An Efficient Algorithm for Bulk-Loading xBR$^+$-trees

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

[a]*Dept. of Informatics, Aristotle University of Thessaloniki, Greece*
[b]*Dept. of Electrical and Computer Engineering, University of Thessaly, Volos, Greece*
[c]*Dept. on Informatics, University of Almeria, Spain*

Abstract

A major part of the interface to a database is made up of the queries that can be addressed to this database and answered (processed) in an efficient way, contributing to the quality of the developed software. Efficiently processed spatial queries constitute a fundamental part of the interface to spatial databases due to the wide area of applications that may address such queries, like geographical information systems (GIS), location-based services, computer visualization, automated mapping, facilities management, etc. Another important capability of the interface to a spatial database is to offer the creation of efficient index structures to speed up spatial query processing. The xBR$^+$-tree is a balanced disk-resident quadtree-based index structure for point data, which is very efficient for processing such queries. Bulk-loading refers to the process of creating an index from scratch, when the dataset to be indexed is available beforehand, instead of creating the index gradually (and more slowly), when the dataset elements are inserted one-by-one. In this paper, we present an algorithm for bulk-loading xBR$^+$-trees for big datasets residing on disk, using a limited amount of main memory. The resulting tree is not only built fast, but exhibits high performance in processing a broad range of spatial queries, where one or two datasets are involved. To justify these characteristics, using real and artificial datasets of various cardinalities, first, we present an experimental comparison of this algorithm vs. a previous version of the same algorithm and *STR*, a popular algorithm of bulk-loading R-trees, regarding tree creation time and the characteristics of the trees created, and second, we experimentally compare the query efficiency of bulk-loaded xBR$^+$-trees vs. bulk-loaded R-trees, regarding I/O and execution time. Thus, this paper contributes to the implementation of spatial database interfaces and the efficient storage organization for big spatial data management.

Keywords: Spatial Database Interfaces, Spatial Indexes, Bulk-Loading, xBR$^+$-trees, Spatial Query Processing

1. Introduction

The various types of queries that can be addressed to a database and answered (processed) in an efficient way make up a key part of the interface to this database. Spatial queries that can be efficiently processed constitute a fundamental part of the interface to spatial databases due to the wide area of applications that may address such queries, like geographical information systems (GIS), location-based services, computer visualization, automated mapping, facilities management, etc. One of the most important characteristics of the interface to a spatial database is to provide the creation of spatial indexes that can be used for the execution of efficient algorithms for spatial queries [1, 2] on big data. In many cases, the efficiency of these index structures depends on whether the execution time of spatial queries without such indexes is less than the total time to execute them when the time to create such indexes is added. For example, these spatial indexes can be created from non-indexed datasets or from intermediate results of the query executions, and such a creation process must not be too time-consuming. To carry out this creation task, a bulk-loading algorithm of a spatial index is needed. A bulk-loading algorithm creates an index from scratch, when all data are known a priori, and therefore it is based on more information than the standard insertion algorithm, where all

*Corresponding author
Email addresses: groumeli@csd.auth.gr (George Roumelis), mvasilako@uth.gr (Michael Vassilakopoulos), acorral@ual.es (Antonio Corral), manolopo@csd.auth.gr (Yannis Manolopoulos)

data are inserted one-by-one. By using this additional knowledge, the bulk-loading algorithm can produce a spatial index (1) in smaller creation time, (2) with better storage utilization, and (3) with better performance of spatial query processing. These three issues should be taken into account in the design of an efficient bulk-loading algorithm for a particular spatial index structure.

If the dataset for which a spatial index is needed is *static* (i.e. when insertions and deletions are rarely executed or even they are not performed at all), then we can devote efforts on building the spatial index in a way that permits the efficient execution of queries. However, it is also desirable that the creation time of the index is as small as possible. Both these goals become even more important when this dataset is big. The fast construction of an optimized spatial index structure regarding certain index characteristics (e.g. storage overhead minimization, storage utilization maximization, etc.) is an interesting challenge, since it is anticipated that, due to these characteristics, spatial query processing performance will be improved. The creation of indexes by inserting elements one-by-one is less efficient than bulk-loading algorithms that can be executed for creating the indexes with the same complexity as external sorting [3].

Bulk-loading refers to the process of creating an index from scratch for a given dataset [3], and here we use this term to characterize the process of building a disk-based spatial index for an entire dataset. Bulk-loading is especially necessary and useful when an index has to be built up for the first time, knowing the data in advance and the datasets are large enough [4]. The bulk-loading methods can be classified into three categories: *sort-based*, *buffer-based* and *sampled-based* methods [3]. The *sorted-based* methods are the ones most studied in the literature and they are widely used in popular spatial DBMS [5, 6, 7], because they roughly consist of sorting the data and create the tree in a bottom-up fashion.

In this research work, we study the efficiency of building quadtree-based index structures. In particular, we focus on the xBR$^+$-tree [8], a balanced disk-based index structure for point data that belongs to the Quadtree family, and hierarchically decomposes the space in a regular manner. The xBR$^+$-tree improves the xBR-tree [9, 10] regarding nodes structure and the splitting process of nodes. Moreover, it outperforms xBR-trees and R*-trees [11] with respect to several well-known spatial queries, such as Point Location, Window Query, K-Nearest Neighbor, etc.

Material that contributes to the foundation of this research work appeared in [12]. The algorithm proposed in that paper (called *PBL*) was the first algorithm for bulk-loading xBR$^+$-trees for large datasets residing on disk, using a limited amount of main memory. This algorithm belongs to *sort-based bulk-loading* methods, since it sorts, according to z-order, groups of data items that either fit in the available internal memory, or in a disk page, depending on the phase of the algorithm. This sorting is performed in parallel to tree building. This algorithm improves the index structure creation process with respect to the xBR$^+$-tree algorithm of loading the elements one-by-one, and allowed better spatial query processing of single dataset spatial queries (Point Location – *PLQ*, Window Query – *WQ*, Distance Range Query – *DRQ*, K-Nearest Neighbors Query – *KNNQ*, Constrained K-Nearest Neighbors Query – *CKNNQ*) and of dual dataset spatial queries (K-Closest Pairs Query – *KCPQ* and Distance Join Query – ε*DJQ*).

In this paper, we present *PBL+*, an improvement of the *PBL* bulk-loading algorithm for bulk loading xBR$^+$-trees [12]. The new algorithm, like its predecessor, consists of four phases, however, in the new algorithm, all the phases of [12] have been improved so that the xBR$^+$-tree is created by making better use of the available main memory, faster and with improved structural characteristics and reduced size that lead to increased performance in processing queries. Moreover, using real and artificial datasets of various (small and big) cardinalities, first, we present an experimental comparison of the new algorithm (PBL+) vs. the algorithm of [12] (PBL) and *Sort-Tile-Recursive* (STR) [13], a popular algorithm of bulk-loading R-trees, regarding tree creation time and the characteristics of the trees created, and second, we experimentally compare the query efficiency of xBR$^+$-trees created by the new algorithm vs. R-trees created by *STR*, regarding I/O and execution time. Thus, through improving spatial storage structures and the efficiency of processing spatial queries, this paper contributes to the implementation of spatial database interfaces and the quality of software for big spatial data management.

This paper is organized as follows. In Section 2, we review related work on bulk-loading techniques, the *PBL* algorithm, its key differences to the *PBL+* algorithm and the most important characteristics of xBR$^+$-trees, from the implementation point of view. Section 3 presents our new bulk-loading method for the xBR$^+$-tree. In Section 4, we present and discuss the results of our experiments. And finally, Section 5 provides the conclusions arising from our work and discusses related future work directions.

2. Related Work and Background

This section reviews previous bulk-loading methods in general, whose main target is to reduce the creation time, the storage utilization and the query cost of the resulting index structure, or all of them. In [3] the bulk-loading methods are roughly classified into three categories: sort-based, buffer-based and sampled-based methods.

- The *sort-based bulk-loading* methods are characterized by the following two steps: first, the dataset is sorted and second, the tree is built in a bottom-up fashion. The advantages of these methods are their simplicity of implementation and their good query performance.
- The *buffer-based bulk-loading* methods use the *buffer-tree* techniques [14], and they can be considered as a hybrid of top-down and bottom-up strategies [15]. They employ *buffers* attached to nodes of the tree, trying to keep the efficiency of the bulk-loading tree creation and reducing the I/O activity. A disadvantage of this technique is that the performance depends on the input ordering of the elements. Moreover, its objective is only to reduce the number of disk accesses, ignoring the CPU cost [16].
- The *sample-based bulk-loading* methods choose a *sample* with size small enough to fit in main memory and build the index structure for the full dataset using estimations provided by the *sample* [17]. The main problem of this method is that it depends strongly on the quality of the sample, which at times can lead to a bulk-loading time larger than that required by a one-by-one insertion [3].

2.1. Sort-based Bulk-loading Methods

There are several bulk-loading methods that belong to the *sort-based* category, but the most representative ones are proposed in [18, 19, 20, 13, 15]. In [18], a method (so-called *packed R-tree*) that uses a heuristic for aggregating rectangles into nodes is introduced. It suggests to sort the data items with respect to minimum value of the objects in a certain dimension. In [19] a variant of the packed R-tree is proposed, so-called *Hilbert-packed R-tree*, wherein the order is based purely on the Hilbert code of the objects' centroids. In order to improve the query performance, heuristics like the one proposed in [20] can be used for local data reorganization. Conclusions from the experiments in [20] suggest that a better space partitioning can be obtained with the Hilbert-packed R-tree by sacrificing 100% storage utilization. In particular, they proposed that nodes be initially filled to 75% in the usual way. If any of the items subsequently scheduled to be inserted into a node cause the node region to be enlarged by too much (e.g., by more than 20%), then no more items are inserted into the node. Another sort-based method for bulk-loading R-trees, called *Sort-Tile-Recursive* (STR) method, is presented in [13], and it can be applied to d-dimensional datasets. Assuming that N is the number of objects, B is the node capacity and d are the dimensions of the input dataset, the method starts sorting the data source with respect to the first dimension (e.g. using the center of the spatial objects). Then, $(N/B)^{1/d}$ contiguous partitions are generated, each of them containing (almost) the same number of objects. In the next step, each partition is sorted individually with respect to the next dimension. Again, partitions are generated of almost equal size and the process is repeated until each dimension has been treated. The final partitions will eventually contain at most B objects. In [13], it was also shown that this bulk-loading method creates R-trees whose search quality is superior to R-trees created with respect to the Hilbert-ordering [19]. However, the method also requires the input being sorted d times. Recently, in [15] a sort-based query-adaptive loading for building R-trees optimally designed for a given query profile is presented. Query-adaptive loading refers to the problem of generating R-trees whose average performance is minimized regarding a given static profile. Finally, in [21] a scalable alternative MapReduce approach to parallel loading of R-trees using a level-by-level design, based on [15], is introduced.

2.2. Buffer-based Bulk-loading Methods

Bulk-Loading methods based on *buffer-trees* [22] can be considered as a hybrid of top-down and bottom-up strategies [15], and they are called *buffer-based bulk-loading* methods. The basic idea of these methods is to delay insertions by temporarily storing input data in *buffers* attached to the nodes of the tree. If buffers are filled up, the batch of insertions is reactivated and the data continue their traversal down to the leaves. There are several outstanding research works in this category [23, 22, 24]. In [23], a generic algorithm for bulk-loading based on *buffer-trees* for a broad class of index structures (e.g. R-trees) is proposed. Instead of sorting, the split and merge routines of the target index structure are exploited for building an efficient temporary structure (based on *buffer-tree*). From this structure,

the desired index structure is built up incrementally bottom-up, one level at a time. [22] achieves a similar effect by using a regular R-tree structure (i.e. where the non-leaf nodes have the same fan-out as the leaf nodes) and attaching buffers to nodes only at certain levels of the tree. In [24], two extensible buffer-based algorithms for bulk operations (bulk-loading and bulk-insertion) in the class of space-partitioning trees (a class of hierarchical data structures that recursively decompose the space into disjoint partitions) are proposed. The main idea of these algorithms is to build an *in-memory tree* of the desired index structure using the standard insertion procedure. Then, the in-memory tree is used to distribute the remaining data items into disk-based buffers. Bulk-loading algorithms are applied recursively to the disk-based buffers.

2.3. Sample-based Bulk-loading Methods

The most remarkable *sample-based bulk-loading* algorithms have been proposed in [16, 25, 3, 26]. In [16] a method to build an M-tree [27] was proposed. This algorithm selects a number of *seed* objects (by sampling) around which other objects are recursively clustered to build an unbalanced tree, which must later be re-balanced. In [25], a kd-tree [28] structure is built up using a fast external algorithm for computing the median (or a point within an interval centered at the median). The sample is used for computing the skeleton of a kd-tree that is kept as an index in an internal node of the index structure. In [3], two generic sample-based bulk-loading algorithms are proposed that recursively partition the input by using a main-memory index of the same type as the target index to be built. In [26], two sample-based bulk-loading algorithms for dynamic metric access methods are presented, and they are based on the idea of covering radius of representative items employed to organize data in hierarchical data structures. These proposed algorithms were designed to Slim-trees [29], although they can be used in other metric access methods. In [30], a generic framework for R-tree bulk-loading based on sampling on a parallel architecture was introduced.

