# Spatial Batch-Queries Processing Using xBR$^+$-trees in Solid-State Drives

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

[1] Data Structuring & Engineering Lab., Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
{groumelis,mvasilako,fevgas}@uth.gr
[2] Department of Informatics, University of Almeria, Almeria, Spain
acorral@ual.es
[3] Faculty of Pure and Applied Sciences, Open University of Cyprus, Nicosia, Cyprus
yannis.manolopoulos@ouc.ac.cy

**Abstract.** Efficient query processing in spatial databases is of vital importance for numerous modern applications. In most cases, such processing is accomplished by taking advantage of spatial indexes. The xBR$^+$-tree is an index for point data which has been shown to outperform indexes belonging to the R-tree family. On the other hand, Solid-State Drives (SSDs) are secondary storage devices that exhibit higher (especially read) performance than Hard Disk Drives and nowadays are being used in database systems. Regarding query processing, the higher performance of SSDs is maximized when large sequences of queries (batch queries) are executed by exploiting the massive I/O advantages of SSDs. In this paper, we present algorithms for processing common spatial (point-location, window and distance-range) batch queries using xBR$^+$-trees in SSDs. Moreover, utilizing small and large datasets, we experimentally study the performance of these new algorithms against processing of batch queries by repeatedly applying existing algorithms for these queries. Our experiments show that, even when the existing algorithms take advantage of LRU buffering that minimizes disk accesses, the new algorithms prevail performance-wise.

**Keywords:** Spatial indexes · xBR$^+$-trees · Query processing
Solid-State Drives

## 1 Introduction

Nowadays, the volume of available spatial data (e.g. location, routing, navigation data, etc.) is continuously increasing world-wide. To exploit these data, efficient processing of spatial queries is of great importance due to the wide area of applications that may address such queries. The most common spatial queries where points are involved are point-location, window, distance-range and $K$ nearest-neighbor queries (PLQs, WQs, DRQs and $K$NNQs, respectively, in the sequel).


E. H. Abdelwahed et al. (Eds.): MEDI 2018, LNCS 11163, pp. 301–317, 2018.
https://doi.org/10.1007/978-3-030-00856-7_20

At a higher level, such queries have been used as the basis of many complex operations in advanced applications, for example, geographical information systems (GIS), location-based systems (LBS), geometric databases, CAD, etc.

The use of efficient spatial indices is very important for performing spatial queries and retrieving efficiently spatial objects from datasets according to specific spatial constraints [5]. Hierarchical indices are useful due to their ability to focus on the interesting subsets of data. This focusing results in an efficient representation and execution times on query processing and thus, it is particularly useful for performing spatial operations. An example of such indices is the Quadtree [20], which is based on the principle of recursive decomposition of space and has become an important access method for spatial applications [21].

The External Balanced Regular (xBR)-tree [23] is a secondary memory structure that belongs to the Quadtree family (widely used in GIS applications, which is suitable for storing and indexing points and, in extended versions, line segments, or other spatial objects). We use an improved version of xBR-tree, called xBR$^+$-tree [19], which is also a disk-resident structure. The xBR$^+$-tree improves the xBR-tree in the node structure and in the splitting process. The node structure of the xBR$^+$-tree stores information which makes query processing more efficient. In addition, the xBR$^+$-tree outperforms R*-tree and R$^+$-tree (in terms of I/O activity and execution time) for the most common spatial queries, like PLQs, WQs, DRQs, $K$NNQs, etc. [18].

The advent of non-volatile memories (NVM) has enabled a brand-new class of storage devices with exciting features that will prevail in the storage market in the near future. Their high read and write speeds, small size, low power consumption and shock resistance are some of the reasons that made them storage medium of choice. NAND flash is undoubtedly the most popular NVM today. Storage devices based on NAND Flash are found both in consumer devices and enterprise data-centers. However, upcoming technologies, such as 3D XPoint from Intel and Micron, make possible even more efficient devices [6]. At the very beginning, raw Flash memory chips were embedded in mobile devices and other electronics. However, soon enough, the increasing needs for efficient storage drove the emergence of Solid-State Drives (SSDs). SSDs are composed by Flash chips, embedded controllers and DRAM [3]. Contemporary devices incorporate from a few to many NAND chips, supplying capacities even of tens of terabytes in high-end systems. Flash controller, usually a 32-bit embedded CPU, executes the firmware that controls SSD operation, while DRAM is utilized to store metadata, information regarding address mapping and for user data caching. Firmware is fundamental for SSD operation [2]. Its main responsibility is to map virtual addresses, as they are seen by the host, to physical addresses in flash chips. For this reason is also known as Flash Translation Layer (FTL). FTL performs tasks for garbage collection, wear leveling and management of bad blocks. SSDs exhibit higher write and especially read performance than Hard Disk Drives. This performance advantage is maximized when issuing commands that massively write to/read from SSDs large sequences of consecutive pages (due to exploiting the internal parallelism of SSDs), instead of issuing commands that

write to/read from SSDs the pages of such sequences in small subsequences, or even, one-by-one [12].

In this paper, we present new algorithms for processing large sequences of common spatial queries (PLQs, WQs, DRQs) using xBR$^+$-trees in SSDs. These algorithms are especially designed to massively read from SSDs large sequences of pages needed for answering such queries. Such large sequences of queries (batch queries) appear frequently in applications. Moreover, using small and large datasets, we experimentally study the performance of these new algorithms against processing of batch queries by repeatedly applying existing algorithms for these queries. In addition, several experiments have been executed, showing that the new algorithms are performance winners.

