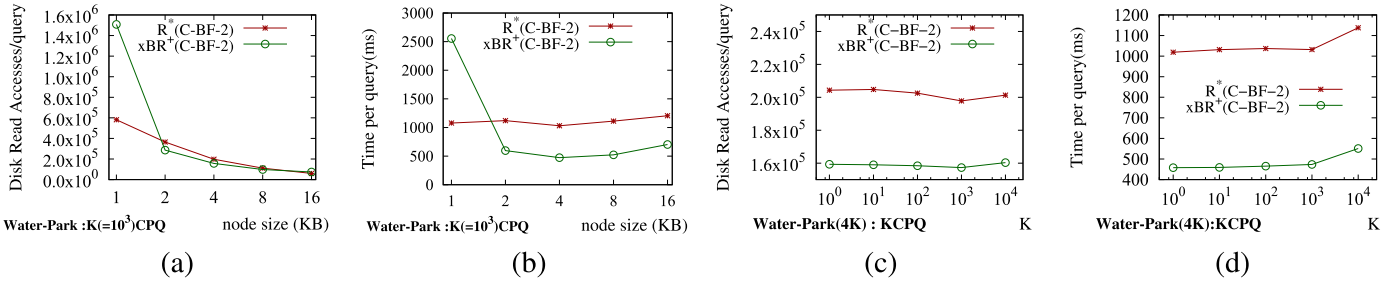

Fig. 10. *K CPQ* on large real datasets Water × Park: disk read accesses (a) and exec. time (b) vs. node size (K=10^3) and disk read accesses (c) and exec. time (d) vs. *K* values of CP (node size = 4K).

In conclusion, we note that:

- There is no meaning for a comparison between the R+-tree and the other two trees because of the very large difference in execution times observed (the R+-tree was mainly designed for *PLQs* and *WQs*).
- Based on the above results, we continue the performance comparison between the R*-tree and xBR+-tree, using the C-BF-2 algorithm for both trees without LRU-buffer (0 pages) for all node sizes, for various values of *K* (1, 10, 10^2, 10^3 and 10^4) and for both performance metrics (i.e. the number of disk read accesses and the execution time).

6.8.2. Performance study of the KCPQ

In Fig. 10a and b, we can see the results of the *KCPQ* with $K = 10^3$, executed on the combination of large real datasets Water × Park, as one representative case. The number of disk read accesses per query point and the execution time vs. the node size are shown. The xBR+-tree needed fewer disk read accesses (*Acc*) than the R*-trees having node sizes between 2KB and 8KB. As the node size increases, the ratio of the I/O difference between the two trees varies. The relative difference $(Acc_{R^*} - Acc_{xBR^+})/Acc_{R^*}$ has values (−1.59, 0.22, 0.20, 0.12, −0.23), the 3 positive one being in favor of the xBR+-tree. For the R*-tree, the ratio of the numbers of disk read accesses between two consecutive node sizes presents a small variation (from 0.5 up to 0.6) and is always decreased. For the xBR+-tree, this ratio presents a similar variation from 0.6 up to 0.8 (except the first case from 1 to 2 KB where it is 0.2). In Fig. 10b, it is shown that the xBR+-tree is faster than R*-tree for all node sizes bigger than 1KB. For both trees the execution time has a minimum value with a node size of 4KB, even though larger node sizes needed fewer disk read accesses. In Fig. 10c and d, we can see the results of the *KCPQ* on the same combination of large real datasets indexed by trees with node size of 4KB, regarding the number of disk read accesses and the execution time vs. the value of *K*. The number of disk read accesses needed by both trees remains stable although the number of *K* is increased exponentially. The xBR+-tree has the best performance regarding disk read accesses in all the cases.

Studying the complete set of results (250 experiments for each tree) for the *KCPQ*, we validate the above performance behavior.

- The number of disk accesses per query for the xBR+-tree was the smallest for most experiments (224/250).
- The xBR+-tree was faster than the R*-tree in all experiments with node sizes of 2KB and 16KB, in 46/50 cases with node size of 4KB, while, in total, it was faster in 231/250 cases.
- The average relative difference of execution time performance between the R*-tree and the xBR+-tree is between 62.7% and 71.9%.

