Bulk Insertions into xBR$^+$-trees

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

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

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

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

Abstract. Bulk insertion refers to the process of updating an existing index by inserting a large batch of new data, treating the items of this batch as a whole and not by inserting these items one-by-one. Bulk insertion is related to bulk loading, which refers to the process of creating a non-existing index from scratch, when the dataset to be indexed is available beforehand. The xBR$^+$-tree is a balanced, disk-resident, Quadtree-based index for point data, which is very efficient for processing spatial queries. In this paper, we present the first algorithm for bulk insertion into xBR$^+$-trees. This algorithm incorporates extensions of techniques that we have recently developed for bulk loading xBR$^+$-trees. Moreover, using real and artificial datasets of various cardinalities, we present an experimental comparison of this algorithm vs. inserting items one-by-one for updating xBR$^+$-trees, regarding performance (I/O and execution time) and the characteristics of the resulting trees. We also present experimental results regarding the query-processing efficiency of xBR$^+$-trees built by bulk insertions vs. xBR$^+$-trees built by inserting items one-by-one.

Keywords: Spatial indexes, Bulk-Inserting, xBR$^+$-trees, Query Processing.

1 Introduction

Nowadays, the volume of available spatial data (e.g. location, routing, navigation, etc.) is continuously increasing world-wide. In many data-intensive spatial applications, dealing with the problem of bulk-insertions of new large datasets into an existing dataset is of particular interest. It is important to add newly collected data into an existing dataset quickly, because new data are continuously being generated and added to existing datasets. The use of efficient spatial indexes is very important for performing spatial queries and retrieving efficiently spatial objects from datasets according to specific spatial constraints. An important aspect in the implementation of such spatial indexes is the time needed to build and update them from a given dataset [16].

⋆ Work funded by the MINECO research project [TIN2013-41576-R]

If the dataset is *dynamic* (i.e. when insertions and deletions are interleaved), then we can devote efforts on creating and updating the spatial index in a way that permits the efficient execution of spatial queries. Furthermore, slow updates of spatial indexes can seriously degrade query response time, which is especially critical in modern interactive and data-intensive spatial database applications. There are three ways in which spatial indexes can be created or updated by a dynamic dataset. First, if the dataset has not been indexed yet, the spatial index can be built from scratch for the entire dataset (this process is known as *bulk-loading*). Second, if the dataset already has a spatial index and a large batch of data is to be added to the index, then the spatial index can be updated with all the new data at once (this process is known as *bulk-insertion*). Third, if the dataset already has an index and a small amount of data is to be added to the index, it can be more efficient to insert the new data items one by one into the existing spatial index (this process is known as *one-by-one-insertion* [7]). In this work, we present a method for speeding up the updating of a spatial index for the second situation (*bulk-insertion*).

In contrast to a bulk-loading algorithm, where a spatial index is built from scratch, a bulk-insertion algorithm aims at updating an existing index structure with a large set of new data. Thus, *bulk-insertion* refers to the process of updating an existing spatial index with a large new dataset, that is, of combining data that is already indexed by a disk-resident spatial index and data that has not yet been indexed. Bulk-insertion is necessary when a spatial index already exists and a large amount of new data needs to be added. An example of this process could be the following. If we are indexing data received from an earth-sensing satellite and new data from a specific region that spatially overlaps with the existing index have arrived, we need to insert these new data into the existing index. Loading indexes by inserting elements one-by-one is less efficient than executing specially designed bulk-insertion algorithms, with smart merging techniques. Bulk-insertion is therefore an interesting option for updating spatial indexes when chunks of new data are inserted as a whole. However, bulk-insertion methods in spatial indexes have not been studied in depth by the database research community.