The sequel is organized as follows. In Sect. 2 we review related work on spatial query processing over xBR-trees, as well as, on indices 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. Section 4 presents new algorithms for batch queries processing using xBR$^+$-tree in SSDs. The results of our experiments are discussed in Sect. 5. Finally, Sect. 6 provides the conclusions arising from our work and discusses future work directions.

## 2 Related Work and Motivation

In this section, we first briefly review the xBR-tree family. Then, the most representative spatial indexes, taking advantage of SSD performance, are revised. Finally, the main motivation of this work is exposed.

### 2.1 The xBR-tree Family

The xBR-tree was initially proposed in [23] as a secondary-memory pointer-based structure that belongs to the Quadtree family. The original xBR-tree was enhanced in [15]. The xBR$^+$-tree [18,19] is a further improved extension of the xBR-tree regarding performance of tree creation and spatial query processing. Bulk-loading and bulk-insertion methods for xBR$^+$-trees are presented in [16] and [17], respectively.

In [18], an exhaustive performance comparison (I/O activity and execution time) of xBR$^+$-trees (non-overlapping trees of the Quadtree family), R$^+$-trees (non-overlapping trees of the R-tree family) and R*-trees (industry standard belonging to the R-tree family) is performed. In this comparative study, several performance aspects are studied, like tree building and processing single point dataset queries (PLQs, WQs, DRQs and $K$NNQs) and distance-based join queries (DJQs), using medium and large spatial (real and synthetic) datasets. As a conclusion, the xBR$^+$-tree is a clear winner for tree building and query performance. The excellent building performance of the xBR$^+$-tree is due to the regular subdivision of space that leads to much fewer and simpler calculations. The higher query performance of the xBR$^+$-tree is due to the combination of the regular subdivision of space, the additional representation of the minimum

rectangles bounding the actual data objects, the algorithmic improvements of certain spatial queries and the storage order of the entries of internal nodes.

### 2.2 Spatial Indexes for Flash SSDs

NAND Flash provides superior performance compared to traditional magnetic disks but has some intrinsic characteristics. It exhibits asymmetry in the read, write, and erasure speeds and a page must be erased first before being re-programmed. Erase operations take place at block level, while reads and writes are performed at page level. SSDs inherit some of these characteristics, thus in most devices read operations are faster than writes, while difference exist among the speeds of sequential and random I/Os as well. Especially, random writes may initiate garbage collection which is impacts the efficiency of the device. On the other hand, the high degree of internal parallelism of latest SSDs substantially contributes to the improvement of I/O performance [13]. Many research efforts have been made for Flash efficient database indexes. The works for spatial data processing mostly concern the R-tree.

The RFTL [24] is the first effort towards a flash efficient implementation of the R-tree. It is based on recording deltas for update operations. An in-memory buffer is utilized to hold the deltas before be persisted in batches. The same method has also been applied for the Aggregated R-tree in [11].

The LCR-tree [10] exploits a small section of SSD to log update operations. In contrary to other works it accumulates all the deltas for a particular node to one page in Flash. This way it ensures only one additional page reading to reach a tree node. The LCR-tree exhibits better performance than the original R-tree and the RFTL in mixed search/insert experimental scenarios. In the FOR-tree [7] authors aim to reduce small random writes by introducing overflow nodes to the R-tree. They propose new search and insert algorithms and a buffering algorithm for efficient caching of original and overflow nodes.

Regarding to non R-tree spatial indexes, the F-KDB [8] is a log-structured implementation of the K-D-B-tree for Flash, the MicroGF [9] is a 2D Grid File like structure for raw Flash, that is embedded in wireless sensor nodes, while a first effort towards to an efficient Grid-File for SSDs is presented in [4].

Furthermore two generic frameworks for spatial indexing have been proposed so far, which can encapsulate different data structures. FAST [22] utilizes the original insertion and update algorithms, buffers updates in RAM and flashes them to the SSD at once. eFIND [1] is a newer generic framework that provides better performance than FAST.

MPSearch [13,14] is a multi-path search algorithm for the $B^+$-tree that performs batch searches considering the characteristics of SSDs to accelerate performance. To the best of our knowledge, there are not any works concerning spatial batch-queries processing for Flash SSDs. Motivated by this observation, in this paper, we develop new algorithms for processing common spatial batch queries (PLQs, WQs, DRQs), using $xBR^+$-trees in SSDs. These algorithms are designed for maximizing performance by exploiting the internal parallelism of SSDs.

## 3 The xBR$^+$-tree Structure

In this section, for the sake of self-containment of the paper, we present the basics of the xBR$^+$-tree. The xBR$^+$-tree [19] is a hierarchical, disk-resident Quadtree-based index 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 tree nodes are disk pages of two kinds: *leaves*, which store the actual multidimensional data 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, having a region related to a subquadrant of the original space. *Shape* is a flag that determines if this region 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. *qside* is the side length of the subquadrant of the original space that corresponds to this child-node. *DBR* (Data Bounding Rectangle) stores the coordinates of the rectangular subregion of this child-node region that contains point data (at least two points must reside on the sides of the *DBR*), while *Pointer* points to this child-node.