Table 11
ε *DJQ* ($\varepsilon = 1.25 \times 10^{-5}$) with R+-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec time (ms)	Node size (KB)	Buffer size (pages)	Algorithm
NAppN × NArrN	0.146	1	2^{10}	H-DF-2
NAppN × NArdN	0.347	1	0	H-DF-2
NArrN × NArdN	0.873	1	2^{10}	H-DF-2
NArrND × NArdND	2.085	1	2^{10}	H-DF-2
250KC1N × 250KC2N	5.994	1	2^8	H-DF-2
500KC1N × 500KC2N	25.16	1	2^8	H-DF-2
500KC2N × 1000KC1N	29.98	1	2^{10}	H-DF-2
1000KC1N × 1000KC2N	55.97	1	2^8	H-DF-2
NArdND × Water	2202	4	2^{10}	C-BF-2

6.8.3. Selection of buffer size and algorithms for the εDJQ

For the ε*DJQ* the same scenario of experiments to the one of *KCPQ* was performed to find out for which buffer size and with which algorithm among S-DF-2, H-DF-2, C-DF-2 and C-BF-2 each tree responded better, regarding the execution time metric.

In Fig. 11a, we can see the results of the ε*DJQ* on the combination of synthetic datasets $1000KC1N \times 1000KC2N$, both indexed by R+-trees, with node size of 4KB, searching for the pairs with distance $\leq 1.25 \times 10^{-5}$ with all four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12} pages, as one representative case. We chose to present the same dataset combinations to the previous query (*KCPQ*) because of the similarity which exists between the two types of queries. It is shown that with the C-DF-2 algorithm the R+-tree is slightly faster than with the other algorithms for all buffer sizes. The lowest execution time value is achieved with buffer size of 2^{10} pages (nodes) for all algorithms and the minimum execution time value, 152,541 ms (152 s), is achieved with the C-DF-2 algorithm.

Considering the complete set of 45/50 experiments for the 5 buffer sizes (the biggest combination Water × Park was not tested for all node sizes because of the big execution time values) we collected the minimum execution time values for each combination and these results are shown in Table 11. It is shown that there is not a single best buffer size, while 9/10 best execution times were achieved with node size equal to 1KB. We conclude that for combinations between small real datasets it is better to have 2^{10} pages in LRU-buffer, while for combinations of small synthetic and large real datasets it is better to have 2^8 buffer pages (nodes). The best algorithm for R+-trees executing the ε*DJQ* is the H-DF-2 (in 9/10 cases).

In Fig. 11b, we can see the results of the ε*DJQ* on the combination of real datasets NArdND × Water, both indexed by R*-trees with node size of 4KB, searching for the pairs with distance $\leq 1.25 \times 10^{-5}$ with the four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12} pages, as one representative case. It is shown that with the C-BF-2 algorithm the R*-tree is from 1.1 up to 1.4 times faster than the best of the other tree algorithms for all buffer sizes.

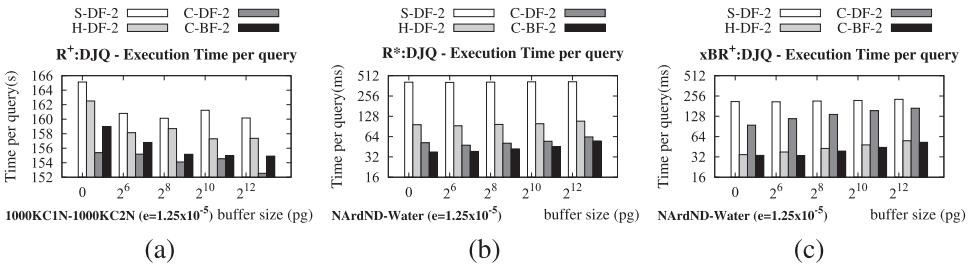

Fig. 11. ε *DJQ* with (a) R$^+$-tree on 1000KC1N × 1000KC2N; (b) R*-tree on NArdND × Water; (c) xBR$^+$-tree on NArdND × Water: exec. time vs. buffer size per Algorithm with $\varepsilon = 1.25 \times 10^{-5}$ and node size = 4KB.

Table 12
ε *DJQ* ($\varepsilon = 1.25 \times 10^{-5}$) with R*-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec time (ms)	Node size (KB)	Buffer size (pages)	Algorithm
NAppN × NArrN	50.619	4	0	C-BF-2
NAppN × NArdN	129.48	4	0	C-BF-2
NArrN × NArdN	157.00	1	2^8	C-BF-2
NArrND × NArdND	319.55	1	2^8	C-BF-2
250KC1N × 250KC2N	176.06	8	2^6	C-BF-2
500KC1N × 500KC2N	356.79	16	2^6	C-BF-2
500KC2N × 1000KC1N	532.82	16	0	C-BF-2
1000KC1N × 1000KC2N	782.53	16	2^6	C-BF-2
NArdND × Water	25.698	1	0	C-BF-2
Water × Park	1,212.5	4	0	C-BF-2

Table 13
ε *DJQ* ($\varepsilon = 1.25 \times 10^{-5}$) with xBR$^+$-tree: Min Exec Time per query for all combinations of datasets.

