GeoInformatica manuscript No.
(will be inserted by the editor)

Porting Disk-based Spatial Index Structures to Flash-based Solid State Drives

Anderson C. Carniel · George Roumelis · Ricardo R. Ciferri · Michael Vassilakopoulos · Antonio Corral · Cristina D. Aguiar

Received: date / Accepted: date

Abstract Indexing data on flash-based Solid State Drives (SSDs) is an important paradigm recently applied in spatial data management. During last years, the design of new *spatial access methods* for SSDs, named *flash-aware spatial indices*, has attracted the attention of many researchers, mainly to exploit the advantages of SSDs in spatial query processing. eFIND is a generic framework for transforming a disk-based spatial index into a flash-aware one, taking into account the intrinsic characteristics of SSDs. In this article, we present a *systematic approach* for porting disk-based data-driven and space-driven access methods to SSDs, through the eFIND framework. We also present the actual porting of representatives data-driven (R-trees, R*-trees, and Hilbert R-trees) and space-driven (xBR$^+$-trees) access methods through this framework. Moreover, we present an extensive experimental evaluation that compares the performance of these ported indices when inserting and querying synthetic and real point datasets. The main conclusions of this experimental study are that the eFIND R-tree excels in inser-

Anderson C. Carniel
Dept. of Computer Science, Federal University of São Carlos, São Carlos, SP 13565-905, Brazil
E-mail: accarniel@ufscar.br

George Roumelis
Dept. of Electrical & Computer Engineering, University of Thessaly, 382 21 Volos, Greece
E-mail: groumelis@uth.gr

Ricardo R. Ciferri
Dept. of Computing, Federal University of São Carlos, São Carlos, SP 13565-905, Brazil
E-mail: rrc@ufscar.br

Michael Vassilakopoulos
Dept. of Electrical & Computer Engineering, University of Thessaly, 382 21 Volos, Greece
E-mail: mvasilako@uth.gr

Antonio Corral
Dept. on Informatics, University of Almeria, 04120 Almeria, Spain
E-mail: acorral@ual.es

Cristina D. Aguiar
Dept. of Computer Science, University of São Paulo, São Carlos, SP 13566-590, Brazil
E-mail: cdac@icmc.usp.br

tions, the eFIND xBR$^+$-tree is the fastest for different types of spatial queries, and the eFIND Hilbert R-tree is efficient for processing intersection range queries.

Keywords Spatial Indexing · Spatial Access Methods · Flash-aware Spatial Index · Flash-based Solid State Drive

1 Introduction

Many database applications require the representation, storage, and management of spatial or geographic information to enrich data analysis. *Spatial database systems* and *Geographic Information Systems* (GIS) provide the foundation for these applications and often employ *spatial index structures* to speed up the processing of *spatial queries* [28; 52; 49], such as *intersection range queries* and *point queries*. The goal of a spatial index is to reduce the search space by avoiding the access of objects that certainly do not belong to the final answer of the query. In general, near spatial objects are grouped into index pages that are organized in a hierarchical structure. To this end, two main approaches are employed [28]: (i) *data partitioning*, and (ii) *space partitioning*. Spatial indices based on the first approach organize the hierarchy oriented by the groups formed from the spatial objects; thus, they are termed *data-driven access methods*. Examples include the R-tree [30] and its variants like the R*-tree [5] and the Hilbert R-tree [38]. Spatial indices belonging to the second approach organize the hierarchy oriented by the division of the space in which the objects are arranged; thus, they are termed *space-driven access methods*. For instance, Quadtree-based indices [58] such as the xBR$^+$-tree [54].

The efficient indexing of multidimensional points has been the main focus of several indices because of the use of points in real spatial database applications [28; 52; 49]. In general, the majority of these indices assumes that the point objects should be indexed in magnetic disks (i.e., *Hard Disk Drives* - HDDs). Hence, they are termed *disk-based spatial indices* since they consider the slow mechanical access and the high cost of search and rotational delay of disks in their design.

On the other hand, advanced database applications are interested in using modern storage devices like *flash-based Solid State Drives* (SSDs) [8; 47; 26]. This includes spatial database systems that employ spatial indices to efficiently retrieve spatial objects (i.e., points) stored in SSDs [23; 39; 9; 10]. The main reason for this interest is because SSDs, in contrast to HDDs, have a smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

However, SSDs have introduced a new paradigm in data management because of their intrinsic characteristics [2; 7; 18; 37; 19]. A well-known characteristic is the asymmetric cost of reads and writes, where a write requires more time and power consumption than a read. Further, SSDs are able to write data to empty pages only, which means that updating data in previously written pages requires an erase-before-update operation. Other factors that impact SSD performance are the processing of interleaved reads and writes, and the execution of reads on frequent locations. These factors are related to the internal controls of SSDs, such as its internal buffers and read disturbance management [37].

To deal with the intrinsic characteristics of SSDs, spatial indices specifically designed for SSDs, termed here as *flash-aware spatial indices*, have been proposed in the literature. Among existing flash-aware spatial indices (see Section 2), eFIND-based indices [11; 15] distinguish themselves. eFIND is a generic framework that transforms a disk-based spatial index into a flash-aware spatial index. It is based on a distinct set of design goals that provides guidelines to deal with the intrinsic characteristics of SSDs. The effectiveness of these guidelines has been validated through experimental evaluations. Another advantage of eFIND is that its data structures do not change the structure of the index being ported, requiring a low-cost integration when implementing eFIND in spatial database systems and GIS.

Although the advantages of eFIND, designing an efficient flash-aware spatial index remains a challenging task. In fact, there are three open problems. First, it is still unclear how to systematically port disk-based spatial indices to SSDs in a way that they exploit the advantages of SSDs. This leads to the second problem, how in-memory structures of eFIND should be adapted to fit well with the structure of the underlying index, which might be a data- or space-driven access method. Finally, the third problem refers to the lack of a performance study that identifies the best index to handle points on SSDs. That is, identify the best hierarchical structure for building indices and for processing spatial queries.

In this article, our goal is to solve these problems by introducing a novel *systematic approach* for porting disk-based spatial index structures to SSDs. The systematic approach is based on the *characterization* of the types of operations that different indexing strategies (i.e., data partitioning and space partitioning) can perform on index pages. In this sense, we focus on identifying when reads and writes are performed by index operations, such as insertions and queries. With this characterization, we leverage an extended and generalized version of the eFIND's data structures and algorithms to implement our systematic approach. We analyze and validate our systematic approach by porting an expressive set of disk-based spatial index structures to SSDs: (i) the R-tree, (ii) the R*-tree, (iii) the Hilbert R-tree, and (iv) the xBR$^+$-tree. Since they are hierarchical structures, in the remainder of this article, we use *node* as an equivalent term to *index page*.

As a result, we highlight the main contributions of this article as follows:

- development of a *systematic approach* that provides the needed guidelines to port a disk-based spatial index to SSDs;
- application of the systematic approach using eFIND for porting the disk-based spatial indices R-tree, R*-tree, Hilbert R-tree, and xBR$^+$-tree to SSDs; thus, we show the *creation of the flash-aware spatial indices eFIND R-tree*, *eFIND R*-tree*, *eFIND Hilbert R-tree*, and *eFIND xBR$^+$-tree*;
- analysis of an *extensive experimental evaluation* that compares the performance of the flash-aware spatial indices when inserting and querying points from synthetic and real datasets;
- identification of the eFIND R-tree as the best flash-aware spatial index to handle insertions, the eFIND xBR$^+$-tree as an efficient structure to execute several types of spatial queries, and the eFIND Hilbert R-tree as an efficient indexing scheme for processing intersection range queries.

The rest of this article is organized as follows. Section 2 surveys related work. Section 3 summarizes the spatial index structures employed in this article and a running example. Section 4 generalizes eFIND aiming at its incorporation into our systematic approach. Section 5 presents our systematic approach for porting disk-based spatial indices to SSDs. Section 6 details the conducted experiments. Finally, Section 7 concludes the article and presents future work.

2 Related Work

This article introduces a systematic approach, which follows the movement of general methods for indexing data, such as GiST [32; 40] and SP-GiST [3]. We present a brief overview of them in Section 2.1. In Section 2.2, we discuss some approaches that port one-dimensional index structures to SSDs. Then, we survey flash-aware spatial indices based on their underlying design: (i) approaches designed for porting a *specific* type of disk-based spatial index to SSDs (Section 2.3), and (ii) approaches that are *generic* and thus port any disk-based index structure to SSDs (Section 2.4).

2.1 Generalized Search Trees

GiST is a data structure that is extensible in terms of data types and definition of index operations. GiST requires the registration of six key methods that encapsulate the structures and behavior of the underlying index structures. For instance, a spatial database system can implement R-trees and variants by registering (i.e., implementing) such methods of GiST. GiST mainly assumes data-driven access methods. To implement space-driven access methods in a general way, SP-GiST can be deployed. SP-GiST defines a set of methods that take into account the similarities of the space-driven access methods, which are mainly related to the internal structure of the tree. In addition, it specifies a set of methods associated with the behavior of the underlying index. GiST and SP-GiST offer algorithms to manipulate the index structures, such as queries, insertions, and deletions, by invoking their key methods as needed.

Similar to GiST and SP-GiST, our systematic approach describes general algorithms for manipulating index operations in data-driven and space-driven access methods. However, differently from them, our systematic approach focuses on indexing spatial objects in SSDs by identifying how nodes are manipulated by the index operations. With this, we are able to provide implementations that take into account the intrinsic characteristics of SSDs. In this article, eFIND is deployed to implement such manipulations since eFIND exploits the advantages of SSDs and shows good performance results compared to FAST, its closest competitor. More details on eFIND and FAST are given in Section 2.4.

2.2 Approaches to Porting One-Dimensional Index Structures to SSDs

Index structures are widely employed to accelerate information retrieval. Such structures applied to alphanumeric data lead to *one-dimensional index structures*. For HDDs, we can cite the traditional B-tree and its variants, the B$^+$-tree and the B*-tree, as examples [20]. With the advances of SSDs, approaches to port one-dimensional index structures to these storage devices have been proposed in the literature; we call them *flash-aware one-dimensional indices*. A common strategy employed by flash-aware one-dimensional indices is to mitigate the negative effects of the poor performance of random writes. Here, we describe key ideas of some existing one-dimensional index structures that port the B-tree (or some variant) to flash memory or SSDs (see [26] for a survey).

The *Lazy-Adaptive tree* [1] ports the B+-tree to raw flash devices by logging updates in data structures stored in the flash memory. Each data structure is associated with a node of the B^+-tree. Updates of a node are appended as log records, which are later mapped in a table to facilitate their access. Hence, this flash-aware one-dimensional index increases the number of access to recover a node for reducing the number of random writes since the updates are possibly scattered in the flash memory. Other one-dimensional indices store the updates in a write buffer and flush them in a batch when space is needed. The *B-tree over the FTL* [64] is based on the *Flash Translation Layer* (FTL) [41]. This index performs a mapping between logical addresses of the FTL and the modified nodes of the B-tree in order to organize the write buffer. Then, the modified nodes are packed in blocks, based on the logical blocks of the FTL, in order to perform a flushing operation. The *FD-tree* [43] organizes the write buffer in different levels of the tree, respecting ascending order. However, depending on the height of the B-tree, the search time may be negatively impacted. Some improvements of the FD-tree are also introduced in [62], which focus on the concurrent control of B-trees in SSDs. The *read/write optimized B^+-tree* [35] also ports the B^+-tree to SSDs. It allows overflowed nodes to reduce random writes and leverages Bloom filters to reduce extra reads to these overflowed nodes.

This article differs from these works since we propose a systematic approach to port *multidimensional* access methods to SSDs. Our approach takes into account spatial index structures based on space and data partitioning.

2.3 Specific Approaches to Porting Spatial Index Structures to SSDs

The flash-aware spatial indices created by the specific approaches widely employ a write buffer to avoid random writes. Whenever the write buffer is full, a flushing operation is performed. We detail the main characteristics of these flash-aware spatial indices as follows.

The *RFTL* [63] ports the R-tree to SSDs and its write buffer is based on the mapping provided by the FTL. That is, it correlates the logical flash pages managed by the FTL with the modified entries of a node of the R-tree. However, the main problem of RFTL is its flushing operation because it flushes all modifications stored in the write buffer, requiring high elapsed times.

The *MicroGF* [44] ports the grid-file [48] to flash-based sensor devices. Due to the low processing capabilities of sensor devices, this index deploys a write buffer only and does not provide solutions for other aspects inherent to SSDs, such as the interference between reads and writes.

The *LCR-tree* [45] leverages a write buffer by using a log-structured format. The benefit of this format is that retrieving a node from the R-tree is optimized and consequently the spatial query processing is improved. However, the log-structured format requires an extra cost of management. Also, the LCR-tree faces the same problems as the RFTL, such as the execution of expensive flushing operations.

The *F-KDB* [42] ports the K-D-B-tree [53] to SSDs by employing a write buffer that stores modified entries as log entries. Logging entries of a node might be stored in different flash pages. Hence, a table in the main memory is used to keep the correspondence between logging entries and its node. The main problem of the

F-KDB is that retrieving nodes is a complex operation, requiring a possibly high number of random reads to access the logging entries.

The *FOR-tree* [34] modifies the structure of the R-tree by allowing overflowed nodes and thus, it abolishes split operations. It also defines a specialized flushing operation that picks some modified nodes to be written to the SSD based on their number of modifications and recency of their modifications. The main problem of the FOR-tree is the management of overflowed nodes. Whenever a specific number of accesses in an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting them into the parent node, growing up the tree if needed. However, the number of accesses of an overflowed root node is never incremented in an insert operation. As a consequence, the construction of a FOR-tree, inserting one spatial object by time, forms an overflowed root node instead of a hierarchical structure. This critical problem disallowed us to create spatial indices over large and medium spatial datasets.

The *Grid file for flash memory* and *LB-Grid* [24; 25] employ a buffer strategy based on the Least Recently Used (LRU) [21] replacement policy to port the grid file to SSDs. They store indexed spatial objects in buckets whose modifications are managed by a logging-based approach; thus, they deploy a write buffer. The buffering scheme is divided into different regions. The first region, called hot, stores recently accessed pages, whereas the second region, called cold, stores the remaining pages. A flushing operation writes to the SSD only those pages that are classified as cold pages. However, the quantity of modifications is not considered, leading to a possibly high number of flushing operations.

Unfortunately, many intrinsic characteristics of SSDs are not taken into account by the aforementioned flash-aware spatial indices. First, they do not mitigate the negative impact of interleaved reads and writes. Second, they assume that reads are the fastest operations in SSDs. However, this is not always the case because of the read disturbance management of SSD. This management requires an extra computational time of SSDs to avoid *read disturbances*, which occur if multiple reads are issued on the same flash page without any previous erase. Consequently, such reads can require a long latency comparable to the latency of writes, as experimentally showed in [37]. Another problem is the lack of data durability. This means that the modifications stored in the write buffer are lost after a system crash or power failure. On the other hand, we propose a generic approach to porting disk-based spatial indices to SSDs that is based on eFIND (see Section 2.4). Thus, such ported indices do not face these problems.

Other works in the literature propose specific flash-aware algorithms for the xBR$^+$-tree, such as spatial batch-queries [56] and bulk-loading strategies [57]. Given a set of spatial queries, an algorithm for spatial batch-queries organizes the nodes to be visited in order to read them as batch operations. Given a set of points, an algorithm for bulk-loading creates an index as an atomic operation attempting to optimize the tree structure. Thus, such studies are focused on very specific types of algorithms involving the xBR$^+$-tree. On the other hand, in this article, we focus on providing a systematic approach to port any spatial index to SSDs. Hence, our solutions can be employed to process transactions like insertions, deletions, and queries in spatial database systems and GIS.

Our previous work [14; 16] ports the xBR$^+$-tree to SSDs using the generic frameworks eFIND and FAST (Section 2.4); thus creating the flash-aware spatial indices *eFIND xBR$^+$-tree* and *FAST xBR$^+$-tree*, respectively. The experiments show that the eFIND xBR$^+$-tree provides the best results because it fits well with the properties and structural constraints of the xBR$^+$-tree (see Section 3.4). However, to accomplish this porting, some modifications in the eFIND's data structures are performed. A limitation of the previous work is that these modifications are not generalized in a form that can be applied to other disk-based spatial index structures. Other limitations are related to the use of eFIND, as detailed in Section 2.4.

2.4 General Approaches to Porting Spatial Index Structures to SSDs

Generic frameworks are promising tools for porting disk-based spatial indices to SSDs. In general, they generalize the write buffer to be used by any underlying index. Further, they also provide solutions for guaranteeing data durability by sequentially storing index modifications contained in the write buffer into a log-structured file. This file is then employed to reconstruct the write buffer after a fatal problem. Further, generic frameworks do not change the structure of the underlying index, requiring a low-cost integration with spatial database systems and GIS. Due to these advantages, this article leverages generic frameworks.