The subquadrant of the original space related to a 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*. Here, 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 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, i.e. the ones corresponding to the next entries of the internal node. The entries in an internal node are saved in sequential groups, consisting of subgroups. The first entry of each group is the parental entry of the rest entries of this group. Each entry of a group is a descendant of the entry on its left, or it is the parent of a new (sub)group. For example, in Fig. 1 an internal node (a root) that points to 5 internal nodes that point to 15 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 1. The region of the rightmost entry (220*) is the NW subquadrant of the SW subquadrant of the SW quadrant of the original space (the * symbol is used to denote the end of a variable size *Address*). The flag *shape* is set at the value 'S' 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 region of the next (on the left) subquadrant is the SW subquadrant of the SW quadrant of the whole space. For this subbquadrant, the *Address* is 22* (non-complete square, denoted by 'noS'). The next two (on the left) entries cover the whole space of the NE quadrant (1*) and the NW quadrant (0*) of the whole space, respectively. Finally, the first entry in the root of this example expresses the whole space minus the four descendant regions (0*, 1*, 22* and 220*), 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.

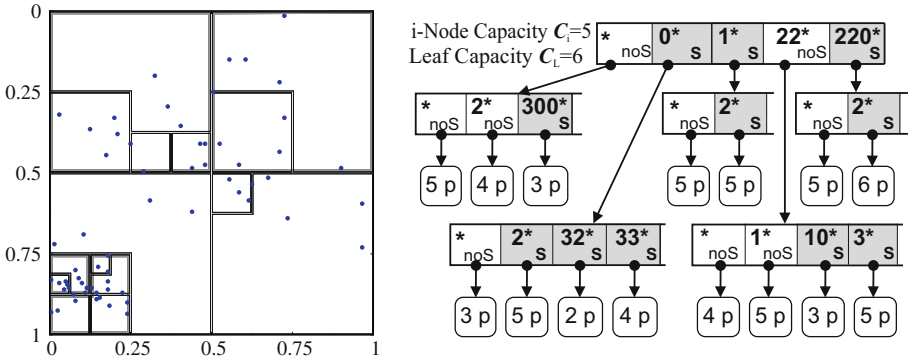


**Fig. 1.** A collection of 64 points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree.

*External* nodes (leaves) of the xBR$^+$-tree simply contain the data elements and have a predetermined capacity $C_L$. When $C_L$ is exceeded, due to an insertion in a leaf, the region of this leaf is partitioned in two subregions.

An example that demonstrates split of a leaf and an internal node follows. In the left upper part of the Fig. 2, an xBR$^+$-tree having one internal (root) node with 5 entries (its cardinality equals the maximum capacity of internal nodes, $C_i = 5$) is depicted. The 5 entries point to 5 leaves containing the first 25 points of the total number of 64 points of the dataset of Fig. 1. The next ($26^{th}$) point $p$ must be inserted in a leaf that already contains 6 points and is pointed by the first entry of the root (*) . Since $C_L = 6$, this leaf overflows and is split in two (itself and a new leaf). The new leaf covers the region of the subquadrant 2* and holds 3 points (left lower part of Fig. 2). The other 4 points remain in the existing leaf (*). An entry for the new leaf (2*) must be inserted in the root node which is already full. The root overflows and is split in two internal nodes (itself and a new node). In order to maintain the cohesion of the tree, a new root node having 2 entries is created. The first entry (*) points to the old root node and the second entry points to the new node (0*). The resulting xBR$^+$-tree, consisting of 3 internal nodes with 6 entries pointing to 6 leafs, is depicted in the right part of Fig. 2. The final tree, after inserting the rest of the 64 points

and the space partitioning of the xBR$^+$-tree are shown in Fig. 1. Details on the algorithms for splitting leaf and internal nodes appear in [19].

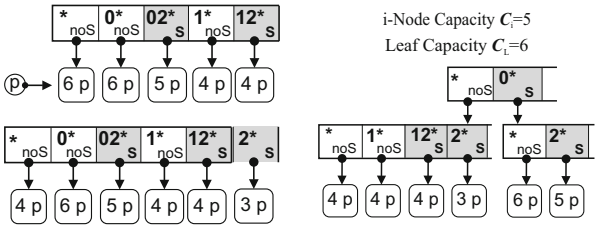


**Fig. 2.** Up-left: an xBR$^+$-tree root pointing to 5 leaves. Down-left: an overflown xBR$^+$-tree root pointing to 6 leaves. Right: the resulting xBR$^+$-tree after splitting of the root.

## 4 Algorithms for Batch-Queries Processing

In the following, we present algorithms for processing the batch versions of three common single-dataset queries, using xBR$^+$-trees in SSDs. These algorithms are designed for maximizing performance when applied on SSDs. They make use of a main memory area (denoted by $M$ in the following), group read accesses needed by several queries of the batch, reorder the pages to be read and, at the same time, avoid unnecessary re-reading of the same pages and issue massive read operations of large sequences of consecutive pages (exploiting the internal parallelism of SSDs).

### 4.1 Algorithm for Processing of Batch Point-Location Queries

In this subsection, we present our new processing method for Batch Point-Location Queries ($BPLQ$) using xBR$^+$-trees in SSDs. The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset $P$ of points and a set of query points $Q$, the **BPLQ** returns the largest subset $R \subseteq Q$ such that $R = \{p : p \in Q \wedge p \in P\}$. The basic idea is as follows. We use the main memory area $M$ (the size of $M$ is defined by the system administrator and its size, a few MBs, is not significant in comparison to the size of the datasets) and we divide $Q$ in subsets such that of each subset can be processed within $M$. Hierarchically, we visit the tree nodes and partition the query points in groups such that each group uniquely falls within one subregion of the current node and massively read the nodes corresponding to the resulting subregions into $M$. This process continues down to the leaf level, where we read the leaves corresponding to the resulting subregions into $M$. For each leaf, we determine all the query points of $Q$ that exist in this leaf. The algorithm is described in more details as follows.

