Bulk-Loading xBR$^+$-trees

George Roumelis[1], Michael Vassilakopoulos[2](✉), Antonio Corral[3], and Yannis Manolopoulos[1]

[1] Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
{groumeli,manolopo}@csd.auth.gr

[2] Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
mvasilako@uth.gr

[3] Department of Informatics, University of Almeria, Almeria, Spain
acorral@ual.es

Abstract. Spatial indexes are important in spatial databases for efficient execution of queries involving spatial constraints. 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 as a whole, when the dataset to be indexed is available beforehand, instead of creating (loading) the index gradually, when the dataset items are available 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 RAM. Moreover, using real and artificial datasets of various cardinalities, we present an experimental comparison of this algorithm vs. the algorithm loading items one-by-one, regarding performance (I/O and execution time) and the characteristics of the xBR$^+$-trees created. We also present experimental results regarding the efficiency of bulk-loaded xBR$^+$-trees vs. xBR$^+$-trees where items are loaded one-by-one for query processing.

Keywords: Spatial indexes · Bulk-loading · xBR$^+$-trees · Query processing

1 Introduction

Spatial indexes are designed to facilitate spatial database operations that involve retrieval of spatial objects according to specific spatial constraints [1]. An important issue in the implementation of such spatial indexes is the time needed to build them from a specific dataset. If the dataset is static (i.e. when insertions and deletions are rarely executed or even they are not performed at all), we can focus on building the spatial index in a way that permits the efficient execution of queries. However, it is also desirable that the index creation time is as short as possible. For this reason, 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

L. Bellatreche et al. (Eds.): MEDI 2016, LNCS 9893, pp. 57–71, 2016.
DOI: 10.1007/978-3-319-45547-1_5

is anticipated that, due to these characteristics, query processing performance will be improved. This is known in the literature as *packing* or *bulk-loading*.

Bulk-loading refers to the process of creating an index from scratch for a given dataset [2], and here we use this term to characterize building of a disk-based spatial index for an entire dataset without any intervening queries [17]. Index bulk-loading of large datasets has been an important direction of database research. Bulk-loading is necessary when an index is built up for the first time. It is well known that loading indexes by inserting items (tuples) one-by-one (i.e. *tuple-loading* [2]) is less efficient than specially designed bulk-loading algorithms that can be executed with the same complexity as external sorting [2]. Bulk-loading is therefore an interesting option for building spatial indexes when the data are known in advance and not many updating operations are executed.

The bulk-loading process, in general, consists of two sub-processes: data partitioning and index construction. In the *data partitioning process*, a dataset is partitioned into subsets whose cardinality is at maximum a certain number of data objects, called *fanout*. Data objects are sorted according to particular methods and are assigned linearized orders. In the *index construction process*, geometric shapes (as Minimum Bounding Rectangles, MBR) are formed from partitioned subsets and inserted as entries to intermediate or terminal nodes of the spatial index structure. *Top-down* approach builds intermediate nodes first, whereas *bottoms-up* builds terminal nodes first.

In this paper, we study the efficiency of building a quadtree-based index structure. In particular, wel focus on the xBR$^+$-tree [3], a balanced disk-based index structure for point data that belongs to the Quadtree family, and hierarchically decomposes space in a regular manner. The xBR$^+$-tree improves the xBR-tree [4,19] in the node structure and the splitting process. Moreover, it outperforms xBR-trees and R*-trees with respect to several well-known spatial queries, such as Point Location, Window Query, K-Nearest Neighbor, etc.

In this paper, we present the first algorithm for bulk-loading xBR$^+$-trees for big datasets residing on disk, using a limited amount of RAM. Moreover, using real and artificial datasets of various cardinalities, we present an experimental comparison of this algorithm vs. the algorithm of loading items one-by-one, regarding creation time and the characteristics of the tree created. We also present experimental results regarding the (I/O and execution time) efficiency of bulk-loaded xBR$^+$-trees vs. xBR$^+$-trees, where items are loaded one-by-one for query processing of single dataset 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 − DJQ).