In this paper, we study the efficiency of updating a Quadtree-based index structure when it already exists and a large amount of data is pending to be inserted. In particular, we focus on the xBR$^+$-tree [10], 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 [14] 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. [10]. With this research work, we complete the design and implementation of methods to create and update an xBR$^+$-tree from a dynamic dataset, since a new bulk-insertion method is proposed. That is, (1) in [11] a bulk-loading method is presented for xBR$^+$-trees, (2) in [10] an efficient *one-by-one-insertion* procedure is defined and implemented for this new spatial index, and finally (3) in [13] a deletion algorithm is added to this index for removing object in a one-by-one fashion.

There are two important aspects of the bulk-insertion methods. First, the bulk-insertion itself should be fast enough and the storage utilization should not be degraded with respect to the existing spatial index (i.e. the quality of the structure should be preserved). Second, the spatial query performance should not be compromised by the bulk-insertion process. Keeping this in mind, in this paper, we present the first algorithm for bulk-insertion into xBR$^+$-trees, for big datasets residing on disk and taking into account the previous important aspects. Moreover, using real and artificial datasets of various cardinalities, we present an experimental comparison of this bulk-insertion algorithm vs. the algorithm of loading items one-by-one in a existing xBR$^+$-tree, regarding creation time and the characteristics of the tree created.

This paper is organized as follows. In Section 2 we review related work on bulk-insertion techniques and provide the motivation of this paper. In Section 3, we describe the most important characteristics of the xBR$^+$-tree and the bulk-loading method for this recent spatial index. Section 4 presents our bulk-insertion method for the xBR$^+$-tree. In Section 5, we discuss the results of our experiments. And finally, Section 6 provides the conclusions arising from our work and discusses related future work directions.

2 Related Work and Motivation

This section reviews previous bulk-insertion techniques, that consist of inserting a set of new data into an already existing spatial index at once rather than inserting one new data element at a time. The main target of the bulk-insertion process is to create a good enough spatial index in order to reduce the loading time, the query cost of the resulting index structure, or both. In [1] the bulk-insertion techniques are roughly classified into two categories: merge-based and buffer-based techniques.

– The *merge-based bulk-insertion* techniques are characterized by the following two steps: first, a new small spatial index is created from the new dataset (if has not been created yet) and second, the new small index is merged into the existing one to complete the bulk-insertion process.
– The *buffer-based bulk-insertion* techniques use the *buffer-tree* [2] as buffering technique for the bulk-insertion process.

Most of the bulk-insertion methods belong to the first category and concern the R-tree [8,15,3,5,4,1].

In [8], a bulk-insertion method in which new leaf nodes are built following the Hilbert-packed R-tree technique is proposed. The new leaf nodes are then inserted one-by-one into the existing R-tree using a dynamic R-tree insertion algorithm.

In [15], the *cubetree* is proposed, which is an R-tree-like structure for OLAP applications that uses a specialized packing algorithm. The bulk-insertion algorithm proposed in [15] works as follows. First, the new dataset to be inserted is sorted in the packing order. The sorted list is merged with the sorted list of objects in the existing dataset, which is obtained directly from the leaf nodes of the existing *cubetree*. A new *cubetree* is then packed using the sorted list resulting from merging.

In [3,4], a bulk-insertion technique for R-trees, called STLT (Small-Tree-Large-Tree), is proposed. The STLT technique constructs an R-tree (small tree) from the new dataset and inserts it into the existing R-tree (large tree). To insert a small tree into a large tree, it chooses an appropriate location to maintain the balance of the resulting large tree. That is, the root node of the small tree is then inserted into the appropriate place in the large R-tree, using a specialized algorithm that performs some local reorganization of the existing tree based on a set of proposed heuristics.

In [5], a variant of STLT, called GBI (Generalized Bulk Insertion), is presented. For this technique, the new input dataset is partitioned into a number of clusters by grouping spatially close data items into the same cluster. After clustering, from each of these clusters, R-trees are built. Finally, these R-trees are inserted into the existing R-tree one at a time. The data items not included in any cluster are classified as outliers and inserted one by one using normal R-tree insertion.