FAST [59] mainly focuses on reducing the number of writes. Hence, FAST provides a specialized flushing algorithm that picks a set of nodes, termed *flushing unit*, to be written to the SSD. A flushing unit is selected by using a *flushing policy*. However, FAST faces several problems. First, its flushing algorithm might pick nodes without modifications, resulting in unnecessary writes. This is due to the static creation of flushing units as soon as nodes are created in the index. Second, its write buffer stores the modifications in a list possibly containing repeated entries, impacting negatively the performance of retrieving modified nodes. Third, FAST does not improve the performance of reads. Finally, it does not provide a solution to the negative impact of interleaved reads and writes.

eFIND [11; 15] is based on a set of design goals that consider the intrinsic characteristics of SSDs to exploit the advantages of these storage devices. To accomplish the design goals, eFIND includes: (i) a generic write buffer that deploys efficient data structures to handle index modifications, (ii) a read buffer that caches frequently accessed nodes (i.e., index pages), (iii) a temporal control that avoids interleaved reads and writes, and (iv) a log-structured approach that guarantees data durability. Further, eFIND specifies a flushing operation that dynamically creates flushing units to be written to the SSD. Because of these data algorithms and strategies, experimental evaluations show that eFIND is more efficient than FAST. However, it is still unclear how to use eFIND to port disk-based spatial indices based on different techniques, such as data partitioning and space partitioning. This is due to the use of eFIND for porting only two indices, the R-tree [15] and the xBR$^+$-tree [14; 16]. Finally, there is a lack of a performance study that indicates the most efficient spatial index structure ported by eFIND.

Differently from [11; 15], which propose a framework for specifying flash-aware spatial index structures based on disk-based structures, and going beyond our previous works [14; 16], which port a specific space-driven access method to SSDs, in this article:

- We propose a novel systematic approach for porting disk-based data-driven and space-driven access methods to SSDs, in general. For this, we characterize how the index operations perform reads from and writes to the SSD.

- We implement the systematic approach by using FAST and eFIND. We particularly focus on describing how eFIND fits in the systematic approach due to its superior performance compared to FAST (see Section 6).
- We extend and generalize eFIND's data structures and algorithms in order to implement the systematic approach. The extensions and generalizations are not focused on one type of spatial index only (such as in [15; 16]). They are conducted to deal with different aspects of the underlying disk-based spatial index structures. For instance, the sorting property of nodes' entries of the Hilbert R-tree and the xBR$^+$-tree. Hence, the data structures are extended to store groups of attributes that are needed to process internal algorithms of the underlying index and to process algorithms of eFIND.
- We show how to apply the systematic approach implemented by eFIND to port the R-tree, the R*-tree, the Hilbert R-tree, and the xBR+-tree by using a running example. As a result, we specify the eFIND R-tree, the eFIND R*-tree, the eFIND Hilbert R-tree, and the eFIND xBR+-tree.
- We conduct an extensive experimental evaluation that compares the implementation of our systematic approach by using FAST and eFIND when porting the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree. This performance evaluation considers: (i) two real datasets, (ii) two synthetic datasets, (iii) two SSDs, and (iv) three different types of workload.

3 An Overview of Spatial Index Structures

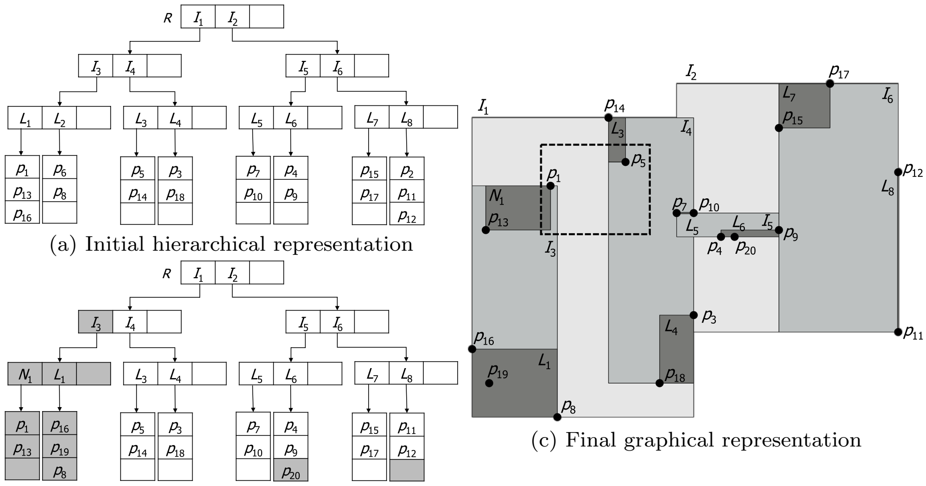
In this section we summarize four spatial index structures employed in this article. They are: (i) the R-tree (Section 3.1), (ii) the R*-tree (Section 3.2), (iii) the Hilbert R-tree (Section 3.3), and (iv) the xBR$^+$-tree (Section 3.4). For each spatial index, we provide its underlying structure and key points for manipulating the indexed spatial objects. Finally, we deploy them to our running example (Section 3.5).

3.1 The R-tree

The R-tree [30] is a classical spatial index that organizes the *minimum bounding rectangles* (MBRs) of the indexed spatial objects in a hierarchical structure; thus, it is a data-driven access method. Figure 1a depicts the hierarchical representation of an R-tree that indexes 18 points (i.e., p_1 to p_{18}), while Figures 1b and c depict the hierarchical and graphical representation of an R-tree that indexes a modified set of 18 points according to our running example (i.e., the previous set of points from which p_{19} and p_{20} have been added, and p_6 and p_2 have been removed).

A node has a minimum and a maximum number of entries indicated by m and M respectively, where $m \leq \frac{M}{2}$. Entries are in the format (id, r). For leaf nodes, id is a unique identifier that provides direct access to the indexed spatial object represented by its MBR r. As for internal nodes, id is the node identifier that supplies the direct access to a child node, and r corresponds to the MBR that covers all MBRs in the child node's entries.

The searching algorithm of the R-tree descends the tree examining all nodes that satisfy a given topological predicate considering a search object. A typical query is the *intersection range query* (IRQ), which returns all spatial objects that



(a) Initial hierarchical representation

(b) Final hierarchical representation

(c) Final graphical representation

Fig. 1 An R-tree in hierarchical representation (a) and the R-tree resulting after applying a set of modifications on it in hierarchical (b) and graphical (c) representations. The hierarchical representation highlights the performed modifications in gray.

intersect a rectangular-shaped object called *query window*. Inserting a spatial object into an R-tree first involves the choice of a leaf node to accommodate its corresponding entry (id, r). The entry is directly inserted in the chosen leaf node if it has enough space. Otherwise, a *split operation* is performed, resulting in the creation of a new leaf node that is later inserted as a new entry in the parent node of the chosen leaf node. A chain of splits might be performed along with the levels of the R-tree, requiring the creation of a new root node if needed.

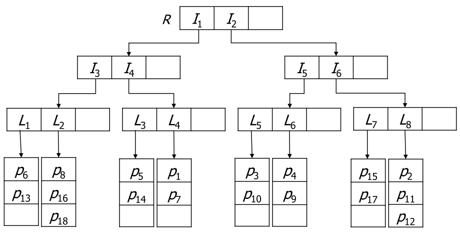
3.2 The R*-tree

The R*-tree [5] is a well-known R-tree variant that aims at improving the hierarchical organization of the indexed spatial objects. Figure 2 depicts the hierarchical and graphical representations of the R*-tree that are analogous to the R-tree ones of Figure 1. The nodes of the R*-tree have the same structure as the R-tree.

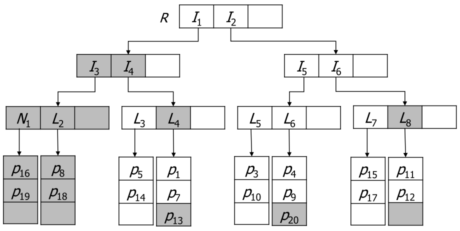
The R*-tree attempts to minimize: (i) the area covered by a rectangle of an entry, (ii) the overlapping area between rectangles of entries, (iii) the margin of the rectangle of an entry, and (iv) the storage utilization. To accomplish them, the R*-tree improves the insert operation of the R-tree and provides a different split algorithm. In special, the R*-tree establishes a *reinsertion policy* (usually 30%), which picks a set of entries of an overflowed node and reinserts them into the tree instead of performing a split. The searching algorithm of the R-tree is not changed.

3.3 The Hilbert R-tree

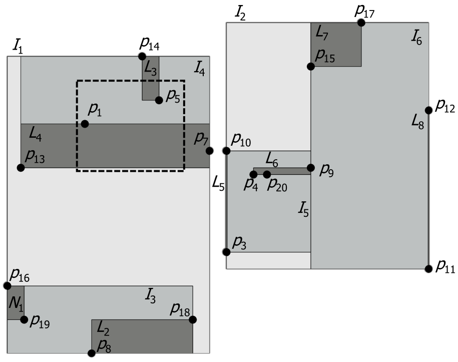
The Hilbert R-tree [38] is another R-tree variant that employs the Hilbert curve when indexing spatial objects. The Hilbert R-tree extends the structure of internal



(a) Initial hierarchical representation



(b) Final hierarchical representation



(c) Final graphical representation

Fig. 2 An R*-tree in hierarchical representation (a) and the R*-tree resulting after applying a set of modifications on it in hierarchical (b) and graphical (c) representations. The hierarchical representation highlights the performed modifications in gray.

nodes of the R-tree (Section 3.1). An internal node consists of entries in the format (id, r, lhv), where id and r have the same meaning as the entries of internal nodes of the R-tree and lhv is the largest Hilbert value among the child node's entries. Leaf nodes of the Hilbert R-tree have the same format as the leaf nodes of the R-tree but are sorted by the Hilbert values of their MBRs.

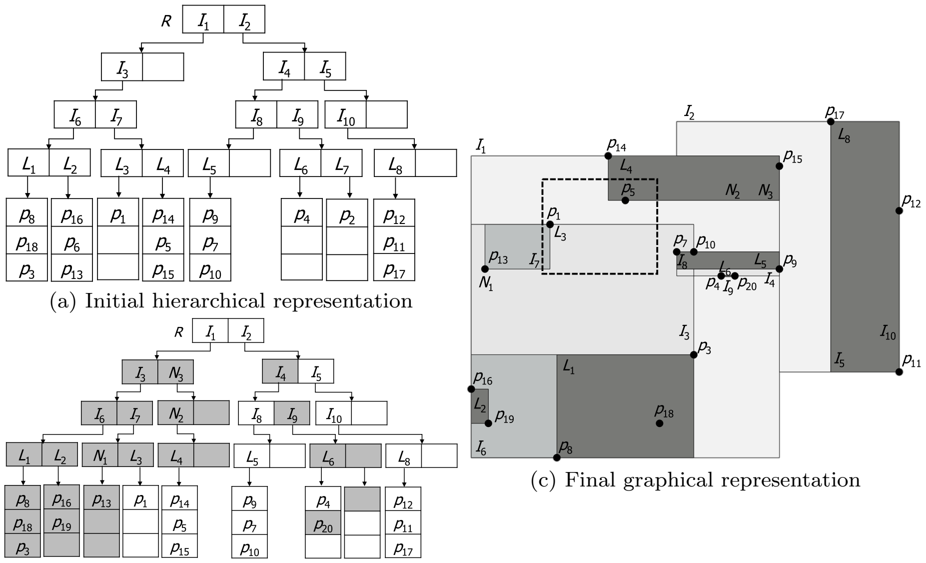
Figure 3 depicts the hierarchical and graphical representations of a Hilbert R-tree in a similar way to the Figures 1 and 2. Because of the extra element in internal nodes and considering that every node has a fixed number of bytes, the maximum capacity of an internal node might be lesser than the maximum capacity of a leaf node. This can be noted in Figure 3, where each internal node can store at most 2 entries.

The structure of the Hilbert R-tree permits that the searching algorithm is the same as the R-tree, and that the insertion is similar to the insertion of a B-tree [21]. It also includes a specific algorithm for handling overflows, which either involves the redistribution of entries among s cooperating siblings of the overflowed node or the execution of an s-to-$s+1$ split policy. Usually, s is equal to 2.

3.4 The xBR$^+$-tree

The xBR$^+$-tree [54] is a hierarchical spatial index based on the regular decomposition of space of Quadtrees [58] able to index multi-dimensional points. Hence, it is a space-driven access method. For two-dimensional points, the xBR$^+$-tree decomposes recursively the space by 4 equal quadrants, called *sub-quadrants*.

Figure 4 depicts the hierarchical and graphical representations of an xBR$^+$-tree on the same objects of Figures 1, 2, and 3. Differently from the R-tree-based indices previously discussed (Sections 3.1 to 3.3), the coordinates on the vertical



(a) Initial hierarchical representation

(b) Final hierarchical representation

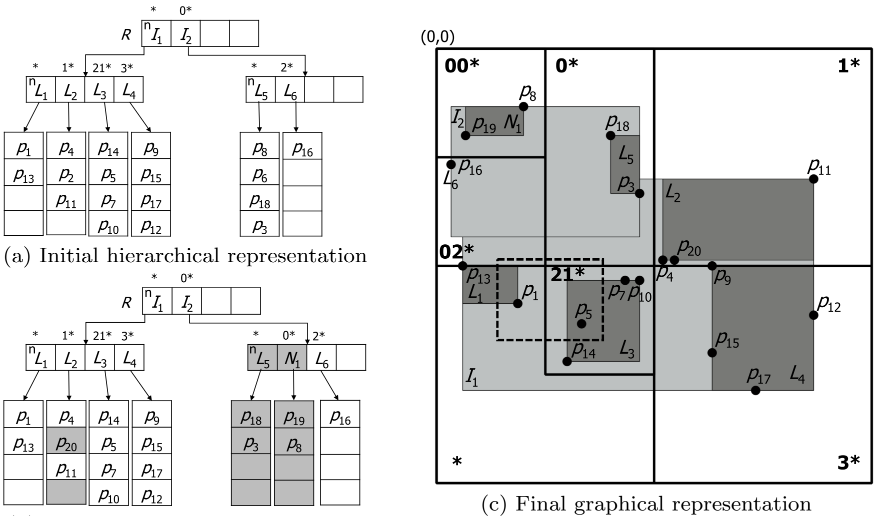
(c) Final graphical representation

Fig. 3 A Hilbert R-tree in hierarchical representation (a) and the Hilbert R-tree resulting after applying a set of modifications on it in hierarchical (b) and graphical (c) representations. The hierarchical representation highlights the performed modifications in gray.

axis (i.e., y) are incremented from top to bottom. Hence, its origin point is the top-leftmost point in the space (as indicated in Figure 4c).

Leaf nodes of the xBR$^+$-tree contain entries in the format (id, p), where p is the point object and id is a pointer to the register of p. These entries are sorted by x-axis coordinates of the points. Internal nodes consist of entries in the following format (id, DBR, $qside$, $shape$). Each entry of an internal node refers to a child node that is pointed by id and represents a sub-quadrant of the original space, minus some smaller descendent sub-quadrants, i.e., ones corresponding to the next entries of the internal node. DBR refers to the data bounding rectangle that minimally encompasses the points stored in such a sub-quadrant. $qside$ stores the side length of the sub-quadrant of the entry. Last, $shape$ is a flag that indicates if the sub-quadrant is either a complete or non-complete square. Each internal node also stores additional metadata in the format (o, s), where o is the origin point of the sub-quadrant and s is the side length. The entries of an internal node are sorted by the Quadtree *addresses* of their sub-quadrants. Each address is formed by directional digits 0, 1, 2, and 3 that respectively symbolize the NW, NE, SW, and SE sub-quadrants of a relative space.

The searching algorithm of the xBR$^+$-tree is similar to the R-tree, starting from the root, it descends the tree examining all nodes that satisfy the search criterion. Inserting a point into an xBR$^+$-tree first involves the choice of a leaf node to accommodate its corresponding entry (id, p). If the chosen node has enough space, it is directly inserted in the correct position. Otherwise, the overflowed node is partitioned into two parts according to a Quadtree-like hierarchical decomposition, and this change is propagated upwards, recursively.



(a) Initial hierarchical representation

(b) Final hierarchical representation

(c) Final graphical representation

Fig. 4 An xBR$^+$-tree in hierarchical representation (a) and the xBR$^+$-tree resulting after applying a set of modifications on it in hierarchical (b) and graphical (c) representations. The hierarchical representation highlights the performed modifications in gray.

3.5 Running Example

In the remainder of this article, we make use of a running example to illustrate how our systematic approach works. This running example consists of the following sequence of index operations applied to the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree shown in Figures 1a, 2a, 3a, and 4a, respectively:

1. Insertion of two points, p_{19} and p_{20};
2. Deletion of two points, p_6 and p_2;
3. Execution of an IRQ that retrieves the points p_1 and p_5;

Figures 1{b, c} to 4{b, c} depict the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree after applying the index operations. In these figures, the query window of the IRQ is represented by a dashed rectangle. In Sections 4 and 5 we discuss how the aforementioned index operations are performed by using our systematic approach.

