Cluster Computing
https://doi.org/10.1007/s10586-019-03013-0

Parallel processing of spatial batch-queries using xBR$^+$-trees in solid-state drives

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

Received: 9 May 2019 / Revised: 24 October 2019 / Accepted: 30 October 2019

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. Moreover, nowadays each CPU contains multiple cores which can be utilized to perform calculations in parallel and further improve performance of query processing. In this paper, we present algorithms for processing common spatial (point-location, window and distance-range) batch queries using xBR$^+$-trees in SSDs. Next, we transform these algorithms to additionally take advantage of the multiple CPU cores. Moreover, utilizing small and large datasets, we experimentally study the performance of these new, SSD based, algorithms against processing of batch queries by repeatedly applying existing algorithms for these queries. We further study the performance of the algorithms that utilize parallelism against the ones taking advantage of SSDs only. Our experiments show that the new algorithms taking advantage of SSDs and even further the ones that also utilize multiple cores prevail performance-wise. Nevertheless, we discuss how these new parallel algorithms can be extended to work in a distributed environment, taking advantage of parallelism between machines, while processing data of larger scales.

Keywords Spatial indexes · xBR$^+$-trees · Query processing · Solid-state drives · Multi-core CPUs

✉ Michael Vassilakopoulos
mvasilako@uth.gr

George Roumelis
groumelis@uth.gr

Polychronis Velentzas
cvelentzas@uth.gr

Antonio Corral
acorral@ual.es

Athanasios Fevgas
fevgas@uth.gr

Yannis Manolopoulos
yannis.manolopoulos@ouc.ac.cy

[1] Data Structuring and Eng. Lab., Department of Electrical and Computer Engineering, University of Thessaly, 38221 Volos, Greece

[2] Department on Informatics, University of Almeria, 04120 Almería, Spain

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

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 KNNQs, respectively, in the sequel). 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 [25], which is based on the principle of recursive decomposition of space and has become an important access method for spatial applications [26].

The external balanced regular (xBR)-tree [28] 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 [20], 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, KNNQs, etc. [23].

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 [8]. 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 [17].

Due to power limitations of chips, the possible increase of CPU clock speed that can be achieved in new generations of CPUs is limited and, therefore, manufacturers improve performance by creating CPUs with multiple cores. Multi-core CPUs are common in today's commodity hardware. On the other hand, processing of spatial queries has been widely investigated for decades, focusing mainly on single threaded code. Processing of spatial queries usually demands significant processing power and developing algorithms that utilize multiple cores can improve overall performance considerably.

In [24], we presented 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 and are also included in this paper. Such large sequences of queries (batch queries) appear frequently in applications. Moreover, in this paper, we elaborate on the algorithms of [24] and create new algorithms that additionally take advantage of the multiple CPU cores. Extending the experimentation presented in [24], based on small and large datasets, we experimentally study the performance of these new, SSD based, algorithms against processing of batch queries by repeatedly applying existing algorithms for these queries and further study the performance of the new algorithms that utilize parallelism against the ones taking advantage of SSDs only. Our experiments show that the new algorithms taking advantage of SSDs and even further the ones that also utilize multiple cores are clear performance winners. However, the performance achieved by the mutiple cores of a CPU, or the amount of data a centralized system can process might not be enough for future applications. Therefore, we also discuss how these new parallel algorithms can be extended to work in a distributed environment, taking advantage of parallelism between machines, while processing data of larger scales.

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, on processing spatial queries using multiple cores 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. Section 5 presents extensions of these new algorithms that further take advantage of multiple cores. The results of our experiments are discussed in Sect. 6. Section 7 discusses how the new parallel algorithms can be extended to work in a distributed environment. Finally, Sect. 8 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 and continue with the most representative spatial indexes, taking advantage of SSD performance and spatial processing on multi-core processors. Finally, the main motivation of this work is exposed.

2.1 The xBR-tree family

The xBR-tree was initially proposed in [28] as a secondary-memory pointer-based structure that belongs to the Quadtree family. The original xBR-tree was enhanced in [19]. The xBR$^+$-tree [20, 23] 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 [21, 22], respectively.

In [23], 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 [16]. Many research efforts have been made for Flash efficient database indexes. The works for spatial data processing mostly concern the R-tree.

The RFTL [29] 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 [15].

The LCR-tree [12] 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 [9] 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 [10] is a log-structured implementation of the K-D-B-tree for Flash, the MicroGF [11] 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 [27] 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 [16, 18] 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 [24], we developed 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.

2.3 Spatial processing on multi-core processors

Multi-core processors usually have a small number of cores (in most cases, between two and eight) as opposed to high performance computing systems. This characteristic constitutes a challenge for the parallelization of algorithms, since extensive use of parallelism can degrade performance, due to the overhead of parallelization and competition for resources, like shared memory. Instead, parallelism on a small scale might prove more appropriate. Spatial operations used for processing spatial data, even I/O bound ones, require significant core processing power; therefore, it is worthwhile to re-design and parallelize such spatial algorithms for multi-core CPU architectures that are commonly available in nowadays computers.

There are several parallelisms that can be mapped to different parallel hardware (e.g., multi-core CPUs, GPUs and MICs) at different levels (e.g., multi-processor, thread-blocks, SIMD elements). In the context of spatial processing, most of the research has been applied on multi-core CPUs, GPUs and MICs. In [32], data-parallel designs and implementations of point-to-polyline shortest distance computation and point-in-polygon topological test on different commodity hardware using real large-scale spatial data are proposed, comparing their performance and discussing important factors that may significantly affect the performance. Moreover, parallel designs and implementations of spatial indexes on commodity parallel hardware are becoming available, like GPU-based R-trees [30], Quadtrees [31] and simple flat Grid files [33] for spatial indexing and spatial filtering.

In [13] a parallel version of the plane-sweep algorithm targeted towards the small number of processing cores available on commonly available multi-core systems is presented. Experimental results show that the proposed algorithm significantly outperforms the serial plane-sweep on such spatial systems. An improved version of parallelizing plane-sweep algorithms for spatial computations on multi-core processors has been published in [14].

To the best of our knowledge, there has not appeared any work in the literature on processing spatial batch-queries which combines the use of SSDs and also takes advantage of multiple CPU cores, to improve performance. Therefore, in this paper, we elaborate on the algorithms of [24] and develop new algorithms for processing such queries that combine the utilization of an efficient index (the xBR$^+$-tree), exploit the massive I/O advantages of SSDs and make use of the multiple cores existing in a modern CPU.