In [1], a new Oracle's approach for performing bulk-insertion in R-trees is presented. The characteristics of this approach are: (1) batched insertion into subtrees resulting in fast insertion times, and (2) fast reorganization of subtrees whenever there is an overlap, to ensure good quality of the final R-tree. This approach extends buffering-based techniques by not materializing the auxiliary structures and pushing the data right to the leaves; besides, it merges subtrees whenever they overlap. In general, this bulk-insertion strategy combines multiple inserts and reduces the number of tree traversals.

The basic idea presented in [2] is to attach *buffers* to the internal nodes of an R-tree (except for the root node) in pre-calculated levels and keep the total size of the buffers to fit in memory. Then, when a new data item is inserted, it is stored in the buffer until it gets full. When the buffer is full, data items in the buffer are pushed down to the buffer at the lower level. Since data items are only inserted into the leaf level, it is only when the data objects arrive at the leaf level that disk accesses occur. By using buffers, a data item is inserted as soon as it arrives without having to gather items to perform bulk operations and it is likely that disk accesses are reduced by delaying insertions using such buffers.

In [6], two new extensible buffer-based bulk-loading and bulk-insertion algorithms for the class of Space Partitioning Trees (SPTs, a class of hierarchical data structures that recursively decompose space into disjoint partitions) are presented. The authors adopt the idea of buffer-trees [2] during bulk insertions into SPTs. The main difference over [2] is that SPTs may not be balanced and hence need non trivial clustering and whenever a sufficient number of data items accumulate at the buffers at the leaf level, the algorithm clusters them to form a new part of the tree. The main idea of this algorithm is to build an in-memory tree of the target SPT. Then, data items are recursively partitioned into disk-based buffers using the in-memory tree.

In [9], the idea of the *seeded clustering* for an R-tree is used for a bulk-insertion algorithm which is performed in two steps: seeded clustering and insertion. In the *seeded clustering* step, the algorithm first builds a *seed tree* by taking a few top levels of nodes from a target R-tree (existing tree). The seed tree guides the way the new input data items are clustered. In the *insertion* step, two different methods are proposed. In the one of them, the algorithm takes each data item from a cluster and inserts it into a target R-tree, one at a time, using the standard R-tree insertion method. Although it inserts data items one by one, it reduces the construction cost dramatically because of localized insertions. In the other method, the algorithm builds an R-tree from each of the clusters and inserts them into the target R-tree one at a time in bulk.

The most representative contribution related to bulk-insertion technique for Quadtree index structures has been published in [7]. The main idea is to adapt the bulk-loading algorithm [7] to the problem of bulk-inserting into an existing PMR Quadtree index. The process of the bulk-insertion algorithm is to build a Quadtree in main-memory for the new dataset with the bulk-loading algorithm [7] and then to merge it with the existing disk-resident PMR-Quadtree. It is a merged-based algorithm, since it essentially merges a new Quadtree being built in memory with an existing disk-resident Quadtree, and writes out a new combined disk-resident Quadtree. Moreover, a transformation of this merge-based algorithm to an update-based algorithm is also proposed.

The main motivation of this work is the proposal of a new merge-based algorithm for bulk-insertion of a space-driven Quadtree variant, the xBR$^+$-tree. Note that the xBR$^+$-tree [10] (for more details, see Section 3.1) 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 quadtree variant has all these characteristics.

3 The xBR$^+$-tree and the Bulk-Loading Method

In this section, we present the basics of the xBR$^+$-tree structure and an abstract description of the xBR$^+$-tree bulk-loading algorithm.

3.1 xBR$^+$-tree

The xBR$^+$-tree [10] (an extension of the xBR-tree [14]) is a hierarchical, disk-resident Quadtree-based index structure for multidimensional points (i.e. it is a totally disk-resident, height-balanced, pointer-based tree 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*. We depict the address only for demonstration purposes. 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 [14]. For example, the *Address* 1 represents the NE quadrant of the current space, while the *Address* 12 the SW subquadrant of the NE quadrant of the current space.