Combination of datasets	Exec Time(s)	Node size	Buffer size	Algorithm
NAppN × NArrN	11.094	16	0	H-DF-2
NAppN × NArdN	27.156	4	0	H-DF-2
NArrN × NArdN	46.244	4	0	H-DF-2
NArrND × NArdND	119.86	2	0	H-DF-2
250KC1N × 250KC2N	21.346	16	0	C-BF-2
500KC1N × 500KC2N	47.708	8	0	H-DF-2
500KC2N × 1000KC1N	73.090	8	0	H-DF-2
1000KC1N × 1000KC2N	102.57	8	0	H-DF-2
NArdND × Water	23.360	2	0	C-BF-2
Water × Park	682.02	4	0	H-DF-2

The lowest execution time value is achieved with buffer size of 0 pages (nodes) for all algorithms and the minimum execution time value, 37.837 ms, is achieved with the C-BF-2 algorithm.

Considering the complete set of all (50) experiments for the 10 dataset combinations and 5 buffer sizes, we collected the minimum execution time values for each combination and these results are shown in Table 12. It is shown that R*-tree responded best in half cases (5/10), including the combinations between large real datasets, without buffering, while in the 4 combinations between synthetic datasets it responded best with 2^6 pages in LRU-buffer (in 3/4 cases). The best algorithm for R*-trees executing the ε*DJQ* is the C-BF-2.

Finally, in Fig. 11c, we can see the results of the ε*DJQ* on the combination of real datasets NArdND × Water, both indexed by xBR$^+$-trees with node size of 4KB, searching for the pairs with distance $\leq 1.25 \times 10^{-5}$ with all four algorithms, using LRU-buffer sizes of 0, 2^6, 2^8, 2^{10}, 2^{12} pages, as one representative case. It is shown that with the C-BF-2 algorithm the xBR$^+$-tree is from 2.8 up to 3.5 times faster than the best among the S-DF-2 and C-DF-2 algorithms and from 1.03 up to 1.13 faster than the H-DF-2 algorithm for all buffer sizes. The lowest execution time value was achieved with buffer size of 2^6 pages (nodes) for all algorithms and the minimum execution time value, 33.437 ms, was achieved with the C-BF-2 algorithm.

Considering the complete set of all (50) experiments, we collected the minimum execution time values for each combination of datasets and these results are shown in Table 13. It is shown that xBR$^+$-tree responded best without buffering in all combinations of datasets. The best algorithm for xBR$^+$-trees executing the ε*DJQ* is the H-DF-2 (8/10 cases).

In conclusion, we note that:

- There is no meaning for a comparison between the R$^+$-tree and the other two trees for ε*DJQ*, because of the very big difference in execution times observed (the R$^+$-tree was mainly designed for *PLQs* and *WQs*).
- We continue the performance comparison between the R*-tree and xBR$^+$-tree, using the C-BF-2 algorithm for the first one and the H-DF-2 algorithm for the second one, without LRU-buffer (0 pages) for all node sizes, for various values of $\varepsilon (0, 1.25 \times 10^{-5}, 2.5 \times 10^{-5}, 5 \times 10^{-5}, 10 \times 10^{-5})$ and for both performance metrics (i.e. the number of disk read accesses and the execution time).