2.4. Bulk-loading Quadtree Structures

In [31, 32, 33, 24], we can find the most significant contributions related to bulk-loading techniques for Quadtree index structures. In [31], the bulk-loading is characterized by the process of building a disk-based spatial index for an entire set of objects without any intervening queries. The proposed approaches, trying to speed up the bulk-loading process on PMR-quadtrees [34], are based on the idea of trying to fill up memory with as much of the Quadtree as possible (using *buffers*) before writing some of its nodes on disk. The input data is sorted in such a way that the portions that are written out to the disk would not be inserted again. A first approach focused on the problem of B-tree level, increasing the amount of buffering done by the B-tree (*B-tree buffering*), similarly to [35]. A second approach focused on the problem of PMR-quadtree level, reducing the number of accesses to the B-tree as much as possible by storing parts of the PMR-quadtree in main memory (*Quadtree buffering*). In [32], improved versions of the bulk-loading algorithms studied in [31] for PMR-quadtrees are presented, assuming that the algorithms are implemented using a *linear quadtree* [36], which is a disk-resident representation that stores objects contained in the leaf nodes of the quadtree in a linear index (B^+-tree [37]) ordered on the basis of a space-filling curve (*z-order* [38]). Finally, an extended version of [31, 32] is presented in [33], where the problem of creating and updating spatial indexes in situations where the database is *dynamic*, is addressed. The functionalities that arise when the database is dynamic are covered by *bulk-loading*, *bulk-insertion* and *dynamic insertion* methods. Additionally, in [33], detailed algorithms for the proposed *sort-based bulk-loading* method, analytic observations, an extensive experimental study and interesting discussions are presented. According to [24], the two extensible buffer-based algorithms for bulk operations can be applied to Quadtrees, because of this is a type of space-partitioning tree, but no implementation and no experimentation were performed.

Our contribution differs from that presented in [33] in the following aspects: (1) The type of Quadtree; in [33] the bulk-loading method has been designed for the *linear quadtree* (a pointerless quadtree) using a B^+-tree (as a part of SAND -Spatial and Non-Spatial Data- research project [39]), while we propose a bulk-loading algorithm for a height-balanced, disk-resident, pointer-based Quadtree, the *xBR$^+$-tree*. (2) The strategy of tree creation; in [33] a top-down approach for bulk-loading was proposed, while we present a bottom-up approach for building the xBR$^+$-tree. (3) The scheme of buffers; in [33] a complex scheme of buffers (Quadtree buffer and B^+-tree buffer) was proposed, while we use an in-memory tree, the *m-xBR$^+$-tree*, for the bulk-loading algorithm. (4) Merge operations; in [33], complex merge operations for both a Quadtree and a B^+-tree are presented, while we use merge operations for the *m*-xBR$^+$-tree and the xBR$^+$-tree already built in secondary memory.

2.5. The PBL algorithm for xBR$^+$-trees

In [12], an algorithm, called Process of Bulk Loading (*PBL*), for bulk-loading xBR$^+$-trees (a totally disk-resident, height-balanced, pointer-based, multiway tree for multidimensional points; a presentation of xBR$^+$-trees appears in Subsection 2.6) for large datasets residing on disk was presented, using a limited amount of main memory.

PBL consists of four phases. These phases are pipelined. Each phase produces an output that is used as input for the next phase. During the first phase (Algorithm 1a), the initial dataset file is transformed to binary format and is split in two files.

Algorithm 1a PBL Phase1

Input: file f
1: **split** f in files f_1, f_2 by the middle of dimension 0
2: **if** sizeof(f_1) > *MemoryLimit* **then**
3: **Phase2**(f_1, 1)
4: **else**
5: **read** f_1 in memory block b
6: **Phase3**(b)
7: **if** sizeof(f_2) > *MemoryLimit* **then**
8: **Phase2**(f_2, 1)
9: **else**
10: **read** f_2 in memory block b
11: **Phase3**(b)

During the second phase (Algorithm 2a), each of the two input item files is partitioned into item blocks of size ≤ *MemoryLimit* in a regular fashion. The resulting blocks are transferred in main memory, as input for the next phase.

Algorithm 2a PBL Phase2

Input: file f_i, dimension d
1: **create** files t_0, t_1 ▷ *temporary files*
2: **split** f_i in files t_0, t_1 by the middle of f_i.*region* in dimension d
3: $d \leftarrow (d+1) \mod \#dimensions$
4: **if** t_0.*size* > *MemoryLimit* **then**
5: **Phase2**(t_0, d) ▷ *recursive call*
6: **else**
7: **read** t_0 in memory block b
8: **Phase3**(b)
9: **if** t_1.*size* > *MemoryLimit* **then**
10: **Phase2**(t_1, d) ▷ *recursive call*
11: **else**
12: **read** t_0 in memory block b
13: **Phase3**(b)

During the third phase (Algorithm 3a), for each block of items, a Quadtree (a four way tree) is built top-down n main memory by splitting this block regularly as long as the resulting regions contain more items than the capacity of xBR$^+$-tree leaves. This Quadtree is gradually transformed to an m-xBR$^+$-tree (main memory xBR$^+$-tree) in a bottom-up fashion.

During the last phase (Algorithm 4a), the m-xBR$^+$-tree is merged with the xBR$^+$-tree already built in secondary memory (created during the previous iteration of the bulk-loading process), discriminating between three different cases among the heights of the trees to be merged. More details about the *PBL* algorithm appear in [12].

This algorithm belongs to *sort-based bulk-loading* methods, because in Phases 1 and 2 it sorts (according to z-order) groups of data items that fit in the available internal memory size (*MemoryLimit*). This means that points are reordered in groups (an enumeration of groups is produced), where each group corresponds to a subquadrant and contains at most points that fit in *MemoryLimit* size, such that, each subsequent group in this enumeration (order) contains points with z-order value larger than the z-order of the points in any precedent group. However, the order

Algorithm 3a PBL Phase3

Input: memory block of points b
1: **build** Quadtree Q for b
2: **pack** Q subtrees that form level-0 nodes of the m-xBR$^+$-tree
3: **save** level-0 nodes of the m-xBR$^+$-tree ▷ *leaves of m-xBR$^+$-tree*
4: **if** #*leaves* of $Q > C$ **then**
5: $NextLevel \leftarrow$ TRUE
6: **else**
7: $NextLevel \leftarrow$ FALSE
8: $l = 0$
9: **while** $NextLevel$ = TRUE **do**
10: **remove** packed Q nodes
11: **if** #*nodes* of lowest level of $Q \leq C$ **then**
12: $NextLevel \leftarrow$ FALSE
13: $l \leftarrow l + 1$
14: **pack** Q subtrees that form level-l nodes of the m-xBR$^+$-tree
15: **Phase4**(m-xBR$^+$-tree)

Algorithm 4a PBL Phase 4

Input: m-xBR$^+$-tree m
Output: xBR$^+$-tree saved in disk
1: $d \leftarrow$ xBR$^+$-tree already saved in disk
2: **if** IsNull($d.root$) **then** ▷ *empty xBR$^+$-tree*
▷ *1st block being processed*
3: **save** m
4: $d.root \leftarrow m.root$
5: **else if** $m.height = d.height$ **then** ▷ *trees of equal height*
6: **if** #*entries* of ($m.root$) + #entries of ($d.root$) $\leq C$ **then**
7: **save** the whole m except for $m.root$
8: **copy** the entries of $m.root$ in $d.root$
9: **else** ▷ *merging of roots will cause an overflow*
10: **create** new root $d.rootNew$ for d
11: **save** m ▷ *save the whole tree m*
12: **insert** $m.root$ and $d.root$ as entries in $d.rootNew$
13: $d.root \leftarrow m.root$
14: **else if** $m.height < d.height$ **then** ▷ *the tree in disk is taller*
15: **find** the closest ancestor A_d of $m.root$ at $m.height$
16: **if** #*entries* of ($m.root$) + #entries of (A_d) $\leq C$ **then**
17: **save** the whole m except for the $m.root$
18: **copy** the entries of $m.root$ in node A_d
19: **else** ▷ *merging of nodes will cause an overflow*
20: **save** m ▷ *save the whole tree m*
21: **copy** the entry pointing to $m.root$ in parent node of A_d
22: **else** ▷ *the tree in main memory is taller*
23: $i \leftarrow 0$
24: **while** $i <$ #*entries* of $m.root$ **do**
25: **Phase4**($m.root.entry[i]$) ▷ *recursive call for a subtree*
▷ *with smaller height*
26: $i \leftarrow i + 1$
27: **save** $d.root$
28: **return** $d.root$

of points within each group is irrelevant. In Phase 3, sorting is continued, but now, at the page size level: points are reordered in pages, where each subsequent page in this order contains points with z-order value larger than the

z-order of the points in any preceding page. This is a type of sorting in 2 dimensions (or more dimensions, since the algorithm of [12] and the algorithm proposed in this paper work for any number of dimensions), or sorting to according to z-order, at *MemoryLimit* / disk page size resolution. This sorting is performed in parallel to tree building. The algorithm proposed here improves the index structure creation process with respect to the xBR$^+$-tree algorithm of loading the elements one-by-one, and allowed better spatial query processing of single dataset spatial queries and of dual dataset spatial queries.

The main motivation of this work is to improve the bulk-loading algorithm for xBR$^+$-tree presented in [12] (*PBL*). The enhancements of the algorithm presented in this paper, called *PBL+*, are summarized in the following.

- In Phases 1 (Algorithm 1b) and 2 (Algorithm 2b), instead of partitioning in two parts (either in disk, or in main memory), we partition in 2^d parts, reducing movements of data items on disk.

Algorithm 1b PBL+ Phase1

Input: file f
1: **create** files t_0, t_1, t_2, t_3 ▷ *four temporary files*
2: **split** f in files f_0, f_1, f_2, f_3 regularly ▷ *in quadrants of space*
3: **for** $i = 0$ .. 3 **do**
4: **if** sizeof(f_i) > *MemoryLimit* **then**
5: **Phase2**(f_i, t_0, t_1, t_2, t_3)
6: **else**
7: **read** f_i in memory block b
8: **Phase3**(b)

Algorithm 2b PBL+ Phase2

Input: file f_i, files t_0, t_1, t_2, t_3
1: **split** f_i in files t_0, t_1, t_2, t_3 regularly ▷ *in quadrants of f_i.region*
2: **for** $i = 0$ .. 3 **do**
3: **if** sizeof(t_i) > *MemoryLimit* **then**
4: **Phase2**(t_i, t_0, t_1, t_2, t_3) ▷ *recursive call*
5: **else**
6: **read** t_i in memory block b
7: **Phase3**(b)

- In Phase 3 (Algorithm 3b), instead of building top-down a Quadtree in main memory, we build bottom-up the m-xBR$^+$-tree, using an auxiliary tree, called T, in the following. T is a degree up-to-four tree (an incomplete Quaddtree) without leaves, that holds only the information that is necessary for creating the internal m-xBR$^+$-tree nodes, making better use of the available main memory and increasing tree creation speed.

- In Phase 4 (Algorithm 4b), when the m-xBR$^+$-tree and the tree already built in secondary memory have equal heights, or when the tree already built in secondary memory is higher, instead of creating a new root pointing to the two existing roots, without making any changes in the regions covered by these roots, we merge the roots of the two trees in a possibly overflown node. If this node is overflown, it is subsequently partitioned, in the best way possible, in two nodes that are pointed by a new root that is created. Moreover, when the m-xBR$^+$-tree is higher, we save it in secondary memory, we adjust the region of the root of the tree just saved to correspond to the whole space and then we apply the previous procedure of merging the roots of the two trees. Following the approach of merging roots and repartitioning the merged region to two regions, instead of leaving the regions of the original roots unchanged, as in [12], we achieve better partitioning of space and increased query performance in the resulting tree.

More details about *PBL+* are given in Section 3.

Algorithm 3b PBL+ Phase3

Input: memory block b

recursively, **split** b regularly to create leaves of the m-xBR$^+$-tree and **save** information for its internal nodes in an auxiliary tree T
save leaves of the m-xBR$^+$-tree
pack T nodes pointing to level-0 nodes of m-xBR$^+$-tree
if #*leaves* of $T > C$ **then**
    $NextLevel \leftarrow$ TRUE
else
    $NextLevel \leftarrow$ FALSE
$l = 0$
while $NextLevel$ = TRUE **do**
    remove packed T nodes
    if #*nodes* of lowest level of $T \leq C$ **then**
        $NextLevel \leftarrow$ FALSE
    $l \leftarrow l + 1$
    pack T nodes pointing to level l nodes of m-xBR$^+$-tree
Phase4(m-xBR$^+$-tree)

2.6. The xBR$^+$-tree

The xBR$^+$-tree [8] (an extension of the xBR-tree [9, 10]) is a hierarchical pointer based, disk-resident Quadtree-based (space-driven access method) index structure for multidimensional points. The xBR-tree family is the unique disk-resident pointer-based Quadtree in the literature together with [40]. For 2d space, the space indexed is a *square* and is recursively subdivided in 4 equal subquadrants. 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.

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* 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, although it is uniquely determined and can be easily calculated using *qside* and *DBR*. Each *Address* represents a subquadrant that 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 each of every dimension. The lower bit represents the subdivision on horizontal (X-axis) dimension, while the higher bit represents the subdivision on vertical (Y-axis) dimension [9, 10]. For example, the *Address* 1 represents the NE quadrant of the current space, while the *Address* 10 the NW subquadrant of the NE quadrant of the current space. The address of the left child is * (has zero digits), since the region of the left child is the whole space minus the region of the right child.

