Computing
https://doi.org/10.1007/s00607-019-00709-4

Bulk-loading and bulk-insertion algorithms for xBR$^+$-trees in Solid State Drives

George Roumelis[1] · Athanasios Fevgas[1] · Michael Vassilakopoulos[1] · Antonio Corral[2] · Panayiotis Bozanis[1] · Yannis Manolopoulos[3]

Received: 19 March 2018 / Accepted: 8 February 2019

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 for point data, which is very efficient for processing spatial 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. 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. In this paper, we modify previous bulk-loading and bulk-insertion algorithms for xBR$^+$-trees to achieve higher performance by taking advantage of the special features of Solid State Drives (SSDs). SSDs have attracted database developers, mainly due to their higher read performance (thanks to their internal parallelism) than Hard Disk Drives. Using real and artificial datasets of various cardinalities, we experimentally compare the modified algorithms against their predecessors and show that the modified algorithms are clear winners regarding performance.

Keywords Spatial indexes · Bulk-loading · Bulk-insertion · xBR$^+$-trees · Solid State Drives

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 insertions of new large datasets into an existing dataset is of particular interest. It is important to add newly collected data into an existing

✉ Michael Vassilakopoulos
mvasilako@uth.gr

Extended author information available on the last page of the article

Published online: 12 March 2019

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 [45].

If the dataset is *dynamic* (i.e. 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 as a whole (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* [21]).

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 combining data that is already indexed by a disk-resident spatial index and data that has not yet been indexed. 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 should therefore be considered as an option for updating spatial indexes when chunks of new data are inserted as a whole.

Non-volatile memories (NVM) are revolutionizing the storage hierarchy, enabling new devices with exciting features compared to their predecessor magnetic disks. NAND flash is the most widely used non-volatile memory nowadays, however upcoming technologies such as Intel's-Micron's 3D XPoint are promising even better performance [19].

Storage devices based on NAND Flash quickly became popular both in consumer and enterprise markets thanks to their unique features. Ultra-fast reads and writes, high densities, low power consumption and shock resistance are some of the reasons that explain their popularity. Nevertheless, NAND Flash exhibits some intrinsic characteristics. Reads and writes are performed at page level whilst erasures at block level. Reads are faster than writes and writes are faster than erases. Unlike magnetic disks, in-place updates are not feasible, meaning that pages must be erased before being re-programmed again. Continuous erase operations degrade the ability of Flash cells to retain their charge. Therefore, Flash blocks can sustain a certain number of program/erase cycles before wearing out.

Bare Flash memory chips were, initially, utilized in mobile devices and consumer electronics. The increasing demands for fast and reliable storage led to the advent of Solid State Drives (SSDs). SSDs are storage devices that comprise a bunch of Flash packages, one or more controllers and DRAM [13]. Latest SSDs incorporate from a few, to dozens of NAND packages achieving capacities as high as several terabytes. A low power 32-bit embedded CPU is usually employed as Flash controller executing the firmware code that manages the SSD. DRAM is used to store address mapping information, metadata and for caching user data as well. Finally, host interface controller interconnects the SSD with the host system. Flash Translation Layer (FTL) firmware is executed on the SSD controller performing fundamental management tasks [10]. FTL maps logical addresses as visible by the file system to physical addresses in the flash memory. To separate logical from physical addresses it implements an out-of-place updates mechanism hiding the complexity of programming Flash memory cells. Moreover, FLT is responsible for wear-leveling, garbage collection and bad block management.

We focus on the xBR$^+$-tree [41], 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 [36] in the node structure and the splitting process. The xBR$^+$-tree is a very efficient structure. It has been extensively compared against the R*-tree and the R-tree regarding tree building and processing of the most common spatial queries [39], medium and large real and synthetic datasets, and it has been shown that it is a big winner in execution time in all cases and a winner in I/O performance in most cases. In [37,40] an efficient bulk-loading method, while, in [38] an efficient bulk-insertion method is presented for xBR$^+$-trees. In this paper, we modify these bulk-loading and bulk-insertion algorithms for xBR$^+$-trees to achieve higher performance by taking advantage of the special features of Solid State Drives (SSDs). SSDs have attracted database developers, mainly due to their higher read performance (thanks to their internal parallelism) than Hard Disk Drives (HDDs).

Moreover, using real and artificial datasets of various cardinalities, we experimentally compare the modified algorithms against their predecessors on SSDs and show that the modified algorithms are clear winners regarding performance.

This paper is organized as follows. In Sect. 2 we review related work on bulk-insertion and bulk-loading techniques, as well as, on indexes taking advantage of SSDs performance and provide the motivation of this paper. In Sect. 3, we describe the most important characteristics of the xBR$^+$-tree. Sections 4 and 5 present the modified bulk-loading/bulk-insertion method for this spatial index that takes advantage of the special features of SSDs for achieving increased performance. In Sect. 6, we present the experimental comparison performed and discuss the results of the experiments. And finally, Sect. 7 provides the conclusions arising from our work and discusses related future work directions.

2 Related work and motivation

In this section, we first present an overview of the most representative contributions on bulk-loading (sorted-based, buffer-based and sample-based ones) and bulk-insertion

(merge-based and buffer-based ones) techniques. Then, we review on spatial indexes taking advantage of SSDs performance.

2.1 Bulk-loading algorithms for spatial indexes

This subsection reviews previous bulk-loading methods, which, according to [41], 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 [5]. They employ external (*buffers*) attached to the internal nodes of the tree except for the root node.
- The *sample-based bulk-loading* methods use a sample of the input that fits into memory to build up the target index.

There are several methods that belong to the *sort-based bulk-loading* category, but the most characteristic ones are proposed in [1,18,20–22,25,29,37,40,43]. In [43], 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 values of the objects in a certain dimension. In [25] 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 [29]. The method starts sorting the data source with respect to the first dimension (e.g. using the center of the spatial objects). Then, a number of 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. In [1] a sort-based query-adaptive loading for building R-trees optimally designed for a given query profile is presented. Finally, in [2] a scalable alternative MapReduce approach to parallel loading of R-trees using a level-by-level design, based on [1], is introduced.

In [22], the proposed approach for bulk-loading on PMR-Quadtrees is based on the idea of trying to fill up memory with as much of the Quadtree as possible before writing some of its nodes on disk. In [20], improved versions of the bulk-loading algorithms studied in [22] for PMR-Quadtrees are presented, assuming that the algorithms are implemented using a *linear Quadtree*. Finally, an extended version of [20,22] is presented in [21], where the problem of creating and updating spatial indexes in situations of a *dynamic* database is addressed and detailed algorithms for the proposed *sort-based bulk-loading* method are presented. According to [18], the two extensible buffer-based algorithms for bulk operations can be applied to Quadtrees, because this is a type of space-partitioning tree, but no implementation and no experimentation were performed. Finally, in [37] a new sort-based approach for bulk-loading xBR$^+$-trees residing on disk, using a limited amount of main memory, is presented. Later, in [40],

an improvement of such bulk-loading algorithm is proposed, reducing movements of data items, making better use of internal memory and achieving better partitioning of space.