This paper is organized as follows. In Sect. 2 we review related work on bulk-loading and provide the motivation of this paper. In Sect. 3, we describe the most important characteristics of xBR$^+$-tree, from the implementation point of view. Section 4 presents our bulk-loading method for the xBR$^+$ -tree. In Sect. 5, we discuss the results of our experiments. And finally, Sect. 6

provides the conclusions arising from our work and discusses related future work directions.

2 Related Work and Motivation

This section reviews previous bulk-loading methods in general, whose main target is to reduce the loading time, the query cost of the resulting index structure, or both. In [2] 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 [6], and they can be considered as a hybrid of top-down and bottom-up strategies. They employ external queues (*buffers*) attached to the internal nodes of the tree except for the root node. An insertion of a record can be viewed as a process temporarily blocked after having arrived at a node. Instead of continuing the traversal down to the leaf, the record is inserted into the *buffer*. Whenever the number of records in a *buffer* exceeds a threshold, a large portion of the records of the buffer is transferred to the next level.
– The *sample-based bulk-loading* methods use a sample of the input that fits into memory to build up the target index. In general, this method randomly samples objects, so-called *representatives*, from the input and builds up a structure. Then, the remaining records of the input are assigned to one of the *representatives*. For each representative, the associated data objects are treated again in the same way.

There are several methods that belong to the *sort-based bulk-loading* category, but the most characteristic ones are proposed in [8–11]. In [8], a method (so-called *packed R-tree*) that uses a heuristic for aggregating rectangles into nodes is introduced. It suggests to sort the data with respect to minimum value of the objects in a certain dimension. In [9] 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. Another sort-based method for bulk-loading R-trees is presented in [10]. 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, where N is the number of objects, B is the node capacity and d are the dimensions of the input dataset. 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 [10], it was also shown that this method of sort-based bulk-loading creates R-trees whose search quality is superior to R-trees created with respect to the Hilbert-ordering. However, the method also requires the input being sorted d times. Recently, in [11] a sort-based query-adaptive loading for building R-trees optimally designed for a given query profile is presented. Finally, in [12] a scalable alternative MapReduce approach to parallel loading of R-trees using a level-by-level design, based on [11], is introduced.

The most representative of *buffer-based bulk-loading* methods are proposed in [6,7,13]. In [13], the R-tree is built recursively bottom-up. In each stage, an intermediate tree structure is built, where the lowest level corresponds to the next level of the final R-tree. The non-leaf nodes in the intermediate tree structures have a high fan-out (determined by available internal memory) as well as a buffer that receives insertions. [6] 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 [7], 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.

The most remarkable *sample-based bulk-loading* algorithms have been proposed in [2,14,15]. In [14] a method to build an M-tree 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 [15], a kd-tree 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 [2], the two generic sample-based bulk-loading algorithms proposed, recursively partition the input by using a main-memory index of the same type as the target index to be built.

The most representative contributions related to bulk-loading techniques for Quadtree index structures are [16–18]. In [16], 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, 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 first approach focuses on the problem on B-tree level, increasing the amount of buffering done by the B-tree (*B-tree buffing*). The second approach focuses 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 [17], improved versions of the bulk-loading algorithms studied in [16] for PMR-quadtrees are presented, assuming that the algorithms are implemented using a linear quadtree, a disk-resident representation that stores objects contained in the leaf nodes of the quadtree in a linear index (B-tree) ordered on the basis of a space-filling curve (*Morton curve*). Finally, the extended version of [16,17] is presented in [18], where the detailed algorithms for the proposed *sort-based* *bulk-loading* method, analytic observations, an extensive experimental study and interesting discussions are presented.

The main motivation of this work is the proposal of a new bottom-up, sort-based like approach for bulk-loading of a space-driven quadtree variant, the xBR$^+$-tree. Note that the xBR$^+$-tree [4,19] (for more details, see Sect. 3) is unlike any other quadtree variant, since it is a totally disk-based, height-balanced, pointer-based, multiway tree for multidimensional points and no other quatree variant has all these characteristics.