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 (its advantages over trees belonging to the R-tree family are summarized in Sect. 2.1). The xBR$^+$-tree [20] 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 (= 2^2) equal subquadrants, while for 3d space, the space indexed is a *cube* and it is recursively subdivided in 8 (= 2^3). In this paper, we focus on 2d data. 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. To study the contents of a node more easily and understand the mechanism behind subtracting of subregions, it is suggested to examine the node entries from right to left. 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 subquadrant, the *Address* is 22* (non-complete square, denoted by ‘noS’, since the descendent region 220* has been subtracted, while handling an overflow during an insertion, or update). 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. To further demonstrate the tree structure and the relation between parent region and child, the child of the last root entry (220*) can be examined. This child divides the region addressed by 220* in two parts: the first part is the SW subquadrant of address 220* (denoted by 2*), corresponding to the absolute address 2202*, and the

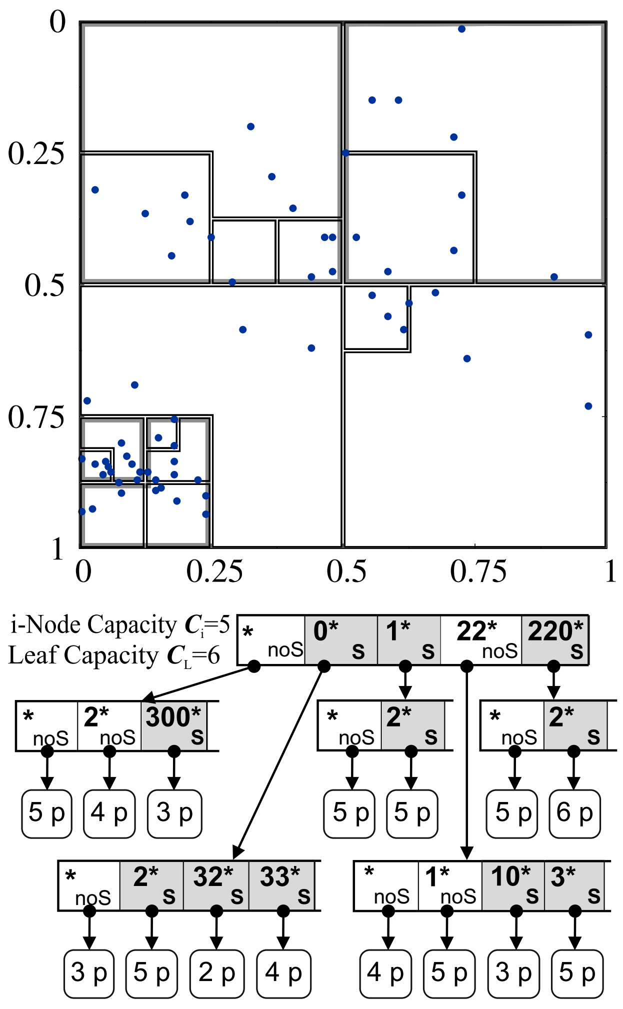


Fig. 1 A collection of 64 points, its grouping to xBR$^+$-tree nodes and its xBR$^+$-tree

second part is the remaining area of address 220* (denoted by *), after subtracting its SW subquadrant. 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_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 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 (26th) 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).

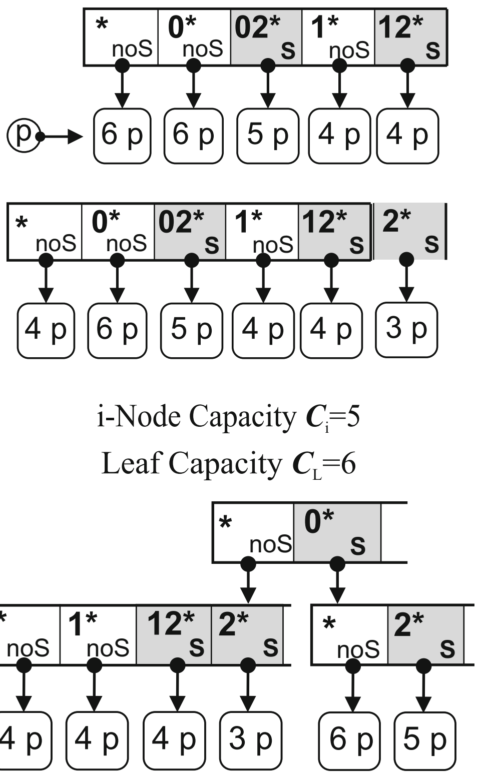


Fig. 2 *Up* an xBR$^+$-tree root pointing to 5 leaves. *Middle* an overflown xBR$^+$-tree root pointing to 6 leaves. *Down* the resulting xBR$^+$-tree after splitting of the root

The new leaf covers the region of the subquadrant 2* and holds 3 points (middle 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 lower 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. Note that the leaf corresponding to the SW (2*) of the NE (1*) subquadrant of the whole space contains 5 points, since there is one point on the top left corner of this subquadrant. The minimum coordinates of a leaf/internal node region belong to these regions, while the maximum ones do not. This means that regions are closed regarding their left and top borders. Details on the algorithms for splitting leaf and internal nodes appear in [20].

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 each subset can be processed within M. Hierarchically, we visit the tree nodes and partition the query points in groups as follows: we examine the node entries from right to left and for each entry, using a variation of quicksort, we partition the query points to the points falling inside the current node entry and the remaining ones; we repeat for the next (to the left) node entry with the remaining points, until we reach the leftmost node entry, or the remaining points are exhausted. This procedure guarantees that each group of points uniquely falls within one subregion of the current node. Next, we massively read the child nodes corresponding to the node entries containing non-empty groups of points 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 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 which have been 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 $R = \{(p, w) : p \in P \wedge p$ falls inside $w \in W\}$.

The basic idea (an extension of the method in Sect. 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 Sect. 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 which has been 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 Sect. 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 bounding squares, we call it *BDRQ-B*.
- 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 ε. 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 minimum distance, we call it *BDRQ-D*.

5 Algorithms for parallel batch-queries processing

In this section, we present the techniques we have utilized, the general idea and the specific algorithms that we developed for processing of the queries studied by taking advantage of the multiple cores of a CPU.

5.1 Parallelization techniques

Although, the queries we study are I/O bound (the factor that dominates performance is the cost of accessing secondary storage) and the algorithms we present in Sect. 4 are designed to take advantage of the internal parallelism of SSDs, the CPU cost of processing such queries is not negligible. In this section we further evolve our algorithms so as to utilize the multiple cores of the CPU as much as possible and minimize the time of CPU processing (the cumulative time of CPU processing may be enlarged, since multiple cores are used, however, the actual time of CPU processing is reduced).

Processing of batch PLQs, WQs, or DRQs includes the following operations.

- Filtering at internal nodes, where we determine which query objects (points, windows, circular ranges, or their bounding squares) satisfy the query criterion for the regions of the entries (children) of each internal node.
- Refinement at leaf nodes, where we determine which query objects (points, windows, circular ranges, or their bounding squares) exist (in the case of query points), or include data points (in the other cases of query objects) in each leaf.

If we have to process multiple internal nodes, filtering can be done in parallel, by assigning a number of such nodes to each CPU core. If, however, we have to process one, or a few internal nodes, filtering can still take advantage of parallelism by assigning a number of entries of this (these) node(s) to different cores, during partitioning the query objects according to the query criterion for the regions of these entries.

Accordingly, since in practice during refinement we always have to process multiple leaves, refinement can be done in parallel by assigning a number of leaves to each CPU core.