4 Generalizing and Adapting the eFIND for the Systematic Approach

In this article, we employ the *efficient Framework for spatial INDexing on SSDs* (eFIND) in our systematic approach aiming at porting disk-based spatial index structures to SSDs due to its sophisticated algorithms and data structures (Section 2.4). To this end, we generalize the eFIND's data structures in Section 4.1, and shortly describe the eFIND's main algorithms in Section 4.2.

4.1 Data Structures

eFIND is based on five design goals that exploit the benefits of SSDs. It leverages specific data structures to achieve a design goal. Here, we go further by generalizing some of these data structures to deal with the different spatial index structures, such as those introduced in Section 3.

Write buffer. Its main goal is to avoid random writes to the SSD by storing the modifications of nodes that were not applied to the SSD yet (design goal 1). eFIND leverages a hash table named *Write Buffer Table* to implement the write buffer. In this article, we generalize this data structure to deal with any type of disk-based spatial index as follows. A hash entry stores the modifications of a node and is represented by the tuple $\langle page\_id, (\mathsf{M}, \mathsf{F}, \mathsf{E})\rangle$. *page_id* is the search key of the hash entry and consists of the identifier of a node. Thus, a hash function (e.g., Jenkins hash function [33]) gets the value of *page_id* as input to determine the place (i.e., bucket) in the *Write Buffer Table* where its corresponding value should be stored. The value of a hash entry is formed by $(\mathsf{M}, \mathsf{F}, \mathsf{E})$, where each element is a list of attributes defined as follows.

M consists of the attributes that store the metadata of the node required for processing internal algorithms of the underlying index. Thus, the attributes may vary. Considering the spatial indices detailed in Section 3, M is empty if the underlying index is the R-tree, the R*-tree, and the Hilbert R-tree. If the underlying index is the xBR$^+$-tree, M is an attribute named *header* that consists of the pair (o, s) corresponding to the metadata stored in internal nodes, where o is the origin point and s is the side length of the sub-quadrant of the node, respectively. Since this pair only applies to internal nodes, M assumes NULL if the node is a leaf node (see Figure 5d).

F includes the needed data for using the *flushing policy* in the *flushing operation* (design goal 2). For the flushing policy, the required attributes may vary. Performance tests showed better results when applying a flushing policy based on the number of modifications using the height of the nodes as a weight [15]. That is, this flushing policy requires the attributes h and *mod_count* for storing the height of the node and its quantity of in-memory modifications, respectively. For the flushing algorithm, eFIND requires the attribute *timestamp*, which stores when the last modification of the node was performed. Hence, in this article F consists of the tuple $(h, mod\_count, timestamp)$.

E refers to the *essential attributes to manage the modifications of the node*; it consists of the pair $(status, mod\_tree)$. *status* stores the type of modification made on the node and can be NEW, MOD, or DEL for representing that the node is a newly created node in the buffer, a node stored in the SSD but with modified entries, or a deleted node, respectively. *mod_tree* assumes NULL, if *status* is equal to DEL. Otherwise, it is a red-black tree storing the most recent version of the node's entries. Each element of this red-black tree is a pair (k, e), where k is the search key and corresponds to the unique identifier of the entry and e stores the latest version of the entry, assuming NULL if it is removed from the node. We employ red-black trees for storing the node's entries because of its amortized cost of executing insertions, deletions, and searches. Further, it allows that only the latest version of an entry be stored in the *Write Buffer Table*; thus, the space of the write buffer is better managed with a low cost of retrieving the most recent version of a node (see Section 5). More importantly, the red-black tree maintains a specific order among the node's entries, an essential aspect when dealing with spatial indices that require a special sort property (e.g., the Hilbert R-tree and the xBR$^+$-tree). Hence, the design of the comparison function of the red-black trees should accomplish the sort property of the underlying index. Considering the spatial indices detailed in Section 3, we provide the following base ideas for implementing their corresponding comparison functions as follows:

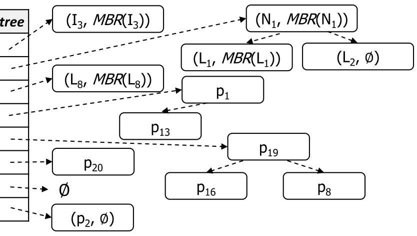
- **The R-tree and the R*-tree.** Their comparison functions implement the ascending order of *id*, which is an element that either gives direct access to the indexed spatial object (if the node is a leaf node) or points to a child node (if the node is an internal node).
- **The Hilbert R-tree.** If the node is a leaf node, its comparison function computes the ascending order of the Hilbert values calculated from r (i.e., the MBR). Otherwise, its comparison function implements the ascending order of *lhv*, which is an element of internal nodes that stores the largest Hilbert value of a child node. In both cases, ties are resolved by sorting the entries by *id*.
- **The xBR$^+$-tree.** If the node is a leaf node, its comparison function implements the ascending order of the x-axis coordinates of the points where ties are resolved by sorting the entries by their y-axis coordinates and then by their *id*. Otherwise, its comparison function implements the ascending order of the directional digits of the entries (using the *qside* and *DBR*), considering the metadata of the internal node (i.e., the pair (o, s)).

It is important to emphasize the role of the comparison function in the cost of performing operations in red-black trees. In our running example, the comparison functions for the R-tree and R*-tree have a constant cost. On the other hand, the Hilbert R-tree and the xBR$^+$-tree require the computation of additional values when evaluating their comparison functions. As a consequence, it may impact the performance evaluations, as discussed in Section 6.

Figure 5 shows the *Write Buffer Tables* for each spatial index of our running example. In this figure, *MBR* is a function for computing the rectangle that encompasses all entries of a node by considering current modifications in the write buffer. For instance, the first line of the hash table in Figure 5a shows that I_1, located in the *height* 2, has the *status* MOD to store the entry $(I_3, \mathit{MBR}(I_3))$. Note that this entry now corresponds to the most recent version of the first entry of I_1 in the eFIND R-tree depicted in Figure 1. This modification occurred in the *timestamp* 10 and is derived from the adjustment of the node I_3 after the reinsertion of the point p_8. The other write buffers (Figures 5b to d) store the needed modifications performed on their corresponding spatial indices to process the index operations of our running example, which are further detailed in Section 5.

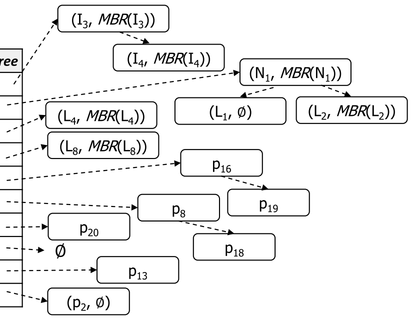
Read buffer. Its main goal is to avoid excessive random reads by caching the nodes stored in the SSD (design goal 3). eFIND leverages another hash table named *Read Buffer Table* to implement the read buffer. It does not employ the same hash table of the write buffer because the read buffer has a different purpose and requires a *read buffer replacement policy* to decide which node should be replaced when the *Read Buffer Table* is full. This buffer is very similar to the classical buffer managers employed by database management systems [22] and is extended to deal with the specific constraints of the underlying index. In this article, we generalize

page_id	h	mod_count	timestamp	status	mod_tree
I_1	2	2	10	MOD	
I_3	1	4	9	MOD	
I_6	1	1	12	MOD	
N_1	0	3	1	NEW	
L_1	0	4	8	NEW	
L_6	0	1	5	MOD	
L_2	0	1	6	DEL	
L_8	0	1	11	MOD	



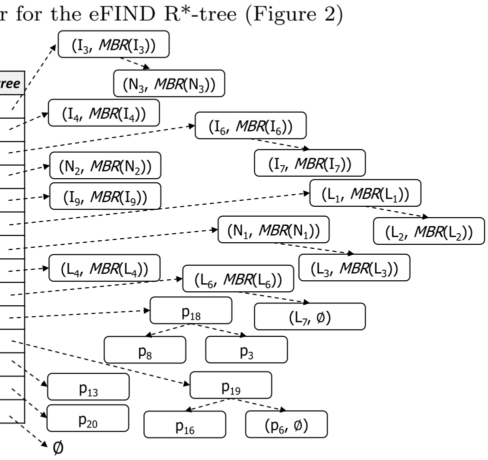
(a) The write buffer for the eFIND R-tree (Figure 1)

page_id	h	mod_count	timestamp	status	mod_tree
I_1	2	2	10	MOD	
I_3	1	3	6	MOD	
I_4	1	1	9	MOD	
I_6	1	1	12	MOD	
N_1	0	3	1	NEW	
L_2	0	4	2	NEW	
L_6	0	1	4	MOD	
L_1	0	1	5	DEL	
L_4	0	1	8	MOD	
L_8	0	1	11	MOD	



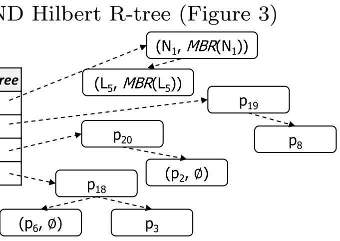
(b) The write buffer for the eFIND R*-tree (Figure 2)

page_id	h	mod_count	timestamp	status	mod_tree
I_1	3	2	10	MOD	
I_2	3	1	19	MOD	
I_3	2	3	15	MOD	
N_3	2	2	9	NEW	
I_4	2	1	18	MOD	
I_6	1	6	14	NEW	
I_7	1	4	7	NEW	
N_2	1	2	5	NEW	
I_9	1	2	17	MOD	
L_1	0	5	2	NEW	
L_2	0	6	13	NEW	
N_1	0	2	1	NEW	
L_6	0	1	11	MOD	
L_7	0	1	16	DEL	



(c) The write buffer for the eFIND Hilbert R-tree (Figure 3)

page_id	header	h	mod_count	timestamp	status	mod_tree
I_2	(0,0)-75	1	2	3	MOD	
N_1	NULL	0	3	1	NEW	
L_2	NULL	0	2	6	MOD	
L_5	NULL	0	6	5	NEW	



(d) The write buffer for the eFIND xBR$^+$-tree (Figure 4)

Fig. 5 Write buffers for storing the modifications of the disk-based spatial indices the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree transforming them to the eFIND R-tree (a), the eFIND R*-tree (b), the eFIND Hilbert R-tree (c), and the eFIND xBR$^+$-tree (d).

the *Read Buffer Table* to deal with any type of disk-based spatial index. A hash entry corresponds to a node stored in the *Read Buffer Table* and consists of a tuple $\langle page\_id, (\mathsf{M}, \mathsf{R}, entries)\rangle$. *page_id* is the search key of the hash entry and stores the identifier of the node. The hash value has the following format $(\mathsf{M}, \mathsf{R}, entries)$, where each element is defined as follows. M consists of the same attributes as M of the definition of a hash entry in the *Write Buffer Table*. That is, it stores the metadata of the node.

R includes the needed data for executing the *read buffer replacement policy*. For instance, the height of the node stored in an attribute named h for implementing the LRU replacement policy prioritizing the nodes near to the root of the tree [11]. If the replacement policy does not require any additional data, then R is empty, optimizing the space of the *Read Buffer Table*. This is the case when adopting the simplified 2 Queues (S2Q) [36] replacement policy, which showed good performance results because it mitigates the problem of loading nodes from the SSD to the main memory [15]. Hence, R is empty in our running example.

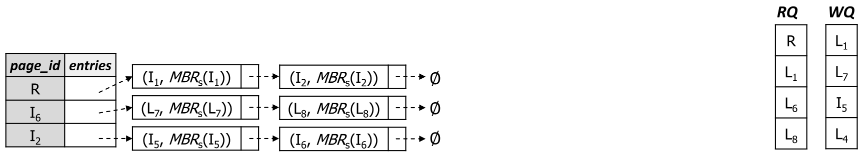
entries refers to a list storing the node's entries. Since the *Read Buffer Table* caches nodes stores in the SSD, this list does not consider the modifications stored in the write buffer. An element of the *entries* has the same format as an entry of the node. The order of the elements of this list corresponds to the order in which they are stored in the SSD. This means that it respects the properties and structural constraints of the underlying index.

Figure 6 shows the *Read Buffer Tables* for each spatial index of our running example. In this figure, MBR_S is a function for computing the rectangle that encompasses all entries of a node by considering entries stored in the SSD only. Thus, it does not consider modifications stored in the write buffer. For instance, the read buffer for the eFIND R*-tree (Figure 6b) contains the cached version of the nodes R, I_1, and I_3, corresponding to the same entries shown in Figure 2a.

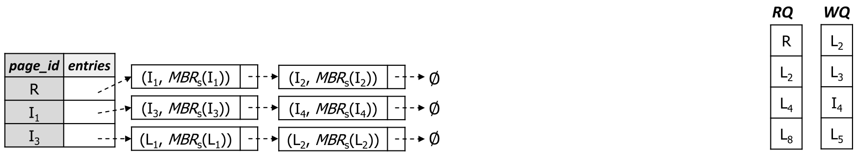
Temporal control. Two queues named RQ and WQ are responsible for implementing the temporal control of eFIND (design goal 4). Each queue is a First-In-First-Out (FIFO) data structure. RQ stores identifiers of the nodes read from the SSD, while WQ keeps the identifiers of the last nodes written to the SSD. Figure 6 shows the queues of the temporal control for each spatial index of our running example. For instance, last read nodes are R, I_3, I_6, and I_8, and the last flushed nodes are L_1, L_8, I_9, and I_5 for the eFIND Hilbert R-tree.

Log file. eFIND sequentially writes to a log file the modifications that are performed on the underlying index before storing it in the *Write Buffer Table* to ensure *data durability* (design goal 5). Since we generalize the *Write Buffer Table*, we also generalize the log file as follows. The log-structured approach of eFIND is based on the write-ahead logging employed by database systems and indexing structures, such as surveyed in [29]. The main goal here is to store only the needed data to recover the *Write Buffer Table*. In the following, we describe the compatibility between a log entry and a hash entry of the write buffer. A log entry consists of a tuple $\langle page\_id, (\mathsf{M}, \mathsf{P}, \mathsf{T})\rangle$, where *page_id* stores the identifier of the node and each element in $(\mathsf{M}, \mathsf{P}, \mathsf{T})$ respectively corresponds to an element of the definition of a hash entry of the *Write Buffer Table*.

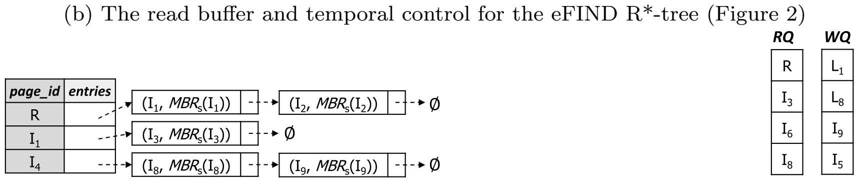
M is the same M from that used in each hash entry of the *Write Buffer Table*. P is a subset of F. In this article, it consists of a single attribute named h that stores the height of the node. The other attributes of F (i.e., *timestamp* and *mod_count*)



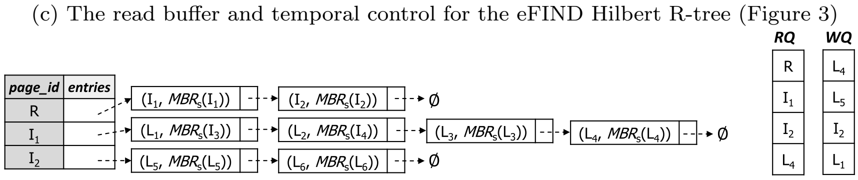
(a) The read buffer and temporal control for the eFIND R-tree (Figure 1)



(b) The read buffer and temporal control for the eFIND R*-tree (Figure 2)



(c) The read buffer and temporal control for the eFIND Hilbert R-tree (Figure 3)



(d) The read buffer and temporal control for the eFIND xBR$^+$-tree (Figure 4)

Fig. 6 Read buffers and queues of the temporal control for the eFIND R-tree (a), the eFIND R*-tree (b), the eFIND Hilbert R-tree (c), and the eFIND xBR$^+$-tree (d).

are not stored in the log file because they are calculated in the main memory every time that a modification is stored in the *Write Buffer Table* (e.g., see Section 5.1).

T is a subset of E and consists of a pair ($type\_mod, result$), where *type_mod* is similar to the *status*, assuming MOD if the entry is added to or removed from the node and NEW if the node is a newly created node, and *result* is equivalent to an element of the red-black tree of the node in the *mod_tree*. That is, the pair (k, e). Because only one element of *mod_tree* is stored by log entry, several log entries may be needed to store all elements of the red-black tree. Nodes flushed to the SSD are also appended to the log file. This strategy allows the compaction of the log, that is, the exclusion of already flushed modification from the log file, reducing its size (Section 4.2). In this case, *status* assumes the value FLUSH, *result* stores the list of flushed nodes, and NULL is assigned to the remaining attributes.