The most representative of *buffer-based bulk-loading* methods are proposed in [4,15,18]. In [15], 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 the *buffer-tree*). From this structure, the desired index structure is built up incrementally bottom-up, one level at a time. Arge et al. [4] achieves a similar effect by using a regular R-tree structure and attaching buffers to nodes only at certain levels of the tree. In [18], 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.

The most remarkable *sample-based bulk-loading* algorithms have been proposed in [6,12,14,32,47]. In [12], 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 [6], 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 [14], 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 [47], two sample-based bulk-loading algorithms for dynamic metric access methods are presented. They are based on the idea of covering radius of representative items employed to organize data in hierarchical data structures. These proposed algorithms were applied to Slim-trees, although they can be used in other metric access methods. In [32], a generic framework for R-tree bulk-loading based on sampling on a parallel architecture was introduced.

2.2 Bulk-insertion algorithms into spatial indexes

This subsection 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 to reduce the loading time, the query cost of the resulting index structure, or both. In [3] 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 it 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* [5] as a buffering technique for the bulk-insertion process.

Like for bulk-loading, the basic idea for *buffer-based bulk-insertion* techniques presented in [5] 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 appropriate so that they 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. In [18], 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 [5] during bulk insertions into SPTs. The main idea of this bulk-insertion 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.

Most of the bulk-insertion methods belong to the first category and concern the R-tree [3,8,9,11,21,26,28,38,42]. In [26], 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 [8,9], a bulk-insertion technique for R-trees, called Small-Tree-Large-Tree (STLT), 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, by a specialized algorithm, an appropriate location to maintain the balance of the resulting large tree. In [11], a variant of STLT, called Generalized Bulk Insertion (GBI), is presented. For this technique, the new input dataset is partitioned into a number of clusters. After clustering, from each of these clusters, R-trees are built, and these R-trees are inserted into the existing R-tree one at a time. Finally, the data items not included in any cluster (outliers) are inserted one by one using normal R-tree insertion. In [28], 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 [3], 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.

In [42], a bulk-insertion algorithm for the *cubetree* is proposed. This algorithm consists in sorting the new dataset to be inserted in the packing order and in merging it with the sorted list of objects in the existing *cubetree*, which is obtained directly from the leaf nodes.

For Quadtrees, a bulk-insertion technique has been published in [21]. The main idea is to adapt the bulk-loading algorithm 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 [21] and then to merge it with the existing disk-resident PMR-Quadtree.

Finally, in [38] we present a new merge-based algorithm for bulk insertion into xBR$^+$-trees that incorporates extensions of previous bulk-loading techniques [40].

2.3 Flash efficient spatial indexes

Although FTL aims to mitigate Flash memory idiosyncrasies, SSD devices inherit some of its characteristics. Many different SSD models exist in the market today with high diversity both in cost and specifications. However, in the most cases read operations are faster than write ones, while sequential I/O (reads and writes) is faster than the corresponding random. Furthermore, random writes may trigger garbage collection operations inducing unpredictable response times. On the other hand, contemporary SSDs exhibit high level of internal parallelism which can provide with high performance gains user applications [35]. All these have motivated researchers to investigate Flash efficient spatial indexes. Most of the works are about R-Tree and its variants.

Authors in [48] present an efficient implementation of R-Tree for Flash storage. They exploit an in-memory buffer to cache updates and persist them in batches when the buffer gets full. The same technique is also exploited for Aggregated R-tree in [34]. LCR-Tree [31] stores log records to a dedicated log section in Flash. It always occupies only one page to store all the deltas for a particular node, thus achieves to access a tree node with only one additional page read.

FOR-tree [24] exploits overflow nodes to postpone node splitting which causes small random writes. A new buffering algorithm is also introduced that adapts to the unbalance nature of FOR-tree.

FAST [44] is a generic framework for spatial indexes. It utilizes the original insertion and update algorithms of the underlying tree, logging the result of the operation rather than the operation itself. It buffers updates in RAM and commits them to the SSD according to a flushing policy. Another such a generic framework is presented in [7].

Authors in [16] study and compare the performance of R*-Tree on HDDs and SSDs: The fraction of data an index on HDD can read before a full scan completes is much lower than the assumed 5%. On SSD this fraction is 20–45% (depends on page size). As the number of threads increases, SSDs perform much better (up to one order of magnitude). Page size does not alter drastically the performance in SSDs as it does in HDDs. As k increases, k-nn queries are performing much better in SDDs than HDDs. Scan outperforms R*-trees on HDDs at 11 dimensions, while the breakpoint for SSDs is 17.

F-KDB [30] is a Flash efficient K-D-B-Tree that also uses logs records to defer small random writes, improving its update performance. A 2D grid file-like structure (MicroGF) for raw Flash in wireless sensor networks is proposed in [30].

Last, a study on the performance of Grid-File in SSDs is presented in [17] and a buffering scheme for batching updates is demonstrated.

Considering the high performance of xBR$^+$-trees in HDDs [39], the efficient bulk-loading [37,40] and bulk-insertion [38] algorithms developed for these trees in HDDs, the improved performance characteristics and the internal parallelism of SSDs in comparison to HDDs, and noting that (to the best of our knowledge) there is no work on bulk-loading/insertion algorithms for spatial indexes in SSDs, in this paper, we develop the first such algorithms for xBR$^+$-trees, by extending previous work on bulk-loading [37,40] and bulk-insertion [38] algorithms for these trees.

Database systems exploit in-memory buffers to increase performance reducing accesses to secondary storage. The advent of SSDs introduced the demand for Flash efficient buffer managers [27]. CFLRU [33] is the first buffering algorithm which considers the asymmetry in execution times of read and write operations, prioritizing the eviction of clean pages. Extending CFLRU a new policy (AD-LRU) proposed taking into account the frequency of page accesses also [23]. Furthermore, a buffering algorithm for tree indexes is introduced in [49] which utilizes different priorities according to the type of node (leaf of internal). In this paper, we employ buffering that is suitable for accelerating the performance of our algorithms (details are given in Sects. 4, 5).