3 The xBR$^+$-tree

The xBR$^+$-tree [3] (an extension of the xBR-tree [4,19]) is a hierarchical, disk-resident Quadtree-based (space-driven access method) index structure for multidimensional points. 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 [4,19]. 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 Fig. 1 an internal node (a root) that points to 2 internal nodes that point to 7 leaves is depicted. The region of the root is the original space, which is assumed to have a quadrangular shape. The region of the right child is the NW quadrant of the original space. The region of the left child is the whole space minus the region of the NW quadrant - a non-complete square. The * symbol is used to denote the end of a variable size address. The *Address* of the right child is 0*, since the region of this child is the NW quadrant of the original space. The *Address* of the left child is * (has zero directional digits), since the region of this child refers to the remaining space. Each of these *Addresses* is expressed relatively to the minimal quadrant that covers the internal node (each *Address* determines a subquadrant of this minimal quadrant). For example, in Fig. 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.

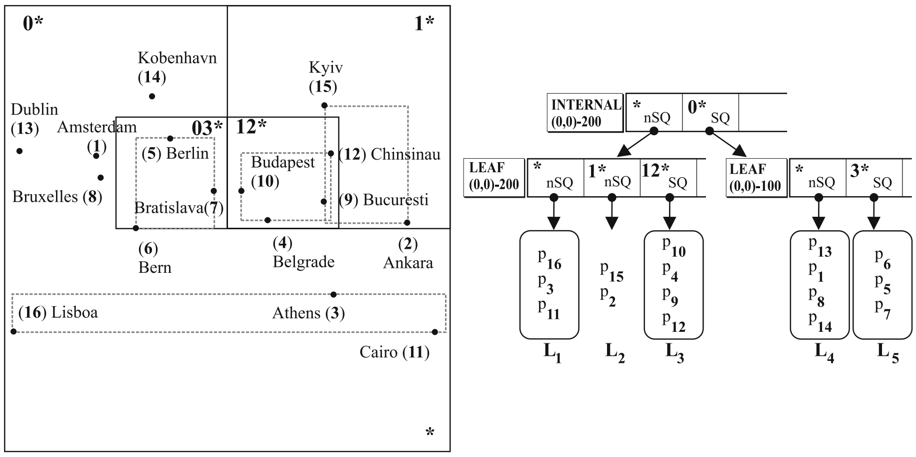

Fig. 1. A collection of points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree.

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.

4 Bulk-Loading xBR$^+$-tree

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

The first phase is a technical one. Since the initial dataset file is usually in text format, we transform this file to binary format (for efficiency during reading from and writing to secondary memory) and at the same time we split it in two files, based on the middle of the axis of the last dimension. These transformation and splitting are a preprocessing step for the next phase (the resulting files are used as input item files for the next phase).

During the second phase we partition each of the two input item files into item blocks of size $\leq$ *MemoryLimit* in a regular fashion (we alternate between splitting axes and always split in the middle of the current subregion). We use a temporary helper file, which is paired with each of the input item files. When the partition of one of these input item files starts, the item file is used for input and the helper is used for output. While there exist blocks of size $>$ *MemoryLimit*, we continue splitting and each time we alternate between splitting axes the roles of input and output files are interchanged. The resulting blocks are transferred in main memory, as input for the next phase.

During the third phase, for each block of items, a Quadtree is built in main memory by splitting nodes as long as they correspond to regions (quadblocks) containing more items than the capacity of xBR$^+$-tree leaves. This Quadtree is gradually transformed to an xBR$^+$-tree in main memory, referred to as m-xBR$^+$-tree for the rest of the paper. Initially, in a bottom-up fashion, the subtrees of this Quadtree that will form the leaves of the m-xBR$^+$-tree having the largest possible occupancy of items are determined. When all the leaves of the m-xBR$^+$-tree have been created in main memory, they are transferred to secondary memory, since they will not be altered for the rest of the bulk-loading process and the pointers pointing them (entries of internal nodes) are updated. This transfer to secondary memory is uninterruptible for the whole set of leaves, since this is more efficient (I/O with the data file hosting the whole set of leaves for large chunks of time) than several transfers of single leaves that are intermixed with other I/O operations. Afterwards, the rest of the Quadtree is transformed to the index part of the m-xBR$^+$-tree (internal nodes), level-by-level. Again, for each level, in a bottom-up fashion, the subtrees of the Quadtree that will form the internal nodes of the m-xBR$^+$-tree of this level having the largest possible occupancy of children are determined. The process followed to form the nodes of the leaf level and the levels of internal nodes has similarities to the way we split overflowed xBR$^+$-tree nodes when we insert items one-by-one into them [3,4,19]. The difference lies in the number of subtrees that form the m-xBR$^+$-tree nodes of a level, as now they may be more than two. This process is repeated a number of times equal to the final height of the m-xBR$^+$-tree (up to the root level) for the current item block.