Moreover, when processing PLQs, query points that fall within the region of a leaf should be sorted by one of their coordinates, before applying plane-sweep to discover which of these points exist within this leaf. We utilize QuickSort which can also be done with parallelism if the number of elements to be sorted exceed a predefined number (through experimentation we concluded that an effective setting for this threshold is 256).

Note that, if each of the available cores is engaged in some kind of parallel processing of the ones mentioned above, using an additional type of parallel processing might not improve the total CPU processing efficiency. For example, if parallel processing between nodes during filtering is employed, using parallel processing between entries also would likely have no positive effect. On the contrary, since employing parallelism has always some overhead, it might have a negative effect. Note also that the effect of each type of parallelism depends on the distribution of the specific dataset and the query being processed.

Therefore, for each type of query, we have tested four configurations for the parallel algorithms we developed. These are depicted in Table 1. The 1st configuration employs no parallel processing. However, the algorithm used is the same as in the other configurations (described in the following) and, depending on the dataset and the query, it may be faster due to the absence of overhead.

5.2 Basic ideas of parallel query processing

The algorithms we presented in Sect. 4 work on a depth-first basis. Processing starts at the root and recursively reaches the leaves. In order to better utilize the parallelization techniques exposed previously, we redesigned our algorithms to work on a breadth-first basis as much as possible. This means that each algorithm processes (exploiting multiple cores) as many nodes of each level as possible, before proceeding to the next level. “As many $\cdots$ as possible” is related to the main memory available. Since main memory is limited, loading the data for a whole level of the tree might not be possible. Therefore, considering the memory limit, for each level (iteratively) we load as many nodes and as many query objects as possible, process them in parallel and continue processing in-depth (recursively), before returning to the level where there are more nodes waiting to be processed. This idea is applied on every new tree level we visit.

Our technique employs a combination of breadth-first and depth-first processing to exploit parallelism. Therefore, we call the technique we use as *restricted breadth depth-first processing*. We believe that this technique can also be applied to other algorithms working on trees, so as to take advantage of parallel processing.

Table 1 The configurations of parallel processing tested

Config.	Filtering	Refinement	Sorting
1	No	No	No
2	Yes (nodes)	No	Yes (>256)
3	Yes (partitioning)	Yes (nodes)	Yes (>256)
4	Yes (nodes)	Yes (nodes)	Yes (>256)

5.3 Parallel algorithm for processing of batch point-location queries

In this subsection, we present our new processing method for parallel batch point-location queries (*PBPLQ*) using xBR$^+$-trees in SSDs. The basic idea is as follows. We use two buffers with predefined sizes by the system administrator, the node buffer M_N and the query point buffer M_Q. We divide Q in subsets such that each subset can be processed within M_Q. For each tree level, starting at the root level, we massively read as many nodes of the current level as possible into M_N. Using parallelism between nodes, or between entries of nodes (depending on the number of nodes read), we partition the query points in groups and distinguish the ones that uniquely fall within a subregion (region of a child entry) of the nodes read. Next, we massively read the child nodes corresponding to the resulting subregions (belonging to possibly multiple nodes of the same level) into M_N. This process continues recursively down to the leaf level, where we read the leaves (children of possibly multiple nodes of the same level) corresponding to the resulting subregions into M_N. We process leaves in parallel and, for each leaf, we determine all the query points of Q that exist in this leaf. If the number of query points to be examined in such a leaf is large enough, parallel sorting can also be employed. As recursive calls return, as many nodes as possible of the respective level that have not been processed yet are loaded into M_N and the process is repeated for them.

The algorithm is described in more details as follows. To be able to handle multiple nodes of the same level, using of stacks is needed. Although, a single stack could be used, for efficiency of copying operations of consecutive multiple stack records, we use two synchronized stacks, a stack of pointers to nodes, called S_N, and a stack holding information of query points groups, called S_{Qi}.

Initialization

- Read the query points that fit in M_Q.
- Push into S_{Qi} a record (stack frame) that specifies all the points of M_Q.
- Read the root into M_N.
- Push into S_N the record for the root.
- Call the PBPLQ algorithm with input S_N and S_{Qi}.

```
Algorithm PBPLQ input $S_N$, $S_{Qi}$
– If # nodes > threshold, execute loop in parallel
      For each record $r$ of $S_N$ and respective record of $S_{Qi}$,
    • Create stacks $S_{N_r}$ and $S_{Qi_r}$
    • For each entry $e$ of the node of $r$,
        • partition (with parallel code, unless parallel execution is already employed) the query points for $e$ (in and out of the region of $e$).
        • if there are points inside the region of $e$, push a record that specifies the child node of $e$ into $S_{N_r}$ and a record that specifies the points inside $e$ into $S_{Qi_r}$.
– Merge all $S_{N_r}$/$S_{Qi_r}$ stacks into one, which replaces $S_N$/$S_{Qi}$.
– If this is the level of leaf parents,
    • While the stacks have records,
        • massively transfer the maximum possible number of pointed leaves into $M_N$, replacing its existing content.
        • If # leaves > threshold, execute loop in parallel
              For each leaf, sort (with parallel code, unless parallel execution is already employed) the query points that correspond to this leaf and by using plane-sweep calculate the result,
– Else (this is another internal level)
    • While the stacks have records,
        • massively transfer the maximum possible number of pointed nodes into $M_N$, replacing its existing content.
        • call recursively PBPLQ, passing as parameters the part of $S_N$ and $S_{Qi}$ that corresponds to the nodes read.
```

5.4 Parallel algorithm for processing of batch window and distance-range queries

In this subsection, we present our processing methods, using xBR$^+$-trees in SSDs, for Parallel Batch Window Queries (*PBWQ*) and our Parallel Batch Distance-Range Queries based on circular ranges (*PBDRQ-D*), or on range bounding squares (*PBDRQ-B*).

The basic idea (an extension of the method in Sect. 5.3) is as follows. We use two buffers with predefined sizes by the system administrator, the node buffer M_N and the query object (window, circular range, or bounding square) buffer M_Q. We divide Q in subsets such that each subset can be processed within M_Q. For each tree level, starting at the root level, we massively read as many nodes of the current level as possible into M_N. Using parallelism between nodes, or between entries of nodes (depending on the number of nodes read), we partition the query objects in groups and distinguish the ones that intersect with a subregion of the nodes read, since each subregion intersected with a query object may be a candidate for containing points of the pairs of the result (R). Next, we massively read the child nodes corresponding to the intersected subregions (belonging to possibly multiple nodes of the same level) into M_N. This process continues recursively down to the leaf level, where we read the leaves (children of possibly multiple nodes of the same level) corresponding to the resulting subregions into M_N. We process leaves in parallel and, for each leaf, we determine the points of this leaf that fall inside any of the query objects of Q. As recursive calls return, as many nodes as possible of the respective level that have not been processed yet are loaded into M_N and the process is repeated for them. The algorithm is described in more details as follows.

Again, we use two synchronized stacks, a stack of pointers to nodes, called S_N, and a stack holding information of query object (window, circular range, or bounding square) groups, called S_{Qi} and two buffers with predefined sizes, the node buffer M_N and the query object buffer M_Q.