3 The xBR$^+$-tree

The xBR$^+$-tree [41] (an extension of the xBR-tree [36]) 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. Data Bounding Rectangle (DBR) 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 [36]. 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

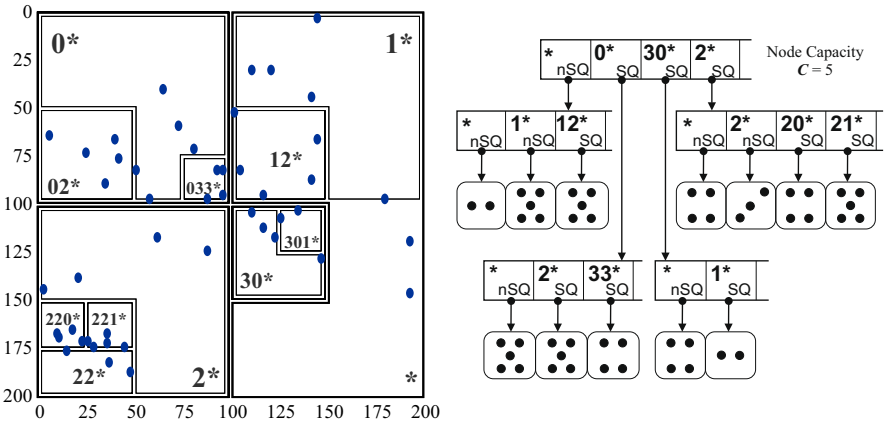

Fig. 1 A collection of 48 points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree

preorder traversal of the Quadtree that corresponds to the internal node). For example, in Fig. 1 right we depict the structure of an xBR$^+$-tree. The tree consists of an internal node (a root) that points to 4 internal nodes that point to 12 leaves is depicted. In Fig. 1 left we depict the data points and the partitioning of space corresponding to this tree. The regions of the entries of the root are drawn with thick lines, while the regions of the entries at the lowest level pointing to the leaves are drawn with thin lines. The region of the root is the original space, which is assumed to have a square 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 subquadrant, 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 NW 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.

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 the xBR$^+$-tree on SSDs

In this section, we present the *Process of Bulk-Loading* xBR$^+$-trees *on SSDs* (*PBLS*), that has evolved from the *Process of Bulk-Loading* xBR$^+$-trees (*PBL+*) [37,40]. This method consists of four phases.

4.1 Phase 1 (transformation of input file format)

During this phase, 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) regularly. This means that, for d dimensions, we divide the range of the space region (a d-dimensional hyper-cube, or d-cube, for short) in the middle of each dimension (e.g. for 2 dimensions, we divide the original square region in the middle of its x-range and in the middle of its y-range), creating 2^d d-cubes of equal size (e.g. for 2 dimensions, 4 quadrants of the original square space), and we transfer, in each of these files, data depending on the d-cube (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 side. 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, and directly proceed to Phase 3 for this block. Otherwise, if the amount of data in this binary file remains large, we proceed to Phase 2 for this file (note that Phase 1 is non-recursive, while Phase 2 is the recursive extension of Phase 1 and its first recursive call is driven by Phase 1).

We use one buffer for each of the files that this phase uses (one for the input file and 2^d for the output files). The input points are transferred to the appropriate output buffer until either the input buffer gets empty (therefore, we load it again with points), or one of the output buffers gets full (therefore, we save this buffer to the connected output file; this approach exploits the speed of SSDs when executing massive writing operations, termed "massive writing advantage of SSDs", or "massive SSD writing", for short, in the following). This process continues until all the input points are partitioned to the output files. In Algorithm 1, we depict Phase 1 of *PBLS*. The use of these buffers appears in Line 2.

4.2 Phase 2 (partitioning input data)

In this phase, 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, for the ease of exposition and w.l.o.g., let's consider that our data are 2d points and that 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. Afterwards, we examine the cardinality of each file and if it is smaller than or equal

Algorithm 1 *PBLS* Phase1

Input: file f

1: **create** files $t_0, t_1, \cdots, t_{2^d-1}$ ▷ *temporary files used for Phase2 calls*
2: **split** f in files $f_0, f_1, , \cdots, f_{2^d-1}$ regularly ▷ *split space in 2^d equally sized d-cubes*
(using one buffer for each file to exploit massive SSD writing)
3: **for** $i = 0 \,..\, 2^d - 1$ **do**
4: **if** sizeof(f_i) $> MemoryLimit$ **then**
5: **Phase2**($f_i, t_0, \cdots, t_{2^d-1}$) ▷ *1st level call of Phase2 (empty temporary files)*
6: **else**
7: **read** f_i in memory block b
8: **Phase3**(b)

to *MemoryLimit* we read this file in a memory block and then we call the routine of Phase 3 for this memory block. Otherwise, we call recursively the current routine, passing as arguments this file as input file and the same temporary files as output files. During the execution of this routine, data will be appended to each of these output files after its current end. In each recursive call, the input file (the first parameter of the line 4 call) is one of the temporary files (one of the rest parameters of the line 4 call). This file will be used as input (for its continuous portion containing the points of the corresponding subquadrant) and as output after its current end. Since the process is always executed in Depth-First mode, the points of a subquadrant in this temporary file are partitioned, while the points that wait for processing are not affected.

Note that, when we call the routine of Phase 3 for a temporary file (corresponding to a child node), we pass as parameter values the group of points contained in this file and the region of its parent. In this way, this group of points will belong to a parental region, and thus, we obtain 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.

During this phase, we use the same buffers as in the previous phase, exploiting the massive writing advantage of SSDs. In Algorithm 2, we depict Phase 2 of *PBLS*.

Algorithm 2 *PBLS* Phase2

Input: file f_i, files $t_0, t_1, \cdots, t_{2^d-1}$

1: **split** f_i in files $t_0, t_1, \cdots, t_{2^d-1}$ regularly ▷ *split f_i.region in 2^d equally sized d-cubes*
(using one buffer for each file to exploit massive SSD writing)
2: **for** $i = 0 \,..\, 2^d - 1$ **do**
3: **if** sizeof(t_i) $> MemoryLimit$ **then**
4: **Phase2**($t_i, t_0, t_1, \cdots, t_{2^d-1}$) ▷ *recursive call (non-empty temporary files)*
5: **else**
6: **read** t_i in memory block b
7: **Phase3**(b)

4.3 Phase 3 (creation of the *m*-xBR$^+$-tree)

The third phase 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 [37] 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 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 (in the case of 2 dimensions, or in general, of 2^d equally sized d-subcubes) 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 of the leaves we can transfer the data included in them to the parental region without causing overflow of the leaf corresponding to 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. In [37,40], we issued an independent write call for each leaf.

In this work, we use a buffer (called *Group Write Buffer*, or GWB) that groups the leaves in sequences of consecutive runs and whenever this buffer gets full, its contents are written to disk (this approach minimizes write calls and maximizes the data segments written for each call and, therefore, exploits the massive writing advantage of SSDs). More specifically, GWB is a buffer that can store a number of node runs, in the form *start node address—length of run*. A read/write call for a node does not read/write this node from/to disk, but inserts it at the last run of the buffer (if it is adjacent with the last node of this run), or creates a new buffer run starting with this node. Note that, due to our algorithms, it is not possible a new run to intersect with previous runs in GWB. When GWB gets full, or a phase using it finishes, its runs of nodes are read from/written to disk using massive I/O calls.