Figure 7 shows the log file for each spatial index of our running example. In this figure, the first column (*log#*) refers to the sequence of the processed modification. Thus, we can follow the sequence of modifications performed to process the index operations of our running example. For instance, the first modification of the eFIND R-tree is the creation of the node N_1 (*timestamp* equal to 1 in Figure 5a) that contains the points p_1 and p_{13}. This sequence is stored in the first three log entries in Figure 7a. Section 5 further details how the modifications are appended in the log files of each spatial index.

log#	page_id	h	type_mod	result
1	N_1	0	*NEW*	-
2	N_1	0	*MOD*	p_1
3	N_1	0	*MOD*	p_{13}
4	L_1	0	*DEL*	-
5	L_1	0	*NEW*	-
6	L_1	0	*MOD*	p_{16}
7	L_1	0	*MOD*	p_{19}
8	I_3	1	*MOD*	$(N_1, \mathit{MBR}(N_1))$
9	I_3	1	*MOD*	$(L_1, \mathit{MBR}(L_1))$
10	L_6	0	*MOD*	p_{20}
11	L_2	0	*DEL*	-
12	I_3	1	*MOD*	$(L_2, \emptyset)$
13	I_1	2	*MOD*	$(I_3, \mathit{MBR}(I_3))$
14	L_1	0	*MOD*	p_8
15	I_3	1	*MOD*	$(L_1, \mathit{MBR}(L_1))$
16	I_1	2	*MOD*	$(I_3, \mathit{MBR}(I_3))$
17	L_8	0	*MOD*	$(p_2, \emptyset)$
18	I_6	1	*MOD*	$(L_8, \mathit{MBR}(L_8))$

(a) The log file for the eFIND R-tree (Figure 1)

log#	page_id	h	type_mod	result
1	N_1	0	*NEW*	-
2	N_1	0	*MOD*	p_{16}
3	N_1	0	*MOD*	p_{19}
4	L_2	0	*DEL*	-
5	L_2	0	*NEW*	-
6	L_2	0	*MOD*	p_8
7	L_2	0	*MOD*	p_{18}
8	I_3	1	*MOD*	$(N_1, \mathit{MBR}(N_1))$
9	I_3	1	*MOD*	$(L_2, \mathit{MBR}(L_2))$
10	L_6	0	*MOD*	p_{20}
11	L_1	0	*DEL*	-
12	I_3	1	*MOD*	$(L_1, \emptyset)$
13	I_1	2	*MOD*	$(I_3, \mathit{MBR}(I_3))$
14	L_4	0	*MOD*	p_{13}
15	I_4	1	*MOD*	$(L_4, \mathit{MBR}(L_4))$
16	I_1	2	*MOD*	$(I_4, \mathit{MBR}(I_4))$
17	L_8	0	*MOD*	$(p_2, \emptyset)$
18	I_6	1	*MOD*	$(L_8, \mathit{MBR}(L_8))$

(b) The log file for the eFIND R*-tree (Figure 2)

log#	page_id	h	type_mod	result
1	N_1	0	*NEW*	-
2	N_1	0	*MOD*	p_{13}
3	L_1	0	*DEL*	-
4	L_1	0	*NEW*	-
5	L_1	0	*MOD*	p_8
6	L_1	0	*MOD*	p_{18}
7	L_1	0	*MOD*	p_3
8	L_2	0	*DEL*	-
9	L_2	0	*NEW*	-
10	L_2	0	*MOD*	p_{16}
11	L_2	0	*MOD*	p_{19}
12	L_2	0	*MOD*	p_6
13	I_6	1	*MOD*	$(L_2, \mathit{MBR}(L_2))$
14	N_2	1	*NEW*	-
15	N_2	1	*MOD*	$(L_4, \mathit{MBR}(L_4))$
16	I_6	1	*DEL*	-
17	I_6	1	*NEW*	-
18	I_6	1	*MOD*	$(L_1, \mathit{MBR}(L_1))$
19	I_6	1	*MOD*	$(L_2, \mathit{MBR}(L_2))$
20	I_7	1	*DEL*	-
21	I_7	1	*NEW*	-
22	I_7	1	*MOD*	$(N_1, \mathit{MBR}(N_1))$
23	I_7	1	*MOD*	$(L_3, \mathit{MBR}(L_3))$
24	I_3	2	*MOD*	$(I_6, \mathit{MBR}(I_6))$
25	I_3	2	*MOD*	$(I_7, \mathit{MBR}(I_7))$
26	N_3	2	*NEW*	-
27	N_3	2	*MOD*	$(N_2, \mathit{MBR}(N_2))$
28	I_1	3	*MOD*	$(N_3, \mathit{MBR}(N_3))$
29	I_1	3	*MOD*	$(I_3, \mathit{MBR}(I_3))$
30	L_6	0	*MOD*	p_{20}
31	I_9	1	*MOD*	$(L_6, \mathit{MBR}(L_6))$
32	L_2	0	*MOD*	$(p_6, \emptyset)$
33	I_6	1	*MOD*	$(L_2, \mathit{MBR}(L_2))$
34	I_3	2	*MOD*	$(I_6, \mathit{MBR}(I_6))$
35	L_7	0	*DEL*	-
36	I_9	1	*MOD*	$(L_7, \emptyset)$
37	I_4	2	*MOD*	$(I_9, \mathit{MBR}(I_9))$
38	I_2	3	*MOD*	$(I_4, \mathit{MBR}(I_4))$

(c) The log file for the eFIND Hilbert R-tree (Figure 3)

log#	page_id	header	h	type_mod	result
1	N_1	-	0	*NEW*	-
2	N_1	-	0	*MOD*	p_{19}
3	N_1	-	0	*MOD*	p_8
4	L_5	-	0	*DEL*	-
5	L_5	-	0	*NEW*	-
6	L_5	-	0	*MOD*	p_{18}
7	L_5	-	0	*MOD*	p_3
8	L_5	-	0	*MOD*	p_6
9	I_2	(0,0)-75	1	*MOD*	$(N_1, \mathit{MBR}(N_1))$
10	I_2	(0,0)-75	1	*MOD*	$(L_5, \mathit{MBR}(L_5))$
11	L_2	-	0	*MOD*	p_{20}
12	L_5	-	0	*MOD*	$(p_6, \emptyset)$
13	L_2	-	0	*MOD*	$(p_2, \emptyset)$

(d) The log file for the eFIND xBR$^+$-tree (Figure 4)

Fig. 7 Log files for guaranteeing data durability for the eFIND R-tree (a), the eFIND R*-tree (b), the eFIND Hilbert R-tree (c), and the eFIND xBR$^+$-tree (d).

4.2 General Algorithms

eFIND provides algorithms to execute the following operations: (i) *maintenance operation*, which is responsible for reorganizing the index whenever modifications are made on the underlying spatial dataset (i.e., insertions, deletions, and updates); (ii) *search operation*, which is responsible for executing spatial queries; (iii) *flushing operation*, which picks a set of modifications stored in the write buffer to be written to the SSD according to a flushing policy; and (iii) *restart operation*, which rebuilds the write buffer after a fatal problem and compacts the log file. To employ eFIND in our systematic approach, we generalize the maintenance and search operations considering our characterization of node handling (see Section 5). We did not change the flushing and restart operations of eFIND, which are detailed in [15] and shortly described as follows.

The flushing operation is responsible for sequentially writing some modified nodes to the SSD. The modified nodes are picked after applying a *flushing policy* to the *flushing units* created from a list of the oldest modified nodes stored in the *Write Buffer Table* that satisfy the criteria of the temporal control of writes such as a sequential or semi-sequential pattern of previous writes made on the SSD. While a flushing unit groups a set of sequential modified nodes, a flushing policy implements the criteria to choose a flushing unit to be written to the SSD. Experiments show the best results when applying a flushing policy that uses the height of the modified node as a weight on its number of modifications [15]. After writing the picked flushing unit, this operation is also registered as a log entry for guaranteeing data durability. Further, frequently accessed nodes are possibly pre-cached in the *Read Buffer Table* according to the temporal control of reads.

The restart operation reconstructs the *Write Buffer Table* after a system crash, fatal error, or failure power. This means that eFIND guarantees *data durability.* This is performed by recovering all the modifications that were not effectively applied to the index stored in the SSD. For this, eFIND reads the log file in reverse order since the modifications and the flushed nodes are written to the log as append-only operations. During this traversal, the modifications of flushed nodes can be ignored since they were already written to the SSD. The idea of removing the modifications of flushed nodes from the log is also employed to compact it. This compaction requires some additional processing for handling maintenance operations and different factors like the write buffer size, log size, and node size affect its performance (as discussed in [59] and [15]).

5 Porting Disk-based Spatial Indices to SSDs

In this section, we detail our systematic approach by focusing on the following operations: (i) insert (Section 5.1), (ii) delete (Section 5.2), and (iii) search (Section 5.3). For each operation, we provide its generic algorithm and characterize how the nodes are modified and accessed when implementing the operation. Then, we propose a set of algorithms, including their complexity analysis, that manage the generalized eFIND's data structures in order to deal with this characterization. To illustrate how our algorithms work, we also provide examples of executions based on our running example (Section 3.5).

5.1 Insert Operations

General algorithm. Considering a spatial index SI being ported by eFIND (i.e., R-tree, R*-tree, Hilbert R-tree, and xBR$^+$-tree), Algorithm 1 inserts a new entry E into SI as follows. First, a leaf node L is selected according to the particular properties of SI (line 1). For instance, the R-tree chooses a leaf node by prioritizing the path of the tree that minimizes the coverage area of the nodes. This step involves the retrieval of nodes. For this, the underlying index has to employ Algorithm 9, which is discussed in Section 5.3. Then, the entry E is inserted into L, leading to two possible cases: either (i) a direct insertion, or (ii) treatment of an overflow. In both cases, a pair $P = (sn, n)$, where sn is a set of nodes and n is a node, is formed and later used to adjust the tree after the insertion of E (line 2). The first case is if L has enough space to accommodate the entry E (lines 3 to 6). Hence, the entry E is inserted into L according to the structural constraints of SI (line 4), this insertion is registered by eFIND (line 5), and P assumes the pair $(\{L\}, \text{NULL})$ where its first element is a set containing L with the new entry and its second element is NULL since there are no other modified nodes.

The second case is if L has its maximum capacity reached (lines 7 to 9); thus, the overflowed node has to be treated by the underlying index SI (line 8). Some indices attempt to apply a redistribution to the entries of L and s sibling nodes instead of executing a split operation. This is the case for the Hilbert R-tree. Thus, the first element of P is the set of modified nodes (i.e., a set H containing L and its s sibling nodes) and the second element of P is NULL. If the redistribution is not possible and for other indices (e.g., the R-tree, the R*-tree, and the xBR$^+$-tree), a split operation is directly performed, leading to the creation of a new node. Then, the entries are distributed among the available nodes. In this case, the first element of P is the set of modified nodes (i.e., H is L and its s sibling nodes for the Hilbert R-tree, and only L for the remaining indices) and the second element of P is the newly created node. After processing the overflow, the pair P is saved by eFIND (line 9).

After inserting the entry E, the tree is adjusted in order to preserve its structural constraints and particular properties (line 10). For this, the tree is traversed from the leaf node L to the root node, adjusting the needed entries in this path. It may include the propagation of split operations because of overflow handling. eFIND is called to register the modifications resulted from these adjustments (Algorithm 2) and to save every pair resulted from the propagation of split operations (Algorithm 3). Finally, Algorithm 1 checks whether the propagation reached the root node (line 11). In this case, a new root node is created (line 12) and saved by eFIND (line 13).

Handling nodes with eFIND. The computation of Algorithm 1 can invoke five specialized algorithms of eFIND to manipulate nodes of the underlying index. They are called by the following characterized cases: (i) the retrieval of nodes (line 1), (ii) the direct insertion of the new entry E into a chosen leaf node (line 5), (iii) the treatment of overflowed nodes (lines 9 and 10), (iv) the adjustment of entries (line 10), and (v) the creation of a new root node (line 13). In this section, we discuss the algorithms responsible for executing the last four characterized cases, whereas the first characterized case is discussed in the spatial query processing (Section 5.3).

Algorithm 1: Inserting an entry into a spatial index

Input: *SI* as the underlying index, E as the entry being inserted

choose a leaf node L to accommodate the entry E (**nodes are read using Algorithm 9**);
let P be a pair (sn, n), where sn is a set of nodes and n is a node;
if *L is not full* **then**
  insert E into L;
  save the direct insertion by calling Algorithm 2;
  let P become the pair $(\{L\}, \text{NULL})$;
else
  let P become the pair (H, NN) resulted from the execution of the overflow handling of *SI* on L after inserting E;
  save the node overflow by calling Algorithm 3;
traverse the tree from leaf level towards the root by adjusting the entries pointing to modified nodes (**saving them using Algorithm 2**) and by propagating splits (**saving them using Algorithm 3**) if any;
if *the root was split* **then**
  create a new root node *NR* whose the entries refer to the old root R and the newly created node N;
  save the new root node by calling Algorithm 4;

Algorithm 2: Saving the modification of a node in the *Write Buffer Table* of eFIND.

Input: O as the operation type, E as the entry being modified or inserted, and N as the node accommodating the entry E

let E' be an entry;
if *O is a delete operation* **then**
  let E' become NULL;
else
  let E' point to E;
append the new log entry $\langle N_{id}, (metadata(N), height(N), (\text{MOD}, (key(E), E')))\rangle$ into the log file;
let *WBEntry* be the hash entry of N in the *Write Buffer Table*;
if *WBEntry is not NULL* **then**
  if *the mod_tree of WBEntry contains an element with key equal to key(E)* **then**
    replace it by the element $(key(E), E')$;
  else
    insert the element $(key(E), E')$ into *mod_tree* of *WBEntry*;
  update the value of *timestamp* of *WBEntry* to *now()*;
  increase the value of *mod_count* of *WBEntry* by 1;
else
  set *WBEntry* to the hash entry $\langle N_{id}, (metadata(N), (now(), height(N), 1), (\text{MOD}, emptyRBTree()))\rangle$;
  store *WBEntry* in the *Write Buffer Table*;
  insert the element $(key(E), E')$ into *mod_tree* of *WBEntry*;
if *Write Buffer Table is full* **then**
  execute a flushing operation (as detailed in [15]);

Algorithm 2 shows how the extended eFIND processes a node modification. Its inputs are the type of modification to be handled (O), the entry (E) being manipulated, and its node (N). This algorithm is employed to execute the cases (ii) and (iv). For the case (ii), the algorithm is handling an insert operation (O)

Algorithm 3: Handling a node overflow

Input: P as a pair (R, NN), where R is a set of modified nodes and NN is a possibly newly created node

1 **if** NN *is not NULL* **then**
2   **save** ***NN*** **by calling Algorithm 4**;
3 **foreach** node ND **in** R **do**
4   **delete** ***ND*** **from the** ***Write Buffer Table*** **by calling Algorithm 7;**
5   **save** ***ND*** **by calling Algorithm 4**;

of an entry E into a node N; for the case (iv), the algorithm is dealing with an adjustment operation (O) of an entry E that is contained in a node N. First, an auxiliary entry (line 1) is used to adequately process the operation, such as a delete operation (Section 5.2). Here, this auxiliary entry points to the input entry (line 5). Next, the modification is registered in the log file in order to guarantee data durability (line 6). This is a main step of the algorithm because it permits to recover the modification if any fatal error occurs before its accomodation in the *Write Buffer Table*. Then, two main cases are alternately possible (lines 8 to 18). The first case is if the node has a corresponding hash entry in the *Write Buffer Table* (lines 8 to 14). Thus, the entry is either replaced (line 10) or inserted (line 12) in its *mod_tree*. This guarantees that only its most recent version is stored in the write buffer. In the sequence, other values of the hash entry are updated, such as the moment of the operation (line 13) and the increment of the number of modifications (line 14). The second case is if the node is receiving its first modification (lines 16 to 18). Thus, the algorithm creates a new hash entry (line 16) to be stored in the *Write Buffer Table* (line 17) and stores the modified entry as the first element of its *mod_tree* (line 18). Finally, the algorithm checks whether a flushing operation has to be executed (lines 19 and 20). This flushing algorithm is the same as presented and discussed in [15] (see Section 4.2).

Algorithm 3 depicts how eFIND saves the pair P resulted from the overflow handling of the underlying index. This algorithm is employed to execute the case (iii). In principle, if there is exists a newly created node (line 1), this node is saved in the *Write Buffer Table* by using Algorithm 4. Next, for each node contained in R of P (line 3), Algorithm 3 deletes its previous version (line 4) and then stores this node as a newly created node in the *Write Buffer Table* (line 5). This strategy redefines the hash entries in the write buffer that are related to nodes affected by a redistribution after handling an overflow. Thus, we store the most recent version of the node instead of expending time to save their particular differences. As a result, it improves the management of the write buffer. This also contributes to simplifying the retrieval of nodes by avoiding the execution of merging operations (see Section 6.3) since the node can be completely modified after handling an overflow.