6.8.4. Performance study of the εDJQ

In Fig. 12a and b, we can see the results of the ε*DJQ* with $\varepsilon = 1.25 \times 10^{-5}$ executed on the combination of large real datasets Water × Park, as one representative case. The number of disk read accesses per query and the execution time vs. the node size are shown. The xBR$^+$-tree needed fewer disk read accesses (*Acc*) than the R*-tree for all node sizes. As the node size increases, the ratio of the I/O difference between the two trees varies. The relation between I/O performance and the node size is quite stable (almost linear). The relative difference $(Acc_{R^*} - Acc_{xBR^+})/Acc_{R^*}$ has values (0.76, 0.71, 0.66, 0.61, 0.49), all in favor of the xBR$^+$-tree. For the R*-tree, the ratio of the numbers of disk read accesses between two consecutive node sizes presents a small variation (from 0.42 up to 0.49 and always is decreased) and for the xBR$^+$-tree this ratio presents a similar level of variation, from 0.51 up to 0.60. In Fig. 12b, it is shown that the xBR$^+$-tree is faster than the R*-tree for all node sizes. For both trees the execution time has minimum value with node size of 4KB, even though larger node sizes needed fewer disk read accesses. In Fig. 12c and d, we can see the results of the ε*DJQ* on the same combination of large real datasets indexed by trees with node size of 4KB, regarding the number of disk read accesses and the execution time vs. the value of ε. The number of disk read accesses needed by xBR$^+$-tree remains very stable although the value of ε is increased exponentially. The xBR$^+$-tree has the best performance regarding disk read accesses in the most cases (3/5), while in the execution time it was the best in all cases.

Studying the complete set of results (250 experiments for each tree) for the ε*DJQ*, we validate the above performance behavior.

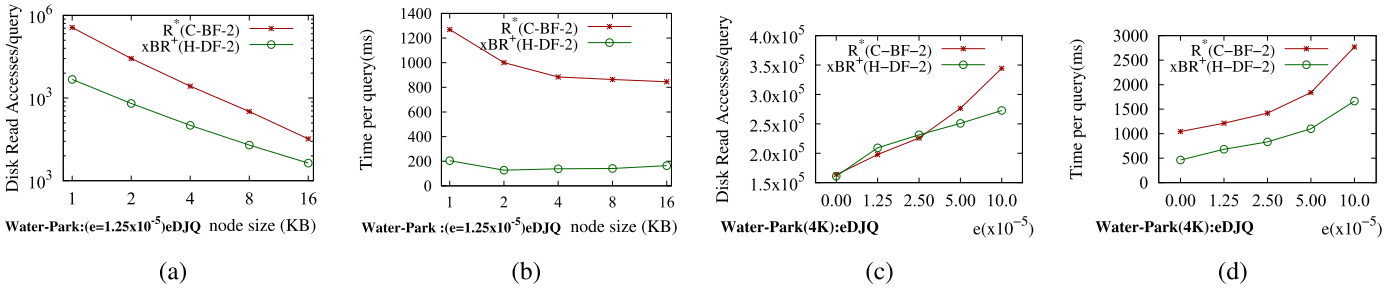

Fig. 12. ε *DJQ* on large real datasets Water × Park: disk read accesses (a) and exec. time (b) vs. node size ($\varepsilon = 1.25 \times 10^{-5}$) and disk read accesses (c) and exec. time (d) vs. ε values (node size=4KB).

Table 14
Synopsis of efficiency of xBR$^+$-tree in all queries vs. R*-tree.

	all node sizes		node size=16KB		
Query name	Time	I/O	Time	gain	I/O
Single-dataset queries:					
WQ	360/360	323/360	72/72	64.7%	68/72
DRQ	360/360	292/360	72/72	68.4%	67/72
KNNQ	209/240	67/240	48/48	49.2%	18/48
CKNNQ	240/240	138/240	48/48	60.5%	30/48
Dual-dataset queries:					
KCPQ	231/250	224/250	50/50	71.9%	45/50
εDJQ	235/250	216/250	49/50	66.4%	43/50

- The number of disk accesses per query for the xBR$^+$-tree was the smallest for most experiments (216/250).
- The xBR$^+$-tree was faster than the R*-tree in all experiments with node sizes 2KB and 4KB, in 49/50 cases with node size of 16KB, while in total it was faster in 235/250 case.
- The average relative difference of execution time performance between the R*-tree and the xBR$^+$-tree is between 66.2% and 68.2%.

6.9. Summary and conclusions of experimental results

The experimental results of tree building are summarized in the following.

- The xBR$^+$-tree needs a little less space (in most cases) and is built in a smaller time than the two R-trees.
- The xBR$^+$-tree building is faster than the R*-tree and the R*-tree is faster than R$^+$-tree for all datasets and node sizes.
- This difference is increasing as the node size increases and becomes bigger for the large real datasets.