Initialization

– Read the query objects (windows / circular ranges / bounding squares) that fit in M_Q.
– Sort the query objects (by descending order, according to their right end, so that plane-sweep can be applied).
– Push into S_{Qi} a record for these query objects (pointing to M_Q).
– Read the root into M_N.
– Push into S_N the record for the root.
– Call the **PBPLQ / PBDRQ-D / PBDRQ-B** algorithm with input S_N and S_{Qi}.

```
Algorithm PBPLQ/PBDRQ-D/PBDRQ-B input $S_N$, $S_{Qi}$
– If # nodes > threshold, execute loop in parallel
      For each record $r$ of $S_N$ and respective record of $S_{Qi}$.
    • Create stacks $S_{N_r}$ and $S_{Qi_r}$.
    • Execute loop in parallel, unless parallel execution is already employed
          For each entry $e$ of the node of $r$,
        • discover the query objects (windows / circular ranges / bounding squares) that intersect with the region (DBR) of $e$.
        • if there query objects intersecting the region of $e$, push related entries into $S_{N_r}$ and $S_{Qi_r}$.
```

- Merge all S_{N_r} / S_{Qi_r} stacks into one, which replaces S_N / S_{Qi}.
- If this is the level of leaf parents,
  - While the stacks have records,
  - massively transfer the maximum possible number of pointed leaves into M_N, replacing its existing content.
  - *If # leaves > threshold, execute loop in parallel* For each leaf, by using plane-sweep calculate the points that fall inside query objects.
- Else (*this is another internal level*)
  - While the stacks have records.
  - massively transfer the maximum possible number of pointed nodes into M_N, replacing its existing content.
  - call recursively the **PBPLQ / PBDRQ-D / PBDRQ-B** algorithm, passing as parameters the part of S_N and S_{Qi} that corresponds to the nodes read.

6 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, which either use one [24], or multiple CPU cores.

We used real spatial datasets of North America representing roads (NArdN with 569082 line-segments) and railroads (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. |NArdN| = 569082 points, |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, |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.

The C programming language was used for implementing the algorithms and the OpenMP library was used for implementing parallelism. 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 (supporting Hyper-Threading technology, with 8 logical cores), 8GB of main memory, an 1TB 7.2K SATA-3 Seagate HDD used for the operating system and a 512GB 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 (DRQs were processed in two versions, using an MBS and the actual circular range). We tested batches consisting of $2^{17}, 2^{18}, 2^{19}$ and 2^{20} queries. We also tested tree node sizes equal to 4KB, 8KB and 16KB. In each experiment, we counted actual disk accesses and total execution time.

The existing algorithms answer batch queries by repetitive application for each query of the batch (One-by-One, or ObO, execution) 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). Following the results presented in [24], in this paper we present only results for ObO execution with LRU enabled.

Therefore, for each query (PLQ, WQ and DRQ in two versions), we tested LRU ObO, the respective new, SSD optimized, single-core algorithm and the respective new multi-core algorithm (using 2, 4 and 8 cores and 4 configurations for each case). The total number of experiments performed equals 6048 (combinations of 9 datasets, 3 node sizes, 4 batch sizes, 4 queries and 14 algorithmic versions

Table 2 PLQ: (disk accesses % performance gain and Total Exec. Time absolute values and % time performance gain) BPLQ versus LRU ObO, for the 500KC dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BPLQ versus LRU ObO												
2^{17}	70.6	72.6	64.4	1995	163.4	91.8	1530	164.0	89.3	1183	180.2	84.8
2^{18}	71.8	73.2	64.8	2428	273.5	88.7	2099	283.6	86.5	1790	323.2	81.9
2^{19}	72.1	73.3	64.9	3135	486.7	84.5	3268	518.1	84.1	2994	592.7	80.2
2^{20}	74.5	75.4	68.1	4666	925.8	80.2	4776	986.2	79.3	5506	1140	79.3

Table 3 PLQ: (disk accesses % performance gain and Total Exec. Time absolute values and % time performance gain) BPLQ versus LRU ObO, for the Park dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BPLQ versus LRU ObO												
2^{17}	32.0	34.7	36.7	1039	454.7	56.3	1095	451.6	58.8	1469	530.9	63.9
2^{18}	33.7	35.9	37.1	1585	668.5	57.8	1709	680.0	60.2	2431	845.5	65.2
2^{19}	34.4	36.1	37.0	2566	1057	58.8	2870	1092	61.9	4232	1413	66.6
2^{20}	37.7	39.0	40.6	4568	1818	60.2	5065	1914	62.2	7918	2562	67.6

Table 4 PLQ: (Total Exec. Time % performance gain) PBPLQ versus BPLQ, for the 500KC dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBPLQ versus BPLQ									
2^{17}	07.1	10.1	13.6	17.1	23.8	30.6	14.7	23.9	21.7
2^{18}	09.5	11.4	14.2	21.0	24.4	32.6	18.8	24.9	27.0
2^{19}	09.9	11.5	13.8	21.3	25.2	32.0	21.2	24.2	24.7
2^{20}	10.5	12.0	14.8	21.3	25.9	32.3	20.5	25.7	25.8

Table 5 PLQ: (Total Exec. Time % performance gain) PBPLQ versus BPLQ, for the Park dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBPLQ versus BPLQ									
2^{17}	14.9	10.8	19.4	18.0	13.6	26.1	17.7	15.9	23.6
2^{18}	20.2	12.1	23.2	20.5	14.9	30.7	19.1	17.8	29.0
2^{19}	18.1	13.3	25.9	21.2	18.6	31.8	21.2	20.5	30.4
2^{20}	19.4	17.9	29.3	22.1	25.1	37.0	23.3	22.7	32.8

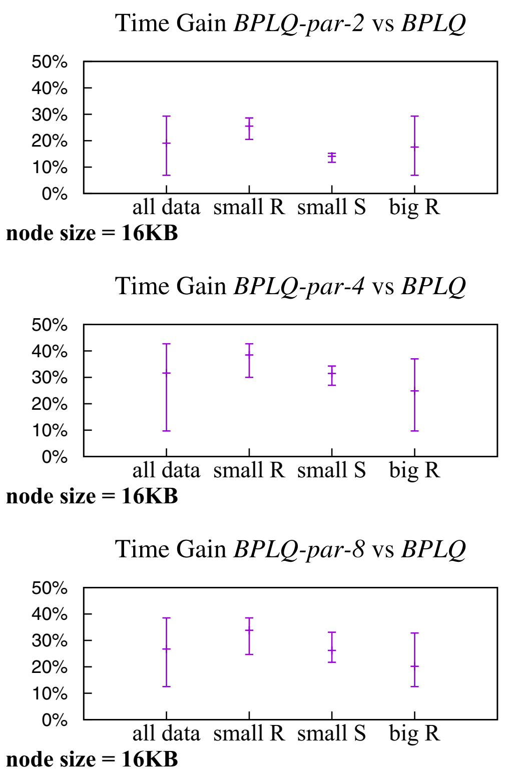


Fig. 3 PLQ: min, max and average gain of PBPLQ versus BPLQ

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. In all experiments performed, the query batches used contain 2^{17}, 2^{18}, 2^{19} and 2^{20} query objects (column Q in tables with experimental results).