– Considering the maximum memory size of $M$ available to our program, we calculate the maximum cardinality of each subset of $Q$ that can be processed within $M$. We divide $Q$ in subsets that do not exceed this maximum cardinality.
– For each of these subsets, we begin at the root.
  • For a subset of query points, we call a procedure that visits a tree node and partitions this subset in groups such that each group uniquely falls within one subregion of this node.
  • If this node points to internal nodes, we calculate and allocate the memory (part of $M$) that is required for reading the node entries that contain query points and massively read the nodes pointed by these entries.
    ∗ For each of the nodes read and the group of points that fall within the region of this node, we recursively apply this procedure.
  • If a node read points to leaf nodes, we calculate and allocate the memory (part of $M$) that required for reading the leaves that contain query points and massively read the leaves pointed by the node entries.
    ∗ For each of the leaves read and the group of points that fall within this leaf, we sort this group of points according to the axis along which the leaf points have been sorted and determine the query points that exist in the leaf (using a plane-sweep based technique to minimize comparisons).

### 4.2 Algorithm for Processing of Batch Window Queries

Here, we present our processing method for Batch Window Queries ($BWQ$) using xBR$^+$-trees in SSDs. The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset $P$ and a set of rectangular query windows $W$, the **BWQ** returns the largest set $R$ that contains pairs of objects $(p, w)$ such that $R = \{(p, w) : p \in P \wedge p \text{ falls inside } w \in W\}$. The basic idea (an extension of the method in Subsect. 4.1) is as follows. We use a main memory area $M$ and we divide $W$ in subsets such that processing of each subset can be done within $M$. Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the query windows corresponding to each entry, since each region intersected with a query window may be a candidate for containing points of the pairs of the result ($R$). For the entries of the current node that contain a non-empty list of query windows, we massively read the nodes corresponding to these entries into $M$. This process continues down to the leaf level, where we read the leaves corresponding to these entries into $M$. For each leaf, we determine all the points of $P$ that exist into the leaf and fall inside the regions of the query windows of the list. The algorithm is described in more details as follows.

– Considering the maximum memory size of $M$ available to our program, we calculate the maximum cardinality of each subset of $W$ that can be processed within $M$. We divide $W$ in subsets that do not exceed this maximum cardinality.

– For each of these subsets of query windows, we begin at the root.
  - For a subset of query windows, we call a procedure that visits a tree node and in each entry of the node we append a list of query windows whose region is intersected with the region of the entry (in Subsect. 4.1, we didn't need such lists, since a query point falls in at most one region).
  - If this node points to internal nodes, we calculate and allocate the memory (part of $M$) that is required for reading the node entries that contain non-empty lists of query windows and massively read the nodes pointed by these entries.
    * For each of the nodes read and the list of query windows that has intersection with the region of this node, we recursively apply this procedure.
  - If a node read points to leaf nodes, we calculate and allocate the memory (part of $M$) that is required for reading the leaves that contain non-empty lists of query windows and massively read the leaves pointed by the node entries.
    * For each of the leaves read and the list of query windows that have intersection with the region of this leaf we apply the refinement step as follows. We sort this list of windows using as key the maximum coordinate of the axis along which the leaf points have been sorted and determine the leaf points that fall inside the regions of the query windows (using a plane-sweep based technique, to minimize comparisons).

### 4.3 Algorithm for Processing of Batch Distance-Range Queries

In this subsection, we present our processing method for Batch Distance-Range Queries ($BDRQ$) using xBR$^+$-trees in SSDs. The definition of this query is as follows: Given an index $\mathcal{I}_P$ of a dataset $P$ and a set of query pairs of form (query point, distance threshold) $Q$, the **BDRQ** returns the largest set $R$ that contains objects $(p,(q,\varepsilon))$ such that $R = \{(p,(q,\varepsilon)) : p \in P, (q,\varepsilon) \in Q \wedge distance(p,q) \leq \varepsilon\}$. The basic idea is as follows. We utilize a main memory area $M$. We divide $Q$ in subsets such that processing of each subset can be done within $M$.

– One method of processing is the following. Every query pair could be seen as a query window with circular schema. Therefore, we can follow the filtering step of the $BWQ$ method (presented in Subsect. 4.2) down to the leaf level for the *Minimum Bounding Square* ($MBS$) of each query pair. At the leaf level, we apply a refinement step for the leaves and the actual query pairs (which are circles). Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the corresponding query pairs for each entry, since each region intersected with the minimum bounding square of a query pair may be a candidate for containing points of the objects of the result ($R$). For the entries of the current node that contain a non-empty list of query pairs we massively read the nodes corresponding to these entries into $M$. This process continues down to the leaf level, where

we read the leaves corresponding to these entries into $M$. For each leaf, we determine all the points of $P$ that exist into the leaf and fall inside the regions of the query pairs of the list. Since, in this method we utilize MBSs, we call it *BDRQ-MBS*.