The fractions of cases where the xBR$^+$-tree is an execution time and I/O performance winner, for each (single, or dual dataset) query, is depicted in Table 14. The second and third columns refer to the aggregate results for all page sizes, while the fourth and sixth columns refer to results when using a page size of 16KB. The fifth column refers to the xBR$^+$-tree gain in execution time, (R*-tree exec. time - xBR$^+$-tree exec. time) / R*-tree exec. time, for the page size of 16KB (e.g. a gain value equal to 66.67% for a query means that the xBR$^+$-tree needs 1/3 of the execution time of the R*-tree to answer this query). By studying these results, we conclude that the xBR$^+$tree is a clear performance winner, in relation to the R*-tree (the best among R-trees). More specifically:

- The xBR$^+$-tree is a big winner in execution time in all cases and a winner in I/O in all cases except of the I/O of the *KNNQ*.
- The xBR$^+$-tree is an almost absolute winner when the page size equals 16KB (for this page size the xBR$^+$-tree is a relative winner in the I/O of the *KNNQ*, too). Note the high percentages of gain for this page size.

Note that the R$^+$-tree was designed specifically for *PLQs* and *WQs* and not for other ones, like *DRQs, KNNQs, KCPQs*, *εDJQs*, etc.

The regular subdivision of space, the additional representation of the minimum rectangles bounding the actual data objects (DBRs), the extra termination condition applicable in certain queries and the storage order of the entries of internal nodes gave the ability to the xBR$^+$-tree to be a more efficient structure than R-trees and even than the R*-tree. More specifically, the building performance of the xBR$^+$-tree can be credited to the following:

- Due to the regular subdivision of space, the calculations needed are much fewer and simpler than those of the R*-tree.

The building time of an xBR$^+$-tree is smaller even than the one needed for building the respective, very simple, R$^+$-tree. The very good performance of the xBR$^+$-tree in queries can be credited to the following:

- The regular subdivision of space leads to laying the (sub)quadrants, created by the data distribution, in the corners of the embedding (sub)space. In this way, the distances between them are maximized and pruning during join query processing is increased.
- Due to the quadrangular shape of the (sub)quadrants, the dimensions of the contained DBRs are minimized. The minimal dimensions of DBRs in conjunction with their laying in the corners of the embedding (sub)space allows the high exploitation of metrics like *mindist* (the R$^+$-trees, due to their structure, do not utilize efficiently such pruning techniques).
- In xBR$^+$-trees, DBRs are exploited as an extra tool of delimiting the subspace containing data objects.
- By examining the entries of an xBR$^+$-tree internal node in reverse preorder traversal of the Quadtree that corresponds to this internal node (a subregion is examined before any enclosing region of this subregion), multiple examinations of overlapping regions are avoided (at least in point location and window queries).
- The disjointness between regions and the combination of the region of each node with the *Shape* property of this node gives the ability of an extra termination condition in window and range queries (this condition cannot be applied in R-trees, due to their structure).

The conclusions arising from the performance results of the alternative DF/BF algorithms for processing queries are summarized in the following:

- For *PLQs* and *WQs*, N-DF algorithms are the only applicable, since the criterion for such queries is boolean (true/false).
- For the rest of the queries, among DF algorithms, the winning algorithm and the respective percentage of cases is depicted in Table 15. It is obvious that the H-DF variants are the most efficient ones in xBR$^+$-trees, in 4/5, and in R*-trees, in 3/5 of the query types. As noted in Section 5.1, this is due to partial sorting of (pairs of) entries by *mindist* when H-DF variants are used.

Table 15
The winning DF algorithm (and the respective percentage of cases) for *DRQs, KNNQs, CKNNQs, KCPQs*, and ε*DJQs*.

	R^+-tree		R^*-tree		xBR^+-tree	
Query name	(%)	Alg	(%)	Alg	(%)	Alg
Single-dataset queries:						
DRQ	69.2	DF	75.3	DF	54.4	DF
KNNQ	64.6	S-DF	81.7	H-DF	99.2	H-DF
CKNNQ	64.6	H-DF	65.8	H-DF	51.7	H-DF
Dual-dataset queries:						
KCPQ	90.0	C-DF-2	54.8	H-DF-2	95.6	H-DF
ε*DJQ*	44.8	C-DF-2	99.6	C-DF-2	95.6	H-DF-2

- BF algorithms perform significantly better on the R^*-tree, since, due to overlapping between regions of nodes at the same level, the minimization of I/O that BF algorithms achieve plays an important role.

7. Conclusions and future work

In Roumelis et al. (2015), the xBR^+-tree was compared to the xBR-tree for single dataset queries and datasets of medium size and "a detailed relative performance study of the xBR^+-tree against the R^*-tree and/or R^+-tree for single dataset and multi-dataset queries" was mentioned as the main future work target, since these structures had never been compared in the literature.