However, the 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 Figure 1 an internal node (a root) that points to 2 internal nodes that point to 5 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

Algorithm 4b PBL+ Phase 4

Input: m-xBR$^+$-tree m
Output: xBR$^+$-tree saved in disk

1: $d \leftarrow$ xBR$^+$-tree already saved in disk
2: **if** IsNull($d.root$) **then** ▷ *empty xBR$^+$-tree*
▷ *1st block being processed*
3: **save** m
4: $d.root \leftarrow m.root$
5: **else if** $m.height = d.height$ **then** ▷ *trees of equal height*
6: **save** the whole m except for $m.root$
7: **if** #*entries* of ($m.root$) + #entries of ($d.root$) $\leq C$ **then**
8: **copy** the entries of $m.root$ in $d.root$
9: **else** ▷ *merging of roots will cause an overflow*
10: **create** new node N_{new} for
#*entries* of ($m.root$) + #entries of ($d.root$)
11: **copy** the entries of $m.root$ and $d.root$ to node N_{new}
12: **split** the overflown N_{new} to regions R_1, R_2
13: **create** new root $d.rootNew$ for the xBR$^+$-tree
14: **insert** the entries of R_1 and R_2 in $d.rootNew$
15: $d.root \leftarrow d.rootNew$
16: **else if** $m.height < d.height$ **then** ▷ *the tree in disk is taller*
17: **find** the closest Ancestor A_d of $m.root$ at $m.height$
18: **save** the whole m except for $m.root$
19: **if** #*entries* of ($m.root$) + #entries of (A_d) $\leq C$ **then**
20: **copy** the entries of $m.root$ to node A_d
21: **else** ▷ *merging of nodes will cause an overflow*
22: **create** new node N_{new} for
#*entries* of ($m.root$)+#entries of (A_d)
23: **copy** the entries of $m.root$ and A_d to node N_{new}
24: **split** the overflown N_{new} in regions R_1, R_2
25: **save** entries of R_1 in node A_d, of R_2 in new node of d
26: **insert** the entries of R_1 and R_2 into parent of A_d
27: **else** ▷ *the tree in main memory is taller*
28: **save** m
29: **insert** the entry pointing to $d.root$ into the
leftmost node at $d.height$ of m
30: $d.root$ = root of the saved m
31: **save** $d.root$
32: **return** $d.root$

expressed relatively to the minimal quadrant that covers the internal node (each *Address* determines a subquadrant of this minimal quadrant). For example, in Figure 1, the *Address* 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.

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.

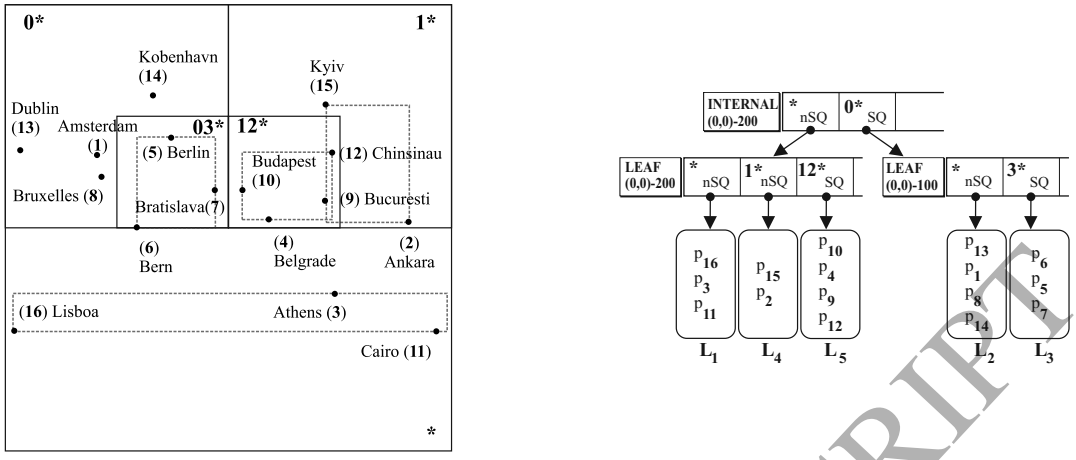

Figure 1: A collection of points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree.

3. Bulk-Loading xBR$^+$-tree (the PBL+ Algorithm)

In this section, we present the method, *PBL+*, that we developed for bulk-loading xBR$^+$-trees. This method consists of four phases.

3.1. Phase 1 (transformation of input file format)

This phase, that is rather technical, is necessary, since data are usually acquainted and stored in text format. Every application that uses such data has to transform them to a suitable format. It is necessary to transform the input file to binary format with fixed width records, for faster accessing and processing.

During this phase (Algorithm 1b), we split the input data file (whether initially in text, or binary format) in 2^d binary files (where d is the number of the space dimensions) (Algorithm 1b, Line 2). We transfer, in each of these files, data depending on the hyperquadrant (quadrant, if d is 2; octant, if d is 3; and so on) to which they belong, by comparing each coordinate of every point with the midpoint of the corresponding axis. Note that the reference space can be defined as wide as needed to satisfy the applications restrictions, however, it is finite. Next, we examine the size of each of the 2^d binary files created. If the file we examine contains an amount of data that is smaller than the predefined limit (*MemoryLimit*), meaning that the total memory required to create a complete xBR$^+$-tree in main memory is available, then we read the contents of this file in a memory block (Algorithm 1b, Line 7), and directly proceed to Phase 3 for this block (Algorithm 1b, Line 8). Otherwise, if the amount of data in this binary file remains large, we proceed to Phase 2 for this file (Algorithm 1b, Line 5).

3.2. Phase 2 (partitioning input data)

In this phase (Algorithm 2b), we group data so that the size of each group does not exceed the predefined *MemoryLimit* size and at the same time the data in a group belong to the same subquadrant.

Next, let's consider that our data are 2d points. As usual in technical texts, axis 0 will be called x axis and axis 1 will be called y axis. Assuming that the points are contained in file f and belong to one quadrant having side midpoints x_m and y_m, we separate them in four (2^2) temporary (helper) files one for each subquadrant (Algorithm 2b, Line 1). The temporary files (t_0, t_1, t_2 and t_3) are passed (from Phase 1) initially empty. Afterwards, we examine the cardinality of each file t_i (Algorithm 2b, Line 3) and if it is smaller than or equal to *MemoryLimit* we read the file t_i in a memory block and then we call the routine of Phase 3 for this memory block (Algorithm 2b, Line 1). Otherwise, we call recursively the current routine, passing as arguments the file t_i as input file and the temporary files (t_0, t_1, t_2 and t_3) which are not empty now (Algorithm 2b, Line 4). In each recursive call the input file is one of the t_i files. This file will be used as input for the continuous portion of t_i containing the points of the corresponding subquadrant and for output after the position of the last input point. Since the process is always executed in Depth-First mode, the points of a subquadrant in the file t_i are partitioned, while the points that wait for processing are not affected.

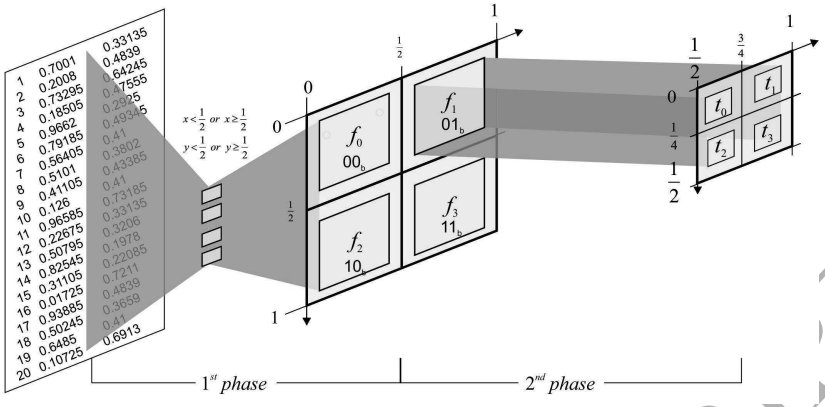

Figure 2: An input file in text format with points in 2^d space is divided in 2^2 binary files.

Note that, during the examination of the cardinality of each file t_i, and for the first of these files for which we call the routine of Phase 3, we pass as parameter values this group (contained in the file) and the whole region which was divided in the t_i files.

In this way, this group of points will belong to a parental region, and thus, we will have obtained better cohesion in the final xBR$^+$-tree that we construct. In these two phases (1 and 2) the bulk-loading algorithm sorts groups of data items that fit in *MemoryLimit* main memory size, according to z-order. This means that groups of points corresponding to subquadrants and at most contain points that fit in *MemoryLimit* size are reordered so that each subsequent group in this order contains points with z-order value larger than the z-order of the points in any preceding group. Note that, the order of points within each group is irrelevant.

3.3. Phase 3 (creation of the m-xBR$^+$-tree)

The third phase (Algorithm 3b) gets as input a group of points with a cardinality smaller than, or equal to *MemoryLimit* that is contained in a subregion of space where it is guaranteed that no other data of the dataset are contained. Instead of proceeding to the creation of a Quadtree in memory [12] that will be used as a guide for the creation of the respective *m*-xBR$^+$-tree in memory, we create one root node (that contains only the range of indexes of the whole group of points) and proceed as follows.

Part A: Creation of the lower level (leaves).

We start a recursive procedure (Algorithm 3b, Line 1) that looks similar, in its general structure, to the one of Phase 2, though it is more complex. In each call we try to reduce the extent of the reference space, so as, eventually the space under processing to consist of 4 subquadrants with cardinalities of points that each one does not exceed the maximum capacity (C) of a leaf of the xBR$^+$-tree.

Eventually, we will reach a parental node with none overflown child. In this case, we check for which of the child regions, which correspond to leaves, we can transfer their data to the parental region without causing overflow of the leaf corresponding the parental region. Afterwards, if there are non-empty children which can be merged without causing overflow of a leaf, then we merge them.

Returning to the previous level of routine calls, the previously overflown parental region is now a child one with acceptable cardinality of points. We continue processing as before, until all the leaves have been created and then we finish the process by saving them (Algorithm 3b, Line 2).

Part B: Creation of the levels above leaves.

The leaf records created by the previous process exist within the auxiliary tree T. Every node has a number of child nodes from zero up to 2^d and an equal number of pointers to the children. It has a field that holds the number of non empty descendant nodes. However, they are not organized in groups that could form nodes of the *m*-xBR$^+$-tree. To accomplish this, we execute the following procedure for each level of nodes (Algorithm 3b, Line 3).

In order to split an overflown xBR$^+$-tree node when we insert items one-by-one into it, the goal is to split it in two nodes with cardinalities of minimum possible difference [8, 9, 10]. If the tree T has a number of nodes which

correspond to non-empty leaves smaller than or equal to C then the whole tree can be packed into a single node of m-xBR$^+$-tree (Algorithm 3b, Line 7 in conjunction with the execution of Line 14 only once). Otherwise (Algorithm 3b, Line 5), the tree is scanned from top to bottom in Depth-First mode and the child-node with the maximum cardinality of non-empty leaves is chosen first. When a root of a sub-tree with cardinality up to C is found, a node of m-xBR$^+$-tree is formed having entries from the non-empty leaves of this sub-tree. By returning to the calling procedure of leaves creation, we have grouped all the leaves in the first level of the m-xBR$^+$-tree that is already created. If the number of nodes of the first level is greater than C then (Algorithm 3b, Line 5 is not executed) we repeat the procedure above (Algorithm 3b, Line 9) considering as leaf nodes the nodes of the m-xBR$^+$-tree of the previous level, but traversing the same tree T (Algorithm 3b, *while* loop in Lines 9-14). We continue creating levels of nodes of the m-xBR$^+$-tree until the number of nodes becomes smaller than or equal to C (, Line 12), the m-xBR$^+$-tree is completely created and Phase 3 for the current group is completed.

In Figure 3, a tree of 5 leaves of the lowest level-0 as an example is shown. The maximum capacity of the leaves and internal nodes is set to the value C=4. The nodes of the largest family of the leaves L_2 and L_3 (the region of L_2 overlaps the region of L_3) are pruned from the tree and one entry for this sub-tree of the higher level-1 (of internal nodes) is created. The rest of nodes of the tree are inserted into the head node of the level-0. The cardinality of the tree with leaf-nodes the nodes of the previous level-0 is 2, less than the maximum capacity, therefore the two entries can be packed in the root node of the m-xBR$^+$-tree.

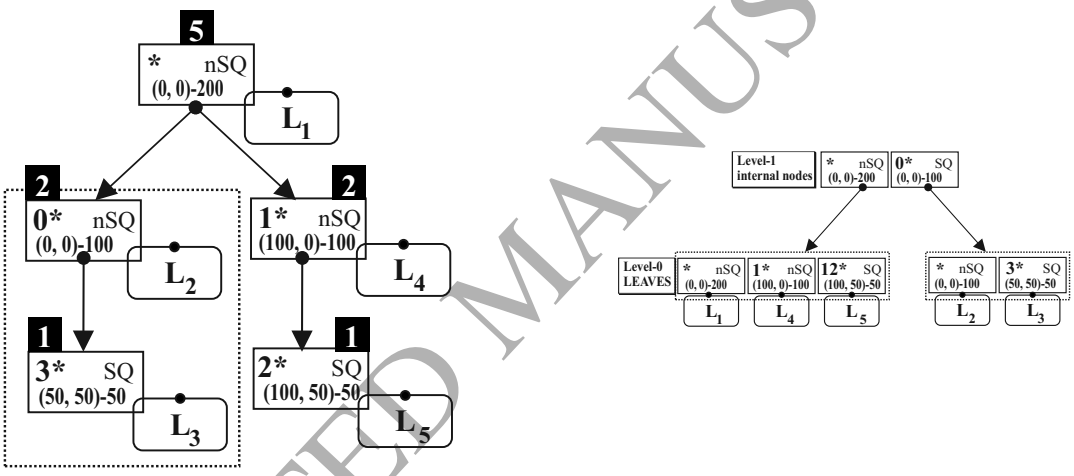

Figure 3: The tree of 5 entries for leaves (level 0) on the left, which are grouped in 2 entries for internal node of the next level 1, on the right.