Algorithm 4 depicts how eFIND stores a newly created node in its *Write Buffer Table*. This algorithm is employed to execute the case (v) and to help the execution of Algorithm 3. First, the newly created node is registered as a new log entry in the log file for data durability purposes (line 1). Note that only the intention of creating a node is registered and not its entries yet. Then, the algorithm uses an auxiliary variable that corresponds to the hash entry of the newly created node

Algorithm 4: Storing a newly created node in the *Write Buffer Table*

Input: N as the newly created node

1 append the new log entry $\langle N_{id}, (metadata(N), height(N), (\text{NEW}, \text{NULL}))\rangle$ into the log file;
2 let *WBEntry* be the hash entry of N in the *Write Buffer Table*;
3 **if** *WBEntry is not NULL* **then**
4   **if** *the status of WBEntry is equal to DEL* **then**
5   set the *status* and *mod_tree* of *WBEntry* to NEW and *emptyRBTree*(), respectively;
6 **else**
7   let *WBEntry* become the hash entry $\langle N_{id}, (metadata(N), (now(), height(N), 1), (\text{NEW}, emptyRBTree()))\rangle$
8   store *WBEntry* in the *Write Buffer Table*;
9 **if** N *is not empty* **then**
10   **foreach** entry E **in** N **do**
11   append the new log entry $\langle N_{id}, (metadata(N), height(N), (\text{MOD}, (key(E), E)))\rangle$ into the log file;
12   insert the element $(key(E), E)$ into *mod_tree* of *WBEntry*;
13   increase the value of *mod_count* of *WBEntry* by 1;
14   update the value of *timestamp* of *WBEntry* to *now()*;
15 **if** *Write Buffer Table is full* **then**
16   execute a flushing operation (as detailed in [15]);

in the write buffer (line 2). By using this variable, two main cases are alternately possible (lines 3 to 8). The first case is if the node has a corresponding hash entry in the *Write Buffer Table* (lines 4 and 5). The entry is effectively stored in the write buffer if it was previously deleted. The second case refers to the non-existence of the hash entry of the newly created node in the write buffer; thus, the algorithm sets the values of the new hash entry (line 7) and stores it in the *Write Buffer Table* (line 8). Afterward, the algorithm adds each entry of the newly created node in the created hash entry of the write buffer if it is not empty (lines 9 to 14). The sequence of operations in this loop is to firstly append a corresponding log entry to guarantee data durability (line 11), to insert the entry in the red-black tree of the hash entry (line 12), and then to increase the number of modifications (line 13). After inserting all entries, the timestamp of the hash entry is also updated (line 14). Finally, the algorithm executes the flushing operation of [15] if the write buffer is full (lines 15 and 16).

Complexity Analysis. Our goal is not to analyze the complexity of algorithms belonging to the underlying spatial index since it goes beyond the scope of this article (see [51; 4] for complexity analysis of R-trees). In this sense, we analyze the complexity of Algorithms 2 to 4 as follows. The time complexity of Algorithm 2 can be determined by $\mathcal{C}_{alg2} = \mathcal{W}_s + \mathcal{H} + \mathcal{O}(\log n)$, where $\mathcal{W}_s$ is the average cost of one sequential write to the SSD in order to log the modification, $\mathcal{H}$ refers to the cost of accessing an element from the hash table that implements the write buffer (i.e., which is usually $\mathcal{O}(1)$), and $\mathcal{O}(\log m)$ is the average cost of updating an element of the red-black tree with m elements. Note that red-black trees have an amortized update cost, as discussed in [46], which is particularly useful for implementing the write buffer. In addition, the time complexity of Algorithm 2 can also include the cost of a flushing operation, as detailed in [15].

The time complexity of Algorithm 3, in the worst case, is determined by $\mathcal{C}_{alg3} = \mathcal{C}_{alg4} + k\mathcal{C}_{alg7} + k\mathcal{C}_{alg4}$, where k is the number of nodes in R. Algorithm 4 has a time complexity similar to Algorithm 2; the difference is that there is the cost of logging and inserting each entry of the newly created node. Hence, $\mathcal{C}_{alg4}$ is given by $\mathcal{W}_s + \mathcal{H} + e\mathcal{W}_s + \mathcal{O}(e \log n)$, where e is the number of entries of the newly created node. The time complexity of Algorithm 7 is presented in Section 5.2.

With respect to the space complexity, Algorithms 2 and 4 has the space complexity of $\mathcal{O}(2n)$, where n is the total number of modified entries. This is due to the data durability, which requires that a copy of each modification be stored in the log file. The space complexity of the write buffer is $\mathcal{O}(a)$, where a is the number of elements (i.e., nodes) in the buffer since it is implemented as a hash table. A red-black tree has a space complexity of $\mathcal{O}(b)$ for storing b (modified) entries of a particular node. It does not require extra space since its keys are based on the identifier of the entry (i.e., a value greater than zero). Hence, the color information can be stored by using the sign bit of the keys. The space complexity of Algorithm 3 is constant.

Examples of Execution. Our running example inserts the points p_{19} and p_{20} into each spatial index depicted in Figures 1a to 4a. After applying Algorithm 1, a set of modifications is appended to the log file and stored in the write buffer of each spatial index ported to the SSD. Instead of repeating the explanation of the algorithm by showing its execution line by line, we highlight the sequence of the modifications performed in the ported spatial indices after each insertion operation as follows:

- **The R-tree (Figure 1a).** A split operation on the node L_1 is performed to insert the point p_{19}, creating the new node N_1. After this operation, the newly created node N_1 contains the points p_1 and p_{13} (*log#* 1 to 3 in Figure 7a and the fourth line in Figure 5a), and after the recreation of the node L_1, it contains the points p_{16} and p_{19} (*log#* 4 to 7 in Figure 7a and the fifth line in Figure 5a). Next, two adjustments are made in the node I_3 (*log#* 8 and 9 in Figure 7a and the second line in Figure 5a). First, a new entry that points to the node N_1 is created and inserted into the node I_3. Second, the entry pointing to the node L_1 has its MBR adjusted. The point p_{20} is directly inserted into the node L_6 (*log#* 10 in Figure 7a and the sixth line in Figure 5a).
- **The R*-tree (Figure 2a).** It executes a split operation to accommodate the point p_{19}, creating the new node N_1 that stores the points p_{16} and p_{19} (*log#* 1 to 3 in Figure 7b and the fifth line in Figure 5b). Further, the node L_2 is recreated to store the points p_8 and p_{18} (*log#* 4 to 7 in Figure 7b and the sixth line in Figure 5b). Then, similar to the R-tree, two adjustments are made in the node I_3 (*log#* 8 and 9 in Figure 7b and the second line in Figure 5b). The point p_{20} is directly inserted into the node L_6 (*log#* 10 in Figure 7b and the seventh line in Figure 5b).
- **The Hilbert R-tree (Figure 3a).** It executes two 2-to-3 split operations to accommodate the point p_{19}. First, it creates the new node N_1 containing the point p_{13} (*log#* 1 and 2 in Figure 7c and the twelfth line in Figure 5c), and then redistributes the points p_8, p_{18} and p_3 to the node L_1 (*log#* 3 to 7 in Figure 7c and the tenth line in Figure 5c) and the points p_{16}, p_{19}, and p_6 to the node L_2 (*log#* 8 to 12 in Figure 7c and the eleventh line in Figure 5c), according to their Hilbert values. Next, it adjusts the MBR of the entry pointing to the node

L_2 (*log#* 13 in Figure 7c and the sixth line in Figure 5c). The second 2-to-3 split occurs when inserting the node N_1 into the node I_6. Thus, it creates the new node N_2 containing the entry pointing to L_4 (*log#* 14 and 15 in Figure 7c and the eighth line in Figure 5c), and then redistributes the entries among the nodes I_6 and I_7 (*log#* 16 to 25 in Figure 7c and the sixth and seventh lines in Figure 5c), according to their largest Hilbert values. To accommodate the new node N_2, another new node is created, named N_3 (*log#* 26 and 27 in Figure 7c and the fourth line in Figure 5c). Then, two entries of the node I_1 are adjusted accordingly (*log#* 28 and 29 in Figure 7c and the first line in Figure 5c), concluding the insertion of the point p_{19}. The insertion of the point p_{20} requires the creation of a new corresponding entry in the node L_6 (*log#* 30 in Figure 7c and the thirteenth line in Figure 5c). As a consequence, its MBR is adjusted in the parent entry's node I_9 (*log#* 31 in Figure 7c and the ninth line in Figure 5c).
– **The xBR$^+$-tree (Figure 4a).** To insert the point p_{19}, the new sub-quadrant 00* that also accommodates the point p_8 is created (*log#* 1 to 3 in Figure 7d and the second line in Figure 5d). This sub-quadrant is derived from a split operation on the node L_5, which then stores the points p_{18}, p_3, and p_6 (*log#* 4 to 8 in Figure 7d and the fourth line in Figure 5d). The node I_2 is modified to accommodate the newly created node and to store the adjusted DBR of the node L_5 (*log#* 9 and 10 in Figure 7d and the first line in Figure 5d). The point p_{20} is directly inserted into the node L_2 (*log#* 11 in Figure 7d and the third line in Figure 5d).

Note that Figures 1b to 4b show the resulting hierarchical representation after also removing two points. Thus, the aforementioned modifications represent an intermediary result of the running example.

5.2 Delete Operations

General algorithm. Considering a spatial index *SI* being ported by eFIND (i.e., an R-tree, R*-tree, a Hilbert R-tree, and an xBR$^+$-tree), Algorithm 5 deletes an entry E from *SI* as follows. First, an exact match query is executed to retrieve the leaf node L containing the entry E (line 1). To this end, the underlying index has to employ the general search algorithm (Algorithm 9), which is discussed in Section 5.3. Next, the entry E is deleted from L, leading to two possible alternately cases: either (i) a direct deletion, or (ii) treatment of an underflow. In both cases, a pair $P = (sn, d)$ is defined, where *sn* is a set of nodes with adjustments and d is a node to be deleted from *SI* (line 2). This pair is also used to propagate further adjustments in the tree after the deletion of E. The first case is if the minimum capacity of L is not affected after removing the entry E (lines 3 to 6). Hence, the entry E is removed from L according to the structural constraints of *SI* (line 4), the deletion is registered by eFIND (line 5), and P assumes the pair $(\{L\}, \text{NULL})$ where its first element is a set containing L after the deletion and its second element is NULL since there are no other modified nodes.

The second case is if an underflow occurs in L after removing the entry E (lines 7 to 9); this case is then treated by the underlying index *SI* (line 8). Considering the indices of this article (Section 3), we shortly describe how they handle an

Algorithm 5: Deleting an entry from a spatial index

Input: *SI* as the underlying index, E as the entry being deleted
1 pick the leaf node L containing the entry E (**nodes are read using Algorithm 9**);
2 let P be a pair (sn, d), where sn is a set of nodes and d is a node;
3 **if** *L's size is greater than the minimum capacity minus one* **then**
4   delete E from L;
5   **save the direct deletion by calling Algorithm 2**;
6   let P become the pair $(\{L\}, \mathrm{NULL})$;
7 **else**
8   let P become the pair (H, D) resulted from the execution of the underflow handling of *SI* on L after deleting E;
9   **save the node underflow by calling Algorithm 6**;
10 traverse the tree from leaf level towards the root by adjusting the entries pointing to modified nodes (**saving them using Algorithm 2**) and by propagating deletion of entries, possibly causing underflow, (**saving them using Algorithm 6**) if any;
11 **if** *the root contains one entry only* **then**
12   let N be the first entry of the root node;
13   let the new root node be N;
14   **delete the old root node by calling Algorithm 7**;
15 execute the additional treatment of *SI*, if any;

underflow. The R-tree and the R*-tree directly delete L and save its entries in a queue stored in the main memory. Then, these entries are reinserted in the tree by using the corresponding insertion algorithm (Section 5.1). The Hilbert R-tree attempts to apply a redistribution to the entries of L and $s - 1$ sibling nodes instead of deleting L. If the redistribution is not possible, this index deletes L and redistributes the remaining entries of L among its $s - 1$ sibling nodes. The xBR$^+$-tree deletes L if there exists one sibling node representing the ancestor or descendant of L with available space, it inserts the remaining entries of L in this sibling node. In general, these indices can delete L and possibly modify other sibling nodes. Because of this behavior, these modifications are stored as the pair P that is saved by eFIND (line 9).

After deleting the entry E, the tree is adjusted in order to preserve its structural constraints and particular properties (line 10). For this, the tree is traversed from the leaf level to the root node, adjusting the needed entries in this path (e.g., the minimum boundary rectangles). It may include the propagation of deletions because of underflow handling. That is, every time that a node is deleted, its corresponding entry in its parent has to be also deleted. eFIND is called to register the modifications resulted from these adjustments (Algorithm 2) and to save every pair resulted from the propagation of deletion operations (Algorithm 6).

Finally, Algorithm 5 checks whether the propagation reached the root node and this node has only one element (line 11). If this is the case, its child node turns the new root node (lines 12 and 13) and is saved by eFIND (line 14). Then, the algorithm executes additional treatment after deleting an entry. This is the case for indices like the R-tree and the R*-tree since they require the reinsertion of entries that were contained in deleted nodes.

Handling nodes with eFIND. The execution of Algorithm 5 can invoke four specialized algorithms of eFIND to manipulate nodes of the underlying index in the following cases: (i) the retrieval of nodes (line 1), (ii) the direct deletion of the entry E from a leaf node (line 5), (iii) the treatment of nodes with underflow

Algorithm 6: Handling an underflow

Input: P as a pair (H, D), where H is a set of modified nodes and D is a possibly deleted node

1 **if** *D is not NULL* **then**
2   **save *D* by calling Algorithm 7**;
3 **foreach** node *ND* **in** *H* **do**
4   **delete *ND* from the *Write Buffer Table* by calling Algorithm 7**;
5   **save *ND* by calling Algorithm 4**;

(lines 9 and 10), (iv) the adjustment of entries (line 10), and (v) the deletion of a root node (line 14). In this section, we discuss eFIND's algorithms responsible for executing the cases (iii) and (v). The cases (ii) and (iv) are covered by the algorithms introduced in the insert operations (Section 5.1), while the case (i) is discussed in the search operations (Section 5.3).

Algorithm 6 depicts how eFIND saves the pair P resulted from the underflow handling of the underlying index. This algorithm is employed to execute the case (iii). The idea behind this algorithm follows the same principle as Algorithm 3. That is, Algorithm 6 firstly saves the deletion by using Algorithm 7 if there exists a deleted node (lines 1 and 2). Next, for each modified node in H (line 3), this algorithm deletes the old version of the modified node (line 4) and then stores the modified node as a newly created node (line 5), improving the space utilization and future search operations.

Algorithm 7 depicts how eFIND stores a deleted node in its *Write Buffer Table*. This algorithm is employed to execute the case (v) and to help the execution of Algorithm 6. First, the deleted node is registered as a new log entry in the log file for data durability purposes (line 1). Next, an auxiliary variable corresponding to the hash entry of the deleted node is defined (line 2). By using this variable, two main cases are alternately possible (lines 3 to 10). In the first case, the node has a corresponding hash entry in the *Write Buffer Table* (lines 4 to 7). Hence, previous modifications are deleted from the write buffer (line 4), creating space for storing other modifications. Then, the status (line 5), the number of modifications (line 6), and the timestamp (line 7) of the hash entry are updated accordingly. The second case is executed if the deleted node has not a corresponding hash entry in the write buffer; thus, the algorithm sets the values of the new hash entry (line 9) and stores it in the *Write Buffer Table* (line 10). Finally, the algorithm executes the flushing operation, if the write buffer is full (lines 11 and 12).

Complexity Analysis. The complexity analysis of Algorithm 5 depends on the underlying index being ported. Hence, we focus on understanding the complexity of Algorithms 6 and 7. The time complexity of Algorithm 6 is similar to the complexity of Algorithm 3 (Section 5.1). In the worst case, its complexity is given by $\mathcal{C}_{alg6} = \mathcal{C}_{alg7} + p\mathcal{C}_{alg7} + p\mathcal{C}_{alg4}$, where p is the number of nodes in D. The time complexity of Algorithm 7 is given by $\mathcal{C}_{alg7} = \mathcal{W}_s + \mathcal{H} + \mathcal{F}$, where $\mathcal{F}$ refers to the cost of freeing the red-black tree of the deleted node, if any. In addition, the time complexity of Algorithm 7 can also include the cost of a flushing operation, as detailed in [15]. As for the space complexity, Algorithm 6 does not require extra space and Algorithm 4 always registers the deletion in the log file one time only.