– According to an alternative processing method, every query pair could be seen as the actual circle it represents. Therefore, we can apply the filtering step as follows. Hierarchically, we visit the tree nodes and for each node we process the regions of the entries contained within, to create a list of the corresponding query pairs for each entry. For each entry $e$ of a tree node, we calculate the minimum distance between the region of the entry and the point of the query pair $q$ ($minDist(e, q)$). Every point $q$ with $minDist(e, q) \leq \varepsilon$ is added into the query list of $e$. It is expected that each region entry having intersection with a query pair may be a candidate to contain points of the objects of the resulting set ($R$). For the entries of the current node that contain a non-empty list of query pairs, we massively read the nodes corresponding to these entries into $M$. To simplify the calculations (reduce the execution time) we calculate the square of $minDist$ between a point and the region of an entry and we compare this metric with the square of the given $\varepsilon$. This process continues down to the leaf level, where we read the leaves corresponding to these entries into $M$. For each leaf, we determine all the points of $P$ that exist into the leaf and fall inside the regions of the query pairs of the list. Since, in this method we utilize $minDist$, we call it *BDRQ-mD*.

## 5 Experimental Results

We run a large set of experiments to compare the repetitive application of the existing algorithms for processing batch queries to the new algorithms, designed especially for batch queries, in SSDs. 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 transformed the MBRs of line-segments from NArdN and NArrN into points by taking the center of each MBR (i.e. $|\text{NArdN}| = 569082$ points, $|\text{NArrN}| = 191558$ points). Moreover, 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, $|\text{NArdND}| = 1138164$ 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 (retrieved from http://spatialhadoop.cs.umn.edu/datasets.html), which 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. All experiments were performed on a Dell Precision T3500 workstation, running CentOS Linux 7 with Kernel 4.15.4 and equipped with a quad-core Intel Xeon W3550 CPU, 8 GB of main memory, an 1TB 7.2K SATA-3 Seagate HDD used for the operating system and a 512 GB SM951A Samsung SSD hosted on PCI-e 2.0 interface, storing our executables and data. Since our algorithms are especially designed for maximizing performance when applied on SSDs, the xBR$^+$-tree was stored on the SSD of our system. However, we tested storing our structure on the HDD, too and obtained execution times 2 orders of magnitude larger than the ones on the SSD. Therefore, we present results of execution on the SSD only.

We run experiments for studying the performance of existing and new algorithms for processing batches of PLQs, WQs and DRQs. We tested batches consisting of $2^{10}, 2^{12}, 2^{14}$ and $2^{16}$ queries. We also tested tree node sizes equal to 4 KB, 8 KB and 16 KB. In each experiment, we counted actual disk accesses and total execution time. Since the existing algorithms answer batch queries by repetitive application for each query of the batch (One-by-One, or ObO, execution), we experimented with and without the use of LRU buffer equal to 256 internal nodes and 256 leaf nodes. This discrimination of the two parts of LRU buffer is necessary, since internal nodes are significantly fewer and a common LRU buffer would get frequently emptied from internal nodes, although the same internal nodes are more likely to be needed for separate queries of the batch. Our experiments showed that this buffer size is adequate for maximizing performance, even for the largest trees tested. The maximum size of $M$ was comparable to LRU size (although, in many cases this maximum size was not utilized by the algorithms studied). Therefore, for each query (PLQ, WQ and DRQ), we tested no-LRU ObO, LRU ObO and the respective new algorithm. The total number of experiments performed equals 972 (combinations of 9 datasets, 3 node sizes, 4 batch sizes, 3 queries and 3 algorithms for each query). Due to space limitations, in the following we present indicative (or, a limited part of the) experimental results, expressing the general trends found.

### 5.1 PLQ Experiments

To study PLQs, we created batches consisting of 50% existing and 50% non-existing points in each dataset. Both existing and non-existing points cover the whole indexed space. In Fig. 3 right, in the upper bar chart, we depict the (%) gain of LRU ObO vs no-LRU ObO, regarding actual disk accesses and total execution time, for the NArdN dataset, the 3 node sizes (4 KB, 8KB and 16 KB) and $2^{12}$ batch size. Note that the gain is defined (for both metrics) as the fraction of the difference of performance of the second and the first algorithms over the performance of the second algorithm (gain $= \frac{noLRU\ ObO - LRU\ ObO}{noLRU\ ObO}$, in this chart). The LRU version is a clear winner (gain more than 90%, for disk accesses and more than 88%, for execution time). In Fig. 3 left, in the upper table, we depict the exact (%) gain figures of LRU ObO vs no-LRU ObO, regarding actual disk accesses and total execution time for all batch sizes (the first column denotes

number of queries, #Q) and the same values for the rest experimental parameters. Note that the line in bold font corresponds to the diagram next to the table. It is clear that the LRU version maximizes performance for all batch sizes. Among the execution of PLQs using these two algorithms for all datasets, the min/max gain for disk accesses was 80.7%/99.4%, while the min/max gain for execution time was 74.4%/98.2%. In all datasets, as batch size increases, gain is also increased.

| #Q | Disk Read Acc | | | Exec Time | | |
|---|---|---|---|---|---|---|
| | 4KB | 8KB | 16KB | 4KB | 8KB | 16KB |
| **LRU ObO vs no-LRU ObO** | | | | | | |
| $2^{10}$ | 89.6 | 88.9 | 91.4 | 87.3 | 84.9 | 87.5 |
| **$2^{12}$** | **92.7** | **92.0** | **93.6** | **90.9** | **88.3** | **92.4** |
| $2^{14}$ | 93.6 | 92.6 | 94.2 | 92.2 | 90.5 | 92.4 |
| $2^{16}$ | 94.1 | 93.0 | 94.5 | 92.8 | 90.9 | 93.2 |
| **BPLQ vs LRU ObO** | | | | | | |
| $2^{10}$ | 0.3 | 0.4 | 0.5 | 11.6 | 23.1 | 28.8 |
| **$2^{12}$** | **17.9** | **18.6** | **17.7** | **33.0** | **45.8** | **37.7** |
| $2^{14}$ | 38.1 | 40.4 | 43.7 | 55.3 | 62.7 | 67.5 |
| $2^{16}$ | 59.9 | 70.1 | 77.9 | 75.2 | 84.9 | 89.6 |