3.4. Phase 4 (merging of trees)

During the last phase (Algorithm 4b), the m-xBR$^+$-tree created in main memory is merged with the xBR$^+$-tree already built in secondary memory (existing xBR$^+$-tree). This xBR$^+$-tree was created during the previous iterations of the bulk-loading process.

During the first execution of this phase, the xBR$^+$-tree in disk is empty (Algorithm 4b, Line 2), thus it is necessary to access all the nodes of the m-xBR$^+$-tree and store them in disk (Algorithm 4b, Line 3). During every subsequent execution of Phase 4, we initially examine the relation of heights of the two trees. Let h_m the height of the m-xBR$^+$-tree created during Phase 3 and h_d the height of the xBR$^+$-tree already in disk.

If $h_m = h_d$ (Algorithm 4b, Line 5), then, after saving the whole m-xBR$^+$-tree, except for its root, we examine the current occupancy of the two roots. Let N_m the occupancy of the root R_m of the m-xBR$^+$-tree and N_d the occupancy of the root R_d of the xBR$^+$-tree.

- If $N_m + N_d \leq C$ (Algorithm 4b, Line 7), then R_m is merged with R_d, copying all its entries to the empty slots of R_d (Algorithm 4b, Line 8). Of course, we examine if the insertion of a region affects the characterization as a rectangle, or not of the existing entries of R_d.

- If $N_m + N_d > C$ (Algorithm 4b, Line 9), then we proceed to a procedure of creating a new node (Algorithm 4b, Lines 10-15), N_{new}, capable of holding $N_m + N_d$ entries, in which we will copy all entries of two roots. Afterwards the overflown node N_{new} will be spit in two regions R_1, R_2 and this procedure will be continued by creating a new root node, R_{d-new}, since an increase of the height of the whole tree is necessary. The entries of regions R_1, R_2 are inserted in the new root node, R_{d-new}, that we subsequently store (Algorithm 4b, Lines 15 and 31).

In Figure 4, the xBR$^+$-tree, at the top of the left side, has height $h_d = 3$ and its root R_d has occupancy $N_d = 2$, similarly, at the bottom of the same side the m-xBR$^+$-tree has height $h_m = 3$ and its root R_m has occupancy $N_m = 2$. In order to merge the two trees with maximum capacity ($C = 4$) we, simply, add the entries M_1 and M_2 into the empty (two) slots in the root R_d and the resulting tree (on the right side) has height without changes $h_d = 3$.

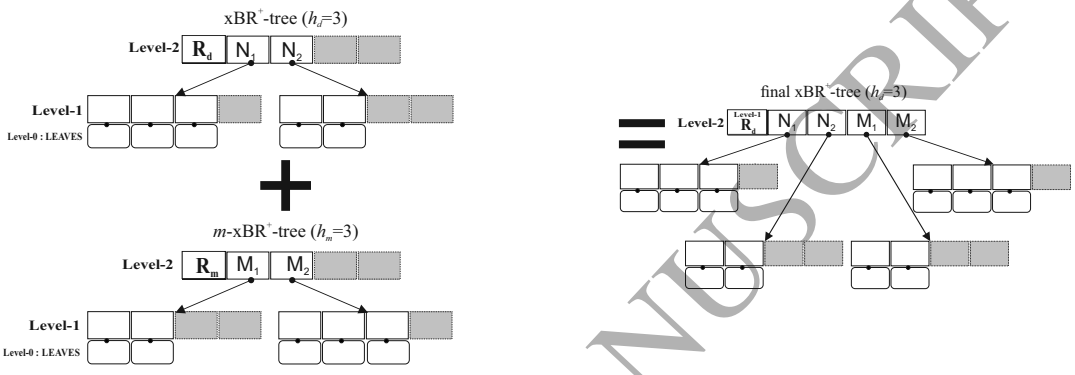

Figure 4: Merge xBR$^+$-tree and m-xBR$^+$-tree of equal height ($h_m = h_d = 3$) on the left, and the resulting xBR$^+$-tree of the same height, on the right.

In Figure 5 an xBR$^+$-tree, at the top of the left side, and an m-xBR$^+$-tree at the bottom of the same side are depicted. On the right side of the Figure 5 the resulting xBR$^+$-tree, after merging, is shown. In this case, the two trees have equal heights ($h_m = h_d = 3$) but the original xBR$^+$-tree has a root (R_d) with occupancy $N_d = 3$, while the root R_m of the m-xBR$^+$-tree has occupancy $N_m = 2$. Since both roots have total occupancy bigger than the maximum capacity ($C = 4$), we merge the two trees in a process of two steps. First we create an internal node having capacity $C_{max} = N_d + N_m$ enough to host all the 5 entries of both roots. Afterwards we split this overflowed node in two nodes, giving into the second one the maximum occupancy, in the example, shown in the figure, this occupancy is 3. The smaller number of entries (2) is added into the old root of the original xBR$^+$-tree. In the second step we create a new node for the new root (R_{d-new}) and two new entries are added. The first entry R_1 points at the old root R_d while the second one R_2 points at the new node created from the split of the overflowed node.

If $h_m < h_d$ (Algorithm 4b, Line 16), then, using a Depth-First traversal of the xBR$^+$-tree (like in a Point Location Query) and starting from R_d, we seek for the region that is the closest ancestor of the region of R_m (Algorithm 4b, Line 17). The search stops at level $h_m + 1$. We mark the pointer to the node of the region A_d that is the closest ancestor (i.e. totally contains) of the region of R_m and resides at level h_m. We also mark the node P_d containing this pointer (the parental node of A_d). {Subsequently, the whole m-xBR$^+$-tree will be stored, except for its root (Algorithm 4b, Line 18). If N_m is the occupancy of R_m and N_d the occupancy of A_d then:

- If $N_m + N_d \leq C$ (Algorithm 4b, Line 19), the root R_m is merged with A_d, by copying its entries to the empty slots of A_d {(Algorithm 4b, Line 20). Of course, we examine if the insertion of a region affects the characterization as a rectangle, or not of the existing entries of A_d.

- If $N_m + N_d > C$ (Algorithm 4b, Line 21), we proceed to a procedure of creating a new node, N_{new}, capable of holding $N_m + N_d$ entries, in which we will copy all entries of two roots. This overflown node is spit in two regions R_1, R_2 and this procedure is continued by storing the updated node A_d with the entries of R_1 and a node with the entries of R_2, and by saving the entries of regions R_1, R_2 in the parent node P_d. Since P_d may overflow (if it already contains C entries), the insertion of the new entry (for R_2) is done following the insertion

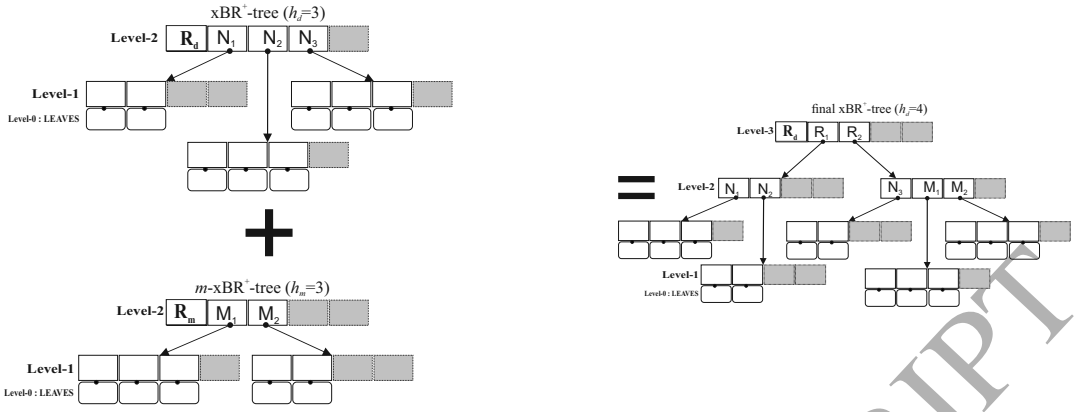

Figure 5: Merge xBR$^+$-tree and m-xBR$^+$-tree of equal height ($h_m = h_d = 3$) on the left, and the resulting xBR$^+$-tree with bigger height, on the right.

to an internal node that is described in [8], but without creating a temporary Quadtree for deciding about the way the overflown node is divided. In this work, we followed a procedure similar to part B of Phase 3 and more specifically the procedure where we seek for the sequence of regions with a cardinality closest to a target occupancy.

In Figure 6, the xBR$^+$-tree, at the top of the left side, has height $h_d = 3$ and its root R_d has occupancy $N_d = 2$, similarly, at the bottom of the same side the m-xBR$^+$-tree has height $h_m = 2$ and its root R_m has occupancy $N_m = 2$. In order to merge the two trees with maximum capacity ($C = 4$) we scan the node R_d and the second entry A_d corresponds to a region that overlaps the region of the root R_m. The occupancy of the node that points the entry A_d is two and the free slots in this node are enough to host the entries of the root R_m. In the right side of the Figure 6 the final xBR$^+$-tree is shown. The entries M_1 and M_2 of the root R_m are added into the node pointed from the entry A_d.

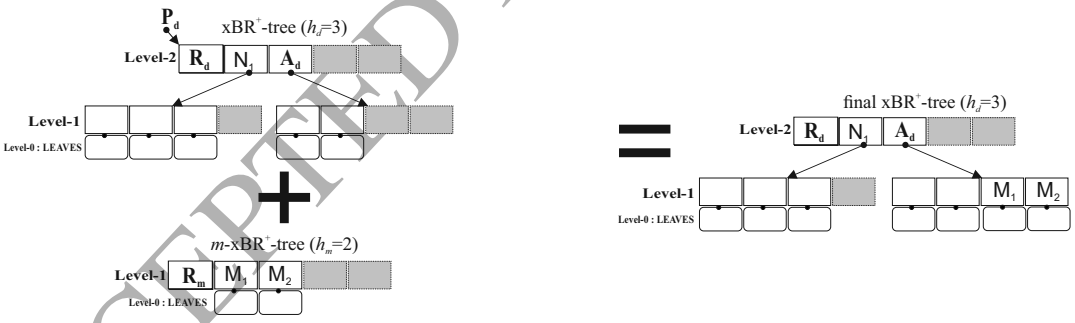

Figure 6: Merge xBR$^+$-tree ($h_d = 3$) with m-xBR$^+$-tree ($h_m = 2$) on the left, and the resulting xBR$^+$-tree of the same height ($h_d = 3$), on the right.

In Figure 7, an xBR$^+$-tree, at the top of the left side, and an m-xBR$^+$-tree at the bottom of the same side are depicted. On the right side of the Figure 7 the resulting xBR$^+$-tree, after merging, is shown. In this case the original xBR$^+$-tree is taller ($h_d = 3$) than the m-xBR$^+$-tree ($h_m = 2$). The root R_d has occupancy $N_d = 2$, while the root R_m of the m-xBR$^+$-tree has occupancy $N_m = 3$. Since both roots have total occupancy bigger than the maximum capacity ($C = 4$), we merge the two trees in a process of two steps. First we create an internal node having capacity $C_{max} = N_d + N_m$ enough to host all the 5 entries of both nodes A_d and R_m. Afterwards we split this overflowed node in two nodes, giving into the second one the maximum occupancy, in the example, shown in the figure, this occupancy is 3. The smaller number of entries (2) is added into the old node A_d of the original xBR$^+$-tree. In the second step we create a new entry for the R_m and it is added into the parent of the A_d node (P_d). The entry A_d is updated if entries of the child moved into the new node created from the split of the overflowed node.

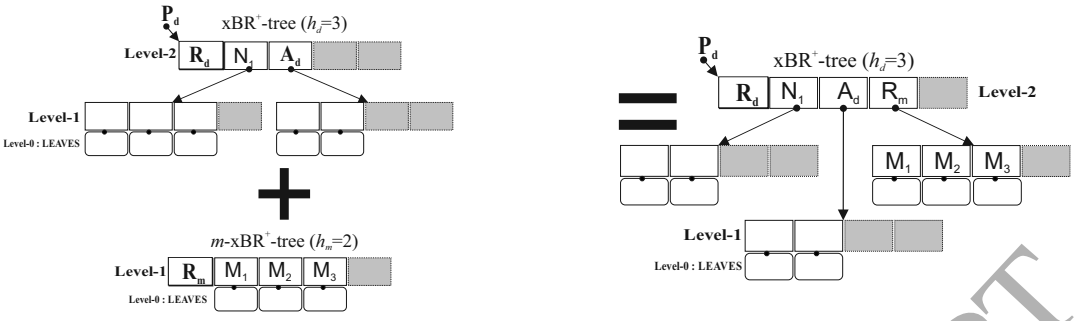

Figure 7: Merge xBR$^+$-tree and m-xBR$^+$-tree of equal height ($h_m = h_d = 3$) on the left, and the resulting xBR$^+$-tree with bigger height, on the right.

Last, if $h_m > h_d$ (Algorithm 4b, Line 27), we save all the nodes of the m-xBR$^+$-tree (Algorithm 4b, Line 28). The root of the m-xBR$^+$-tree becomes the root of the xBR$^+$-tree. Afterwards, starting from the new root, we follow the left-most path in order to find the node of the $h_d + 1$ level. Then, we add a new entry that corresponds to the region of the old root (Algorithm 4b, Line 29), and the xBR$^+$-tree is updated with its new root (Algorithm 4b, Line 30).