Note that, to create a valid tree, writing of the leaves of the *m*-xBR$^+$-tree to disk through GWB should be done as soon as all these leaves have been determined (which takes place in Phase 3).

Part B: Creation of the levels above leaves
This part of Phase 3 takes place in main memory, without the need to execute any I/O operation, so the algorithm of [40] remains unchanged. The xBR$^+$-tree leaf-node entries created by the previous process, but not the data of the leaves which are already saved in the disk, 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.

Our goal, in case an xBR$^+$-tree node has overflown when inserting items one-by-one into it, is to split this node in two nodes with cardinalities of minimum possible difference [36,41,46]. 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. Otherwise, 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 the 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 we repeat the procedure above, considering as leaf nodes the nodes of the m-xBR$^+$-tree of the previous level, but traversing the same tree T. We continue creating levels of nodes of the m-xBR$^+$-tree until the number of nodes becomes smaller than or equal to C, the m-xBR$^+$-tree is completely created and Phase 3 for the current group is completed.

In Algorithm 3, we depict Phase 3 of *PBLS*. The use of GWB appears in Line 2.

Algorithm 3 *PBLS* Phase3

Input: memory block b
1: recursively, **split** b regularly to create leaves of the m-xBR$^+$-tree and **save** information for its internal nodes in an auxiliary tree T
2: **save** leaves of the m-xBR$^+$-tree (using GWB to exploit massive SSD writing)
3: **pack** T nodes pointing to level-0 nodes of m-xBR$^+$-tree
4: **if** #$leaves$ of $T > C$ **then**
5: $NextLevel \leftarrow$ TRUE
6: **else**
7: $NextLevel \leftarrow$ FALSE
8: $l = 0$
9: **while** $NextLevel =$ TRUE **do**
10: **remove** packed T nodes
11: **if** #$nodes$ of lowest level of $T \leq C$ **then**
12: $NextLevel \leftarrow$ FALSE
13: $l \leftarrow l + 1$
14: **pack** T nodes pointing to level l nodes of m-xBR$^+$-tree
15: **Phase4**(m-xBR$^+$-tree)

4.4 Phase 4 (merging of trees)

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 iterations of the bulk-loading process.

During the first execution of this phase, the xBR$^+$-tree in disk is empty, thus it is necessary to access all the nodes of the m-xBR$^+$-tree and store them in disk. As in

the previous phases, we use the GWB to take advantage of the speed of SSDs when executing massive writing operations. Note that we must take care to write all the contents of the GWB into the disk. 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 $h_m = h_d$, then, after saving the whole m-xBR$^+$-tree using the GWB, except for its root, we examine if the merging of the two roots leads to an overflow, or not. For more details, see [40].

If $h_m < h_d$, then, using a Depth-First traversal of the xBR$^+$-tree (like in a Point Location Query) and starting from the root of the xBR$^+$-tree (R_d), we seek for the region that is the closest ancestor of the region of the root of the m-xBR$^+$-tree (R_m). 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 containing this pointer (the parental node of A_d). Subsequently, the whole m-xBR$^+$-tree will be stored using the GWB, except for its root. Merging of A_d and R_m may lead to overflow, or not. For more details, see [40].

Last, if $h_m > h_d$, we save all the nodes of the m-xBR$^+$-tree using the GWB. 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 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 and the xBR$^+$-tree is updated with its new root.

In Algorithm 4, we depict Phase 4 of *PBLS*. The use of GWB appears in Lines 3, 6, 18 and 28.

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.

Note also that, in this work, contrary to [37,40], we utilized additional buffers in Phases 1 and 2 for reading and writing to exploit the massive I/O advantage of SSDs. Moreover, wherever nodes should be written to disk (leaves, as in Phase 3, or internal nodes, as in Phase 4), we utilized the GWB. This buffer groups the nodes in sequences of consecutive runs and whenever it gets full, it is written to disk. This approach minimizes write calls and maximizes the data segments written for each call and, therefore, exploits the massive writing advantage of SSDs, as will be shown in the Sect. 6.

5 Bulk-insertion algorithm for xBR$^+$-tree

In this section, we present the method we developed for bulk insertions into xBR$^+$-trees in SSDs, called *Process of Loading* xBR$^+$-trees by *Bulk Insertions on SSDs* (*PLBIS*), that has evolved from the *Process of Loading* xBR$^+$-trees *by Bulk Insertions* (*PLBI*) [38].

Algorithm 4 *PBLS* 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 (using GWB to exploit massive SSD writing)
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 $m.root$ (using GWB to exploit massive SSD writing)
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 $m.root$ (using GWB to exploit massive SSD writing)
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 (using GWB to exploit massive SSD writing)
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** $d.root$

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, we use GWB (which, in this case, is a reading buffer) to load the contents of the leaves pointed by the entries of I in as long as possible runs. These leaves are transferred to another buffer, where they will be merged with B_I. That is, for each region entry E of I, we merge B_E with the child node (leaf) corresponding to E.
- After merging of all the children of I has been completed, we examine if any of these children (leaves) has overflown.
- If none of these children has overflown, we write them, using GWB, to the SSD. For each one of these children, 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 delete all the children to recreate them with new contents. To speed up the process, we do not perform deletions in SSD, but we create a main memory linked-list that contains the addresses which have been freed. Next, we create and save to SSD the new leaves, using the GWB, as in Phase 3, Part A of *PBLS*. During this process, we utilize the addresses stored in the linked-list. Then, we create an m-xBR$^+$-tree (a main memory xBR$^+$-tree) m for I (Phase 3, Part B of *PBLS*) and gradually merge m with the xBR$^+$-tree. More details, see [38]. 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.

Contrary to [37], we adopted improvements that have been incorporated in [40], regarding building of the m-xBR$^+$-tree using an auxiliary tree and merging of the subtrees of the m-xBR$^+$-tree with the xBR$^+$-tree (for more details, see [38]).

In Algorithm 5, we depict *PLBIS*. The use of GWB appears in Lines 10, 20 and 25 (as in Phase 4 of *PBLS*). Note that, initially this resursive algorithm is called with the root of the xBR$^+$-tree and S_M.

Note that, in this work, contrary to [38], wherever nodes should be read from/written to disk, we utilize the GWB. This buffer groups the nodes in sequences of consecutive runs during reading from/writing to disk. This approach minimizes read/write calls and maximizes the data segments read/written for each call and, therefore, exploits the massive reading/writing advantage of SSDs, as will be shown in the Sect. 6. Note that, in our algorithms, each sub-sequence of consecutive reading operations from SSD should have completed before the next write operation is met in the sequence of operations. Therefore, reading is asynchronous only between write operations.