6.1 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 Table 2, for the 500KC dataset and 3 page sizes, we depict the gain (as a percentage) of BPLQ over LRU ObO, regarding disk accesses. We also depict the absolute total execution times (in ms) of LRU ObO and BPLQ and the execution time gain (as a percentage) of BPLQ over LRU ObO. 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 $\left(gain = \frac{LRUObO-BPLQ}{LRUObO}\right)$ and expresses the speedup obtained by first algorithm, in relation to the time cost of the first algorithm.

As another indicative result set, in Table 3, we depict analogous data for the Park dataset. It is evident from both tables that BPLQ saves a significant number of disk accesses over LRU ObO and it is even more efficient regarding execution time. Analogous remarks can be made for the other dataset cases.

In Tables 4 and 5, for the same datasets as the ones of Tables 2 and 3, respectively, we depict the total execution time gain (as a percentage) of PBPLQ over BPLQ, for 3 page sizes and 2, 4 and 8 cores used (for each number of cores, this gain has been calculated using the best of the 4 configurations of Table 1). Studying these results, it can be noted that the parallel algorithms tend to maximize their gain over BPLQ for the larger page size. For both datasets depicted, the parallel algorithms are clearly faster than BPLQ and justify their use, although the query studied is I/O bound.

In Fig. 3, for the larger page size, we depict the smaller, larger and average total execution time gain (represented by the lower, upper and middle dash of each vertical line, respectively) of the parallel algorithms over BPLQ for all datasets, small real datasets, small synthetic datasets and big real datasets. The top, middle and lower diagram corresponds to parallel algorithms utilizing 2, 4 and 8 cores, respectively. These diagrams further justify that it is worth

Table 6 WQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BWQ versus LRU ObO, for the NArdND dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BWQ versus LRU ObO												
2^{17}	4.4	12.2	26.3	1447	386.0	73.3	1322	271.1	79.5	1192	246.8	79.3
2^{18}	4.4	12.2	26.3	1927	483.9	74.9	2064	445.6	78.4	1900	433.2	77.2
2^{19}	4.4	12.2	26.4	2689	789.0	70.7	2890	836.4	71.1	2747	820.9	70.1
2^{20}	4.4	12.2	26.4	4145	1499	63.8	5781	1612	72.1	4806	1598	66.7

Table 7 WQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BWQ versus LRU ObO, for the Water dataset

Q #	Disk read Acc			Exec time								
	4KB	8KB	16KB	4KB			8KB			16KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BWQ versus LRU ObO												
2^{17}	0.2	0.2	0.3	1841	673.9	63.4	1387	597.5	56.9	1369	630.8	53.9
2^{18}	0.1	0.1	0.2	3011	1033	65.7	2254	913.0	59.5%	2313	1044	54.9
2^{19}	0.1	0.1	0.2	4630	1516	67.3	3476	1344	61.3	3802	1717	54.8
2^{20}	0.1	0.1	0.2	6906	2248	67.4	5062	2062	59.3	6299	2908	53.8

Table 8 WQ: (Total Exec. Time % performance gain) PBWQ versus BWQ, for the NArdND dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBWQ versus BWQ									
2^{17}	06.5	05.3	10.0	09.2	10.8	18.3	08.5	10.3	17.6
2^{18}	10.3	07.1	10.4	14.2	12.7	19.6	13.1	11.6	18.4
2^{19}	11.5	10.5	11.8	16.8	14.6	21.6	16.4	13.5	19.8
2^{20}	14.6	09.9	13.4	19.5	15.7	23.1	18.3	14.9	21.4

Table 9 WQ: (Total Exec. Time % performance gain) PBWQ versus BWQ, for the Water dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBWQ versus BWQ									
2^{17}	04.0	05.1	08.7	06.1	08.2	12.3	05.1	06.4	12.8
2^{18}	06.7	08.7	15.6	08.0	12.8	19.5	07.3	11.6	20.0
2^{19}	09.0	13.2	20.8	11.0	17.7	26.1	10.7	17.2	25.9
2^{20}	11.1	17.1	22.9	14.8	23.1	29.5	13.4	22.4	28.9

utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch PLQs. For example, for 8 cores, considering all datasets, the minimum gain is over 10% and the maximum gain is close to 40%. This is important, considering that the PLQ is I/O bound.

6.2 WQ experiments

To study WQs, we created batches with query windows that cover the whole indexed space. Tables 6 and 7 are analogous to Tables 2 and 3, for the NArdND and Water datasets, respectively. Depending on the dataset, BWQ saves disk accesses over LRU ObO, while for both datasets and it is significantly more efficient than LRU ObO regarding execution time.

Tables 8 and 9 are analogous to Tables 4 and 5, for the NArdND and Water datasets, respectively. Studying these results, we reach analogous conclusions the ones for PLQ: the parallel algorithms maximize their gain over BWQ for the larger page size. For both datasets depicted, the parallel algorithms are clearly faster than BWQ and justify their use, although the query studied is I/O bound.

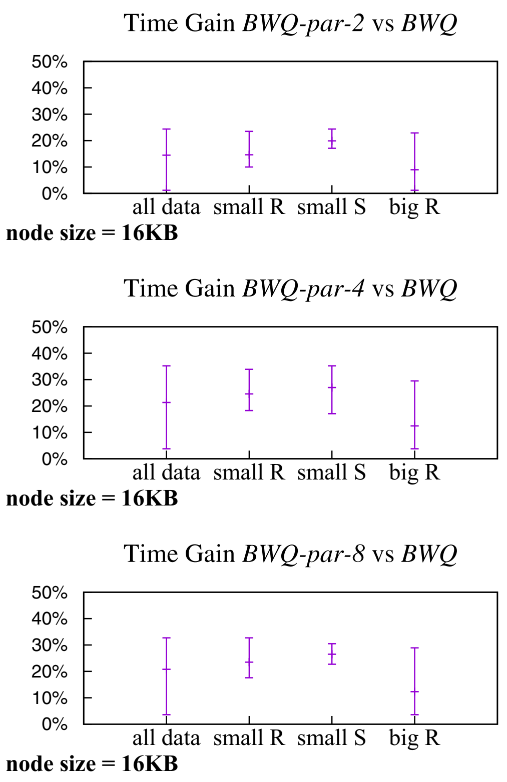


Fig. 4 WQ: min, max and average gain of PBWQ vs BWQ

Figure 4 is analogous to Fig. 3. The diagrams of this figure, like in the case of PLQ, further justify that it is worth utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch WQs. For example, for 8 cores, considering all datasets, the minimum gain is positive, the average gain is over 20% and the maximum gain is around 35%.

6.3 DRQ experiments

To study DRQs, we created batches with query windows that cover the whole indexed space. We run experiments for both BDRQ-D and BDRQ-B and the respective parallel versions.

Tables 10, 14 and 11, 15 are analogous to Tables 2 and 3, for the 1000KC/NArd and Build/Park datasets, respectively. Depending on the dataset, BDRQ-D/PBDRQ-B saves disk accesses over LRU ObO, while for both datasets and it is significantly more efficient than LRU ObO regarding execution time.