In this paper, we accomplished this target based on single, as well as, on dual dataset queries, utilizing existing and new algorithms and performing experiments on medium, as well as, large datasets. More specifically, in this paper, we presented algorithms for *PLQs* and *WQs* used in the above three structures. We also presented for these structures N-DF, S-DF and BF existing algorithms and the new H-DF algorithm for *DRQs, KNNQs* and *CKNNQs*. Moreover, we presented the first algorithms for *KCPQs* and ε*DJQs* on the xBR^+-tree and a new alternative DF algorithm (H-DF-2) for *KCPQs* and ε*DJQs* for all the three trees. We also highlighted the differences between alternative algorithms.

Moreover, by a detailed performance comparison (I/O and execution time) of xBR^+-trees (non-overlapping trees of the quadtree family), R^+-trees (non-overlapping trees of the R-tree family) and R^*-trees (industry standard belonging to the R-tree family) for tree building, processing single point dataset queries (*PLQs, WQs, DRQs, KNNQs* and *CKNNQs*) and distance-based join queries (*KNNQs*, ε*DJQs*), using medium and large spatial (real and synthetic) datasets, we showed that the xBR^+-tree is a clear winner in tree building, a big winner in execution time in all cases and a winner in I/O in all cases, except for the I/O of the *KNNQ* (it is an almost absolute winner when the page size equals 16KB).

The building performance of the xBR^+-tree is due to the regular subdivision of space that leads to much fewer and simpler calculations. The higher query performance of the xBR^+-tree is due to the combination of the regular subdivision of space, the additional representation of the minimum rectangles bounding the actual data objects (DBRs) and the extra termination condition applicable in certain queries and the storage order of the entries of internal nodes gave

In the future we plan to:

- Compare the three trees for data of dimensionality larger than 2,
- Create extensions of the xBR^+-tree for non-point data objects and algorithms for processing queries on them and compare to competitive structures,
- Create extensions of the xBR^+-tree for parallel and distributed environments,
- Create algorithms to bulk-load xBR^+-trees.

Acknowledgments

Work of the second, third and fourth author funded by the MINECO research project [TIN2013-41576-R] and the Junta de Andalucia research project [P10-TIC-6114].