Algorithm 5 *PLBIS*

Input: xBR$^+$-tree node I, set of points B_I
Output: xBR$^+$-tree saved in disk
1: Ret = TRUE
2: **if** I is INTERNAL **then**
3: **for each** $E \in I$ **do**
4: **determine** B_E ▷ *the points of B_I that fall inside E*
5: **read** N_E ▷ *the node pointed by E*
6: Ret = **PLBIS**(N_E, B_E) ▷ *recursive call*
7: **if** Ret = FALSE **then**
8: **return** Ret
9: **else** ▷ I is at the lowest level
10: **read** all the leaves pointed by I (using GWB to exploit massive reading from SSD)
11: **add** the contents of GWB to buffer B ▷ *needed for merging point sets at L.14*
12: OverFlown ← FALSE
13: **for each** $E \in I$ **do**
14: **determine** B_E ▷ *the points of B_I that fall inside E*
15: **add** B_E into the leaf pointed by E
16: **if** the leaf pointed by E has overflown **then**
17: **add** points of all leaves pointed by I to B
18: OverFlown ← TRUE
19: **break**
20: **if** OverFlown = FALSE **then**
21: **save** all the leaves pointed by I (using GWB to exploit massive writing to SSD)
22: **update** parent DBR of all entries of I, cascading update to ancestors
23: **else**
24: **delete** all leaves pointed by I
25: **create** an m-xBR$^+$-tree for B ▷ *similarly to Phase 3 of PBLS*
26: **merge** m-xBR$^+$-tree with xBR$^+$-tree ▷ *similarly to Phase 4 of PBLS*
27: **return** FALSE
28: **return** Ret

This means that, before executing the next write operation, GWB, using massive I/O, executes the current sub-sequence of consecutive read operations.

6 Experimental results

We designed and run a large set of experiments to compare *PBL+* and *PLBI* to their SSD variants (*PBLS* and *PLBIS*, respectively). We used real spatial datasets of North America, representing roads (NArdN with 569082 line-segments) and rail-roads (NArrN with 191,558 line-segments). To create sets of 2d points, we have transformed the MBRs of line-segments from NArdN and NArrN into points by taking the center of each MBR (i.e. |NArdN| = 569,082 points, |NArrN| = 191,558 points). Moreover, 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| = 1,138,164 points. The data of these three files were normalized in the range $[0, 1]^2$. We have also created synthetic clustered datasets of 250,000, 500,000 and 1,000,000 points, with 125 clusters in each dataset (uniformly distributed in the range $[0, 1]^2$), where for a set having N points, $N/125$ points were gathered around the center of each cluster, according to Gaussian distribution. We also used three

big real datasets.[1] They represent water resources of North America (Water) consisting of 5,836,360 line-segments and world parks or green areas (Park) consisting of 11,503,925 polygons and world buildings (Build) consisting of 114,736,539 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. All experiments were performed on a Dell Precision T3500 workstation running CentOS Linux 7 with Kernel 4.15.4 as operating system. The workstation is equipped with a quad-core Intel Xeon W3550 CPU, 8GB of main memory, an 1TB 7.2K SATA-3 Seagate HDD (hosting the operating system) and a 512GB SM951A Samsung SSD hosted on PCI-e 2.0 interface (used for experimentation).

Note that, the trees created by *PBL+* and *PBLS* are the same (same contents of internal nodes and leaves). The difference of the two algorithms lies in the way I/O operations are performed. The same holds for the trees created by *PLBI* and *PLBIS*. Therefore, in the following we compare (performance-wise) to each-other the two algorithms (*PBL+* and *PBLS*) for building xBR$^+$-trees by bulk-loading and the two algorithms (*PLBI* and *PLBIS*) for building xBR$^+$-trees by bulk-insertions. Since the methods we created concern the building of trees only and the trees built by the two bulk-loading/bulk-insertion methods are identical, comparing the trees for relative query performance (as in [37,38,40]) would not add any insight to the assessment of the new methods. Note also that, we utilized Direct I/O (O_DIRECT) for accessing data on SSD to prevent any influence of Linux caching system in our results.

We run experiments for tree building, counting creation time, the numbers of logical I/O operations and the number of nodes read from/written to disk. The experiments were run using SSD with GWB (in *PBLS* and *PLBIS*) equal to 64, 256 and 1024 nodes and LRU (in *PBL+* and *PLBI*) equal to 256 nodes.

To study tree building by *PLBI*/*PLBIS*, we split the whole (unsorted) dataset to a sequence of subsets (segments), we use the *PBL+*/*PBLS* algorithm 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*/*PLBIS*. The size of each *Segment* (*S*) is equal to *MemoryLimit* (*ML*), which is a percentage of the cardinality of the corresponding dataset. For each of the 9 above datasets, we constructed xBR$^+$-trees, using *ML* equal to 2% and 4% and *node size* equal to 4 KB, 8 KB and 16 KB.

In Fig. 2 left, we present, using a small real dataset (NArdN), the execution time of *PBL+* on SSD with a 256 node LRU buffer and *PBLS* on SSD with a 64, 256 and 1024 node buffer, for node size equal to 4, 8 and 16 KB. ML was equal to 2%. In Fig. 3 left, we present analogous results to Fig. 2 left, using a small synthetic dataset (1000KCN). ML remained equal to 2%. In Fig. 4 left, we present analogous results to Fig. 2 left, using a large real dataset (BuildN). ML was equal to 4% (for demonstration purposes, we depicted results for two different ML percentages; the performance of both percentages was close). The last three bars show how faster is the new bulk-loading algorithm in comparison to the old one (first bar) on SSDs (for large files, the new algorithm is up to 53 times faster than the old one). Note that in the last three bars we used different sizes of GWB (64, 256 and 1024 nodes). When we have small

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

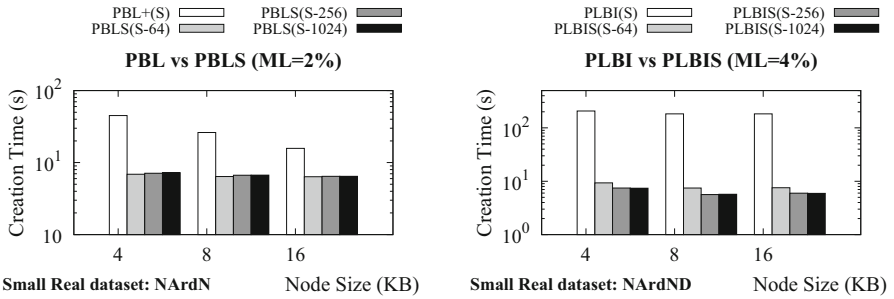

Fig. 2 *PBL+* versus *PBLS* (left column), *PLBI* versus *PLBIS* (right column)