During the last phase, 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 iteration of the bulk-loading process. For the merging process to be feasible, two properties (that are satisfied by the previous phases of the bulk-loading process) should hold. Either the regions R_0 of the existing xBR$^+$-tree holding P_0 points and R_1 of the m-xBR$^+$ -tree holding P_1 points are completely disjoint ($R_0 \cap R_1 = \emptyset$), or the existing xBR$^+$-tree has no data items into the R_1 region ($\forall p \in R_0, p \notin R_1$). The merge process proceeds as follows:

– If there is an empty xBR$^+$-tree in disk (since this is the first block processed), the internal nodes of the m-xBR$^+$-tree are transferred to disk in a post-order fashion and become the existing xBR$^+$-tree. As mentioned before, the leaves of the m-xBR$^+$-tree have already been saved into disk.
– If the the existing xBR$^+$-tree and the m-xBR$^+$-tree have the same height, all internal nodes of the m-xBR$^+$-tree are transferred to disk in a post-order fashion, except for the root. The root of the xBR$^+$-tree ($|R_0| = n_0$) must be merged with the root of the m-xBR$^+$-tree ($|R_1| = n_1$). There are two cases in this phase. If $n_0 + n_1 \leq C$ then all entries of R_1 are added to R_0. Else, if $n_0 + n_1 > C$, a new root of the xBR$^+$-tree must be created having the roots of the two trees as children. In this case, the merge process leads to an increase of the height of the existing xBR$^+$-tree.
– If the m-xBR$^+$-tree has a smaller height (h_1) than the existing xBR$^+$-tree (h_0), all internal nodes of the m-xBR$^+$-tree are transferred to disk in a post-order fashion, except for the root. The xBR$^+$-tree is searched from its root in order to find the internal node (with region N_0, where $|N_0| = n_0$) of height h_1, that can host the region of the root of the m-xBR$^+$-tree ($|R_1| = n_1$). If $n_0 + n_1 \leq C$ then all entries of R_1 are added in N_0. Else, if $n_0 + n_1 > C$, a new entry for R_1 is created and added to the parent node of N_0. As long as overflows continue to appear, this process continues upwards and may even cause an increase of the tree height.
– If the m-xBR$^+$-tree has a larger height than the existing xBR$^+$-tree ($h_1 > h_0$), the m-xBR$^+$-tree is traversed from its root down to the h_0 level (without transferring any nodes above h_0 level to disk) and the process of merging trees of one of the last two previous cases (trees of equal heights, or m-xBR$^+$-tree with a smaller height) is executed for each of the nodes (roots of subtrees) of the m-xBR$^+$-tree at the h_0 level and the existing xBR$^+$-tree.

The bulk-loading process continues as long as m-xBR$^+$-trees that host an item sub-dataset (block) are created and terminates when the whole dataset has been processed.

Note that the four phases are pipelined (phase one produces an output file as input to phase two that produces a block as input to phase three that produces an m-xBR$^+$-tree as input to phase four). Note also that partitioning data items in a regular (Quadtree-like) fashion is a way of two dimensional sorting. 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.

5 Experimental Results

We designed and run a large set of experiments to compare the Process of Bulk-Loading xBR$^+$-trees (*PBL*) to the Process of Loading items in xBR$^+$-trees One-by-One (*PLObO*). We used 4 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 railroads (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. $|\text{NArdN}| = 569082$ points, $|\text{NArrN}| = 191558$ 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. $|\text{NArdND}| = 1138164$ points, $|\text{NArrND}| = 1138188$ 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. We also used two 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 and world parks or green areas (Park) consisting of 11503925 polygons. To create sets of points, we used the centers of the line-segment MBRs from Water and the centroids of polygons from Park. The experiments were run on a Linux machine, with Intel core duo 2×2.8 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 *DJQ*), counting disk read accesses (I/O) and total execution time.