References

Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B., 1990. The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD Conference.Atlantic City, NJ, pp. 322–331.
Brinkhoff, T., Horn, H., Kriegel, H.-P., Schneider, R., 1993. A storage and access architecture for efficient query processing in spatial database systems. In: SSD Conference. Singapore, pp. 357–376.
Brown, P., 2001. Object-Relational Database Development: A Plumber's Guide. Informix Press.
Chen, Y., Patel, J.M., 2007. Efficient evaluation of all-nearest-neighbor queries. In: ICDE Conference. Istanbul, Turkey, pp. 1056–1065.
Cheung, K.L., Fu, A.W.-C., 1998. Enhanced nearest neighbour search on the R-tree. ACM SIGMOD Record 27 (3), 16–21.
Comer, D., 1979. The ubiquitous B-tree. ACM Comput. Surv. 11 (2), 121–137.
Corral, A., Almendros-Jiménez, J.M., 2007. A performance comparison of distance-based query algorithms using R-trees in spatial databases. Inf. Sci. 177 (11), 2207–2237.
Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M., 2000. Closest pair queries in spatial databases. In: SIGMOD Conference. Dallas, TX, pp. 189–200.
Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M., 2004. Algorithms for processing *k*-closest-pair queries in spatial databases. Data Knowl. Eng. 49 (1), 67–104.
Corti, P., Kraft, T.J., Mather, S.V., Park, B., 2014. PostGIS Cookbook. Packt Publishing.
Eldawy, A., Mokbel, M.F., 2015. Spatialhadoop: a MapReduce framework for spatial data. In: ICDE Conference. Seoul, South Korea, pp. 1352–1363.
Faloutsos, C., Barber, R., Flickner, M., Hafner, J., Niblack, W., Petkovic, D., Equitz, W., 1994. Efficient and effective querying by image content. J. Intell. Inf. Syst. 3 (3/4), 231–262.
Finkel, R.A., Bentley, J.L., 1974. Quad trees: a data structure for retrieval on composite keys. Acta Informatica 4 (1), 1–9.
Gaede, V., Günther, O., 1998. Multidimensional access methods. ACM Comput. Surv. 30 (2), 170–231.
Greener, S., Ravada, S., 2013. Applying and Extending Oracle Spatial. Packt Publishing.
Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. In: SIGMOD Conference. Boston, MA, pp. 47–57.
Hjaltason, G.R., Samet, H., 1995. Ranking in spatial databases. In: SSD Conference. Portland, ME, pp. 83–95.
Hjaltason, G.R., Samet, H., 1998. Incremental distance join algorithms for spatial databases. In: SIGMOD Conference. Chicago, IL, pp. 237–248.
Hjaltason, G.R., Samet, H., 1999. Distance browsing in spatial databases. ACM Trans. Database Syst. 24 (2), 265–318.
Hoel, E.G., Samet, H., 1992. A qualitative comparison study of data structures for large line segment databases. In: SIGMOD Conference. San Diego, CA, pp. 205–214.
Hoel, E.G., Samet, H., 1995. Benchmarking spatial join operations with spatial output. In: VLDB Conference. Zurich, Switzerland, pp. 606–618.
Jagadish, H.V., 1991. A retrieval technique for similar shapes. In: SIGMOD Conference. Denver, CO, pp. 208–217.
Kanth, K.V.R., Ravada, S., Abugov, D., 2002. Quadtree and R-tree indexes in Oracle Spatial: a comparison using GIS data. In: SIGMOD Conference. Madison, WI, pp. 546–557.
Kim, Y.J., Patel, J.M., 2010. Performance comparison of the R*-tree and the quadtree for *k*-NN and distance join queries. IEEE Trans. Knowl. Data Eng. 22 (7), 1014–1027.
Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E.L., Protopapas, Z., 1996. Fast nearest neighbor search in medical image databases. In: VLDB Conference. Bombay, India, pp. 215–226.
Kothuri, R.V., Godfrind, A., Beinat, E., 2007. Pro Oracle Spatial for Oracle Database 11g. APress.
Leutenegger, S.T., Edgington, J.M., López, M.A., 1997. STR: a simple and efficient algorithm for R-tree packing. In: ICDE Conference.Birmingham, UK, pp. 497–506.
Lomet, D.B., Salzberg, B., 1990. The hb-tree: a multiattribute indexing method with good guaranteed performance. ACM Trans. Database Syst. 15 (4), 625–658.
Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y., 2006. R-trees: Theory and Applications. Springer, London, UK.
Mehrotra, R., Gary, J.E., 1993. Feature-based retrieval of similar shapes. In: ICDE Conference. Vienna, Austria, pp. 108–115.
Nelson, R.C., Samet, H., 1986. A consistent hierarchical representation for vector data. In: SIGGRAPH Conference.Dallas, TX, pp. 197–206.
Obe, R., Hsu, L., 2015. PostGIS in Action, 2 Manning.
Ohsawa, Y., Sakauchi, M., 1990. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In: ICDE Conference. Los Angeles, USA, pp. 296–303.
Rigaux, P., Scholl, M., Voisard, A., 2000. Introduction to Spatial Databases: Applications to GIS. Morgan Kaufmann.

Roumelis, G., Corral, A., Vassilakopoulos, M., Manolopoulos, Y., 2016. New plane-sweep algorithms for distance-based join queries in spatial databases. GeoInformatica 1–58. doi:10.1007/s10707-016-0246-1. First Online.
Roumelis, G., Vassilakopoulos, M., Corral, A., 2011. Nearest neighbor algorithms using xBR-trees. In: Panhellenic Conference on Informatics. Kastoria, Greece, pp. 51–55.
Roumelis, G., Vassilakopoulos, M., Corral, A., 2011. Performance comparison of xBR-trees and R*-trees for single dataset spatial queries. In: ADBIS Conference. Vienna, Austria, pp. 228–242.
Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., Manolopoulos, Y., 2015. The xBR$^+$-tree: an efficient access method for points. In: DEXA Conference. Valencia, Spain, pp. 43–58.
Roussopoulos, N., Kelley, S., Vincent, F., 1995. Nearest neighbor queries. In: SIGMOD Conference. San Jose, CA, pp. 71–79.
Samet, H., 1984. The quadtree and related hierarchical data structures. ACM Comput. Surv. 16 (2), 187–260.
Samet, H., 1990a. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, Boston, MA.
Samet, H., 1990b. The Design and Analysis of Spatial Data Structures. Addison-Wesley, Boston, MA.
Samet, H., 2007. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco, CA.
Schwartz, B., Zaitsev, P., Tkachenko, V., 2012. High Performance MySQL - Optimization, Backups, and Replication, 3 O'Reilly.
Sellis, T., Roussopoulos, N., Faloutsos, C., 1987. The R$^+$-tree: a dynamic index for multi-dimensional objects. In: VLDB Conference. Brighton, UK, pp. 507–518.
Shekhar, S., Chawla, S., 2003. Spatial Databases - A Tour. Prentice Hall.
Shin, H., Moon, B., Lee, S., 2003. Adaptive and incremental processing for distance join queries. IEEE Trans. Knowl. Data Eng. 15 (6), 1561–1578.
Vassilakopoulos, M., Manolopoulos, Y., 2000. External balanced regular xBR-trees: new structures for very large spatial databases. In: Advances in Informatics: Selected papers of the 7th Panhellenic Conference on Informatics. World Scientific Publ. Co., pp. 324–333.
Yin, X., Düntsch, I., Gediga, G., 2011. Quadtree representation and compression of spatial data. Trans. Rough Sets 13, 207–239.