Algorithm 7: Storing a deleted node in the *Write Buffer Table*

Input: N as the node being deleted

append the new log entry $\langle N_{id}, (metadata(N), height(N), (\mathrm{DEL}, \mathrm{NULL}))\rangle$ into the log file;
let *WBEntry* be the hash entry of N in the *Write Buffer Table*;
if *WBEntry is not NULL* **then**
    free its *mod_tree*, if any;
    set the status of *WBEntry* to DEL;
    increase the value of *mod_count* of *WBEntry* by 1;
    update the value of *timestamp* of *WBEntry* to *now()*;
else
    set *WBEntry* to the hash entry $\langle N_{id}, (metadata(N), (now(), height(N), 1), (\mathrm{DEL}, \mathrm{NULL}))\rangle$;
    store *WBEntry* in the *Write Buffer Table*;
if *Write Buffer Table is full* **then**
    execute a flushing operation (as detailed in [15]);

Examples of Execution. Our running example deletes the indexed points p_6 and p_2 after inserting the two points p_{19} and p_{20} (Section 5.1). By applying Algorithm 5 to process these operations, a set of modifications are appended to the log file and stored in the write buffer of each spatial index ported to the SSD. We highlight the sequence of the modifications after each delete operation as follows:

- **The R-tree.** To delete the point p_6, it processes an underflow operation on the node L_2, deleting it (*log#* 11 and 12 in Figure 7a and the seventh and second lines in Figure 5a) and adjusting the MBR of the entry pointing to the node I_3 (*log#* 13 in Figure 7a and the first line in Figure 5a). Then, the point p_8 is reinserted into the R-tree in the node L_1 (*log#* 14 in Figure 7a and the fifth line in Figure 5a). This reinsertion provokes one adjustment in its parent entry (*log#* 15 in Figure 7a and the second line in Figure 5a) and another adjustment in an entry of the node I_1 (*log#* 16 in Figure 7a and the first line in Figure 5a). The point p_2 is directly removed from the node L_8 that has its MBR adjusted (*log#* 17 and 18 in Figure 7a and the last and third lines in Figure 5a).
- **The R*-tree.** Similarly to the R-tree, it processes an underflow operation on the node L_1 to delete the point p_6 (*log#* 11 and 12 in Figure 7b and the eighth and second lines in Figure 5b), adjusting its parent entry (*log#* 13 in Figure 7b and the first line in Figure 5b). Next, it reinserts the point p_{13} into the R*-tree (*log#* 14 in Figure 7b and the ninth line in Figure 5b), requiring two adjustments in the upper levels of the tree (*log#* 15 and 16 in Figure 7b and the third and first lines in Figure 5b). The deletion of the point p_2 is directly performed on the node L_8 (*log#* 17 in Figure 7b and the last line in Figure 5a), which has its corresponding parent entry adjusted afterwards (*log#* 18 in Figure 7b and the fourth line in Figure 5b).
- **The Hilbert R-tree.** It deletes the point p_6 from the node L_2 (*log#* 32 in Figure 7c and the eleventh line in Figure 5c), adjusting the MBR of entries in the two levels upwards (*log#* 33 and 34 in Figure 7c and the sixth and third lines in Figure 5c). Then, it deletes the node L_7 when removing the point p_2 (*log#* 35 and 36 in Figure 7c and the last line in Figure 5c). This consequently provokes the adjustment of entries in the nodes I_4 and I_2 (*log#* 37 and 38 in Figure 7c and the fifth and second lines in Figure 5c).

Algorithm 8: Searching spatial objects indexed by a spatial index

Input: N as the node being visited, S as the search object, T as the topological predicate
Output: R as a list of entries

1 **if** *N is an internal node* **then**
2   **foreach** *entry E in N* **do**
3   **if** *the MBR of E and S satisfy T* **then**
4   **let** ***NN*** **be the node pointed by** ***E*** **that is retrieved by calling Algorithm 9**;
5   **call Algorithm 8 for** ***NN*** **recursively**;
6 **else**
7   **foreach** *entry E in N* **do**
8   **if** *the MBR of E and S satisfy T* **then**
9   append E into R;

– **The xBR$^+$-tree.** It deletes the points p_6 and p_2 directly from their respective nodes L_5 and L_2 (*log#* 12 and 13 in Figure 7d and the fourth and third lines in Figure 5d).

5.3 Search Operations

General algorithm. Considering a spatial index being ported by eFIND (i.e., an R-tree, R*-tree, a Hilbert R-tree, and an xBR$^+$-tree), Algorithm 8 returns a list R containing the entries after traversing the tree by starting from its root node N. For this, a search object S and a topological predicate T (e.g., contains, intersects) are employed. The algorithm starts checking whether the current node being traversed is internal or leaf (lines 1 to 9). For internal nodes (lines 1 to 5), Algorithm 8 chooses the path in the tree whose entry satisfies the topological predicate for the search object S (line 3). In this case, the node pointed by this entry is retrieved by eFIND (line 4) and then Algorithm 8 is called recursively. For leaf nodes (lines 6 to 9), only those entries satisfying the criterion of the search operation is appended in the list of entries (lines 8 and 9). Algorithm 8 can be optimized by the underlying index of eFIND. For instance, the xBR$^+$-tree offers some specialized algorithms to deal with different types of spatial queries [55].

Handling nodes with eFIND. The execution of Algorithm 8 invokes the specialized algorithm of eFIND responsible for retrieving nodes from the underlying index (line 4). Furthermore, Algorithms 1 and 5 also employ this specialized algorithm when traversing nodes in order to insert or delete entries. In this section, we discuss how to retrieve a node by using eFIND.

Algorithm 9 specifies the procedure employed by eFIND to retrieve a node and is equivalent to the algorithm presented in [15]. We included this algorithm in the article for completeness purposes. First, the algorithm takes the identifier of a node as input and returns the most recent version of this node. There are three alternative cases. The first one is whether the node is stored in the *Write Buffer Table* with status equal to NEW or DEL (lines 1 and 2); thus, it is directly returned by using the pointer stored in the write buffer since it does not contain further modifications (line 3). The second case refers to a not modified node; the algorithm verifies if this node contains a cached version in the *Read Buffer Table* (lines 4 to 7), avoiding a read operation to be performed on the SSD (returning the node in line 14). Otherwise, the node is read from the SSD and inserted in the *Read Buffer Table* (lines 8 to 10, and returning the node in line 14). In both cases, the *Read Buffer Table* is possibly reorganized by the read buffer replacement policy (line 11). The last case is if the node has modifications stored in the write buffer. Here, a merge operation is needed in order to combine the entries stored in the modification tree and the existing entries of the node (lines 12 and 13). After applying this merging, the algorithm returns the most recent version of the node (line 14).

In this article, we extend and better analyze an important aspect not studied in our previous work: the merge operation (line 13). Algorithm 10 returns the most recent version of a node N and takes two sorted arrays L_1 and L_2 as input respectively representing the modified entries stored in the *Write Buffer Table*, and the entries stored in the previous version of N. Note that these two arrays are not empty. The first array would be empty if N has not modifications; but in this case, Algorithm 9 directly returns N either from the *Read Buffer Table* (line 7) or from the SSD (line 9). The second array would be empty if there exists a hash entry of N in the *Write Buffer Table* with *status* equal to NEW; but in this case, Algorithm 9 directly returns the node pointed by the entry of the write buffer (line 3). Both arrays are sorted since the first flushing operation on a node always happens when its status in the *Write Buffer Table* is equal to NEW. Hence, the comparison function employed by the red-black tree of the node guarantees that its entries are sorted, and this sorting is preserved after a flushing operation.

The merge operation is based on the classical merge operation between sorted files [27]. Let i, j be two integer values, where i indicates the position in the first array and j indicates the position in the second array (line 1). Let also N be an empty node (line 2). A loop is then processed, starting with $i = j = 0$ (lines 3 to 10). First, the algorithm evaluates the order of the current entries being analyzed (line 4), that is, $L_1[i]$ and $L_2[j]$, by executing the comparison function employed by the red-black trees of the underlying index (Section 4.1). It guarantees the structural constraints and properties of nodes of the underlying index. If $L_1[i]$ goes before $L_2[j]$ (line 5), this means that the merge operation appends $L_1[i]$ to N and increments i by 1 (line 6) since an element of the first array has been processed. If the inverse happens, that is, $L_2[j]$ goes before $L_1[i]$ (line 7), the merge operation appends $L_2[j]$ to N and increments j by 1 (line 8). If $L_1[i]$ and $L_2[j]$ point to the same entry (i.e., their unique identifier are equal), the merge operation appends only $L_1[i]$ to N if its value (i.e., *mod_result* in the *mod_tree*) is different to NULL and increment both i and j by 1 (line 10). This is done because the result should only maintain the latest version of the entry and non-null entries. The loop is finished if i (j) is equal to the number of entries in the first (second) list. Finally, the entries that were not evaluated by the loop are appended to N (lines 11 to 14), which is returned as the final step of the merge operation (line 15).

Complexity Analysis. Since the complexity of Algorithm 8 depends on the underlying index, we focus on analyzing the complexity of Algorithms 9 and 10. The time complexity of Algorithm 9, in the best case, is the cost of accessing the hash table that implements the write buffer. That is, $\mathcal{C}_{alg9} = \mathcal{H}$. In the worst case, the

Algorithm 9: Retrieving a node by using eFIND (slightly adapted from [15])

Input: I as the identifier of the node to be returned
Output: N as the node with identifier equal to I

let *WBEntry* be the hash entry in the *Write Buffer Table* with key I;
if *WBEntry has status equal to NEW or DEL* **then**
  return the node pointed by *WBEntry*;
let N be an empty node;
let *RBEntry* be the hash entry in the *Read Buffer Table* with key I;
if *RBEntry is not NULL* **then**
  let N become the node pointed by *RBEntry*;
else
  let N become its version read from the SSD;
  insert N into the *Read Buffer Table* and its identifier in the *RQ*;
apply the read buffer replacement policy;
if *WBEntry has status equal to MOD* **then**
  return the result of a merge operation between the entries contained in the *mod_tree* of *WBEntry* and the entries of N by invoking Algorithm 10;
return N;

Algorithm 10: Merging the modifications of a node

Input: *SI* as the underlying index, L_1 and L_2 as two arrays of entries, where L_1 contains entries from *mod_tree* and L_2 contains entries from the last stored version of N
Output: N as the most recent version of the node

let i and j be two integers equal to 0;
let N be an empty node;
while $i < length(L_1)$ *and* $j < length(L_2)$ **do**
  let r become the result of the comparison function between $L_1[i]$ and $L_2[j]$ according to structural constraints and properties of *SI*;
  if $r < 0$ **then**
    append $L_1[i]$ into N and increment i by 1;
  else if $r > 0$ **then**
    append $L_2[j]$ into N and increment j by 1;
  else
    append $L_1[i]$ into N and increment both i and j by 1;
for i **to** $length(L_1)$ **by** 1 **do**
  append $L_1[i]$ into N;
for j **to** $length(L_2)$ **by** 1 **do**
  append $L_2[j]$ into N;
return N;

time complexity of Algorithm 9 is given by $\mathcal{C}_{alg9} = 2\mathcal{H}+\mathcal{R}+\mathcal{C}_{alg10}$, where $\mathcal{R}$ refers to the average cost of a read operation to the SSD. Note that Algorithm 9 may have the time complexity of $2\mathcal{H}$ or $2\mathcal{H}+\mathcal{R}$ (i.e., they occur if the node has not modification). The time complexity of $\mathcal{C}_{alg10}$ can determined by $\mathcal{O}(l_1 + l_2)$, where l_1 and l_2 represent the number of entries stored in the main memory and in the SSD, respectively. Recall that the use of the comparison function defined by the underlying index (Section 4.1), which checks the order of entries, also impacts the complexity of Algorithm 10.

As for the space complexity, Algorithm 9 does not require extra space. On the other hand, Algorithm 10 requires additional memory to keep the merged c entries of the node. Thus, it can assume the space complexity $\mathcal{O}(c)$.

Examples of Execution. Our running example executes one IRQ in each ported spatial index, after applying the insertions (Section 5.1) and deletions (Section 5.2). Algorithm 8 is employed to execute this IRQ in each spatial index, resulting in the following sequence of operations:

- **The R-tree.** It starts reading the its root node R from the *Read Buffer Table* (first line in Figure 6a). Then, it descends the tree by accessing the node I_1 since the IRQ intersects its MBR. For this, a merging operation (Algorithm 10) between the entries stored in the *mod_tree* of the I_1 and the entries stored in the SSD is performed, resulting in the most recent version of this node. That is, this merge operation returns the node containing the modified version of the entry I_3 (stored in the first line in Figure 5a) and the stored version of the entry I_4. Next, the node I_3 is read from the SSD, which has also modifications stored in its corresponding *mod_tree* to be merged (Algorithm 10). Afterward, the leaf node N_1 is directly accessed from the *Write Buffer Table* since it is a newly created node. It stores the point p_1 in the result of the spatial query. Then, recursively the node I_4 is read from the SSD because its MBR also intersects the query window of the IRQ. The last accessed node is L_4, read from the SSD. Then, the point p_5 is appended to the final result of the query.
- **The R*-tree.** It firstly reads the root node R and then its child node I_1, both stored in the *Read Buffer Table* (first two lines in Figure 6b). Next, it accesses the node I_4. For retrieving this node, a merging operation (Algorithm 10) is performed to integrate the modified entries stored in the *Write Buffer Table* (third line in Figure 5b) and the stored entries. Then, the node L_3 is read from the SSD since it does not contain modifications. From this node, the point p_5 is added to the result. Afterward, its sibling node L_4 is retrieved by performing the merging operation (considering the modified entry in the ninth line in Figure 5b), adding the point p_1 to the result.
- **The Hilbert R-tree.** Starting from the root node R, it descends the tree by accessing the node I_1. These nodes are retrieved from the *Read Buffer Table* (first two lines in Figure 6c). Then, two paths are followed. The first path descends the tree by retrieving the nodes I_3, I_7, and L_3. Expect for the node L_3 that is read from the SSD, the remaining nodes have modifications merged (Algorithm 10) to their stored versions (using the third and seventh lines in Figure 5c). After reading the leaf node of this path, it adds the point p_1 to the result of the spatial query. The second path accesses the newly created nodes N_3 and N_4 directly from the *Write Buffer Table* (fourth and eighth lines in Figure 5c), and then retrieve the node L_4 that is read from the SSD. It finishes by adding the point p_5 to the result.
- **The xBR$^+$-tree.** It follows a single path to solve the spatial query. It starts from the root node R and then reads the node I_1, both cached in the *Read Buffer Table* (first two lines in Figure 6d). Next, the leaf nodes L_1 and L_3 are accessed because their data bounding rectangles intersect the query window of the IRQ. Since they do not have modifications and are not cached in the read buffer, they are directly read from the SSD. After accessing each leaf node, the points p_1 and p_5 returned as result.

6 Experimental Evaluation

In this section, we empirically measure the efficiency of porting disk-based spatial index structures by using our systematic approach. For this, we port the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree by using eFIND and FAST. It shows that our systematic approach can be deployed by using different frameworks. In particular, FAST-based spatial indices are considered the main competitors of the eFIND-based spatial indices, which were discussed in this article. To create the FAST-based spatial indices, we adapted the FAST's data structures and algorithms in a similar way to the adaptations performed on eFIND. Section 6.1 shows the experimental setup. Performance results when building spatial indices, performing spatial queries, and computing mixed operations are discussed in Sections 6.2, 6.3, and 6.4, respectively.

6.1 Experimental Setup

Datasets. We used four spatial datasets, stored in PostGIS/PostgreSQL [50]. Two of them contain real data collected from OpenStreetMaps following the methodology in [12]. The first one is a real spatial dataset, called *brazil_points2019*, containing 2,139,087 points inside Brazil (approximately, 156MB). The second one, called *us_midwest_points2019*, contains 2,460,597 points inside the Midwest of the USA (approximately, 180MB). The other two spatial datasets are synthetic, called *synthetic1* and *synthetic2*, containing respectively 5 and 10 million points (approximately, 326MB and 651MB, respectively). Each synthetic dataset stores points equally distributed in 125 clusters uniformly distributed in the range $[0, 1]^2$. The points in each cluster (i.e., 40,000 points for *synthetic1* and 80,000 points for *synthetic2*) were located around the center of each cluster, according to Gaussian distribution. It follows the same methodology as the experiments conducted in [55]. The use of spatial datasets with different characteristics and volume allows us to analyze the spatial indices under distinct scenarios.