5.1 Experiments for Tree Building

In Table 1, for the *PBL*, we present for four indicative datasets (two big and one smaller real datasets and one synthetic dataset) the effect of the *MemoryLimit (ML)* as a percentage of the cardinality of each dataset to the tree characteristics: tree height (H), internal nodes occupancy percentage (Iocc), leaf nodes occupancy percentage (Locc), size of the tree in disk (Size) and the total creation time of the tree (Time). For each each dataset, we added a line that presents the same tree characteristics and the total creation time of tree created by *PLObO*.

Regarding the efficiency of the *PBL*, we observe the following.

– The tree takes the best (smallest) height value using as *MemoryLimit* a percentage of at least 1 % for all datasets, except for the big real dataset, Park. For Park a *MemoryLimit* greater than or equal to 2 % is needed.

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

Table 1. Tree creation characteristics, using the *PBL* and the *PLObO*.

ML (%)	H	Iocc (%)	Locc (%)	Size (MB)	Time (s)
Dataset : Water					
1/4	6	55.62	65.08	351	16.93
1/2	5	59.82	65.08	350	16.41
1	4	65.52	65.08	350	17.37
2	4	67.52	65.08	350	16.41
4	4	68.50	65.08	350	16.34
8	4	68.80	65.08	350	16.63
—	4	57.38	64.69	353	116.0
Dataset : Park					
1/4	7	55.55	63.35	711	33.87
1/2	6	64.03	63.35	709	34.77
1	5	68.07	63.35	708	35.05
2	4	69.03	63.35	708	34.68
4	4	69.25	63.35	708	33.43
8	4	69.34	63.35	708	32.74
—	4	57.20	64.95	693	208.0

ML %)	H	Iocc (%)	Locc (%)	Size (MB)	Time (s)
Dataset : 1000KCN					
1/4	7	19.72	64.20	63	1.31
1/2	5	48.29	64.20	61	1.28
1	4	57.30	64.19	61	1.43
2	4	60.03	64.19	61	1.26
4	4	64.76	64.19	61	1.25
8	4	65.66	64.19	61	1.45
—	4	63.73	64.57	60	13.0
Dataset : NArdND					
1/4	7	20.30	65.15	71	1.67
1/2	4	52.42	65.15	68	1.61
1	4	56.15	65.15	68	1.58
2	4	59.45	65.15	68	1.50
4	4	63.74	65.15	68	1.76
8	4	64.51	65.15	68	1.63
—	4	56.83	64.87	69	16.0

- The internal nodes occupancy is increasing as the *MemoryLimit* increases. This is expected if we think that the most compact tree would be created if all the data were present in main memory. However, we observe that the rate of increase of internal nodes occupancy decreases significantly for exponential rate of increase of the *MemoryLimit*.
- The leaves occupancy (and the highly correlated tree size), as it is expected, is independent to the *MemoryLimit*, since the number of items that correspond to *MemoryLimit* is always larger than C, the capacity of the leaves.
- For all *MemoryLimit* values in each dataset, the creation time takes similar values.

The observations are analogous for the rest of the datasets used. Considering the above observations, we propose a *MemoryLimit* value of 2 %. This value achieves either the best, or a good enough value for any tree characteristic.

Comparing the *PBL* to the *PLObO*, we observe the following:

- The smallest height of the *PBL* (for an appropriate value of *MemoryLimit*) is equal to the height of the *PLObO*, for all datasets.
- The *PBL* builds more compact indexes (larger internal nodes occupancy) than the *PLObO*, for most datasets, using *MemoryLimit* larger than 0.5 %.
- The leaves occupancy (and the highly correlated tree size) is comparable in all cases for the two processes.
- The time that the *PBL* takes to create the tree (this is mainly time for I/O) is more than 6 times smaller than the *PLObO*.

The conclusions are analogous for the rest of the datasets used.