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 4 internal nodes that point to 12 leaves is depicted. The region of the root is the original space, which is assumed to have a quadrangular shape with origin (0,0) on the upper left corner and side length 200. The region of the rightmost entry is the SW quadrant (2*) of the original space (the * symbol is used to denote the end of a variable size address). The region of the next (on the left) subquadrant is the NW subquadrant of the SE quadrant of the whole space. For this subbquadrant, the *Address* is 30*. The flag *shape* is set at the value 'SQ' which expresses that this subquadrant is a complete square and thus, no part of its region will be found anywhere in the index, except for the child nodes of the subtree rooted at this entry. The next (on the left) entry covers the whole space of the SW quadrant of the whole space (0*). Finally, the first entry in the root node of this example, expresses the whole space minus the three descendant regions (0*, 30* and 2*), and of course it is a non-complete square area. 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.

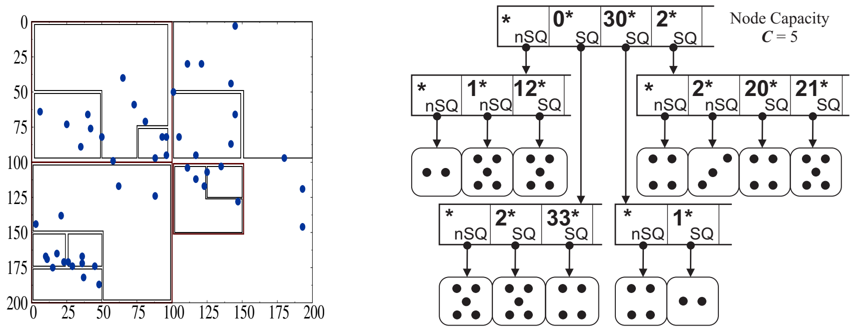

Fig. 1. A collection of 48 points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree.

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.

3.2 Bulk-Loading Method for xBR$^+$-tree

In [11], an algorithm, called Process of Bulk Loading (*PBL*), for bulk-loading xBR$^+$-trees for large datasets residing on disk was presented, using a limited amount of main memory.

PBL consists of four phases. These phases are pipelined and each phase produces an output that is used as input for the next one.

1. During the first phase (*Transformation of input file format*), the initial dataset file is transformed to binary format and is split in two items files.
2. During the second phase (*Partitioning input data*), each of the two input items files is partitioned into items blocks of size $\leq$ *MemoryLimit* in a regular fashion. The resulting blocks are transferred in main memory, as input for the next phase.
3. During the third phase (*Creating the m-xBR$^+$-tree*), for each block of items, a Quadtree (a four way tree) is built top-down in 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.
4. During the last phase (*Merging of trees*), 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.

Improvements of all the above phases are presented in [12]. However, the main algorithmic flow and splitting into phases of [11] remain unchanged in [12].

4 Bulk-Insertion Algorithm for xBR$^+$-tree

In this section, we present the method we developed for bulk insertions into xBR$^+$-trees, called Process of Loading xBR$^+$-trees by Bulk Insertions (*PLBI*). The basic idea is as follows. For each set of points to be inserted in the xBR$^+$-tree, insert these points in the leaves of the tree (the points that fall within the area of the same leaf are handled all together), accessing the leaves from the root in depth-first mode. If a leaf overflows, then create a separate in-memory tree for the subtree rooted at the parent of this leaf and merge the in-memory tree with the rest of the tree.

The algorithm is described in more details as follows. Let's consider a set of points S to be inserted in the xBR$^+$-tree. We utilize a main memory area M of size *MemoryLimit*. If the space needed for S is larger than M, we transfer S to M in subsets that fit within M and process each such subset independently. Otherwise, we transfer to M and process the whole of S. Processing of a set of points S_M within M (in general, a subset of S) is recursive (following depth-first traversal).