Configurations. We employed our systematic approach to creating different configurations of the ported spatial index structures based on the frameworks eFIND and FAST. As a result, we evaluated the following flash-aware spatial indices: (i) the *eFIND R-tree*, (ii) the *eFIND R*-tree*, (iii) the *eFIND Hilbert R-tree*, (iv) the *eFIND xBR$^+$-tree*, (v) the *FAST R-tree*, (vi) the *FAST R*-tree*, (vii) the *FAST Hilbert R-tree*, and (viii) the *FAST xBR$^+$-tree*. The R-tree used the quadratic split algorithm, the R*-tree employed the reinsertion policy of 30%, and the Hilbert R-tree leveraged the 2-to-3 split policy. We varied the employed node (i.e., page) sizes from 2KB to 16KB. The buffer and log sizes were 512KB and 10MB, respectively. We employed the best parameter values of FAST, as reported in [59]: the FAST* flushing policy. We also employed the best parameter values of eFIND, as reported in [15]: the use of 60% of the oldest modified nodes to create flushing units, the flushing policy using the height of nodes as weight, the allocation of 20% of the buffer for the read buffer, and flushing unit size equal to 5. Hence, we built and evaluated 32 different configurations. We did not include non-ported spatial indices (e.g., original R-tree) since other works in the literature have shown that the number of reads and writes of such indices is high and negatively impact on the SSD performance [63; 59; 34; 13].

Workloads. We executed three types of workloads on each spatial dataset: (i) index construction by inserting point objects one-by-one, (ii) execution of 1,000 point queries and 3,000 intersection range queries (IRQs), and (iii) execution of insertions and queries. A point query returns the points that are equal to a given point. An IRQ retrieves the points contained in a given rectangular query window, including its borders. Three different sets of query windows were used, representing respectively 1,000 rectangles with 0.001%, 0.01%, and 0.1% of the area of the total extent of the dataset being used by the workload. We generated different query windows for each dataset using the algorithms described in [12]. This method allows us to measure the performance of spatial queries with distinct selectivity levels. We consider the selectivity of a spatial query as the ratio of the number of returned objects and the total objects; thus, the three sets of query windows built IRQs with low, medium, and high selectivity, respectively. For each configuration and dataset, the workloads were executed 5 times. We avoided the page caching of the system by using direct I/O. For computing statistical values of insertions, we collected the average elapsed time. For computing statistical values of spatial queries, we calculated the average elapsed time to execute each set of query windows.

Running Environment. We employed a server equipped with an Intel Core® i7-4770 with a frequency of 3.40GHz and 32GB of main memory. We made use of two SSDs: (i) Kingston V300 of 480GB, and (ii) Intel Series 535 of 240GB. The Intel SSD is a high-end SSD that provides faster reads and writes than the low-end Kingston SSD. We employed the Intel SSD to execute all the workloads and configurations. This provided us an overview of the performance behavior of the underlying framework implementing our systematic approach. Next, we used the Kingston SSD to compare eFIND-based configurations, allowing us to analyze the performance of eFIND-based spatial indices by considering different architectures of SSDs. The operating system used was Ubuntu Server 14.04 64 bits. We also used FESTIval [17] to execute the workloads.

6.2 Building Spatial Indices

Figure 8 shows that eFIND fits well in our systematic approach since a particular disk-based spatial index ported by eFIND provided better performance than the same disk-based spatial index ported by FAST. The eFIND R-tree delivered the best results in most cases, followed by the eFIND xBR$^+$-tree, which provided the second-best results. Compared to the FAST R-tree, the eFIND R-tree showed performance gains ranging from 40% to 70.3%. A performance gain shows how much a configuration reduced the elapsed time from another configuration. We highlight the long processing times of the FAST xBR$^+$-tree (mainly in the dataset *synthetic2*) due to the complexity of adapting FAST to deal with the special constraints of the xBR$^+$-tree, as discussed in [16]. Since the eFIND-based spatial indices provided the best results, our analysis focuses on detailing their performance behavior, including experiments conducted in the Kingston SSD.

Figure 9 depicts the performance results obtained in the Kingston SSD. We can note that the underlying characteristics of the ported index structures (Section 3) exert a strong influence on the experiments. For the real spatial datasets,

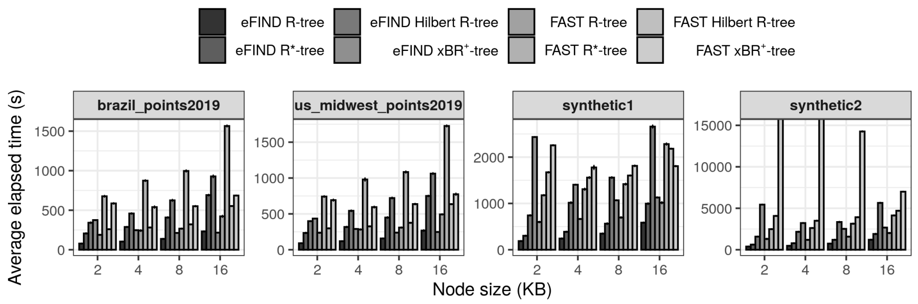


Fig. 8 Performance results when building the flash-aware spatial indices in the Intel SSD. Note that the FAST xBR$^+$-tree presented long processing times for building indices on the dataset *synthetic2*; thus, we have cut the y-scale in this case to better visualize the results. The eFIND-based spatial indices showed better performance than FAST-based spatial indices. The eFIND R-tree and the eFIND xBR$^+$-tree delivered the best results in several situations.

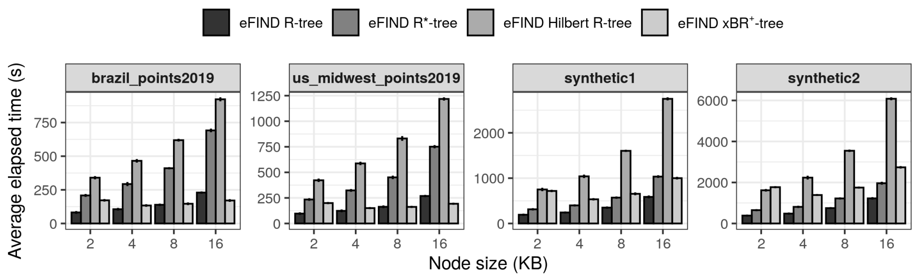


Fig. 9 Performance results when building the eFIND-based spatial indices in the Kingston SSD. In most cases, the eFIND R-tree showed the best results.

two different behaviors were observed. Compared to the other eFIND-based spatial indices and considering the node sizes from 2KB to 8KB, the eFIND R-tree provided performance gains from 33.8% to 79.1% for the Intel SSD and from 5.2% to 80.4% for the Kingston SSD. On the other hand, for the node size equal to 16KB, the eFIND xBR$^+$-tree overcame the eFIND R-tree with reductions up to 7.6% for the Intel SSD and up to 28.3% for the Kingston SSD. Analyzing the cost of building spatial indices using this size is particularly useful when considering the spatial query processing (see Section 6.3).

As for the synthetic spatial datasets, the eFIND R-tree was the fastest spatial index in both SSDs. Its performance gains against the other eFIND-based spatial indices were very expressive. It ranged from 36.4% to 92.9% for the Intel SSD (Figure 8), and from 37.6% to 79.9% for the Kingston SSD (Figure 9).

The poor performance of the eFIND R*-tree and the eFIND Hilbert R-tree is related to the management of overflowed nodes. The overhead of the eFIND R*-tree is due to its reinsertion policy, requiring more reads in insert operations compared to the R-tree. As discussed in the literature (see Section 2), the excessive number of reads impairs the performance of applications in SSDs. Concerning the eFIND Hilbert R-tree, its bad performance is because of the redistribution policy.

It is comparable to the cost of a split operation of the R-tree since s sibling nodes should be written together with a possible adjustment of their parent node. Further, the split operation of the eFIND Hilbert R-tree possibly requires four writes because of the 2-to-3 split policy. Thus, the eFIND Hilbert R-tree required long processing times to build spatial indices in both SSDs.

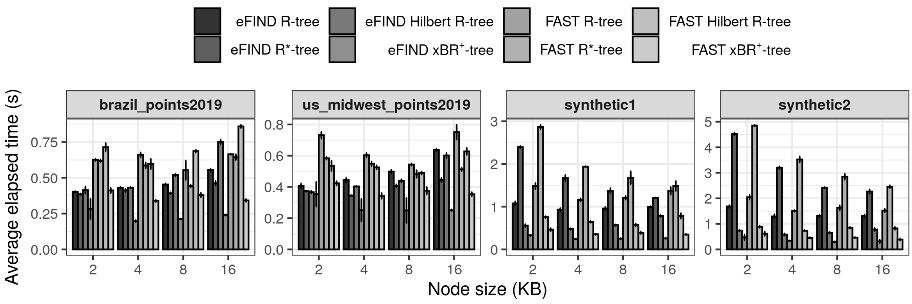
Another important observation is that the special constraints of the underlying index may impair the performance when retrieving nodes by using the eFIND's algorithms and data structures. For instance, the requirement of a sophisticated comparison function to guarantee the sorting property among entries of internal and leaf nodes. We note this influence when analyzing the experimental results of the Hilbert R-tree and xBR$^+$-tree. They require that nodes' entries are sorted by their Hilbert values and directional digits, respectively. eFIND makes use of this comparison function every time that a modified node is recovered by the index (Algorithm 9). Hence, it mainly impacts the performance of the insertions. To improve it, there are efforts in the literature that propose specific bulk-insertions and bulk-loading algorithms. For xBR$^+$-trees, examples of such algorithms are given in [57].

Several configurations presented the best results by employing the node size of 2KB. This is due to the high cost of writing flushing units with larger index pages (e.g., 16KB) since a write made on the application layer can be split into several internal writes to the SSD. Further, the data volume also impacted the construction time, as expected. Hence, building flash-aware spatial indices required more time as the node size and the data volume also increased.

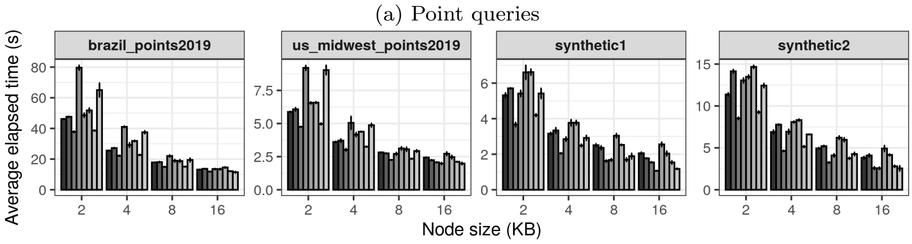
6.3 Query Processing

Figure 10 shows that eFIND-based spatial indices outperformed their corresponding FAST-based spatial indices. The eFIND xBR$^+$-tree delivered the best results when processing the point queries, whereas the eFIND Hilbert R-tree, in most cases, provided the best results when processing the IRQs. Note that the FAST xBR$^+$-tree delivered the best performance results among the FAST-based spatial indices to process the point queries. This reveals that the space partitioning strategy of the xBR$^+$-tree distinguishes itself by delivering lesser elapsed times for computing point queries on SSDs. To process the point queries, the eFIND xBR$^+$-tree showed performance gains ranging from 16.4% to 44.2%, if compared to the FAST xBR$^+$-tree. As for the IRQs, the eFIND Hilbert R-tree showed reductions up to 17.6%, 17%, and 16.3% for the low, medium, and high selectivity levels, respectively, if compared to the FAST Hilbert R-tree. Due to the superior performance of the eFIND-based spatial indices, our next analysis focuses on detailing their performance results, including experiments conducted in the Kingston SSD.

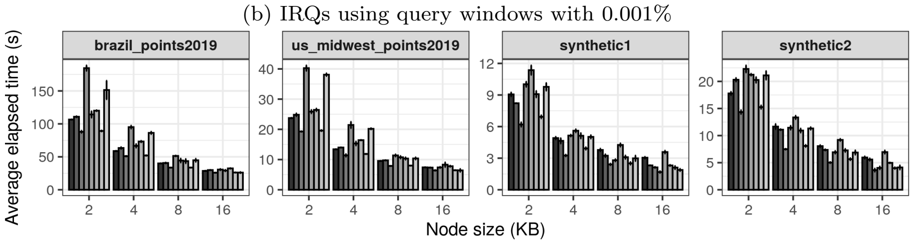
Figure 11 shows the performance results when processing the spatial queries in the Kingston SSD. As for the point queries, the eFIND xBR$^+$-tree showed performance gains from 3.6% to 89.5% for the Intel SSD and from 15% to 94.4% for the Kingston SSD, if compared to the other eFIND-based spatial indices. In general, a point query requires the traversal of a small number of paths in the tree. Thus, processing point queries using node sizes equal to 4KB and 8KB provided better results.



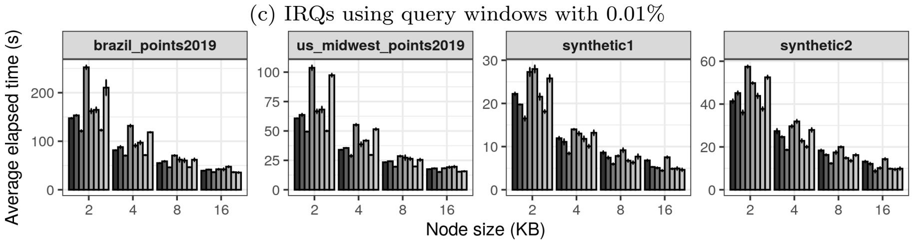
(a) Point queries



(b) IRQs using query windows with 0.001%



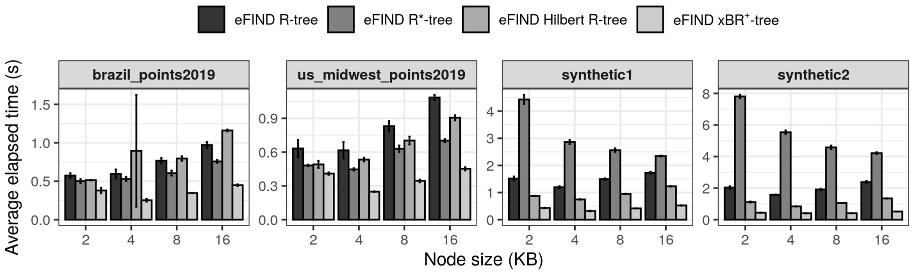
(c) IRQs using query windows with 0.01%



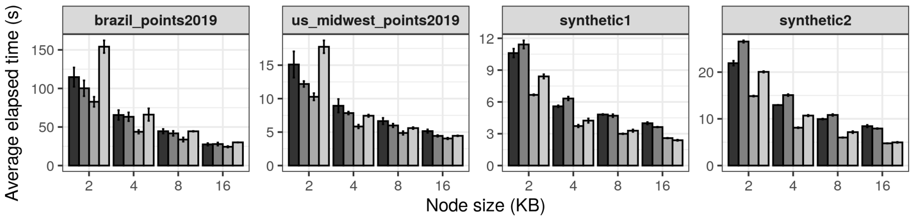
(d) IRQs using query windows with 0.1%

Fig. 10 Performance results when executing the point queries and IRQs in the Intel SSD. It showed that the best results were delivered by the eFIND-based spatial indices.

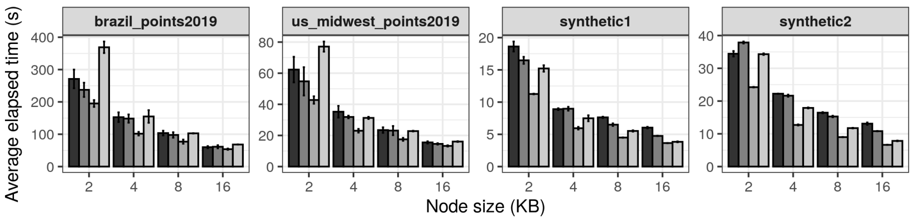
Concerning the execution of IRQs, all configurations showed better performance when employing the node size equal to 16KB because more entries are loaded into the main memory with a few reads. Hence, we consider this node size in the following. We can note that the eFIND Hilbert R-tree and the eFIND xBR$^+$-tree overcame the other flash-aware spatial indices. Due to the differences in the underlying structure of the SSDs, we obtained different performance behaviors. For the Intel SSD, the eFIND xBR$^+$-tree outperformed the eFIND Hilbert R-tree



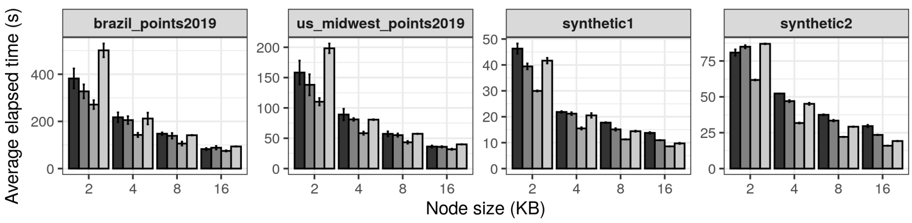
(a) Point queries



(b) IRQs using query windows with 0.001%



(c) IRQs using query windows with 0.01%



(d) IRQs using query windows with 0.1%

Fig. 11 Performance results when executing the point queries and IRQs in the Kingston SSD. As for the point queries, the eFIND xBR$^+$-tree overcame the other configurations. As for the IRQs, the best results were obtained when employing the node size of 16KB. In this case, the eFIND Hilbert R-tree and the eFIND xBR$^+$-tree delivered the best results.

to process IRQs with low selectivity in most cases (Figure 10b), with performance gains up to 30.9%. On the other hand, the eFIND Hilbert R-tree imposed reductions between 10.1% and 17.4% for the other selectivity levels (Figure 10c and d). For the Kingston SSD (Figure 11), the eFIND Hilbert R-tree was better than the eFIND xBR$^+$-tree in the majority of cases by gathering reductions up to 18.9%, 21.1%, and 20.2% for the low, medium, and high selectivity levels, respectively.