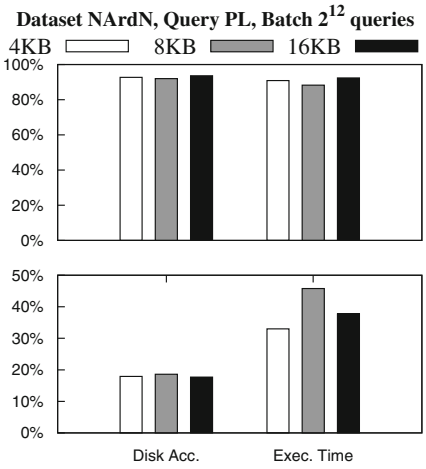


**Fig. 3.** PLQ: % performance gain (regarding Disk Accesses and Exec. Time) of LRU ObO vs no-LRU ObO and BPLQ vs LRU ObO, for the NArdN dataset.

In Fig. 3 right, in the lower bar chart, we depict the (%) gain of BPLQ vs LRU ObO, regarding actual disk accesses and total execution time, for the same experimental settings as in the upper bar chart. BPLQ is approximately 18% more efficient than LRU ObO regarding disk accesses and more than 33% more efficient than LRU ObO regarding execution time. Note that gain increases in the case of execution time, meaning that BPLQ minimizes computations even more than minimizing disk accesses, in relation to LRU ObO. In Fig. 3 left, in the lower table, we depict the exact (%) gain figures of BPLQ vs LRU ObO, regarding actual disk accesses and total execution time for all batch sizes and the same values for the rest experimental parameters. Note that again the line in bold font corresponds to the diagram next to the table. It is clear that BPLQ maximizes performance for all batch sizes. Among the execution of PLQs using these two algorithms for all datasets, the min/max gain for disk accesses was 0.2%/95.3%, while the min/max gain for execution time was 2.8%/95.8%. Again, in all datasets, as batch size increases, gain is also increased (in the case of BLPQ vs LRU ObO, the relative gain improvement is larger than the one between no-LRU ObO vs LRU ObO).

### 5.2 WQ Experiments

To study WQs, we created batches with query windows that cover the whole indexed space. Figure 4 (which is analogous to Fig. 3) depicts indicative results for the WQ and the 1000KCN dataset. As it is evident from the upper chart of this figure, the LRU version is a clear winner (gain more than 68% for disk accesses and more than 66% for execution time). It is also clear from the upper table of this figure that the LRU version maximizes performance for all batch sizes. Among the execution of WQs using these two algorithms for all datasets, the min/max gain for disk accesses was 1.4%/95.7%, whereas the min/max gain for execution time was −2.7%/93.8%. In all datasets, gain increases with increasing batch size. Note that the minimum values of gain with respect to actual disk accesses are observed in the cases of smaller batch sizes consisting of query windows with large dispersion. This way, the use of LRU buffering becomes ineffective because it causes an increase in the execution time of the queries (the overhead of buffer management overcomes the benefit of saving accesses). It is evident from the lower chart of Fig. 4 that BWQ is more than 31% more efficient than LRU ObO, regarding disk accesses and more than 75% more efficient than LRU ObO, regarding execution time. Note that gain increases in the case of execution time, meaning that BWQ minimizes computations even more than minimizing disk accesses, in relation to LRU ObO. The dominance of BWQ is verified for all batch sizes, in the lower table of Fig. 4. Among the execution of WQs using these two algorithms for all datasets, the min/max gain for disk accesses was 0%/97.6%, while the min/max gain for execution time was 42.1%/97.1%. Again, in all datasets, as batch size increases, gain is also increased.

| #Q | Disk Read Acc | | | Exec Time | | |
|---|---|---|---|---|---|---|
| | 4KB | 8KB | 16KB | 4KB | 8KB | 16KB |
| **LRU ObO vs no-LRU ObO** | | | | | | |
| $2^{10}$ | 36.6 | 39.4 | 50.8 | 34.2 | 38.4 | 50.8 |
| $2^{12}$ | 58.7 | 59.8 | 67.1 | 55.6 | 58.6 | 66.6 |
| **$2^{14}$** | **72.5** | **68.3** | **70.9** | **69.9** | **66.9** | **70.4** |
| $2^{16}$ | 74.7 | 69.4 | 71.5 | 73.3 | 67.7 | 70 |
| **BWQ vs LRU ObO** | | | | | | |
| $2^{10}$ | 0.6 | 1 | 2.3 | 43.9 | 45.4 | 48.9 |
| $2^{12}$ | 5.7 | 8.1 | 24.7 | 51.6 | 63.3 | 77.2 |
| **$2^{14}$** | **31.7** | **56.1** | **75.1** | **79.3** | **88.8** | **92.7** |
| $2^{16}$ | 78.2 | 87.7 | 93.4 | 93.5 | 95.7 | 96.7 |