In Figure 8, the xBR$^+$-tree, at the top of the left side, has height $h_d = 2$, and is shorter than the m-xBR$^+$-tree at the bottom of the same side has height $h_m = 3$. On the right side of the Figure 8 the resulting xBR$^+$-tree, after merging, is shown having height $h_d = 3$. Since the root R_d corresponds to all defined space so overlaps the region of the R_m which however is at higher level than the R_d root. First, we set the root R_m as new root of the merged xBR$^+$-tree. We shift the entries of the R_m one place on the right in order to leave free slot to host the entry that points to the R_d node.

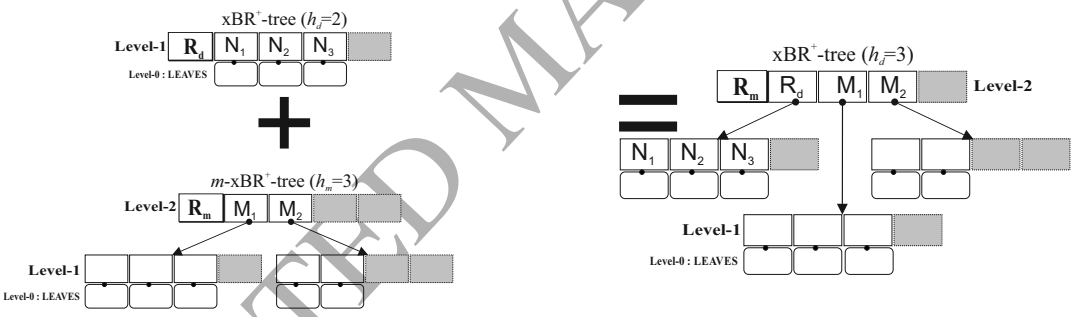

Figure 8: Merge xBR$^+$-tree ($h_d = 2$) with taller m-xBR$^+$-tree ($h_m = 3$) on the left, and the resulting xBR$^+$-tree of height $h_d = 3$, on the right.

Note that the four phases are pipelined (Phase 1 produces an output file as input to Phase 2 that produces a block as input to Phase 3 that produces an m-xBR$^+$-tree as input to Phase 4). Note also that partitioning data items in a regular (Quadtree-like) fashion is a way of two dimensional sorting in z-order. Moreover, m-xBR$^+$-trees are built bottom-up. Therefore, we characterize our technique as a bottom-up, sort-based like approach for bulk-loading the xBR$^+$-tree.

4. Experimental Results

We designed and run a large set of experiments to compare the Process of Bulk-Loading xBR$^+$-trees (*PBL*) of [12], the Process of Bulk-Loading xBR$^+$-trees as presented in Section 3 (*PBL+*) and the process of Bulk-Loading R-trees (*STR*) as published in [13]. The *STR* algorithm is a very popular and fast, frequently used as base line, algorithm for Bulk-Loading comparisons. We used 6 real spatial datasets of North America, representing cultural landmarks (NAclN with 9203 points) and populated places (NAppN with 24491 points), roads (NArdN with 569082 line-segments) and rail-roads (NArrN with 191558 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| =

569082 points, |NArrN| = 191558 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| = 1138188 points, |NArrND| = 383180 points). The data of these 6 files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 125000, 250000, 500000 and 1000000 points, with 125 clusters in each dataset (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. Finally, we have also used three big real datasets[1] 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 5836360 line-segments, world parks or green areas (Park) consisting of 11503925 polygons, and world buildings (Build) consisting of 114736539 polygons. To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park and Build. The experiments were run on a Linux machine, with Intel core duo 2x2.8 GHz processor and 4 GB of RAM.

We run experiments for tree building, counting tree characteristics and creation time. We also run experiments for several single dataset spatial queries (*PLQ*,*WQ*, *DRQ*, *KNNQ*, *CKNNQ*) and for two dual dataset spatial queries (*KCPQ* and ε*DJQ*), counting disk read accesses (I/O) and total execution time.

4.1. Experiments for tree building

In [12] it is shown that, using *PBL* bulk-loading algorithm, the xBR$^+$-tree structure is built with good quality characteristics, if a *MemoryLimit* bigger than 1% of the size of dataset is set. Therefore in this paper we selected to create indexes of xBR$^+$-tree with *MemoryLimit* equal to 2% of the size of dataset. On the other hand, the *STR* algorithm was given an equal size of main memory to the memory size of the *PBL+* algorithm.

In Table 1, for *PBL*, we present for five indicative datasets (three big and one smaller real datasets and one synthetic dataset) the effect of the node size (*size*) to the tree characteristics: tree height (H), internal nodes occupancy percentage (Iocc), leaf nodes occupancy percentage (Locc), size of the data memory in main memory (M.Size), size of the tree in disk (D.Size) and the total creation time of the tree (Time), for all three algorithms. The conclusions for the rest of the datasets are analogous.

Table 1: Tree creation characteristics, using the *PBL*, *PBL+* and the *STR*.

size (KB)	Alg	H	Int Nodes	Iocc (%)	Leaf Nodes	Locc (%)	M.Size (MB)	D.Size (MB)	Time (s)
Dataset : NArd									
1	*PBL*	6	2683	68.3	33967	67.0	1.71	35.79	0.719
	PBL+	5	2529	68.6	32182	70.7	0.83	33.90	0.748
	STR	5	1357	99.9	27100	100	0.78	27.79	0.950
Dataset : 1000KC									
2	*PBL*	5	1223	64.1	30151	65.0	2.36	61.28	1.808
	PBL+	5	1087	66.5	27820	70.5	1.38	56.46	0.995
	STR	4	582	99.8	23810	100	1.38	47.64	1.729
Dataset : Water									
4	*PBL*	4	1624	67.2	87917	65.1	13.98	349.77	13.59
	PBL+	4	1484	69.1	82603	69.3	7.27	328.46	13.14
	STR	4	819	99.8	68664	100	7.27	271.42	24.84
Dataset : Park									
8	*PBL*	4	822	65.9	89089	63.3	36.50	702.43	35.44
	PBL+	4	730	68.4	82140	68.7	14.26	647.42	34.92
	STR	4	403	99.4	67671	100	14.26	531.83	51.41
Dataset : Build									
16	*PBL*	4	1979	70.1	460038	61.0	435.44	7219.0	1373.3
	PBL+	4	1771	70.2	412428	68.0	138.86	6471.9	678.4
	STR	4	991	99.9	336471	100	138.86	5272.8	992.1

Comparing *PBL* to *PBL+* based on this table, we observe the following:

[1]Retrieved from http://spatialhadoop.cs.umn.edu/datasets.html.

- The trees have almost equal heights. If the node size is greater than 2KB then this is true for all datasets.
- *PBL+* builds more compact indexes with smaller number of internal nodes (equal or larger internal nodes occupancy) than the *PBL*, for all node sizes and all datasets.
- Additionally, the *PBL+* builds more compact indexes with smaller number of leaf nodes and larger leaf nodes occupancy (close to 70%) than the *PBL*, for all node sizes and all datasets.
- Because of these two properties (Iocc and Locc), the trees created with *PBL+* have a smaller overall size than the corresponding trees created with *PBL* on disk.
- The *PBL+* needs less memory space for data than the *PBL*, for all node sizes and all datasets. This difference is maximized in big datasets, having a range of 48-65% for all sizes of nodes.
- The time that the *PBL+* takes to create the tree is almost equal or smaller than the *PBL*. Only for the biggest dataset (*Build*) there is a large difference but this is mainly due to I/O.

Comparing the *PBL+* to the *STR* based again on Table 1, we observe the following:

- The trees have almost equal height. As expected for the *STR*, the index is shorter because of the best occupancy of internal and leaf nodes.
- The *STR* builds the most compact indexes with the smallest number of internal nodes (almost 100% of internal nodes occupancy).
- Additionally the *STR* builds the most compact indexes with the smallest number of leaf nodes occupancy (up to 100% especially for the big datasets).
- Because of these two properties of the trees created by *STR* have the smallest overall size than the corresponding trees created with *PBL+* on disk. Note that the R-trees built with *STR* algorithm appear as ideal for storing static data. But if it is necessary to add additional data, they are forced to split their already full nodes. This results to the typical for R-trees nodes occupancy (in a range of 65-70%). On the other hand, there is no reduction of the degree of overlap, especially in the internal nodes.
- The time that the *STR* takes to create the tree is larger than the *PBL/PBL+*, in all cases of datasets or node sizes.

We implemented the *STR* algorithm in a way to ensure fair comparison conditions. We chose to implement the sorting of each data block required at each stage of *STR* according to the size of this block. If the data block was larger than a predefined limit, we used external merge sort with several buffers. If the data block was smaller than this limit, then we transferred it to main memory and applied quick sort. This limit was defined so that an equal size of main memory was used for *STR* and *PBL+*, for the each dataset and node size. So we tried to speed up the *STR* algorithm and reduce the disk activity, which is a retarding factor in execution time.

The way to decide the boundaries of each data block that forms a leaf or a set of records of an internal node between *PBL/PBL+* and *STR* algorithms has a similar time cost. In all cases, simple arithmetical operations are needed. Therefore, the main factor of the difference in creation time between the xBR$^+$-tree and R-tree is the difference of the disk activity and the movements of data in main memory. In *PBL/PBL+* algorithms, sorting continues down to MemoryLimit / disk page size resolution. In *STR* algorithm, sorting continues down to single item resolution. This difference is the main factor for the faster tree creation by *PBL/PBL+* algorithms.

Table 2 presents an overall comparison of the three bulk-loading methods with respect to the six performance measures of the tree creation, as listed in the leftmost column. For each row in this table, an entry with value "**1**" indicates that the corresponding bulk-loading method performs the best regarding the corresponding measure, while an interval "[a, b]" in other entries indicates that the corresponding method is *a* to *b* times more expensive than the best one.

For example, the second data row summarizes the *Iocc* measure for all methods. We can see that the *STR* has a value "**1**", which indicates that it outperforms the other methods. As for *PBL+*, the interval "[0.297, 0.333]" indicates that the internal node occupancy percentage (Iocc) of *PBL+* is 29.7% less, in the best case, and 33.3% less *Iocc*, in the

worst case, than the one of *STR*. The sixth data row shows the creation time. We can observe that *PBL+* outperforms the other methods; for *STR*, the interval "[1.2, 1.9]" indicates that the creation time of *STR* is about 1.2 times larger, in the best case, and about 1.9 times larger, in the worst case, than that of *PBL+*. Therefore, we can deduce that smaller endpoint numbers and a smaller range of the interval indicate a better and more stable performance.

A general conclusion arising from the Table 2 is that all trees have almost the same height, *STR* builds more compact trees and consumes less resources (memory and disk) and, *PBL+* is the fastest bulk-loading method.

Table 2: Performance summary for tree creation characteristics.

Performance measure	*PBL*	*PBL+*	*STR*
Height	**1** (except 1 & 2KB)	**1** (except 2KB)	**1**
Iocc	[0.298, 0.357]	[0.297, 0.333]	**1**
Locc	[0.33, 0.39]	[0.293, 0.32]	**1**
Mem Size	[1.7, 3.1]	**1** (except 2KB)	**1**
Disk Size	[1.3, 1.4]	[1.1, 1.2]	**1**
Time	[1, 2]	**1**	[1.2, 1.9]

4.2. Experiments for single dataset spatial queries

For each of the single data queries we executed 65 experiments (13 datasets × 5 node sizes). Each experiment was performed several times so that the input data scanned all the specified reference space. In order to examine the efficiency of each structure in each query, we created different sets of input query windows. Every set of query windows consisted of equal squares that were distributed throughout the whole space without overlapping and was created as follows. To create a set of 2^{2k} query windows, the space [0,1] was divided in $\sqrt{2^{2k}}$ equal segments in each dimension. Within each cell of the created grid, one square query window with side equal to $\frac{1}{3}$ of this cell side was positioned in a random position. When the query input consisted of reference points and not windows, we used the centroids of these query windows. The query input data for each experiment was given in a random order.

Since the tree created by *PBL+* is built faster and has superior characteristics (less internal nodes and less leaves of higher occupancy, thus, processing of queries will access less nodes performing similar calculations and be faster) to the tree created by the *PBL*, therefore to study the performance of query processing we will only use the tree created by *PBL+* vs. the tree created by *STR*.

For *PLQs*, we executed two sets of experiments. In the first set, we used as query input the nearest point belonging in the dataset to each of the $2^{14} = 16384$ query points (existing points). When searching for existing points the number of disk accesses in xBR$^+$-trees is equal to their height while in R-tree that number may be larger than the height of the tree. Since the height of the R-tree is smaller than or equal to the height of the xBR$^+$-tree, the number of disk accesses per query in that experiment, searching for 2^{14} points, may be smaller for any of the two compared structures.

This set of experiments is summarized in the first two data rows of Table 3. In this table, for each query, regarding disk read accesses and execution time, we present percentages (%) of the experimental cases where trees created by the two bulk-loading processes perform equivalently (data columns 1 and 4, respectively) and where trees created by *PBL+* / *STR* have a performance that is more than 5% better than their rival (data columns 2 and 5 / 3 and 6, respectively). It is evident that, for this set of experiments, regarding the number of disk accesses both types of trees perform almost equivalently in the most cases (60%).

- In 22% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 14% of the cases the xBR$^+$-tree reported less I/O activity. In over half of the cases (64%) both type of trees needed almost an equal average number ($\leq$ 5%) of disk accesses.

- In big datasets there was a tie in half of the cases (46.7%) while in the rest of them (53.3%) the xBR$^+$-tree was the winner.

The results of total time needed for the execution of each experiment pointed out that the xBR$^+$-tree created by *PBL+* was the big winner (76.9% of the cases).