5.2 Experiments for Single Dataset Spatial Queries

For *PLQs*, we executed two sets of 60 experiments (12 datasets $\times$ 5 node sizes). In the first set, we used as query input the original datasets (existing points). When searching for existing points the number of disk accesses in xBR$^+$-trees is equal to their height. This set of experiments is summarized in the 1st data line of Table 2. In this table, for each query, regarding disk read accesses and execution time, we present percentages of experimental cases where trees created by the two processes perform equivalently (Columns 2 and 5, respectively) and where trees created by *PBL*/*PLObO* have a performance that is more than 5 % better than their rivals (Columns 3 and 6/4 and 7, respectively). The *MemoryLimit* used was equal to 4 %, although a value of 2 % gives analogous results. It is evident that, for this set of experiments, both types of trees perform almost equivalently. In the second set, we used as query input the centroids of the query windows (non-existing points). While searching for non existing points in the dataset, the disk accesses may be less than the tree-height of xBR$^+$-trees (due to *DBRs*). This set of experiments is summarized in the 2nd data line of Table 2. It is clear that trees created by *PBL*, on the average, perform better in both metrics.

Table 2. Percentages of cases of Disk Accesses and Execution Time winners.

Query	Number of disk read accesses			Execution time		
		PBL	PLObO		PBL	PLObO
	tie diff $\leq$ 5 %	wins diff $>$ 5 %	wins diff $>$ 5 %	tie diff $\leq$ 5 %	wins diff $>$ 5 %	wins diff $>$ 5 %
PLQ-existing points	95.0	00.0	05.0	61.7	23.3	15.0
PLQ-non-existing points	58.3	35.0	06.7	40.0	36.7	23.3
WQ	85.0	13.9	01.1	68.1	21.9	10.0
DRQ	84.7	14.2	01.1	69.7	19.2	11.1
KNNQ	58.3	24.6	17.1	45.8	35.8	18.3
CKNNQ	50.4	43.8	05.8	55.4	31.7	12.9
KCPQ	20.4	79.6	00.0	24.8	75.2	00.0
DJQ	22.8	77.2	00.0	25.6	74.0	00.4

For *WQs*, we executed 360 experiments (12 datasets $\times$ 5 node sizes $\times$ 6 query window sizes). In Table 3, we depict the average number of disk read accesses and average execution time (in μs), per query window, of 3 real and 2 synthetic datasets having node size of 8KB, as particular examples[2]. The experiments were executed for 4096 query windows (having size 1/4096 of the total space) for each data set. Regarding disk accesses, the two trees perform almost equivalently in 3/5 cases, while, in the other 2/5 cases, the trees created by *PBL* perform

[2] Due to space limitations, results for particular datasets are presented for some queries, only.

better. Regarding execution time, the two trees perform almost equivalently in 2/5 cases, while, in the other 3/5 cases, the trees created by *PBL* perform better. All the 360 experiments are summarized in the 3rd data line of Table 2. It is clear that the trees created by *PBL*, on the average, perform better in both metrics.

For *DRQs*, five algorithms, four versions of depth-first (DF) and one version of best-first (BF) were tested in 360 experiments (12 datasets $\times$ 5 node sizes $\times$ 6 sets of query circles), each. Both trees responded best with DF algorithm in most cases (so, the performance comparison was based on this algorithm). All the 360 experiments are summarized in the 4rd data line of Table 2. It is clear that the trees created by *PBL*, on the average, perform better in both metrics.

For *K-NNQs*, five algorithms, four versions of DF and one version of BF were tested in 240 experiments (12 datasets $\times$ 5 node sizes $\times$ 4 K-values, using 4096 query points, in all cases), each. Both trees responded best with the HDF algorithm (DF search that utilizes a local *MinHeap*, keyed by *mindist* between query point and *DBR*) in most cases (so, the performance comparison was based on this algorithm). In Table 4, we depict the results of 3 real and 2 synthetic datasets having node size of 8 KB, as particular examples. The experiments were executed for 4096 query points (asking for K=1000 nearest neighbors to each query point) for each data set. In these experiments, the trees created by *PBL* always perform better regarding disk read accesses. Regarding execution time, the two trees perform almost equivalently in 1/5 cases, while, in the rest 4/5 cases, the trees created by *PBL* perform better. All the 240 experiments are summarized in the 5rd data line of Table 2. It is clear that the trees created by *PBL*, on the average, perform better in both metrics.