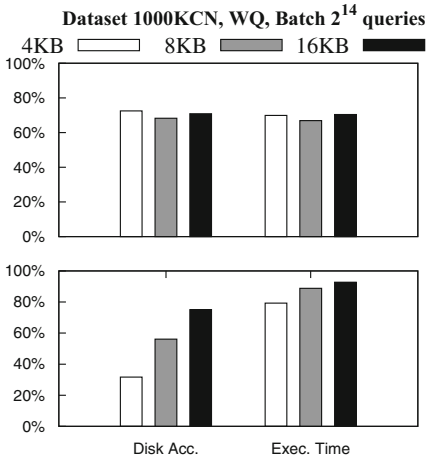


**Fig. 4.** WQ: % performance gain (regarding Disk Accesses and Exec. Time) of LRU ObO vs no-LRU ObO and BWQ vs LRU ObO, for the 1000KCN dataset.

### 5.3 DRQ Experiments

To study DRQs, we created batches with query windows that cover the whole indexed space. In the experimental results, the new algorithm used was BDRQ-mD, since, BDRQ-MBS showed worse execution time performance, especially for big datasets. Although, BDRQ-MBS is faster regarding CPU processing, this is overcome by the smaller number of disk accesses needed by BDRQ-mD. Figure 5 (which is analogous to Figs. 3 and 4) depicts indicative results for the DRQ and the Park dataset. As it is evident from the upper chart of this figure, the LRU version is a clear winner (gain more than 78% for disk accesses and more than 77% for execution time). It is also clear from the upper table of this figure that the LRU version maximizes performance for all batch sizes. Among the execution of DRQs using these two algorithms for all datasets, the min/max gain for disk accesses was 1.2%/95.6%, while the min/max gain for execution time was −0.7%/93.7%. In all datasets, as batch size increases, gain is also increased. Note that the minimum values of gain for the metric of actual disk accesses are observed in the cases of smaller batch sizes consisting of query distance ranges with large dispersion. This way, the use of LRU buffering becomes ineffective as the query execution time is increased. It is evident from the lower chart of Fig. 5 that BDRQ is from 18% more efficient than LRU ObO regarding disk accesses and more than 64% more efficient than LRU ObO regarding execution time. Note that gain increases in the case of execution time, meaning that BDRQ minimizes computations even more than minimizing disk accesses, in relation to LRU ObO. The dominance of BDRQ is verified for all batch sizes, in the lower table of Fig. 5. Among the execution of DRQs using these two algorithms for all datasets, the min/max gain for disk accesses was 0%/97.6%, while the min/max

| #Q | Disk Read Acc | | | Exec Time | | |
|---|---|---|---|---|---|---|
| | 4KB | 8KB | 16KB | 4KB | 8KB | 16KB |
| **LRU ObO vs no-LRU ObO** | | | | | | |
| $2^{10}$ | 9.9 | 19.4 | 25.8 | 9 | 16.2 | 21.3 |
| $2^{12}$ | 28.4 | 45 | 53.5 | 28.1 | 41.3 | 47.9 |
| $2^{14}$ | 53.6 | 69.7 | 75.3 | 54.1 | 66.2 | 68.7 |
| **$2^{16}$** | **78.6** | **86.9** | **88.7** | **77.6** | **83.9** | **82.9** |

| | | | | | | |
|---|---|---|---|---|---|---|
| **BDRQ-mD vs LRU ObO** | | | | | | |
| $2^{10}$ | 0.6 | 0.4 | 1 | 80.5 | 76.2 | 69 |
| $2^{12}$ | 2 | 2.4 | 3.4 | 75.6 | 70.1 | 62.3 |
| $2^{14}$ | 5.8 | 7 | 9.3 | 71.4 | 64 | 63.1 |
| **$2^{16}$** | **18.5** | **21.4** | **27.1** | **66.3** | **64** | **64.7** |

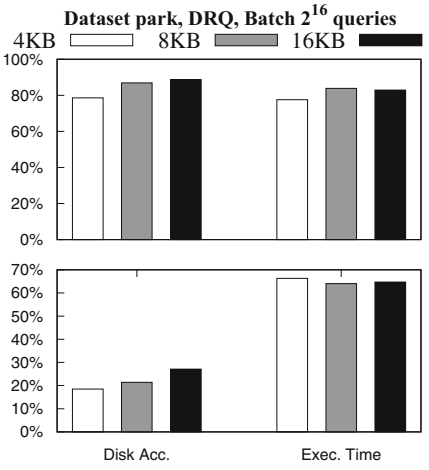


**Fig. 5.** DRQ: % performance gain (regarding Disk Accesses and Exec. Time) of LRU ObO vs no-LRU ObO and BDRQ vs LRU ObO, for the Park dataset.

gain for execution time was 31.4%/97.1%. Again, in all datasets, gain increases with batch size increasing.

## 6 Conclusions and Future Work

In this paper, for the first time in the literature, we present algorithms for common spatial batch queries on single datasets, using xBR$^+$-trees in SSDs. Processing of spatial queries in SSDs has not received considerable attention in the literature, so far. The new algorithms proposed outperform the repetitive application of existing algorithms by exploiting the massive I/O advantages of SSDs, both regarding actual disk access and execution time, even if the I/O of existing algorithms are assisted by LRU buffering.