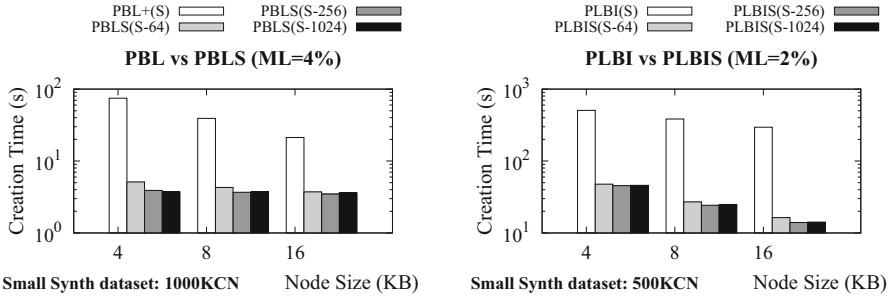

Fig. 3 *PBL+* versus *PBLS* (left column), *PLBI* versus *PLBIS* (right column)

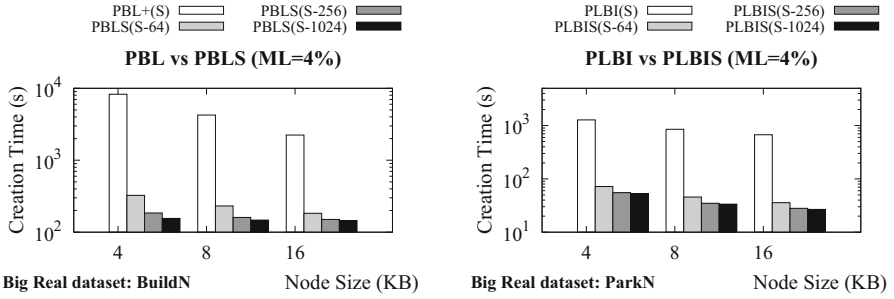

Fig. 4 *PBL+* versus *PBLS* (left column), *PLBI* versus *PLBIS* (right column)

datasets (given that ML is a percentage of the size of the dataset), it is possible that the number of node write calls (node read calls are very few) is the same for all buffer sizes (since their sizes are larger than required). In this case, the performance of the two alternatives is almost the same (with the smaller buffer size requiring smaller management cost). When we have large datasets, the number of node write calls is smaller for the larger buffer and the performance of the related alternative is higher.

As noted in the last paragraph of Sect. 4, the significantly higher performance of *PBLS* shown in the previous experiments is due to the use the additional buffers in Phases 1 and 2 for reading and writing and GWB in Phases 3 and 4 for writing to

disk. The use of these buffers exploits the massive I/O advantage of SSDs and boost performance significantly.

In the right part of Fig. 2, we present, using NArdND (a small real dataset), the execution time of *PLBI* on SSD a 256 node LRU buffer, and *PLBIS* on SSD with a 64, 256 and 1024 node buffer, for node size equal to 4, 8 and 16 KB. ML was equal to 2%. Analogously to Fig. 2 right, the right chart of Fig. 3 shows the results of the creation time for small synthetic dataset (500KCN). ML remained equal to 2%. In Fig. 4 right, we present analogous results to Fig. 2 right, using a large real dataset (ParkN). ML was equal to 4%. (again, for demonstration purposes, we depicted results for two different ML percentages; a percentage of 4%, in general, leads to better performance). The last three bars show how faster is the new bulk-insertion algorithm in comparison to the old one (first bar) on SSDs (for large files, the new algorithm is up to 14 times faster than the old one). Note that in the last three bars we used different sizes of GWB (64, 256 and 1024 nodes). The alternative of 1024 nodes is faster. *PLBIS* not only has node write, but also node read calls and buffers of all sizes get fully used. Thus, the larger buffer has a stronger effect.

As noted in the last paragraph of Sect. 5, the significantly higher performance of *PLBIS* shown in the previous experiments is due to the use of the GWB for reading nodes from and for writing nodes to disk, since nodes are grouped in sequences of consecutive runs and the massive I/O advantage of SSDs is exploited.

In Table 1, for all datasets, we depict the percentage of disk access gained by *PBLS* (logical accesses issued minus actual accesses performed over logical accesses issued) in internal and leaf nodes, for all three disk page sizes, when using GWB with size equal to 256 and 1024 nodes. ML was equal to 2% (the results for 4% are similar). Note that the gain in internal nodes for small datasets is limited, while it is equally important to the gain in leaves for larger datasets.

In Table 2, for all datasets, we depict analogous results to Table 1, regarding *PLBIS*. ML was equal to 4% (the results for 2% are similar). Note that the gain in internal nodes for small and larger datasets is limited, while it is very important in leaves for all datasets.

Both tables show how important was the use of GWB for saving disk accesses and accelerating our algorithms.

In Table 3, we comparatively study the scalability of our algorithms. The data sets are sorted by cardinality (the size of each dataset is depicted in million of points). The size of LRU for the old algorithms and the size of GWB for the new ones is equal to 256 nodes, while the node size is 4 KB in all cases. For each data set (size) the execution times of each couple of algorithms (*PBL+* vs. *PBLS* and *PLBI* vs. *PLBIS*) and the time gain of the latter algorithm over the former are shown. Note that, the gain of *PBLS* over *PBL+* gets higher as the size of the dataset increases and exceeds 65%/97% for the smallest/largest dataset, while the gain of *PLBIS* over *PLBI* is very large (larger than 92%) for all datasets and varies slightly depending on the data set distribution.

In summary, trees created by new algorithms (*PBLS* and *PLBIS*) are built significantly faster than the trees created by the old algorithms (*PBL+* and *PLBI*). Moreover, in most cases, creation time was faster for $ML = 4\%$ than for $ML = 2\%$, for $GWB = 1024\, nodes$ than for $GWB = 256\, or\, 64\, nodes$, and for larger than for smaller

Table 1 Algorithm *PBLS*: gain of accesses (logical-actual)/logical, size of ML = 2%