Tables 12, 16 and 13, 17 are analogous to Tables 4 and 5, for the 1000KC/NArd and Build/Park datasets, respectively. Studying these results, we reach analogous conclusions the ones for PLQ: the parallel algorithms maximize their gain over BDRQ-D/BDRQ-B for the larger page size. For all datasets depicted, the parallel algorithms are clearly faster than BDRQ-D/BDRQ-B and justify their use, although the query studied is I/O bound (Tables 14, 15, 16, 17).

Figures 5 and 6 is analogous to Fig. 3. The diagrams of this figure, like in the case of PLQ, further justify that it is worth utilizing the parallel algorithms and making use of the multiple CPU cores for processing batch DRQs. For example, for 8 cores and PBDRQ-D / PBDRQ-B, considering all datasets, the minimum gain is positive, the average gain is around 35%/20% and the maximum gain is over 40%/35%.

7 Processing in a distributed environment

So far, we presented centralized algorithms that take advantage both of SSDs and multiple CPU cores, to accelerate spatial query processing. If the data to be processed is too large to be handled in a centralized system, a

Table 10 DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-D versus LRU ObO, for the 1000KC dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BDRQ-D versus LRU ObO												
2^{17}	35.9	45.6	50.6	2054	298.8	85.4	1762	289.8	83.6	1732	316.1	81.8
2^{18}	35.4	45.1	50.3	2497	475.3	81	2288	502.5	78	2427	559.4	76.9
2^{19}	35.1	44.9	50.2	3702	882.7	76.2	3437	949.5	72.4	4616	1091	76.4
2^{20}	35.0	44.9	50.2	5182	1751	66.2	5715	1871	67.3	7026	2515	64.2

Table 11 DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-D versus LRU ObO, for the Build dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BDRQ-D versus LRU ObO												
2^{17}	04.1	03.1	01.7	20314	3021	85.1	13631	2576	81.1	9021	2174.0	75.9
2^{18}	12.1	09.2	05.8	35928	4725	86.8	23801	4038	83%	15175	3440.4	77.3
2^{19}	25.4	21.0	14.1	66770	7136	89.3	42339	6129	85.5	26075	5059	80.6
2^{20}	45.3	39.6	28.4	127312	9919	92.2	77916	8931	88.5	44075	7285	83.5

Table 12 DRQ: (Total Exec. Time % performance gain) PBDRQ-D versus BDRQ-D, for the 1000KC dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBDRQ-D versus BDRQ-D									
2^{17}	09.8	12.7	19.9	18.6	21.1	32.5	16.3	18.8	26.9
2^{18}	12.0	15.0	21.2	20.6	24.6	34.7	19.1	20.0	27.8
2^{19}	15.7	16.7	23.7	25.4	26.4	36.4	23.7	21.8	29.0
2^{20}	20.3	19.3	35.6	28.7	28.4	44.3	25.0	22.9	38.3

Table 13 DRQ: (Total Exec. Time % performance gain) PBDRQ-D versus BDRQ-D, for the Build dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBDRQ-D versus BDRQ-D									
2^{17}	05.6	04.8	03.1	07.2	06.4	05.8	06.9	04.9	04.5
2^{18}	04.9	05.4	03.9	08.0	07.7	08.3	06.8	07.7	06.9
2^{19}	07.4	07.1	05.7	10.7	10.1	11.0	08.6	09.7	09.6
2^{20}	08.9	09.8	08.4	12.2	14.0	16.1	11.5	14.1	15.3

Table 14 DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-B versus LRU ObO, for the NArd dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4KB			8KB			16KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BDRQ-B versus LRU ObO												
2^{17}	11.3	24.9	32.5	944.3	216.6	77.1	830.2	201.0	75.8	852.9	229.8	73.1
2^{18}	11.4	25.0	34.2	1260	359.7	71.4	1181	371.6	68.5	1352	446.4	67.0
2^{19}	11.4	25.0	34.2	1851	695.5	62.4	1875	716.8	61.8	2374	1025	56.8
2^{20}	11.5	25.0	34.2	3049	1424	53.3	3273	1437	56.1	5134	2128	58.6

Table 15 DRQ: (Disk Accesses % performance gain and Total Exec. Time absolute values and % performance gain) BDRQ-B versus LRU ObO, for the Park dataset

Q #	Disk read Acc			Exec time								
	4 KB	8 KB	16 KB	4 KB			8 KB			16 KB		
	Gain (%)	Gain (%)	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)	Time	Time	Gain (%)
BDRQ-B versus LRU ObO												
2^{17}	02.9	03.6	05.0	3546	1094	69.1	2607	926.5	64.5	2466	918.2	62.8
2^{18}	03.3	03.2	03.9	5664	1613	71.5	4297	1305	69.6	3995	1408	64.8
2^{19}	07.1	03.9	03.9	8900	2299	74.2	6284	1794	71.5	6409	2123	66.9
2^{20}	14.9	08.4	04.7	13111	3273	75.0	9194	2627	71.4	10143	3598	64.5

Table 16 DRQ: (Total Exec. Time % performance gain) PBDRQ-B versus BDRQ-B, for the Nard dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBDRQ-B versus BDRQ-B									
2^{17}	05.7	11.1	18.1	15.9	14.7	27.1	05.7	11.1	18.1
2^{18}	07.9	11.9	19.7	17.9	15.6	30.8	07.9	11.9	19.7
2^{19}	13.3	13.9	30.2	21.5	18.3	42.0	13.3	13.9	30.2
2^{20}	17.6	17.0	34.0	25.7	20.3	43.8	17.6	17.0	34.0

Table 17 DRQ: (Total Exec. Time % performance gain) PBDRQ-B versus BDRQ-B, for the Park dataset

Q #	Exec time gain								
	2 cores			4 cores			8 cores		
	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)	4 KB (%)	8 KB (%)	16 KB (%)
PBDRQ-B vs BDRQ-B									
2^{17}	04.3	01.4	04.8	05.3	04.8	06.1	04.3	01.4	04.8
2^{18}	05.9	02.2	06.3	07.9	06.0	07.8	05.9	02.2	06.3
2^{19}	09.7	03.9	09.0	12.7	09.9	10.8	09.7	03.9	09.0
2^{20}	14.1	11.0	15.6	17.8	17.2	17.7	14.1	11.0	15.6

distributed computing environment could be utilized. More specifically, a cluster of computers interconnected through a fast LAN could handle data which are larger by orders of magnitude.

For several problems, related algorithms cannot be easily transferred to such a shared-nothing computing environment. However, this is not the case for the algorithms we have presented. Due to the nature of the queries studied, these algorithms can be easily applied in such a distributed setting. If data is distributed among nodes, each node could index its data using an xBR$^+$-tree, based on SSDs. The query batch would have to be transmitted to every node of the cluster, each node would compute the answer, as far as the data it stores are concerned, and the results from all nodes could be merged in a node that will act as a coordinator of query processing. This is possible, since the answer of any of the queries we studied for a node would be independent to the answers for other nodes.