- We call the recursive procedure for the xBR$^+$-tree root and S_M and make recursive calls down to the level of the parents of the leaves. The input of a recursive call is a node and a set of points which is the subset of points of S_M that fall within the region of this node.
- For each (internal) node I visited and the corresponding set of points B_I, we examine the region entries of I, from the rightmost to the leftmost, and, for each such region entry E, we determine the subset B_E of B_I that falls within E.
- If I is not a parent of leaves, for each region entry E, we apply recursively the same procedure for the child node corresponding to E and B_E.
- If I is a parent of leaves, for each region entry E, we insert B_E in the child node (leaf) corresponding to E.
- After insertions to all children of I have been completed, we examine if any of these children (leaves) has overflown.
- If none of these children has overflown, for each one of them, we update its *DBR* value and, if needed, we update *DBR* values of its ancestors (possibly, up to the root level) and the procedure returns to the previous stage of recursion.
- If any child has overflown, we create an m-xBR$^+$-tree (a main memory xBR$^+$-tree) m for I (Phase 3 of [11]) and gradually merge m with the xBR$^+$-tree, as follows.
  - We locate the leftmost non-leaf node I_m of m and replace I with I_m (this is feasible, since I and I_m both correspond to the region of the root of m).

- We merge, one-by-one, the sub-trees that correspond to the siblings of I_m, from left to right, with the xBR$^+$-tree (Phase 4 of [11]). At this stage, processing of I has been completed.
- Up to the root level of m, we move to the parent of the node for which processing has been completed and we merge, one-by-one, the sub-trees that correspond to the siblings of this node with the xBR$^+$-tree (Phase 4 of [11]).

When merging of m with the xBR$^+$-tree has been completed, the procedure returns to the previous stage of recursion, and if no changes in the structure of the index of the xBR$^+$-tree have been done, the procedure will be continued with the next entry E and its corresponding subset B_E. Otherwise, the procedure will be restarted from the root of the xBR$^+$-tree with the unprocessed data subset.

In Figure 1, a collection of 48 points in a squared space with origin (0,0) and side length of 200 is depicted. In the right part of this figure, the xBR$^+$-tree index for this dataset created by *PLBI* is depicted. The first 16 points were inserted in one segment using *PBL* and the rest 32 points were inserted in two equal segments (of 16 points each) using *PLBI*.

Contrary to [11] and adopting improvements of [11] that have been incorporated in [12]:

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

– In order to merge sub-trees of m with the xBR$^+$-tree, if the sub-tree of m under processing and the xBR$^+$-tree have equal heights, 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, or more nodes that are pointed by a new root that is created. If the xBR$^+$-tree is higher, we first locate in the xBR$^+$-tree the direct ancestor of the root of the m sub-tree under processing (that resides at the same level as this root) and then perform a merge between equal height subtrees. If, however, the sub-tree of m under processing is higher, we save it in secondary memory, we adjust the region of this tree to correspond to the whole space and then we apply the previous procedure of merging the roots of the two trees. Following the merge and re-partition approach, instead of leaving the regions of the original roots unchanged [11], we achieve better partitioning of space and increased query performance in the resulting tree.

Note that applying the same procedure for building the m-xBR$^+$-tree and for merging of trees as in [11] is possible. However, the alternatives ([12]) followed here improve performance.

5 Experimental Results

We designed and run a large set of experiments to compare *PLBI* to the Process of Loading xBR$^+$-trees by inserting items One-by-One (*PLObO*). We used real spatial datasets of North America, representing 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 NArdN, we chose the two points with *min* and *max* coordinates of the MBR of each line-segment (i.e. we created a new dataset, |NArdND| = 1138164 points. The data of these three files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 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 three big real datasets[4]. They represent water resources of North America (Water) consisting of 5836360 line-segments and 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 queries, *Point Location Query (PLQ)*,*Window Query (WQ)*, *Distance Range Query (DRQ)*, *K Nearest Neighbor Query (KNNQ)*, *Constrained K Nearest Neighbor Query (CKNNQ)*, and for two dual dataset queries, *K Closest Pairs Query (KCPQ)* and *Disatance Join Query (DJQ)*, counting disk read accesses (I/O) and total execution time.