Finally, for *CK-NNQs*, five algorithms, four versions of DF and one version of BF were tested in 240 experiments (12 datasets $\times$ 5 node sizes $\times$ 4 K-values, using 4096 query circles, in all cases), each. Both trees responded best with BF algorithm in most cases, especially in execution time (so, the performance comparison was based on this algorithm). All the 240 experiments are summarized in the 6th data line of Table 2. It is clear for this query, too, that trees created by *PBL*, on the average, perform better in both metrics.

Overall, trees created by *PBL*, perform better regarding both metrics, for all the single dataset queries, except for the *PLQ* for existing points, where the two trees appear almost equivalent. The explanation for the improved performance

Table 3. WQs: average number of Disk Accesses and Execution Time, per query.

Dataset name	Disk read accesses		Relative diff (%)	Time (μs)		Relative diff (%)
	PBL	PLObO		PBL	PLObO	
NArrN	1.925	2.154	10.6	6.59	7.056	6.62
NArdN	2.830	3.102	8.76	9.90	10.48	5.55
500KCN	3.340	3.406	1.93	12.2	12.44	1.85
1000KCN	4.056	4.081	0.60	15.0	15.28	1.83
Water	12.34	12.72	2.98	34.0	35.97	5.54

Table 4. KNNQs: average number of Disk Accesses and Execution Time, per query.

Dataset name	Disk read accesses		Relative diff (%)	Time (μs)		Relative diff (%)
	PBL	PLObO		PBL	PLObO	
NArdN	17.008	22.714	25.1	131.69	202.85	35.1
500KCN	17.478	18.877	7.41	137.78	146.53	5.97
1000KCN	16.932	18.385	7.90	145.76	148.56	1.88
Water	18.875	32.313	41.6	154.92	328.53	52.9
Park	21.237	27.424	22.6	172.36	281.28	38.7

of trees created by *PBL* is related to the structural difference between the two trees. The *PBL* can achieve better grouping of subregions, since all data/entries are known before each node is created.

5.3 Experiments for Dual Dataset Spatial Queries

For *K-CPQs/DJQs*, four algorithms, three versions of DF and one version of BF were tested in 250 experiments (10 combinations of datasets $\times$ 5 node sizes $\times$ 5 K-values, in all cases), each. Both trees responded best with HDF algorithm (that utilizes *minmindist* between *DBRs*, instead of *mindist* between query point and *DBR*) in most cases (so, the performance comparison was based on this algorithm).

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

Dataset name	Disk read accesses		Relative diff (%)	Time (ms)		Relative diff (%)
	PBL	PLObO		PBL	PLObO	
NArrND×NArdND	14351	24559	41.6	100.7	143.8	30.0
500KC2N×1000KC1N	16969	18123	6.37	109.7	116.2	5.58
1000KC1N×1000KC2N	21913	22788	3.84	136.7	141.0	3.04
NArdND×Water	2095	10987	80.9	23.05	104.2	77.9
Water×Park	56783	111960	49.3	360.4	570.7	36.8

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 two trees perform almost equivalently in 1/5 cases, while, in the rest 4/5 cases, the trees created by *PBL* perform better, in both metrics. All the 250 experiments are summarized in the 7th/8th data line of Table 2. It is clear that trees created by *PBL* perform, on the average, significantly better in both metrics. The explanation for the significantly improved performance of trees created by *PBL* is related to the better grouping of subregions and the fact that the execution of *K-CPQs/DJQs* corresponds to multiple *K-NNQs/DRQs*, maximizing the benefits resulting from the *PBL*.