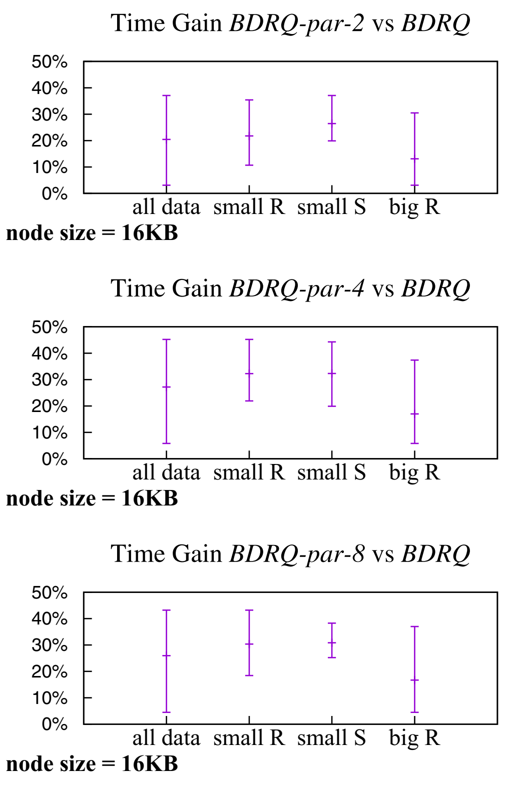


Fig. 5 WQ: min, max and average gain of PBWQ vs BWQ

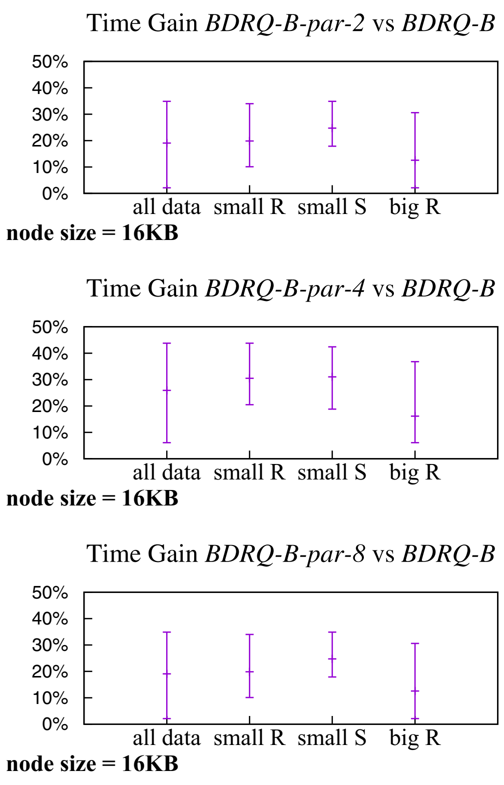


Fig. 6 WQ: min, max and average gain of PBWQ vs BWQ

Although, this approach will work, efficiency requires that data are not blindly distributed among nodes. Each node should keep data that are spatially close, even in case partial intersection between the areas covered by nodes is allowed. In the case, the coordinating node should transmit to each computing node only the part of the query batch that spatially intersects the area of this node. Therefore, the coordinating node should be aware of the areas covered by computing nodes.

Such processing could be embedded in a parallel and distributed system, like SpatialHadoop (http://spatialhadoop.cs.umn.edu/) that already incorporates spatial indexing methods and distributes data to nodes according to such a method. In [6, 7], spatial query processing techniques have been added to SpatialHadoop. We could build on [6, 7] to embed into Spatiahadoop SSD-based xBR$^+$-trees that process spatial queries, taking advantage of multiple cores.

8 Conclusions and future work

In this paper, extending the algorithms presented in [24], for the first time in the literature, we present algorithms for common spatial batch queries on single datasets, using xBR$^+$-trees in SSDs that take advantage of multiple cores. Processing of spatial queries in SSDs has not received considerable attention in the literature, so far. Even more, the utilization of multiple cores, based on a combination of breadth-first and depth-first tree traversals, is a new approach that further accelerates processing. The algorithms proposed in [24] exploit the massive I/O advantages of SSDs and 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. The parallel extensions of these algorithms clearly outperform the ones of [24], although the three queries studied are I/O bound. The new algorithms can be applied to a parallel and distributed environment and deal with very big data.

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, using mutiple cores.
- Developing algorithms for spatial queries in SSDs that utilize other structures (e.g. of the R-tree family).
- Developing parallel algorithms, utilizing GPU cores, for spatial queries in SSDs.

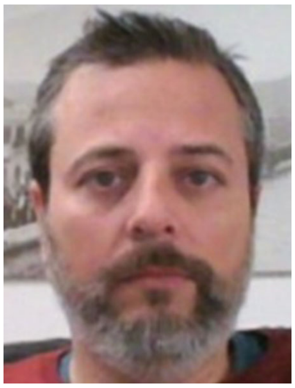
– Embedding the parallel algorihtms developed in a parallel and distributed environment that distributes data obeying spatial locality.