5.1 Experiments for Tree Building

To study tree building by *PLBI*, we split the whole (unsorted) dataset to a sequence of subsets (segments), we use the *PBL* algorithm ([11]) to create an xBR$^+$-tree from scratch by loading the first one of these segments as one block and, subsequently, we insert each of the rest of these segments using *PLBI.* The size of each *Segment (S)* is a percentage of *MemoryLimit (ML)*, which is a percentage of the cardinality of the corresponding dataset. For each of the 9 above datasets, we constructed 45 xBR$^+$-trees, using *ML* equal to 1%, 2% and 4%, *S* equal to 20%, 40%, 60%, 80% and 100% and *node size* equal to 1KB, 4KB and 8KB.

In Table 1, for four indicative cases of datasets (two big and one smaller real datasets and one synthetic dataset) and *ML* values, we present the effect of

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

Table 1. Tree creation characteristics, using *PLBI* and *PLObO* .

ML (%)	*S* (%)	H	Iocc (%)	Locc (%)	Size (MB)	Time (s)
Dataset : Park						
2	20	4	64.93	65.58	685	35.29
2	40	4	66.15	65.58	684	33.50
2	60	4	66.56	65.58	684	33.84
2	80	4	66.89	65.58	684	35.85
2	100	4	67.2	65.58	684	35.03
–	–	4	61.93	65.31	688	107.24
Dataset : Build						
1	20	5	67.72	65.66	6815	462.4
1	40	5	68.58	65.65	6814	463.2
1	60	5	69.09	65.65	6813	546.7
1	80	5	69.36	65.65	6813	548.6
1	100	5	69.69	65.65	6812	534.6
–	–	5	61.9	65.6	6833	1378

ML (%)	*S* (%)	H	Iocc (%)	Locc (%)	Size (MB)	Time (s)
Dataset : NArdND						
2	20	4	63.1	66.04	67.3	2.38
2	40	4	63.85	66.04	67.3	2.02
2	60	4	64.04	66.04	67.3	1.96
2	80	4	64.43	66.04	67.3	1.94
2	100	4	64.82	66.04	67.3	1.99
–	–	4	58.11	64.3	69.2	7.74
Dataset : 1000KCN						
4	20	4	66.1	64.43	60.6	3.47
4	40	4	66.1	64.43	60.6	3.36
4	60	4	66.1	64.43	60.6	3.35
4	80	4	66.33	64.43	60.6	3.26
4	100	4	66.1	64.43	60.6	3.29
–	–	4	63.44	64.48	60.6	7.33

creating an xBR$^+$-tree with node size equal to 4KB by *PLBI* algorithm on 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 dataset, we added a line that presents the same tree characteristics by *PLObO*.

Regarding the characteristics of the trees created by the two methods, we observe the following.

– Both trees have the same height.
– Internal nodes occupancy percentage of *PLBI* trees is, on the average, 5% higher than the one of *PLObO* trees.
– Leaf nodes occupancy percentage of *PLBI* trees is, on the average, 2% higher than the one of *PLObO* trees.
– Size in disk of *PLBI* trees is up to 3% less than the one of *PLObO* trees (resulting mainly from the higher leaf nodes occupancy of *PLBI* trees).

The conclusions are analogous for the rest of the trees built. Regarding the total creation time of the trees created by the two methods, *PLBI* is the big winner (on the average, *PLBI* is 50% faster). The speed improvement is maximized for large datasets. In general, the *PLBI* speed is slightly increased as the *Segment* size increases. In some cases, the opposite trend is observed, since the distribution of data may cancel the benefit of using a larger *Segment* size.