6 Conclusions and Future Work

In this paper, for the first time in the literature, we present an algorithm for bulk-loading xBR$^+$-trees for big datasets residing on disk, using a limited amount of RAM. This bottom-up, sort-based like algorithm was implemented and extensive experimentation was performed for comparing the characteristics and the query performance of trees created by the new algorithm and trees created by the traditional way of inserting items one-by-one. These experiments show that, using a RAM buffer $\geq 2\,\%$ of the dataset size, bulk-loaded trees that have comparable structural characteristics to non bulk-loaded trees and perform better/significantly better in processing single/dual dataset queries are created.

In the future, we plan to examine alternative ways of bulk-loading xBR$^+$-trees (for example, to avoid using a RAM based Quadtree for gradually building the respective xBR$^+$-tree in main memory, working directly with items and entries). Moreover, we plan to compare bulk-loaded xBR$^+$-trees to other types of spatial indexes (like R*-trees produced by the bulk-loading algorithm of [10]). Last but not least, we plan to embed bulk-loaded xBR$^+$-trees in SpatialHadoop[3].

References

1. Shekhar, S., Chawla, S.: Spatial Databases - A Tour. Prentice Hall, Upper Saddle River (2003)
2. Van den Bercken, J., Seeger, B.: An evaluation of generic bulk loading techniques. In: VLDB Conference, pp. 461–470 (2001)
3. Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., Manolopoulos, Y.: The xBR$^+$-tree: an efficient access method for points. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) DEXA 2015. LNCS, vol. 9261, pp. 43–58. Springer, Heidelberg (2015)
4. Roumelis, G., Vassilakopoulos, M., Corral, A.: Performance comparison of xBR-trees and R*-trees for single dataset spatial queries. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 228–242. Springer, Heidelberg (2011)
5. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD Conference, pp. 322–331 (1990)
6. Arge, L., Hinrichs, K.H., Vahrenhold, J., Vitter, J.S.: Efficient bulk operations on dynamic R-trees. Algorithmica **33**(1), 104–128 (2002)
7. Van den Bercken, J., Seeger, B., Widmayer, P.: A generic approach to bulk loading multidimensional index structures. In: VLDB Conference, pp. 406–415 (1997)
8. Roussopoulos, N., Leifker, D.: Direct spatial search on pictorial databases using packed R-trees. In: SIGMOD Conference, pp. 17–31 (1985)
9. Kamel, I., Faloutsos, C.: On packing R-trees. In: CIKM Conference, pp. 490–499 (1993)
10. Leutenegger, S.T., Edgington, J.M., Lopez, M.A.: STR: a simple and efficient algorithm for R-Tree packing. In: ICDE Conference, pp. 497–506 (1997)

[3] http://spatialhadoop.cs.umn.edu/.

11. Achakeev, D., Seeger, B., Widmayer, P.: Sort-based query-adaptive loading of R-trees. In: CIKM Conference, pp. 2080–2084 (2012)
12. Achakeev, D., Schmidt, M., Seeger, B.: Sort-based parallel loading of R-trees. In: BigSpatial Workshop, pp. 62–70 (2012)
13. Van den Bercken, J., Seeger, B., Widmayer, P.: A generic approach to bulk loading multidimensional index structures. In: VLDB Conference, pp. 406–415 (1997)
14. Ciaccia, P., Patella, M.: Bulk loading the M-tree. In: Australian Database Conference, pp. 15–26 (1998)
15. Berchtold, S., Böhm, C., Kriegel, H.-P.: Improving the query performance of high-dimensional index structures by bulk load operations. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 216–230. Springer, Heidelberg (1998)
16. Hjaltason, G.R., Samet, H., Sussmann, Y.J.: Speeding up bulk-loading of quadtrees. In: ACM GIS Conference, pp. 50–53 (1997)
17. Hjaltason, G.R., Samet, H.: Improved bulk-loading algorithms for quadtrees. In: ACM GIS Conference, pp. 110–115 (1999)
18. Hjaltason, G.R., Samet, H.: Speeding up construction of PMR quadtree-based spatial indexes. VLDB J. **11**(2), 109–137 (2002)
19. Vassilakopoulos, M., Manolopoulos, Y.: External balanced regular (x-BR) trees: new structures for very large spatial databases. In: Advances in Informatics: Selected Papers of the 7th Panhellenic Conference on Informatics. World Scientific Publishing Co., pp. 324–333 (2000)