George Roumelis studied Physics in Aristotle University of Thessaloniki (AUTH), Greece and is currently working as a teacher and a vice-principle in a local high school of Thessaloniki, Greece. He obtained a master's degree in Informational Systems from the Open University of Cyprus (2011) and is currently a PhD candidate in the Informatics Department of AUTH, working on spatial databases. His main research interests include access methods, query processing and spatial and spatio-temporal databases. He has published several original papers in international conferences and journals. His interests also include software development and support for educational and administration units in the public educational system of Greece.

Michael Vassilakopoulos obtained a five-year Diploma in Computer Eng. and Informatics from the University of Patras (Greece) and a PhD in Computer Science from the Department of Electrical and Computer Eng. of the Aristotle University of Thessaloniki (Greece). He has been with the University of Macedonia, the Aristotle University of Thessaloniki, the Technological Educational Institute of Thessaloniki, the Hellenic Open University, the Open University of Cyprus, the University of Western Macedonia, the University of Central Greece and the University of Thessaly. For three years he served the Greek Public Administration as an Informatics Engineer. Currently, he is an Associate Professor of Database Systems at the Department of Electrical and Computer Engineering of the University of Thessaly. He has participated in/coordinated several RTD projects related to Databases, GIS, WWW, Information Systems and Employment. His research interests include databases, data structures, algorithms, data mining, employment analysis, information systems, GIS and current trends of data management.

Antonio Corral is an Associate Professor at the Department of Informatics, University of Almeria (Spain). He received his PhD (2002) in Computer Science (European Doctorate) from the University of Almeria (Spain). He has participated actively in several research projects in Spain (INDALOG, vManager, etc.) and Greece (CHOROCHRONOS, ARCHIMEDES, etc.). He has published in referred scientific international journal (Data & Knowledge Engineering, The Computer Journal, GeoInformatica, Information Sciences, etc.), conferences (SIGMOD, SSD, ADBIS, SOFSEM, PADL, etc.) and book chapters. His main research interests include access methods, query processing and spatial and spatio-temporal databases.

Yannis Manolopoulos is Professor with the Department of Informatics of the Aristotle University of Thessaloniki. He has been with the University of Toronto, the University of Maryland at College Park and the University of Cyprus. He has also served as Rector of the University of Western Macedonia in Greece, Head of his own department, and Vice-Chair of the Greek Computer Society. His research interest focuses in Data Management. He has co-authored 5 monographs and 8 textbooks in Greek, as well as >300 journal and conference papers. He has received >11.000 citations from >1700 distinct academic institutions (h-index=49). He has also received 4 best paper awards from SIGMOD, ECML/PKDD, MEDES and ISSPIT conferences and has been invited as keynote speaker in 13 international events. He has served as main co-organizer of several major conferences (among others): ADBIS 2002, SSTD 2003, SSDBM 2004, ICEIS 2006, EANN 2007, ICANN 2010, AIAI 2012, WISE 2013, CAISE 2014, MEDI 2015, ICCCI 2016, TPDL 2017. He has also acted as evaluator for funding agencies in Austria, Canada, Cyprus, Czech Republic, Estonia, EU, Hong-Kong, Georgia, Greece, Israel, Italy, Poland and Russia. Currently, he serves in the Editorial Boards of (among others) The VLDB Journal, The World Wide Web Journal, The Computer Journal.