5.2 Experiments for single dataset spatial queries

For all query experiments, we used node size equal to 4K, for trees created by both algorithms (the *PLBI* trees were created using *ML* equal to 2% and *S* equal to 60%). For *PLQs*, we executed two sets of experiments for the 9 datasets. In

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

Query	Number of Disk Read Accesses			Execution Time		
	tie diff$\leq$ 5%	*PLBI* wins diff$>$ 5%	*PLObO* wins diff$>$ 5%	tie diff$\leq$ 5%	*PLBI* wins diff$>$ 5%	*PLObO* wins diff$>$ 5%
PLQ-existing points	100.0	0.0	0.0	33.3	66.7	0.0
PLQ-non-existing points	77.8	22.2	0.0	0.0	77.8	22.2
WQ	88.9	11.1	0.0	11.1	75.6	13.3
DRQ	86.7	13.3	0.0	13.3	75.6	11.1
KNNQ	72.2	25.0	2.8	13.9	75.0	11.1
CKNNQ	77.8	22.2	0.0	5.6	80.6	13.9
KCPQ	33.3	66.7	0.0	0.0	100.0	0.0
DJQ	50.0	50.0	0.0	6.7	93.3	0.0

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 *PLBI* / *PLObO* have a performance that is more than 5% better than their rivals (Columns 3 and 6 / 4 and 7, respectively). 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 *PLBI*, on the average, perform better in both metrics.

For *WQs*, we executed 54 experiments (9 datasets $\times$ 6 query window sizes). Each experiments was executed for 4096 query windows (having size 1/4096 of the total space) for each dataset. All experiments are summarized in the 3rd data line of Table 2. It is clear that the trees created by *PLBI*, on the average, perform better in both metrics.

For *DRQs*, the depth-first (DF) was used (since this is the best one [11]) in 54 experiments (9 datasets $\times$ 6 sets of query circles). All experiments are summarized in the 4th data line of Table 2. It is clear that the trees created by *PLBI*, on the average, perform better in both metrics.

For *KNNQs*, the HDF algorithm was used (since this is the best one [11]) in 36 experiments (9 datasets $\times$ 4 K-values, using 4096 query points, in all cases). All experiments are summarized in the 5th data line of Table 2. It is clear that the trees created by *PBL*, on the average, perform better in both metrics.

Finally, for *CKNNQs*, the BF algorithm was used (since this is the best one [11]) 36 experiments (9 datasets $\times$ 4 K-values, using 4096 query circles, in all cases). All experiments are summarized in the 6th data line of Table 2. It is clear for this query, too, that trees created by *PLBI*, on the average, perform better in both metrics.

Overall, trees created by *PLBI*, 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 of trees created by *PBLI* is related to the structural difference between the two trees. The *PBLI* algorithm can achieve better grouping of subregions, since all data/entries are known before each node is created and this improved grouping affects positively the time spent for CPU computations during query processing.

5.3 Experiments for dual dataset spatial queries

For *KCPQs* / *DJQs*, the HDF algorithm was used (since this is the best one [11]) in 30 experiments (6 combinations of datasets $\times$ 5 K-values, in all cases). All experiments are summarized in the 7th / 8th data line of Table 2. It is clear that trees created by *PLBI* perform, on the average, significantly better in both metrics. The explanation for the significantly improved performance of trees created by *PLBI* is related to the better grouping of subregions and the fact that the execution of *KCPQs* / *DJQs* corresponds to multiple *KNNQs* / *DRQs*, maximizing the benefits resulting from *PLBI*.

6 Conclusions and Future Work

In this paper, for the first time in the literature, we present an algorithm for bulk insertions into xBR$^+$-trees, using a limited amount of RAM. This algorithm was implemented and extensive experimentation was performed for comparing the characteristics, the creation time and the query performance of trees loaded by the new algorithm and trees loaded by the traditional way of inserting items one-by-one. These experiments show that, trees loaded by bulk insertions have comparable structural characteristics to traditionally loaded trees, are created in significantly less time (the bulk insertion method is the big winner of creation time performance) and perform better / significantly better in processing single / dual dataset queries.