For all three queries studied, all batch sizes and all datasets, the new algorithms always exhibit better performance than the existing algorithms, for both metrics (actual disk accesses and execution time). For all three queries studied, the new algorithms exhibit maximum gain of execution time that exceeds 95% for large batches and big datasets. Therefore, the processing proposed is best suited to heavily queried big data. Nevertheless, in the case of WQs and DRQs, maximum gain of execution time is significant (more than 41% and 32%, respectively) even for small batches and small datasets.

Future work plans include:

- Developing and studying the performance of algorithms for other common spatial queries (e.g. $K$ Nearest Neighbors, queries involving two datasets, like $K$ Closest Pairs, Distance-Range Joins, All $K$ Nearest Neighbors, etc.) in SSDs.
- Developing algorithms for spatial queries in SSDs that utilize other structures (e.g. of the R-tree family, or Grid files).
- Developing parallel algorithms, utilizing multiple CPUs/GPU cores, for spatial queries in SSDs.

**Acknowledgments.** Work of Antonio Corral, Michael Vassilakopoulos and Yannis Manolopoulos funded by the MINECO research project [TIN2017-83964-R].

## References

1. Carniel, A.C., Ciferri, R.R., de Aguiar Ciferri, C.D.: A generic and efficient framework for spatial indexing on flash-based solid state drives. In: Kirikova, M., Nørvåg, K., Papadopoulos, G.A. (eds.) ADBIS 2017. LNCS, vol. 10509, pp. 229–243. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66917-5_16
2. Cho, S., Chang, S., Jo, I.: The solid-state drive technology, today and tomorrow. In: ICDE Conference, pp. 1520–1522 (2015)
3. Cornwell, M.: Anatomy of a solid-state drive. Commun. ACM **55**(12), 59–63 (2012)
4. Fevgas, A., Bozanis, P.: Grid-file: towards to a flash efficient multi-dimensional index. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) DEXA 2015. LNCS, vol. 9262, pp. 285–294. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22852-5_24

5. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
6. Hady, F.T., Foong, A.P., Veal, B., Williams, D.: Platform storage performance with 3d XPoint technology. Proc. IEEE **105**(9), 1822–1833 (2017)
7. Jin, P., Xie, X., Wang, N., Yue, L.: Optimizing R-tree for flash memory. Expert Syst. Appl. **42**(10), 4676–4686 (2015)
8. Li, G., Zhao, P., Yuan, L., Gao, S.: Efficient implementation of a multi-dimensional index structure over flash memory storage systems. J. Supercomput. **64**(3), 1055–1074 (2013)
9. Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D., Najjar, W.A.: Efficient indexing data structures for flash-based sensor devices. TOS **2**(4), 468–503 (2006)
10. Lv, Y., Li, J., Cui, B., Chen, X.: Log-compact R-tree: an efficient spatial index for SSD. In: Xu, J., Yu, G., Zhou, S., Unland, R. (eds.) DASFAA 2011. LNCS, vol. 6637, pp. 202–213. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20244-5_20
11. Pawlik, M., Macyna, W.: Implementation of the aggregated R-tree over flash memory. In: Yu, H., Yu, G., Hsu, W., Moon, Y.-S., Unland, R., Yoo, J. (eds.) DASFAA 2012. LNCS, vol. 7240, pp. 65–72. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29023-7_7
12. Roh, H., Kim, S., Lee, D., Park, S.: As B-tree: a study of an efficient B+-tree for SSDs. J. Inf. Sci. Eng. **30**(1), 85–106 (2014)
13. Roh, H., Park, S., Kim, S., Shin, M., Lee, S.: $B^+$-tree index optimization by exploiting internal parallelism of flash-based solid state drives. PVLDB **5**(4), 286–297 (2011)
14. Roh, H., Park, S., Shin, M., Lee, S.: Mpsearch: multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs. IEEE Data Eng. Bull. **37**(2), 3–11 (2014)
15. Roumelis, G., Vassilakopoulos, M., Corral, A.: Performance comparison of xBR-trees and R*-trees for single dataset spatial queries. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 228–242. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23737-9_17
16. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Bulk-loading $xBR^+$-trees. In: Bellatreche, L., Pastor, Ó., Almendros Jiménez, J.M., Aït-Ameur, Y. (eds.) MEDI 2016. LNCS, vol. 9893, pp. 57–71. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45547-1_5
17. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Bulk insertions into $xBR^+$-trees. In: Ouhammou, Y., Ivanovic, M., Abelló, A., Bellatreche, L. (eds.) MEDI 2017. LNCS, vol. 10563, pp. 185–199. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66854-3_14
18. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Efficient query processing on large spatial databases: a performance study. J. Syst. Softw. **132**, 165–185 (2017)
19. Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., Manolopoulos, Y.: The $xBR^+$-tree: an efficient access method for points. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) DEXA 2015. LNCS, vol. 9261, pp. 43–58. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22849-5_4
20. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. **16**(2), 187–260 (1984)
21. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading (1990)

22. Sarwat, M., Mokbel, M.F., Zhou, X., Nath, S.: FAST: a generic framework for flash-aware spatial trees. In: Pfoser, D. (ed.) SSTD 2011. LNCS, vol. 6849, pp. 149–167. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22922-0_10
23. Vassilakopoulos, M., Manolopoulos, Y.: External balanced regular (x-BR) trees: new structures for very large spatial databases. In: Advances in Informatics: Selected papers of the 7th Panhellenic Conference on Informatics, pp. 324–333. World Scientific (2000)
24. Wu, C., Chang, L., Kuo, T.: An efficient R-tree implementation over flash-memory storage systems. In: ACM-GIS Conference, pp. 17–24 (2003)