Data set name	Type of nodes	Size of GWB 256 nodes size of nodes			Size of GWB 1024 nodes size of nodes		
		4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
NArrN	Internal	0.00	0.00	0.00	0.00	0.00	0.00
	Leaf	95.96	91.78	84.15	95.96	91.78	84.15
NArdN	Internal	21.64	0.00	0.00	21.64	0.00	0.00
	Leaf	98.26	96.65	93.19	98.26	96.65	93.19
NArdND	Internal	52.82	2.25	0.00	52.82	2.25	0.00
	Leaf	99.07	98.26	96.64	99.14	98.26	96.64
250KCN	Internal	0.00	0.00	0.00	0.00	0.00	0.00
	Leaf	96.82	93.56	87.14	96.82	93.56	87.14
500KCN	Internal	23.75	0.00	0.00	23.75	0.00	0.00
	Leaf	98.41	96.82	93.58	98.41	96.82	93.58
1000KCN	Internal	53.80	0.00	0.00	53.80	0.00	0.00
	Leaf	99.17	98.41	96.82	99.20	98.41	96.82
WaterN	Internal	90.91	70.02	14.68	90.91	70.02	14.68
	Leaf	99.54	99.48	99.32	99.83	99.74	99.48
ParksN	Internal	93.69	82.09	41.78	93.69	82.09	41.78
	Leaf	99.57	99.52	99.44	99.86	99.81	99.69
BuildN	Internal	99.21	97.19	92.01	99.39	97.19	92.01
	Leaf	99.61	99.60	99.59	99.90	99.89	99.89

node sizes. The gain of *PBLS* over *PBL+* depends on the dataset size, while the gain of *PLBIS* over *PBLS* is irrelevant to data set size and slightly depends on dataset distribution. Although, the old algorithms use buffering (LRU buffer equal to 256 nodes), buffering used in the new algorithms (additional buffers in Phases 1 and 2 of *PBLS* for reading and writing and GWB for writing in *PBLS* and for reading and writing in *PLBIS*) significantly improve performance.

7 Conclusions and future work

SSDs are already very popular in commodity and enterprise systems. However, taking the highest advantage of their increased performance requires indexes especially designed for them. In this paper, for the first time in the literature, we present algorithms for bulk-loading (*PBLS*) and bulk-insertion (*PLBIS*) for an index structure (xBR$^+$-trees) in SSDs. To the best of our knowledge, these are the first algorithms of their kind that were specially designed for SSDs. Through extensive experimentation, the new algorithms are shown to significantly outperform their predecessors that were designed for HDDs (*PBLS* outperforms *PBL+* by at least 65.5% and *PLBIS* outperformes *PLBI* by at least 92.4%).

Table 2 Algorithm *PLBIS*: gain of accesses (logical-actual)/logical, size of ML = 4%

Data set name	Type of nodes	Size of GWB 256 nodes size of nodes			Size of GWB 1024 nodes size of nodes		
		4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
NArrN	Internal	1.64	0.00	0.00	1.64	0.00	0.00
	Leaf	93.68	95.60	97.84	93.68	95.60	98.02
NArdN	Internal	3.07	2.27	0.00	3.07	2.27	0.00
	Leaf	96.77	97.12	97.60	96.82	97.16	97.78
NArdND	Internal	3.55	2.27	0.00	3.55	2.27	0.00
	Leaf	97.35	97.66	97.77	97.43	97.70	97.99
250KCN	Internal	2.70	0.00	0.00	2.70	0.00	0.00
	Leaf	87.05	92.81	96.55	87.05	92.81	96.68
500KCN	Internal	2.70	0.00	0.00	2.70	0.00	0.00
	Leaf	87.18	92.61	96.26	87.18	92.62	96.43
1000KCN	Internal	2.95	2.74	0.00	2.95	2.74	0.00
	Leaf	87.17	92.82	95.98	87.18	92.83	96.17
WaterN	Internal	3.62	3.17	1.98	3.62	3.17	1.98
	Leaf	98.54	98.84	98.95	98.75	99.07	99.22
ParksN	Internal	8.77	3.52	3.05	8.77	3.52	3.05
	Leaf	98.55	98.70	98.67	98.76	98.90	98.93
BuildN	Internal	35.30	8.88	3.51	35.31	8.89	3.51
	Leaf	99.34	99.34	99.33	99.61	99.61	99.62

Table 3 *PBL+* versus *PBLS* (ML = 2%) and *PLBI* versus *PLBIS* (ML = 4%): execution time and time gain (old-new)/old; node size = 4 KB, LRU and GWB = 256 nodes

Data set		Exec time (s)		Gain (%)	Exec time (s)		Gain (%)
Name	Size	*PBL+*	*PBLS*		*PLBI*	*PLBIS*	
NarrN	0.192	19.444	6.705	65.52	80.60	3.823	95.26
NardN	0.569	45.022	7.104	84.22	118.3	4.384	96.29
NardND	1.138	87.205	9.244	89.40	207.3	7.476	96.39
250KCN	0.250	22.282	5.342	76.03	232.3	17.59	92.43
500KCN	0.500	40.769	6.178	84.85	391.7	29.69	92.42
1000KCN	1.000	78.092	7.141	90.86	713.3	54.56	92.35
WaterN	5.836	427.05	19.56	95.42	664.3	25.19	96.21
ParkN	11.50	838.46	28.61	96.59	1,279	55.04	95.70
BuildN	114.7	8270.7	194.5	97.65	9,595	597.1	93.78

Future work plans include:

– The development of specialized algorithms for processing several types of spatial queries using xBR$^+$-trees on SSDs.

- The development of specialized algorithms for processing bulk spatial queries using xBR$^+$-trees on SSDs.
- The development of specialized bulk-loading and bulk-insertion algorithms for other index structures on SSDs and a comparison to the ones developed for the xBR$^+$-tree.
- To compare spatial index structures on SSDs regarding query-processing performance.
- To study of alternative caching techniques and compare to GWB for building xBR$^+$-trees.

Acknowledgements Work of M. Vassilakopoulos, A. Corral and Y. Manolopoulos funded by the MINECO research projects [TIN2013-41576-R] and [TIN2017-83964-R].

References

1. Achakeev D, Seeger B, Widmayer P (2012) Sort-based query-adaptive loading of R-trees. In: CIKM conference, pp 2080–2084
2. Achakeev D, Seidemann M, Schmidt M, Seeger B (2012) Sort-based parallel loading of R-trees. In: BigSpatial workshop, pp 62–70
3. An N, Kanth KVR, Ravada S (2003) Improving performance with bulk-inserts in Oracle R-trees. In: VLDB conference, pp 948–951
4. Arge L, Hinrichs KH, Vahrenhold J, Vitter JS (1999) Efficient bulk operations on dynamic R-trees. In: ALENEX workshop, pp 328–348
5. Arge L, Hinrichs KH, Vahrenhold J, Vitter JS (2002) Efficient bulk operations on dynamic R-trees. Algorithmica 33(1):104–128
6. Berchtold S, Böhm C, Kriegel H (1998) Improving the query performance of high-dimensional index structures by bulk-load operations. In: EDBT conference, pp 216–230
7. Carniel AC, Ciferri RR, de Aguiar Ciferri CD (2017) A generic and efficient framework for spatial indexing on flash-based solid state drives. In: ADBIS conference, pp 229–243
8. Chen L, Choubey R, Rundensteiner EA (1998) Bulk-insertions info R-trees using the small-tree-large-tree approach. In: ACM-GIS conference, pp 161–162
9. Chen L, Choubey R, Rundensteiner EA (2002) Merging R-trees: efficient strategies for local bulk insertion. GeoInformatica 6(1):7–34
10. Cho S, Chang S, Jo I (2015) The solid-state drive technology, today and tomorrow. In: ICDE conference, pp 1520–1522
11. Choubey R, Chen L, Rundensteiner EA (1999) GBI: A generalized R-tree bulk-insertion strategy. In: SSD conference, pp 91–108
12. Ciaccia P, Patella M (1998) Bulk loading the M-tree. In: ADC conference, pp 15–26
13. Cornwell M (2012) Anatomy of a solid-state drive. Commun ACM 55(12):59–63
14. den Bercken JV, Seeger B (2001) An evaluation of generic bulk loading techniques. In: VLDB conference, pp 461–470
15. den Bercken JV, Seeger B, Widmayer P (1997) A generic approach to bulk loading multidimensional index structures. In: VLDB conference, pp 406–415
16. Emrich T, Graf F, Kriegel H, Schubert M, Thoma M (2010) On the impact of flash SSDs on spatial indexing. In: DaMoN conference, pp 3–8
17. Fevgas A, Bozanis P (2015) Grid-file: towards to a flash efficient multi-dimensional index. In: DEXA conference, pp 285–294
18. Ghanem TM, Shah R, Mokbel MF, Aref WG, Vitter JS (2004) Bulk operations for space-partitioning trees. In: ICDE conference, pp 29–40
19. Hady FT, Foong AP, Veal B, Williams D (2017) Platform storage performance with 3d XPoint technology. Proc IEEE 105(9):1822–1833
20. Hjaltason GR, Samet H (1999) Improved bulk-loading algorithms for quadtrees. In: ACM-GIS conference, pp 110–115