Future work plans include:

- The development of a cost model to analytically study the performance of various operations on xBR$^+$-trees (insertion of items one-by-one, bulk insertion, bulk loading, processing of different single and dual dataset queries).
- The development of parallel bulk-insertion methods for xBR$^+$-trees, utilizing multiple CPUs / GPU cores.
- An extension of the experimental study to more real datasets (to stregthen the validity of the results in real applications) and to data of higher dimensions.

– Embedding of xBR$^+$-trees created by bulk insertion in SpatialHadoop[5] and study their performance in relation of other space partitioning strategies, already existing in this system.
– Creating variations of bulk-insertion methods for xBR$^+$-trees that will take advantage of the characteristics of SSD disks.
– Relax the constraint that an overflown node region in an xBR$^+$-tree has to be split to equal parts (examine splitting to possibly unequal rectangles, instead of equal subquadrants) and test the performance of bulk insertion, bulk loading and insertion one-by-one.

References

1. N. An, K.V.R. Kanth and S. Ravada: "*Improving Performance with Bulk-Inserts in Oracle R-trees*". VLDB Conf., pp. 948-951, 2003.
2. L. Arge, K.H. Hinrichs, J. Vahrenhold and J.S. Vitter: "*Efficient Bulk Operations on Dynamic R-trees*". Algorithmica 33(1), pp. 104-128, 2002.
3. L. Chen, R. Choubey and E.A. Rundensteiner: "*Bulk-Insertions info R-Trees Using the Small-Tree-Large-Tree Approach*". ACM-GIS Conf., pp. 161-162, 1998.
4. L. Chen, R. Choubey and E.A. Rundensteiner: "*Merging R-Trees: Efficient Strategies for Local Bulk Insertion*". GeoInformatica 6(1), pp. 7-34, 2002.
5. R. Choubey, L. Chen and E.A. Rundensteiner: "*GBI: A Generalized R-Tree Bulk-Insertion Strategy*". SSD Conf., pp. 91108, 1999.
6. T.M. Ghanem, R. Shah, M.F. Mokbel, W.G. Aref and J.S. Vitter: "*Bulk Operations for Space-Partitioning Trees*". ICDE Conf., pp. 29-40, 2004.
7. G.R. Hjaltason and H. Samet: "*Speeding up Construction of PMR Quadtree-based Spatial Indexes*". VLDB Journal 11(2), pp. 109-137, 2002.
8. I. Kamel, M. Khalil and V. Kouramajian: "*Bulk insertion in dynamic R-trees*". SDH Conf., pp. 3B.313B.42, 1996.
9. T. Lee, B. Moon and S. Lee: "*Bulk insertion for R-trees by seeded clustering*". Data Knowl. Eng. 59(1), pp. 86-106, 2006.
10. G. Roumelis, M. Vassilakopoulos, T. Loukopoulos, A. Corral and Y. Manolopoulos: "*The xBR$^+$-tree: An Efficient Access Method for Points*". DEXA Conf., pp. 43-58, 2015.
11. G. Roumelis, M. Vassilakopoulos, A. Corral and Y. Manolopoulos: "*Bulk-Loading xBR$^+$-tree*". MEDI Conf., pp. 57-71, 2016.
12. G. Roumelis, M. Vassilakopoulos, A. Corral and Y. Manolopoulos: "*An Efficient Algorithm for Bulk-Loading xBR$^+$-trees*", to appear in Computer Standards & Interfaces, 2017.
13. G. Roumelis, M. Vassilakopoulos and A. Corral: "*The Deletion Operation in xBR-Trees*". PCI Conf., pp. 138-143, 2012.
14. G. Roumelis, M. Vassilakopoulos and A. Corral: "*Performance Comparison of xBR-trees and R*-trees for Single Dataset Spatial Queries*". ADBIS Conf., pp. 228-242, 2011.
15. N. Roussopoulos, Y. Kotidis and M. Roussopoulos: "*Cubetree: Organization of and Bulk Updates on the Data Cube*". SIGMOD Conf., pp. 89-99, 1997.
16. S. Shekhar and S. Chawla: "*Spatial Databases - a Tour*", Prentice Hall, 2003.

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