- In small datasets, 20% of the cases showed the R-tree needed a smaller average time, while in 72% of the cases the xBR$^+$-tree was faster. In a small number of cases (8%) both types of trees needed an almost equal average execution time ($\leq$ 5%).
- In big datasets there was a tie in only one case (6.7%) while in the rest of them (93.3%) the xBR$^+$-tree was the winner.

Table 3: Percentages of cases of Disk Accesses and Execution Time winners in *PLQs*.

PLQ	Size of data set	Number of Disk Accesses			Execution Time		
		tie d$\leq$ 5%	*PBL+* wins d> 5%	*STR* wins d> 5%	tie d$\leq$ 5%	*PBL+* wins d> 5%	*STR* wins d> 5%
Existing points	Small	64.0	14.0	22.0	8.00	72.0	20.0
	Big	46.7	53.3	0.00	6.67	93.3	0.00
Non-existing points	Small	22.0	60.0	18.0	8.00	80.0	12.0
	Big	20.0	73.3	6.7	0.00	86.7	13.3

In the second set of experiments for *PLQs*, we used as query input $2^{14} = 16384$ query points (non-existing points). While searching for non existing points in the dataset, the disk accesses may be less than the tree-height for both trees (due to *DBRs* of the xBR$^+$-trees and *MBRs* of the R-trees). This set of experiments is summarized in the 2nd group of data rows (3 and 4) of Table 3. It is clear that trees created by *PBL+*, on the average, perform better in both metrics for all datasets. Regarding the number of disk accesses the xBR$^+$-tree performed better in most cases (63.1%).

- In small datasets, 18% of the cases showed that the R-tree needed a smaller average number of disk accesses, while in 60% of the cases the xBR$^+$-tree required less disk accesses. In the rest of the cases (22%) both types of trees needed an almost equal average number ($\leq$ 5%) of disk accesses.
- In big datasets there was a tie in 20% of the cases. In a smaller number of cases (6.7%) the R-tree needed a smaller number of disk accesses, while in the rest of them (73.3%) the xBR$^+$-tree was the winner.

The results of total time needed for the execution of each experiment pointed out that the xBR$^+$-tree created by *PBL+* was the big winner (81.5% of the cases).

- In small datasets there were 12% of cases in which the R-tree needed smaller average time, while in 80% of cases the xBR$^+$-tree was faster. In a small number of cases (8%) both types of trees needed an almost equal average execution time ($\leq$ 5%).
- In big datasets there was a tie only in two cases (13.3%), while in the rest of them (86.7%) the xBR$^+$-tree was the winner.

For *WQs*, we executed five sets of experiments using as query input one set of $2^6, 2^8, 2^{10}, 2^{12}$ and 2^{14} query windows. This query performed using a Depth-First scan of the trees, and there is only one criterion in order to visit the child node pointed by the current entry of the current root under consideration, and that is if the query window has intersection with the current region. Thus, as the number of query windows becomes larger the performance of xBR$^+$-tree vs. R-tree tends to approach that of the above *PLQs*.

The experiments of *WQ* are summarized in the bar charts of Figure 9. The left column of charts refers to small datasets, while the right one to big datasets. The percentage of the number of experimental cases where the two bulk-loading processes perform equivalently (tie), *PBL+* and *STR* has a performance that is more than 5% better than its

rival is depicted by empty, gray and black bars, respectively, regarding disc accesses (top row of charts) and execution time (bottom row of charts). It is evident that, for the small datasets, regarding the number of disk accesses, the xBR$^+$-tree created by *PBL+* performs better than the R-tree created by *STR* algorithm in 4/5 of the sizes of query windows. Additionally, we observe that the performance of the xBR$^+$-tree is improved as the extents of the query windows are reduced (increasing the number of query windows). Instead, we observe that for big datasets the R-tree outperforms the xBR$^+$-tree in 4/5 of the sizes of query windows. However, we observe that the trend of improving the performance of xBR$^+$-tree as the extents of the query windows are reduced appears again.

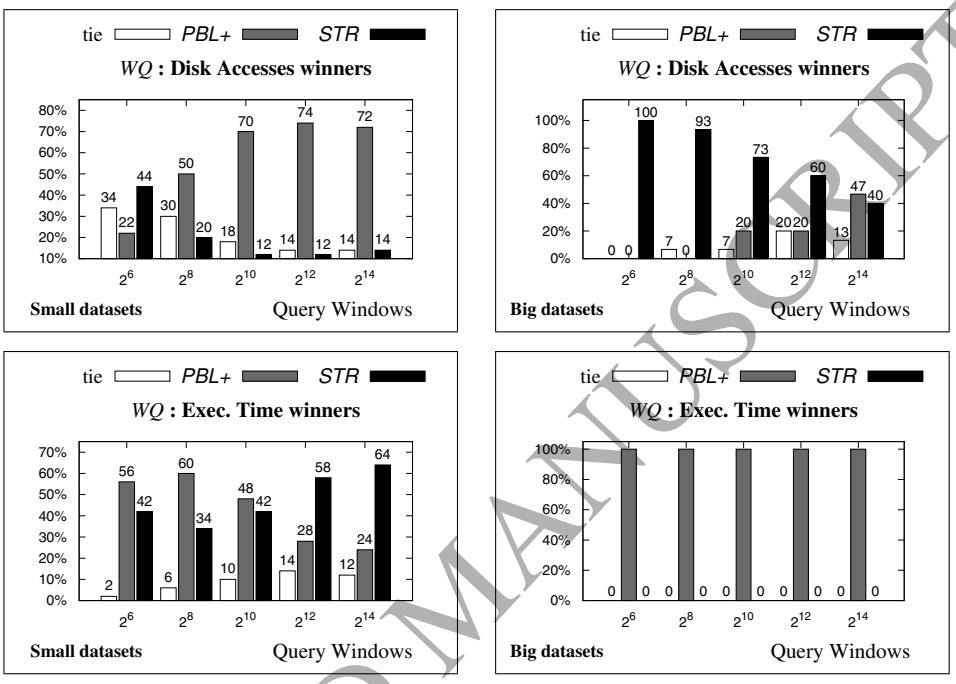

Figure 9: Percentages of cases of Disk Accesses and Execution Time winners in *WQs*.

On the average of all sets of queries, the two trees performed as follows.

- In 20.4% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 57.6% of the cases the xBR$^+$-tree required less I/O activity. In the remaining cases (22%) both type of trees needed almost an equal average number ($\leq 5\%$) of disk accesses.
- In big datasets there was a tie in 9.3% of cases, xBR$^+$-tree was the winner in 17.3% of cases, while in the rest of them (73.3%) the R-tree was the winner. Note the percentage of 47% of the xBR$^+$-tree for the set of the smallest query windows (2^{14}) vs. the 40% of the R-tree.

The results of total time needed for the execution of each experiment pointed out that the xBR$^+$-tree was faster in small datasets in fewer cases than the R-tree, while in the big datasets xBR$^+$-tree won in all cases. On the average of all sets, regarding execution time, the two trees performed as follows.

- The R-tree was a winner for the small datasets in more cases (48%) than the xBR$^+$-tree (43.2% of the cases).
- In big datasets the xBR$^+$-tree was the big winner (won in all the sets of query windows).

For *DRQs*, we executed five sets of experiments using as query input one set of $2^6, 2^8, 2^{10}, 2^{12}$ and 2^{14} query ranges. Each query range of a set of 2^{2k} query ranges was centered at the centroid of a query window (used in WQs) and defined by the radius of the incircle of this query window ($\varepsilon \leq (\frac{1}{2} \times \frac{1}{3} \times \frac{1}{\sqrt{2^{2k}}})$). Additionally, five algorithms, four versions of Depth-First (DF) and one version of Best-First (BF) were tested in 325 experiments (13 datasets $\times$ 5

node sizes × 5 sets of query circles), each. The xBR$^+$-tree created with *PBL+* algorithm responded best (in 118/325 of the cases) using the HDF algorithm (this algorithm uses a local minimum binary heap, prioritized by the minimum distance between the query point and *DBRs* to define the order of processing in Depth-First traversal).

The experiments of *DRQ* are depicted in the bar charts of Figure 10, following the same layout as in Figure 9. It is evident that, for the small datasets, regarding the number of disk accesses the xBR$^+$-tree performs better than the R-tree created by *STR* algorithm in 4/5 of the set of query ranges. Additionally we observe that the performance of the xBR$^+$-tree is improved as the distance ranges are reduced (increasing the number of query ranges). Instead, we observe that for big datasets the R-tree outperforms the xBR$^+$-tree in all set of query ranges. However, we observe that the trend of improving the performance of xBR$^+$-tree, as the extents of the query ranges are reduced, appears again.

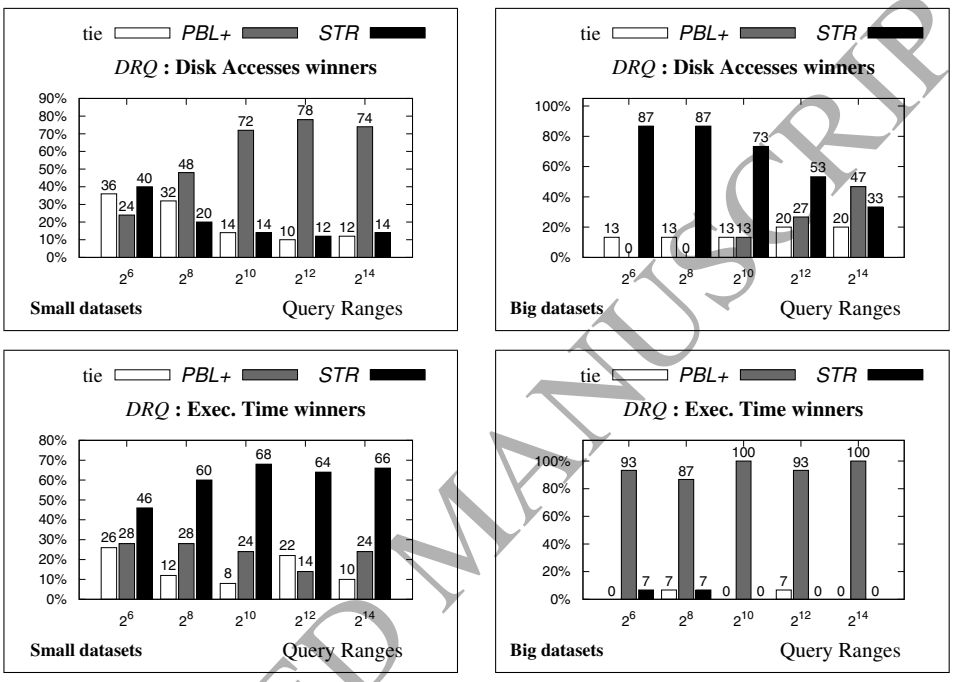

Figure 10: Percentages of cases of Disk Accesses and Execution Time winners in *DRQs*.

On the average of all sets of queries the two trees performed as follows.

- In 20% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 59.2% of the cases the xBR$^+$-tree required less I/O activity. In the remaining cases (20.8%) both type of trees needed almost an equal average number ($\leq 5\%$) of disk accesses.
- In big datasets there was a tie in 16.0% of cases, xBR$^+$-tree was the winner in 17.3% of cases, while in most of the rest of cases (66.7%) the R-tree was the winner. Note the percentage of 46.7% of the xBR$^+$-tree for the set of the smallest query windows (2^{14}) vs. 33.3% of the R-tree.

On the average of all sets, the results of total time needed for the execution of each experiment pointed out that the R-tree was fastest in small datasets (61.6% of the cases), while in the big datasets xBR$^+$-tree won in 97.3% of the cases.

Note that although *WQs* and *DRQs* have a similar description, processing of queries is different. Processing of *WQs* follows an overlap comparison of the item regions of each node with the query window and wherever an intersection is discovered the pointer to the respective child node is followed. Processing of *DRQs* is based on calculation of distances either between a point and a rectangle, or between points and the structure of xBR$^+$-trees, that does not have overlapping between nodes, is best suited to these calculations. For both queries, for specific datasets, it is possible to have a reverse behavior of the two performance metrics. That is, for a specific dataset, the number of disk accesses

may be larger (worse) for one of the two trees, while the execution time is better for this tree. This may arise when CPU processing of nodes of this tree is faster and dominates execution time. This phenomenon can explain that, for *WQs*, the execution time winner for big datasets is the xBR$^+$-tree, although the R-tree has smaller I/O. In general, the partitioning of data items, the number of disk accesses and CPU processing of nodes are the factors that determine the execution time winner.

For *KNNQs*, we executed four sets of experiments using as query input one set of 2^{12} query points and searching for K-values of $1, 10, 100, 1000$ of nearest neighbors. Additionally, five algorithms, four versions of Depth-First (DF) and one version of Best-First (BF) were tested in a project of 260 experiments (13 datasets $\times$ 5 node sizes $\times$ 4 (K-values or sets of query points)), each. Since both trees responded best (xBR$^+$-tree in 142/260 of the cases, R-tree in 210/260 of the cases) using the BF algorithm, Therefore, the performance comparison was based on BF algorithm.

The experimental results of *KNNQ*, are depicted in the bar charts of Figure 11, following the same layout as in Figure 9. It is evident that, for the small datasets, regarding the number of disk accesses the xBR$^+$-tree performs better than the R-tree created by *STR* algorithm in 2/4 of the K-values. Additionally we observe that the performance of the xBR$^+$-tree is better for $K < 10^3$, while for bigger K-values, the number of Disk Accesses is smaller for the R-tree. Instead, we observe that for big datasets the xBR$^+$-tree outperforms the R-tree in all K-values.