21. Hjaltason GR, Samet H (2002) Speeding up construction of PMR quadtree-based spatial indexes. VLDB J 11(2):109–137
22. Hjaltason GR, Samet H, Sussmann YJ (1997) Speeding up bulk-loading of quadtrees. In: ACM-GIS conference, pp 50–53
23. Jin P, Ou Y, Harder T, Li Z (2012) AD-lRU: an efficient buffer replacement algorithm for flash-based databases. Data Knowl Eng 72:83–102
24. Jin P, Xie X, Wang N, Yue L (2015) Optimizing R-tree for flash memory. Expert Syst Appl 42(10):4676–4686
25. Kamel I, Faloutsos C (1993) On packing R-trees. In: CIKM conference, pp 490–499
26. Kamel I, Khalil M, Kouramajian V (1996) Bulk insertion in dynamic R-trees. In: SDH conference, pp 3B.31–3B.42
27. Koltsidas I, Viglas SD (2011) Spatial data management over flash memory. In: SSTD conference, pp 449–453
28. Lee T, Moon B, Lee S (2006) Bulk insertion for R-trees by seeded clustering. Data Knowl Eng 59(1):86–106
29. Leutenegger ST, Edgington JM, Lopez MA (1997) STR: a simple and efficient algorithm for R-tree packing. In: ICDE conference, pp 497–506
30. Li G, Zhao P, Yuan L, Gao S (2013) Efficient implementation of a multi-dimensional index structure over flash memory storage systems. J Supercomput 64(3):1055–1074
31. Lv Y, Li J, Cui B, Chen X (2011) Log-compact R-tree: an efficient spatial index for SSD. In: DASFAA workshops, pp 202–213
32. Papadopoulos A, Manolopoulos Y (2003) Parallel bulk-loading of spatial data. Parallel Comput 29(10):1419–1444
33. Park S, Jung D, Kang J, Kim J, Lee J (2006) CFLRU: a replacement algorithm for flash memory. In: Proceedings of the 2006 international conference on compilers, architecture and synthesis for embedded systems. ACM, pp 234–241
34. Pawlik M, Macyna W (2012) Implementation of the aggregated R-tree over flash memory. In: DASFAA workshops, pp 65–72
35. Roh H, Park S, Kim S, Shin M, Lee S (2011) B^+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. PVLDB 5(4):286–297
36. Roumelis G, Vassilakopoulos M, Corral A (2011) Performance comparison of xBR-trees and R*-trees for single dataset spatial queries. In: ADBIS conference, pp 228–242
37. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2016) Bulk-loading xBR^+-trees. In: MEDI conference, pp 57–71
38. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2017) Bulk insertions into xBR^+-trees. In: MEDI conference, pp 185–199
39. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2017) Efficient query processing on large spatial databases: a performance study. J Syst Softw 132:165–185
40. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2018) An efficient algorithm for bulk-loading xBR^+-trees. Comput Stand Interfaces 57:83–100
41. Roumelis G, Vassilakopoulos M, Loukopoulos T, Corral A, Manolopoulos Y (2015) The xBR^+-tree: an efficient access method for points. In: DEXA conference, pp 43–58
42. Roussopoulos N, Kotidis Y, Roussopoulos M (1997) Cubetree: Organization of and bulk updates on the data cube. In: SIGMOD conference, pp 89–99
43. Roussopoulos N, Leifker D (1985) Direct spatial search on pictorial databases using packed R-trees. In: SIGMOD conference, pp 17–31
44. Sarwat M, Mokbel MF, Zhou X, Nath S (2011) FAST: a generic framework for flash-aware spatial trees. In: SSTD conference, pp 149–167
45. Shekhar S, Chawla S (2003) Spatial databases—a tour. Prentice Hall, Upper Saddle River
46. Vassilakopoulos M, Manolopoulos Y (2000) External balanced regular (x-BR) trees: new structures for very large spatial databases. In: Fotiadis DI, Nikolopoulos SD (eds) Advances in informatics: selected papers of the 7th hellenic conference on informatics (HCI '99). World Scientific, Singapore, pp 324–333. https://doi.org/10.1142/9789812793928_0029
47. Vespa TG, Traina C Jr, Traina AJM (2010) Efficient bulk-loading ondynamic metric access methods. Inf Syst 35(5):557–569
48. Wu C, Chang L, Kuo T (2003) An efficient R-tree implementation over flash-memory storage systems. In: ACM-GIS conference, pp 17–24

49. Yang C, Jin P, Yue L, Yang P (2016) Efficient buffer management for tree indexes on solid state drives. Int J Parallel Program 44(1):5–25

Affiliations

George Roumelis[1] · Athanasios Fevgas[1] · Michael Vassilakopoulos[1] · Antonio Corral[2] · Panayiotis Bozanis[1] · Yannis Manolopoulos[3]

George Roumelis
groumelis@uth.gr

Athanasios Fevgas
fevgas@uth.gr

Antonio Corral
acorral@ual.es

Panayiotis Bozanis
pbozanis@uth.gr

Yannis Manolopoulos
yannis.manolopoulos@ouc.ac.cy

1 Data Structuring and Engineering Laboratory, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece

2 Department of Informatics, University of Almeria, Almería, Spain

3 Faculty of Pure and Applied Sciences, Open University of Cyprus, Nicosia, Cyprus