Acknowledgements 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 C., Cristina D.: A generic and efficient framework for spatial indexing on flash-based solid state drives. In: ADBIS Conference, pp. 229–243 (2017)
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: DEXA Conference, pp. 285–294 (2015)
5. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
6. García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M.: Voronoi-diagram based partitioning for distance join query processing in spatialhadoop. In: Proceedings of the 8th International Conference on MEDI 2018 Model and Data Engineering, 24–26 October 2018, Marrakesh, Morocco. pp. 251–267 (2018)
7. García-García, F., Corral, A., Iribarne, L., Vassilakopoulos, M., Manolopoulos, Y.: Efficient large-scale distance-based join queries in spatialhadoop. GeoInformatica **22**(2), 171–209 (2018)
8. Hady, F.T., Foong, A.P., Veal, B., Williams, D.: Platform storage performance with 3d XPoint technology. Proc. IEEE **105**(9), 1822–1833 (2017)
9. Jin, P., Xie, X., Wang, N., Yue, L.: Optimizing R-tree for flash memory. Expert Syst. Appl. **42**(10), 4676–4686 (2015)
10. 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)
11. 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)
12. Lv, Y., Li, J., Cui, B., Chen, X.: Log-compact R-tree: An efficient spatial index for SSD. In: DASFAA Workshops, pp. 202–213 (2011)
13. McKenney, M., McGuire, T.: A parallel plane sweep algorithm for multi-core systems. In: ACM SIGSPATIAL Conference, pp. 392–395 (2009)
14. McKenney, M., Frye, R., Dellamano, M., Anderson, K., Harris, J.: Multi-core parallelism for plane sweep algorithms as a foundation for GIS operations. GeoInformatica **21**(1), 151–174 (2017)
15. Pawlik, M., Macyna, W.: Implementation of the aggregated R-tree over flash memory. In: DASFAA Workshops, pp. 65–72 (2012)
16. Roh, H., Park, S., Kim, S., Shin, M., Lee, S.-W.: B^+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. PVLDB **5**(4), 286–297 (2011)
17. 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)
18. Roh, H., Park, S., Shin, M., Lee, S.-W.: Mpsearch: multi-path search for tree-based indexes to exploit internal parallelism of flash ssds. IEEE Data Eng. Bull. **37**(2), 3–11 (2014)
19. Roumelis, G., Vassilakopoulos, M., Corral, A.: Performance comparison of xBR-trees and R*-trees for single dataset spatial queries. In: ADBIS Conference, pp. 228–242 (2011)
20. Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., Manolopoulos, Y.: The xBR^+-tree: an efficient access method for points. In: DEXA Conference, pp. 43–58 (2015)
21. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Bulk-loading xBR^+-trees. In: MEDI Conference, pp. 57–71 (2016)
22. Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y.: Bulk insertions into xBR^+-trees. In: MEDI Conference, pp. 185–199 (2017)
23. 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)
24. Roumelis, G., Vassilakopoulos, M., Corral, A., Fevgas, A., Manolopoulos, Y.: Spatial batch-queries processing using xBR^+-trees in solid-state drives. In: Proceedings of the 8th international conference on medi 2018, model and data engineering, 24–26 October 2018. Marrakesh, Morocco, pp. 301–317 (2018)
25. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. **16**(2), 187–260 (1984)
26. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison-Wesley, Boston (1990)
27. Sarwat, M., Mokbel, M.F., Zhou, X., Nath, S.: FAST: a generic framework for flash-aware spatial trees. In: SSTD Conference, pp. 149–167 (2011)
28. 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)
29. Wu, C.-H., Chang, L.-P., Kuo, T.-W.: An efficient R-tree implementation over flash-memory storage systems. In: ACM-GIS Conference, pp. 17–24 (2003)
30. You, S., Zhang, J., Gruenwald, L.: Parallel spatial query processing on gpus using r-trees. In: ACM SIGSPATIAL Conference, pp. 23–31 (2013)
31. Zhang, J., You, S.: Speeding up large-scale point-in-polygon test based spatial join on gpus. In: ACM SIGSPATIAL Conference, pp. 23–32 (2012)
32. Zhang, J., You, S.: Large-scale geospatial processing on multi-core and many-core processors: Evaluations on cpus, gpus and mics. CoRR, arXiv:abs/1403.0802 (2014)
33. Zhang, J., You, S., Gruenwald, L.: Parallel online spatial and temporal aggregations on multi-core cpus and many-core gpus. Inf. Syst. **44**, 134–154 (2014)

George Roumelis studied Physics in Aristotle University of Thessaloniki (AUTH), Greece and is currently working as a teacher and a principle in a local high school of Thessaloniki, Greece. He obtained a master's degree in Information Systems from the Open University of Cyprus (2011) and a Ph.D. on Spatial Databases from the Informatics Department of AUTH (2017). His main research interests include access methods, query processing and spatial and spatio-temporal databases. He has published several original papers in scientific journals and international conferences. His interests also include software development and support for the educational and administration units in the public educational system of Greece.



Polychronis Velentzas is a Ph.D. candidate at the Department of Electrical & Computer Engineering, University of Thessaly, Greece. He holds a degree in Informatics and Telecommunications from the National and Kapodistrian University of Athens (Greece) and a MSc degree in Informatics and Computer Eng. from the University of Thessaly (Greece). He is currently working as a lead developer-analyst for the Research Committee of University of Thessaly. He has participated in several research projects funded by the EU and the Greek state. His research interests include Big Data analysis, using parallel and distributed processing.

Michael Vassilakopoulos obtained a five-year Diploma in Computer Eng. and Informatics from the University of Patras (Greece) in 1990 and a Ph.D. in Computer Science from the Department of Electrical and Computer Eng. of the Aristotle University of Thessaloniki (Greece) in 1995. He has been with the University of Macedonia, the Aristotle University of Thessaloniki, the Technological Educational Institute of Thessaloniki, the Hellenic Open University, the Open University of Cyprus, the University of Western Macedonia, the University of Central Greece and the University of Thessaly. For 3 years he served the Greek Public Administration as an Informatics Engineer. Currently, he is an Associate Professor of Database Systems at the Department of Electrical and Computer Engineering of the University of Thessaly. He has participated in/coordinated several R&D projects related to Databases, GIS, WWW, Information Systems and Employment. His research interests include Databases, Data Structures, Algorithms, Data Mining, Employment Analysis, Information Systems, GIS, Parallel and Distributed Computing, Big Data and other current trends of Data Management.

Antonio Corral is an Associate Professor at the Department of Informatics, University of Almeria (Spain). He received his Ph.D. (2002) in Computer Science from the University of Almeria (Spain). He has participated actively in several research projects in Spain (INDALOG, vManager, CoSmart, etc.) and Greece (CHOROCHRONOS, ARCHIMEDES, etc.). He has published in referred scientific international journals (Data & Knowledge Engineering, GeoInformatica, The Computer Journal, Information Sciences, Computer Standards & Interfaces, JSS, etc.), conferences (SIGMOD, SSD, ADBIS, SOFSEM, PADL, DEXA, OTM, MEDI, SAC, etc.) and book chapters. His main research interests include access methods, algorithms, query processing, databases and distributed query processing.

Athanasios Fevgas is a Ph.D. candidate at the Department of Electrical & Computer Engineering, University of Thessaly, Greece. He holds a degree in Information Engineering from the Technological Educational Institute of Thessaloniki (Greece) and a MSc degree in Informatics and Computer Eng. from the University of Thessaly (Greece). He is an IEEE and Computer Society student member. He has participated in several research projects funded by the EU and the Greek state. His research interests include algorithms for non-volatile memories, database storage and indexing, out-of-core direct solvers for sparse matrices.

Yannis Manolopoulos is Professor and Vice-rector of the Open University of Cyprus. He has been with the University of Toronto, the University of Maryland at College Park, the University of Cyprus and Aristotle University of Thessaloniki, where he served as Head of the Department of Informatics. He has also served as Rector of the University of Western Macedonia in Greece and Vice-Chair of the Greek Computer Society. His research interest focuses in Data Management. He has co-authored 5 monographs and 8 textbooks in Greek, as well as >300 journal and conference papers. He has received 13,600 citations from 1800 distinct academic institutions (h-index=53). He has also received 5 best paper awards from SIGMOD, ECML/PKDD, MEDES (2) and ISSPIT conferences and has been invited as keynote speaker in 15 international events. He has served as main co-organizer of several major conferences (among others):

ADBIS 2002, SSTD 2003, SSDBM 2004, ICEIS 2006, EANN 2007, ICANN 2010, AIAI 2012, WISE 2013, CAISE 2014, MEDI 2015, ICCCI 2016, TPDL 2017, DAMDID 2017, DASFAA 2018, EAIS 2018, WIMS 2018. He has also acted as evaluator for funding agencies in Austria, Canada, Cyprus, Czech Republic, Estonia, EU, Hong-Kong, Georgia, Greece, Israel, Italy, Poland and Russia. Currently, he serves in the Editorial Boards of the following journals (among others): Information Systems, World Wide Web, Computer Journal.