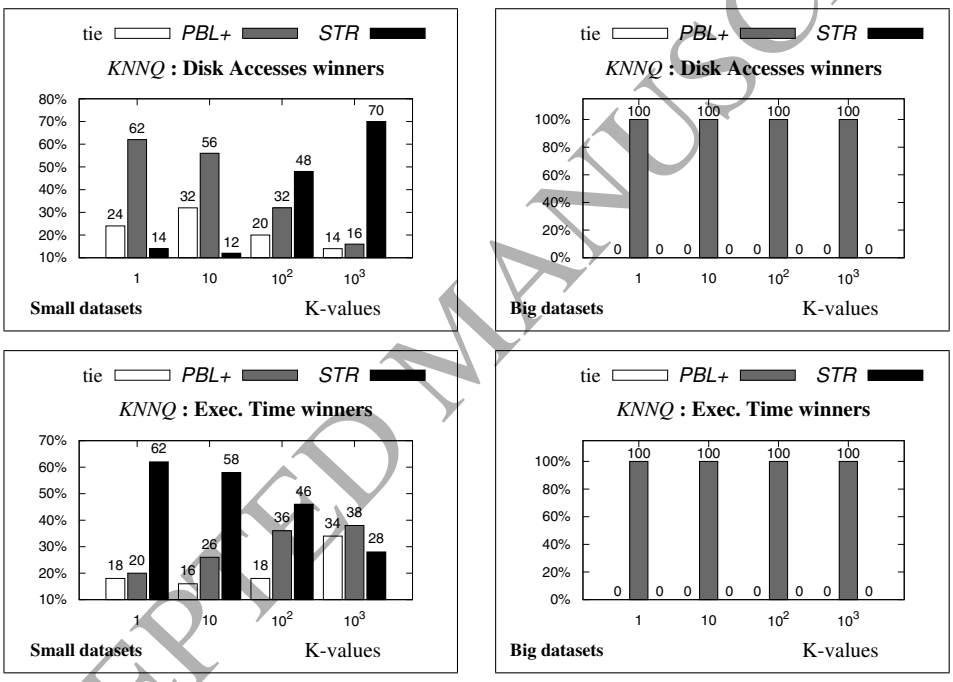

Figure 11: Percentages of cases of Disk Accesses and Execution Time winners in *KNNQs*.

On the average of all sets of queries the two trees performed as follows.

- In 36% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 41.5% of the cases the xBR$^+$-tree required less cost of read operations. In the remaining cases (22.5%) both type of trees needed almost an equal average number ($\leq 5\%$) of disk accesses.
- For big datasets and in all cases (100%) of the xBR$^+$-tree was the absolute winner.

Regarding the execution time, for small datasets, the R-tree was faster than the xBR$^+$-tree in the most cases with percentages which reduced as the K-values increased (62%, 58%, 46%, 28% of the cases). This decrease was partly absorbed by the xBR$^+$-tree (20%, 26%, 36%, 38% of the cases) and the other in tie (18%, 16%, 18%, 34% of the cases). Instead, we observe that for big datasets the xBR$^+$-tree outperforms the R-tree for all K-values.

On the average of all K-values the two trees performed as follows.

- In 48.5% of the cases of small datasets the R-tree needed less execution time, while in 30% of the cases the xBR$^+$-tree was fastest. In the remaining cases (21.5%) the two types of trees needed almost an equal execution time ($\leq 5\%$).

- In big datasets and in all cases (100%) the xBR$^+$-tree was the absolute winner.

For *CKNNQs*, we executed four sets of experiments using as query input one set of 2^{12} query points searching for K-values of $1, 10, 100, 1000$ of nearest neighbors having distances not larger than $\frac{1}{3} \times \frac{1}{2} \times \frac{1}{\sqrt{2^{12}}}$. Additionally, five algorithms, four versions of Depth-First (DF) and one version of Best-First (BF) were tested in a project of 260 experiments (13 datasets $\times$ 5 node sizes $\times$ 4 K-values or sets of query points), each. Since both trees responded best (xBR$^+$-tree in 86/260 of the cases, R-tree in 104/260 of the cases) using the BF algorithm, therefore, the performance comparison was based on BF algorithm.

The experimental results of *CKNNQ* are depicted in the bar charts of Figure 12, following the same layout as in Figure 9. It is evident that, for all experiments and for all datasets (small and big), regarding the number of disk accesses the xBR$^+$-tree performs better than the R-tree created by *STR* algorithm.

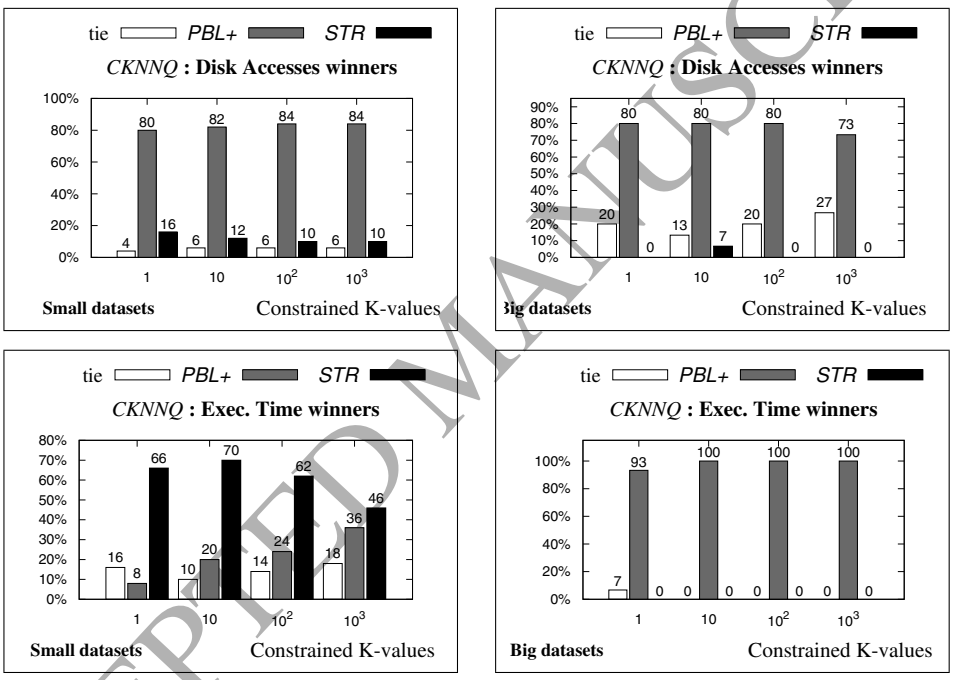

Figure 12: Percentages of cases of Disk Accesses and Execution Time winners in *CKNNQs*.

On the average of all constrained K-values the two trees performed as follows.

- In 12% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 82.5% of the cases the xBR$^+$-tree required less I/O operations. In the remaining cases (5.5%) both type of trees needed almost an equal average number ($\leq 5\%$) of disk accesses.

- In big datasets, in 1.7% of the cases of small datasets the R-tree needed a smaller average number of disk accesses, while in 78.3% of the cases the xBR$^+$-tree needed a smaller number of disk accesses. In the remaining cases (20%) both type of trees needed almost an equal average number ($\leq 5\%$) of disk accesses.

Regarding the execution time, for small datasets, the R-tree was faster than the xBR$^+$-tree in all the cases with percentages which reduced as the constrained K-values increased (66%, 70%, 62%, 46% of the cases). This decrease was partly absorbed by the xBR$^+$-tree (18%, 20%, 24%, 36% of the cases) and the other in tie (16%, 10%, 14%,

18% of the cases). Instead, we observe that for big datasets the xBR$^+$-tree outperforms the R-tree in all constrained K-values with percentages greater than 93%.

On the average of all K-values the two trees performed as follows.

- In 61% of the cases of small datasets the R-tree needed less execution, while in 24.5% of the cases the xBR$^+$-tree was the fastest. In a few number of the cases (14.5%) both type of trees needed almost an equal execution time ($\leq$ 5%).
- In big datasets and in almost all cases (98.3%) the xBR$^+$-tree was the winner.

Similarly to Table 2, Table 4 presents an overall comparison of the two bulk-loading methods (*PLB+* and *STR*) with respect to the five spatial queries when one index is used. The performance measures that we have taken into account are I/O activity (number of disk accesses) and total execution time (ET) in seconds, and we have also considered the behavior of these queries for *Small* and *Big* datasets.

In this table, for a specific query performance measure, dataset size and bulk-loading method, the interval "[a, b]" indicates the minimum (a) and the maximum (b) values of percentages of cases in the different sets of experiments performed where this method exhibited a better value for this performance measure with respect to the best bulk-loading method (indicated by "1"). For example, the interval [12%, 44%] at the third data column and second data row of this table indicates that the percentages of cases in the different sets of experiments performed for *WQ* I/O and Small datasets where *STR* had a smaller number of disk access than *PBL+* ranged between 12% and 44%. The interval [0%, 0%] at the fourth data column and ninth data row of this table indicates that, for the different sets of experiments performed for *KNNQ* ET and Big datasets, the execution time of *STR* was never better than the one of *PBL+* (i.e. *PBL+* was always faster than *STR* in all the executed sets of experiments). Therefore, we can deduce that the closer to 0% the maximum value of this interval is, the worse the performance of the corresponding method is, in comparison to the best method.

A general conclusion arising from Table 4 is that *PBL+* is the best bulk-loading method for Big datasets and for all spatial queries, taking into account both performance measures (I/O and execution time), except for *WQ* and *DRQ* where *STR* proves to be better in I/O activity. On the other hand, *STR* is the best regarding execution time, when we have Small datasets, except for *PLQs*, while *PBL+* is the best regarding the number of disk accesses, except for *KNNQs* where *STR* is the best.

Table 4: Performance summary for single dataset spatial queries.

Spatial	*PBL+*		*STR*	
Query	*Small*	*Big*	*Small*	*Big*
PLQ I/O	**1**	**1**	[18%, 22%]	[0%, 6.7%]
WQ I/O	**1**	[0%, 47%]	[12%, 44%]	**1**
DRQ I/O	**1**	[0%, 47%]	[12%, 40%]	**1**
KNNQ I/O	[16%, 62%]	**1**	**1**	[0%, 0%]
CKNNQ I/O	**1**	**1**	[10%, 16%]	[0%, 7%]
PLQ ET	**1**	**1**	[12%, 20%]	[0%, 13.3%]
WQ ET	[24%, 60%]	**1**	**1**	[0%, 0%]
DRQ ET	[14%, 28%]	**1**	**1**	[0%, 7%]
KNNQ ET	[20%, 38%]	**1**	**1**	[0%, 0%]
CKNNQ ET	[8%, 36%]	**1**	**1**	[0%, 0%]

4.3. Experiments for dual dataset spatial queries

For *KCPQs* / ε*DJQs*, four algorithms, three versions of DF and one version of BF were tested in 225 experiments (9 combinations of datasets $\times$ 5 node sizes $\times$ 5 K-values or different ε values).

For *KCPQs*, the xBR$^+$-tree was faster with the BF algorithm in most cases (107/225), while the R-tree was faster with the DF algorithm, that utilized the classic plain sweep algorithm [41, 42], in most cases (89/225). Therefore, the performance comparison was based on these algorithms.

In Table 5, we depict the results of 3 combinations of real and 2 combinations of synthetic datasets having node size of 8KB, as particular examples. The experiments were executed for K=1000 closest pairs for each combination. In these experiments, the xBR$^+$-tree created by *PBL+* algorithm performed best in all combinations in both performance metrics. In these experiments, the xBR$^+$-tree needed smaller number of Disk Accesses than the R-tree in most cases (96.6%) of experiments, while in 3.6% of the cases the R-tree required less I/O operations. In the remaining cases (0.9%) both type of trees needed almost an equal average number ($\leq$ 5%) of disk accesses. The average relative difference percentage of Disk Accesses for each node size was over 50% (58.6%, 55.2%, 59.7%, 55.0%, 50.8% for node size 1KB, 2KB, 4KB, 8KB and 16KB, respectively) in favor of the xBR$^+$-tree. Regarding execution time, in all the 225 experiments the xBR$^+$-tree was faster than the R-tree. The average relative difference percentage of execution time for each node size was over 70% (71.7%, 74.5%, 70.8%, 74.8%, 76.0% for node size 1KB, 2KB, 4KB, 8KB and 16KB, respectively) in favor of the xBR$^+$-tree. It is clear that trees created by *PBL+* perform, on the average, significantly better in both performance metrics.

Table 5: *KCPQs*: average number of Disk Accesses and Execution Time, per query.

Dataset	Disk Accesses		Relative	time (ms)		Relative
Name	*PBL+*	*STR*	diff (%)	*PBL+*	*STR*	diff (%)
NArrND×NArdND	**14631**	29332	50.12	**290**	1579	81.62
500KC2N×1000KC1N	**14886**	33122	55.06	**314**	1990	84.21
1000KC1N×1000KC2N	**19383**	45636	57.53	**530**	3880	86.33
Water×Park	**55762**	119926	53.50	**8067**	22937	64.83
Water×Build	**114059**	265810	57.09	**40505**	63964	36.68

For $\varepsilon DJQs$, the xBR$^+$-tree was faster with the HDF algorithm in most cases (66/225), while the R-tree responded best (in execution time) with DF algorithm in most cases (96/225). Therefore, the performance comparison was based on these algorithms.

In Table 6, we depict the results of same combinations of datasets as in *KCPQs*. In the particular experiments shown in Table 6 the value of ε was set $\varepsilon \leq 5 \times 10^{-5}$ for the first three combinations, and $\varepsilon \leq 5 \times 10^{-8}$ for the rest two combinations of datasets.

Table 6: $\varepsilon DJQs$: average number of Disk Accesses and Execution Time, per query.

Dataset	Disk Accesses		Relative	time (ms)		Relative
Name	*PBL+*	*STR*	diff (%)	*PBL+*	*STR*	diff (%)
NArrND×NArdND	**26287**	29746	11.63	**529**	1653	68.00
500KC2N×1000KC1N	**16793**	32946	49.03	**409**	2136	80.85
1000KC1N×1000KC2N	**24832**	45384	45.28	**793**	4043	80.37
Water×Park	**54390**	119442	54.46	**21345**	23132	7.72
Water×Build	**101382**	263934	61.59	**46611**	63810	26.95

In these experiments, the xBR$^+$-tree created by *PBL+* algorithm was the best in both performance metrics. The xBR$^+$-tree needed a smaller number of Disk Accesses than the R-tree in most cases (92.9%), while in 5.8% of the cases the R-tree required less I/O operations. In the remaining cases (1.3%) both types of trees needed almost an equal average number ($\leq$ 5%) of disk accesses. The average relative difference percentage of Disk Accesses for each node size was over 49.5% (60.2%, 53.9%, 57.8%, 54.6%, 49.7% for node size 1KB, 2KB, 4KB, 8KB and 16KB, respectively) in favor of the xBR$^+$-tree. Regarding execution time, the xBR$^+$-tree was faster than R-tree in almost all cases (99.1%), while in the rest 0.89% of the cases the R-tree was faster than the xBR$^+$-tree with a small relative time difference ($\leq$ 5%). The average percentage of execution time for each node size was over 63% (69.3%, 65.4%, 63.1%, 67.4%, 70.8% for node size 1KB, 2KB, 4KB, 8KB and 16KB, respectively) in favor of the xBR$^+$-tree. It is clear that trees created by *PBL+* perform, on the average, significantly better in both metrics.

The explanation for the significantly better performance of trees created by *PBL+* is related to the better grouping of subregions and the fact that the execution of *KCPQs* / ε*DJQs* corresponds to multiple *KNNQs* / ε*DRQs*, maximizing the benefits resulting from the *PBL+*.

Table 7 has the same structure as Table 4 for comparison of the two distance-based join queries (*KCPQ* and ε*DJQ*). In this table the performance measures that we have taken into account are I/O activity (number of disk accesses) and total execution time (ET) in seconds for *Small* and *Big* datasets.

In this table, the interval "[a, b]" indicates that the corresponding method is a to b times more expensive than the best one. For example, the interval "[1.5, 2.8]" means the execution time of the *KCPQ* when Big datasets are considered is about 1.5 times larger for *STR* in the case of *Water* $\times$ *Park* and about 2.8 times larger in the case of *Water* $\times$ *Build*, than the one for *PBL+*.

A general conclusion arising from Table 7 is that the xBR$^+$-tree built by *PBL+* needs less disk accesses and is faster than the R-tree built by *STR*, regardless of the size of the datasets.

Table 7: Performance summary for dual dataset spatial queries.

	PBL+		STR	
Distance Join Query	*Small*	*Big*	*Small*	*Big*
KCPQ I/O	**1**	**1**	[2.0, 2.3]	[2.1, 2.3]
εDJQ I/O	**1**	**1**	[1.1, 2.0]	[2.2, 2.6]
KCPQ ET	**1**	**1**	[5.4, 7.3]	[1.5, 2.8]
εDJQ ET	**1**	**1**	[3.1, 5.1]	[1.1, 1.4]

5. Conclusions, Discussion and Future Work

In this paper, we presented an improvement of the algorithm of [12] (*PBL*), which was the first algorithm for bulk-loading xBR$^+$-trees for large datasets residing on disk, using a limited amount of RAM. The new algorithm (*PBL+*), improves all the four phases of its predecessor (it reduces movements of data items, makes better use of internal memory and achieves better partitioning of space).

Using real and artificial datasets of various cardinalities, we presented an extensive experimental comparison of *PBL+*, *PBL* and *STR*, one of the most popular algorithms for bulk-loading R-trees, regarding tree creation time and the characteristics of the trees created. This comparison unveiled that

- *PBL+* is faster than *PBL* and *STR*,
- *PBL+* creates trees that exhibit improved structural characteristics and better storage utilization (reduced size) than *PBL*; these characteristics lead to increased performance in processing queries and satisfy the main targets of an efficient bulk-loading algorithm.

Moreover, we experimentally compared the query efficiency of xBR$^+$-trees created by *PBL+* vs. R-trees created by *STR*, regarding disk accesses and execution time. It was shown that

- the trees created by *PBL+* exhibit better performance for the majority of queries and datasets (this is due to the fact that the xBR$^+$-tree created divides space in non-overlapping of areas and, in general, needs simpler CPU calculations for processing queries),
- especially for big datasets, the trees created by *PBL+* exhibit better execution time in almost all cases,
- the trees created by *PBL+* will retain their characteristics in a dynamic environment, with many insertions and deletions, contrary to the R-trees created by *STR* that will degrade very soon to smaller node occupancy and (as a consequence) reduced query performance, as data items are inserted or deleted.

This paper, through improving the creation of spatial storage structures and the efficiency of processing of spatial queries, contributes to the implementation of spatial database interfaces and the quality of software for big spatial data management.

The algorithm presented in this paper could be adapted to parallel and distributed environments, like Apache Hadoop [2], or Apache Spark [3]. Since in *PBL+* data is partitioned in disjoint quadrangular areas, separate computing nodes could build *m*-xBR$^+$-trees for such areas. These trees could be subsequently merged to form a total xBR$^+$-tree. Alternatively, a computing environment with multiple CPUs or GPUs and large main memory could be utilized for parallelizing the *PBL+* algorithm. Several, non-trivial, changes should be applied to the *PBL+* algorithm for such environments. These include parallel partitioning of input data and multiway merging of *m*-xBR$^+$-trees.

Moreover, *PBL+* could be adapted to other tree structures that do not utilize overlapping between regions of nodes, like R$^+$-trees, or other members of the Quadtree family. However, *PBL+* cannot be easily applied to trees with overlapping regions (like R*-trees, or R-trees).

Our current future work plans include:

- The development of bulk-insertion methods for xBR$^+$-trees. That is, when a large batch of data is scheduled to be inserted to an existing xBR$^+$-tree, to process the insertion as a whole and not to insert the data items one-by-one.
- An extension of the experimental study to data of higher dimensions.
- Embedding of bulk-loaded xBR$^+$-trees in SpatialHadoop[4] and study their performance in relation of other space partitioning strategies, already existing in SpatialHadoop.
- Create variations of bulk-loaded xBR$^+$-trees that will take advantage of the characteristics of SSD disks.

Acknowledgments

Work of Antonio Corral, Michael Vassilakopoulos and Yannis Manolopoulos funded by the MINECO research project [TIN2013-41576-R].

References

[1] R. H. Güting, An introduction to spatial database systems, VLDB J. 3 (4) (1994) 357–399.
[2] S. Shekhar, S. Chawla, Spatial databases - a tour, Prentice Hall, 2003.
[3] J. V. den Bercken, B. Seeger, An evaluation of generic bulk loading techniques, in: VLDB Conference, 2001, pp. 461–470.
[4] H. Tan, W. Luo, H. Mao, L. M. Ni, On packing very large r-trees, in: MDM Conference, 2012, pp. 99–104.
[5] N. An, K. V. R. Kanth, S. Ravada, Improving performance with bulk-inserts in oracle r-trees, in: VLDB Conference, 2003, pp. 948–951.
[6] M. Y. Eltabakh, R. Eltarras, W. G. Aref, Space-partitioning trees in postgresql: Realization and performance, in: ICDE Conference, 2006, p. 100.
[7] Y. Fang, M. Friedman, G. Nair, M. Rys, A. Schmid, Spatial indexing in microsoft SQL server 2008, in: SIGMOD Conference, 2008, pp. 1207–1216.
[8] G. Roumelis, M. Vassilakopoulos, T. Loukopoulos, A. Corral, Y. Manolopoulos, The xBR$^+$-tree: An efficient access method for points, in: DEXA Conference, 2015, pp. 43–58.
[9] G. Roumelis, M. Vassilakopoulos, A. Corral, Performance comparison of xbr-trees and r*-trees for single dataset spatial queries, in: ADBIS Conference, 2011, pp. 228–242.
[10] M. Vassilakopoulos, Y. Manolopoulos, External balanced regular (x-br) trees: New structures for very large spatial databases, in: Advances in Informatics: Selected papers of the 7th Panhellenic Conf. on Informatics, World Scientific Publ. Co., 2000, pp. 324–333.
[11] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: An efficient and robust access method for points and rectangles, in: SIGMOD Conference, 1990, pp. 322–331.
[12] G. Roumelis, M. Vassilakopoulos, A. Corral, Y. Manolopoulos, Bulk-loading xBR$^+$-trees, in: MEDI Conference, 2016, pp. 57–71.
[13] S. T. Leutenegger, J. M. Edgington, M. A. Lopez, Str: A simple and efficient algorithm for r-tree packing, in: ICDE Conference, 1997, pp. 497–506.
[14] L. Arge, K. H. Hinrichs, J. Vahrenhold, J. S. Vitter, Efficient bulk operations on dynamic r-trees, Algorithmica 33 (1) (2002) 104–128.

[2] http://hadoop.apache.org/
[3] http://spark.apache.org/
[4] http://spatialhadoop.cs.umn.edu/

[15] D. Achakeev, B. Seeger, P. Widmayer, Sort-based query-adaptive loading of r-trees, in: CIKM Conference, 2012, pp. 2080–2084.
[16] P. Ciaccia, M. Patella, Bulk loading the m-tree, in: ADC Conference, 1998, pp. 15–26.
[17] C. A. Lang, A. K. Singh, Modeling high-dimensional index structures using sampling, in: SIGMOD Conference, 2001, pp. 389–400.
[18] N. Roussopoulos, D. Leifker, Direct spatial search on pictorial databases using packed r-trees, in: SIGMOD Conference, 1985, pp. 17–31.
[19] I. Kamel, C. Faloutsos, On packing r-trees, in: CIKM Conference, 1993, pp. 490–499.
[20] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, J. Yu, Client-server paradise, in: VLDB Conference, 1994, pp. 558–569.
[21] D. Achakeev, M. Seidemann, M. Schmidt, B. Seeger, Sort-based parallel loading of r-trees, in: BigSpatial Workshop, 2012, pp. 62–70.
[22] L. Arge, K. H. Hinrichs, J. Vahrenhold, J. S. Vitter, Efficient bulk operations on dynamic r-trees, in: ALENEX Workshop, 1999, pp. 328–348.
[23] J. V. den Bercken, B. Seeger, P. Widmayer, A generic approach to bulk loading multidimensional index structures, in: VLDB Conference, 1997, pp. 406–415.
[24] T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, J. S. Vitter, Bulk operations for space-partitioning trees, in: ICDE Conference, 2004, pp. 29–40.
[25] S. Berchtold, C. Böhm, H. Kriegel, Improving the query performance of high-dimensional index structures by bulk-load operations, in: EDBT Conference, 1998, pp. 216–230.
[26] T. G. Vespa, C. T. Jr., A. J. M. Traina, Efficient bulk-loading on dynamic metric access methods, Information Systems 35 (5) (2010) 557–569.
[27] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: VLDB Conference, 1997, pp. 426–435.
[28] J. L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517.
[29] C. Traina, A. J. M. Traina, B. Seeger, C. Faloutsos, Slim-trees: High performance metric trees minimizing overlap between nodes, in: EDBT Conference, 2000, pp. 51–65.
[30] A. Papadopoulos, Y. Manolopoulos, Parallel bulk-loading of spatial data, Parallel Computing 29 (10) (2003) 1419–1444.
[31] G. R. Hjaltason, H. Samet, Y. J. Sussmann, Speeding up bulk-loading of quadtrees, in: ACM-GIS Conference, 1997, pp. 50–53.
[32] G. R. Hjaltason, H. Samet, Improved bulk-loading algorithms for quadtrees, in: ACM-GIS Conference, 1999, pp. 110–115.
[33] G. R. Hjaltason, H. Samet, Speeding up construction of PMR quadtree-based spatial indexes, VLDB Journal 11 (2) (2002) 109–137.
[34] R. C. Nelson, H. Samet, A population analysis for hierarchical data structures, in: SIGMOD Conference, 1987, pp. 270–277.
[35] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, R. Kanneganti, Incremental organization for data recording and warehousing, in: VLDB Conference, 1997, pp. 16–25.
[36] I. Gargantini, An effective way to represent quadtrees, Communications of the ACM 25 (12) (1982) 905–910.
[37] D. Comer, The ubiquitous b-tree, ACM Comput. Surv. 11 (2) (1979) 121–137.
[38] J. A. Orenstein, T. H. Merrett, A class of data structures for associative searching, in: PODS Conference, 1984, pp. 181–190.
[39] W. G. Aref, H. Samet, Extending a DBMS with spatial operations, in: SSD Conference, 1991, pp. 299–318.
[40] C. A. Shaffer, P. R. Brown, A paging scheme for pointer-based quadtrees, in: SSD Conference, 1993, pp. 89–104.
[41] G. Roumelis, M. Vassilakopoulos, A. Corral, Y. Manolopoulos, A new plane-sweep algorithm for the k-closest-pairs query, in: SOFSEM Conference, 2014, pp. 478–490.
[42] G. Roumelis, A. Corral, M. Vassilakopoulos, Y. Manolopoulos, New plane-sweep algorithms for distance-based join queries in spatial databases, GeoInformatica 20 (4) (2016) 571–628.