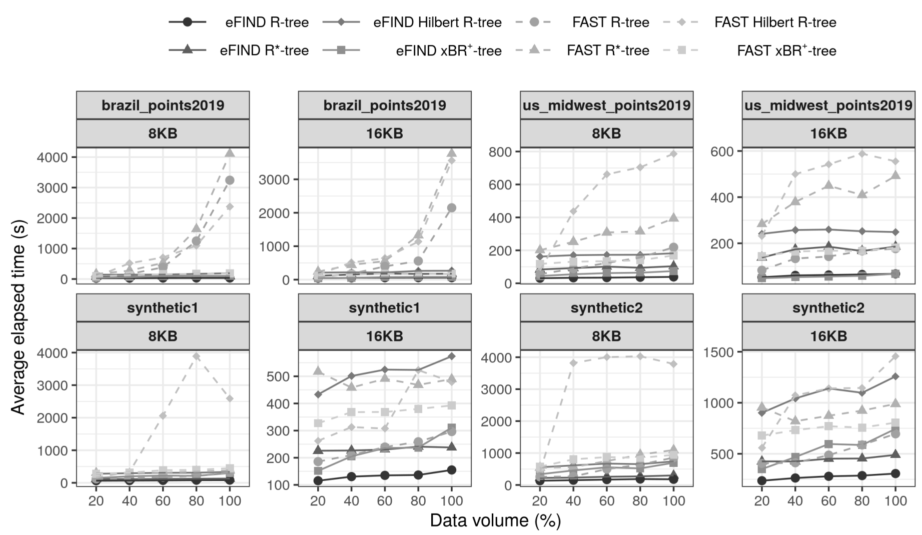


Fig. 12 Performance results for inserting point objects by gradually increasing the data volume. Each spatial dataset and node size are showed in the header of each chart. In most cases, the eFIND R-tree provided the fastest processing time.

In most cases, processing IRQs on the synthetic datasets required much less time than on the real datasets because of their specific spatial distribution. IRQs returning more points (i.e., with high selectivity) exhibited higher elapsed times. This is due to the traversal of multiple large nodes in the main memory, requiring more CPU time than queries with low selectivity. Hence, the performance behavior of IRQs is quite different from the performance behavior of the point queries.

6.4 Mixing Insertions and Queries

In this section, we analyze the performance of the configurations to handle insertions and queries by gradually increasing the volume of the spatial dataset. To this end, we executed a workload that has three sequential steps; the workload sequentially (i) indexes 20% of the point objects stored in the spatial dataset, (ii) computes the point queries, and (iii) executes the IRQs. This sequence is repeated until all the point objects of the corresponding dataset are indexed. Thus, the workload has 5 phases of insertions and queries, where each phase means that the data volume increases 20%. We executed this workload by using the ported spatial indices with eFIND and FAST in the Intel SSD.

Figures 12 to 14 depict the performance results considering the node sizes equal to 8KB and 16KB only. Thus, we can analyze the performance of the flash-aware spatial indices in each step of the workload, that is, the execution of insertions (Figure 12), point queries (Figure 13), and IRQs (Figure 14). The use of the node size equal to 8KB allows us to deliver a good balance between the performance of insertions and queries, whereas the node size equal to 16KB shows better performance when executing queries, such as discussed in Section 6.3.

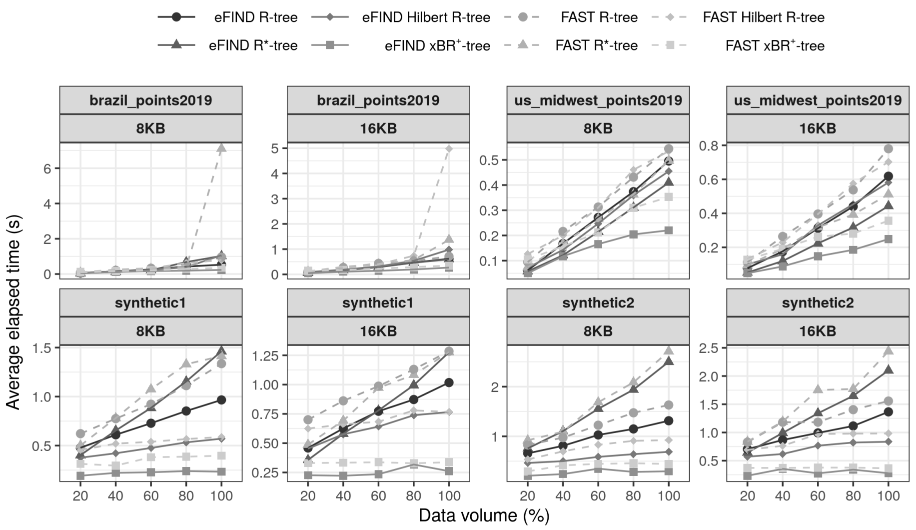
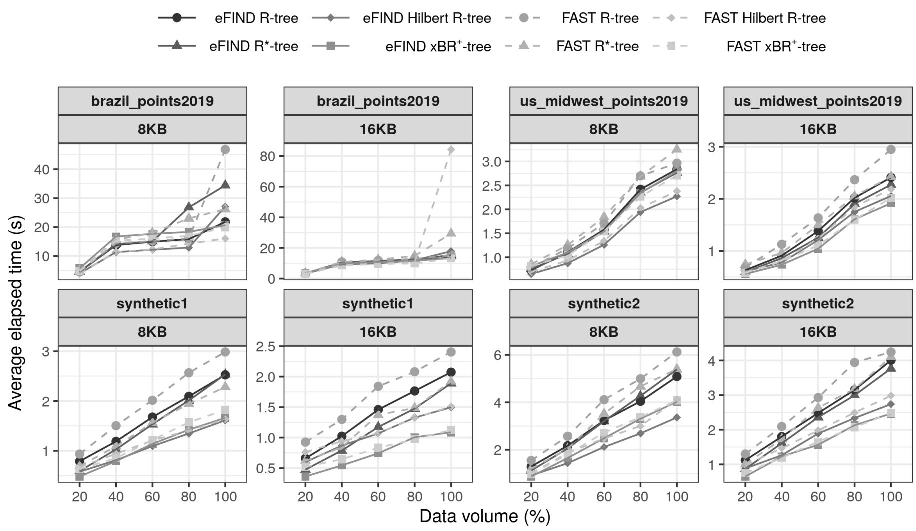


Fig. 13 Performance results for executing point queries by gradually increasing the data volume. Each spatial dataset and node size are showed in the header of each chart. The point queries were executed after inserting the point objects (Figure 14). The eFIND xBR$^+$-tree delivered the best elapsed times.

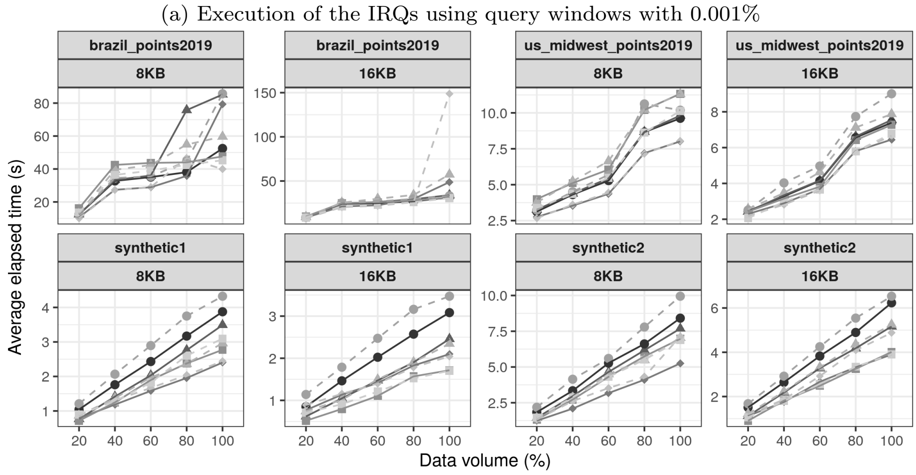
The results of the experiments reported in this section show similar behavior to the performance results in Sections 6.2 and 6.3. In general, a disk-based spatial index ported by eFIND outperformed its corresponding FAST version. In this sense, we highlight eFIND-based spatial indices that showed good performance results in each phase of the workload. In most cases, the eFIND R-tree provided the best performance to index point objects (Figure 12). Compared to the other eFIND-based spatial indices, the eFIND R-tree showed reductions up to 82.1% for the real datasets and up to 76.9% for the synthetic datasets in each step of the workload. The eFIND xBR$^+$-tree often gathered the best results to execute point queries (Figure 13). It provided a reduction up to 96.6% for the real datasets and up to 78.3% for the synthetic datasets in each step, if compared to the other eFIND-based spatial indices. Finally, the fastest processing times for processing the IRQs were also acquired by the eFIND Hilbert R-tree and the eFIND xBR$^+$-tree. A similar behavior indicates that the proposed approach to porting spatial index structures to SSDs is consistent when increasing the handled data volume.

7 Conclusions and Future Work

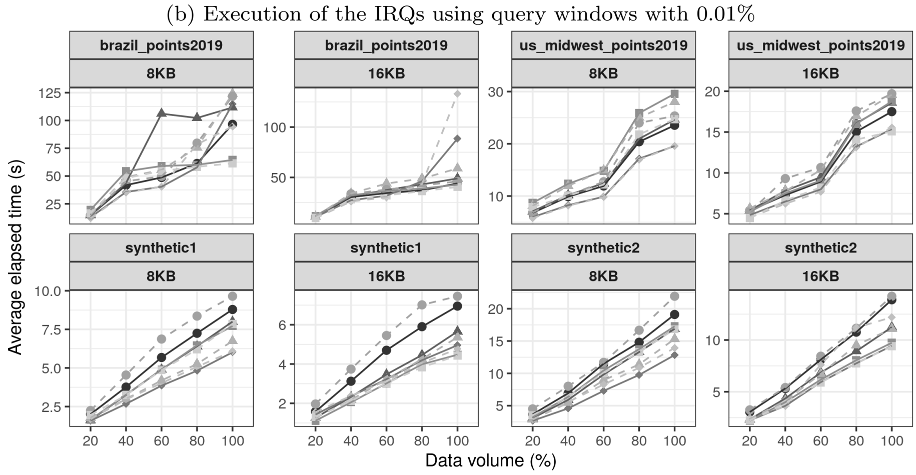
In this article, we have proposed a novel *systematic approach* for porting disk-based spatial indices to SSDs. To this end, we have characterized how the index nodes are written and read in index operations like insertions, deletions, and queries. We have used this characterization in an expressive set of disk-based spatial index structures, including the R-tree, the R*-tree, the Hilbert R-tree, and the xBR$^+$-tree.



(a) Execution of the IRQs using query windows with 0.001%



(b) Execution of the IRQs using query windows with 0.01%



(c) Execution of the IRQs, using query windows with 0.1%

Fig. 14 Performance results for executing IRQs with different sizes of query window by gradually increasing the data volume. Each spatial dataset and node size are shown in the header of each chart. The IRQs were executed after computing the point queries. In general, the best results were obtained by the eFIND Hilbert R-tree and the eFIND xBR$^+$-tree.

We have described how our systematic approach is deployed by eFIND due to its performance advantages showed in our experiments. Hence, we have presented how the data structures and algorithms of eFIND were generalized and extended to fit in our systematic approach. In our running example, we have created the following *flash-aware spatial indices*: (i) the eFIND R-tree, (ii) the eFIND R*-tree, (iii) the eFIND Hilbert R-tree, and (iv) the eFIND xBR^+-tree. To the best of our knowledge, this is the first work that shows how to port different spatial index structures to SSDs by using the same underlying framework.

Our systematic approach can also be applied to other data- and space-driven access methods. For this, two main steps are needed. The first step is to identify the additional attributes to be stored in the underlying data structures of eFIND (i.e., write and read buffers, and log file). This includes the design of the comparison function that accomplishes the sort property of the underlying index if any. The second step is to generalize and characterize the modifications made on the nodes of the underlying index in order to fit the specialized algorithms implemented by using eFIND. This step can be based on our generalization, which provides general algorithms for insertions, deletions, and queries, as well as, other generalizations like GiST and SP-GiST. As a result, our systematic approach can be used to port disk-based spatial indices that were not included in this article, such as the R^+-tree [60], the K-D-B-tree [53], and the X-tree [6].

Our experiments analyzed the efficiency of the ported spatial indices through an extensive empirical evaluation that also implemented the systematic approach by using FAST. Hence, we have evaluated the R-tree, the R*-tree, the Hilbert R-tree, and the xBR^+-tree ported by FAST and eFIND. They were evaluated by using two real spatial datasets and two synthetic spatial datasets, and by executing three different types of workloads. We highlight the following results:

- The eFIND fits well in the systematic approach and the spatial index structures ported by it provided the best performance results;
- The eFIND R-tree delivered the best results when executing insertions;
- The eFIND xBR^+-tree was very efficient when processing point queries;
- The eFIND Hilbert R-tree, followed by the eFIND xBR^+-tree, gathered the most preeminent results when processing IRQs.

We also highlight that such findings were consistent when gradually increasing the data volume of the spatial datasets. Further, the use of the node size equal to 8KB allowed us to deliver a good balance between the performance of insertions and queries, whereas the node size equal to 16KB showed better performance when executing queries. Hence, the choice of the node size depends on the focus of the application.

Future work will deal with many topics. The approach proposed in this article was designed to take advantage of the intrinsic characteristics of the SSDs. The first topic of our future work is to analyze how the systematic approach implemented by eFIND performs on HDDs by conducting theoretical and empirical studies and by including possible adaptations. We also plan to study the performance of spatial indices ported to SSDs by using large spatial datasets and evaluating other common spatial queries, like k-nearest neighbors. In addition, we plan to provide support for the ACID properties [31], allowing us the complete integration of our approach into spatial database systems. Further, we aim at conducting performance evaluations by employing *flash simulators* [61; 13], which emulate the behavior of real SSDs in the main memory. Future work also includes the extension of our systematic approach to port spatial index structures to *non-volatile main memories* (NVMM) like ReRAM, STT-RAM, and PCM [65]. These memories are byte-addressable, allowing us to access persistent data with CPU load and store instructions. Finally, the last topic of future work is to apply the proposed systematic approach, with its integration with eFIND, to port one-dimensional index structures to SSDs and NVMMs. This includes the generalization of data structures and algorithms to deal with one- and multi-dimensional data.

Acknowledgements This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work has also been supported by CNPq and by the São Paulo Research Foundation (FAPESP). Cristina D. Aguiar has been supported by the grant #2018/22277-8, FAPESP. The work of Michael Vassilakopoulos and Antonio Corral is funded by the MINECO research project [TIN2017-83964-R].

References

1. Agrawal D, Ganesan D, Sitaraman R, Diao Y, Singh S (2009) Lazy-adaptive tree: An optimized index structure for flash devices. VLDB Endowment 2(1):361–372
2. Agrawal N, Prabhakaran V, Wobber T, Davis JD, Manasse M, Panigrahy R (2008) Design tradeoffs for SSD performance. In: USENIX 2008 Annual Technical Conf., pp 57–70
3. Aref WG, Ilyas IF (2001) SP-GiST: An extensible database index for supporting space partitioning trees. Journal of Intelligent Information Systems 17:215–240
4. Arge L, Berg MD, Haverkort H, Yi K (2008) The Priority R-tree: A practically efficient and worst-case optimal r-tree. ACM Trans Algorithms 4(1)
5. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: An efficient and robust access method for points and rectangles. In: ACM SIGMOD Int. Conf. on Management of Data, pp 322–331
6. Berchtold S, Keim DA, Kriegel HP (1996) The X-Tree: An index structure for high-dimensional data. In: Int. Conf. on Very Large Databases, pp 28–39
7. Bouganim L, Jónsson B, Bonnet P (2009) uFLIP: Understanding flash IO patterns. In: Fourth Biennial Conf. on Innovative Data Systems Research
8. Brayner A, Monteiro Filho JM (2016) Hardware-aware database systems: A new era for database technology is coming - vision paper. In: Brazilian Symp. on Databases, pp 187–192
9. Carniel AC, Ciferri RR, Ciferri CDA (2016) The performance relation of spatial indexing on hard disk drives and solid state drives. In: Brazilian Symp. on GeoInformatics, pp 263–274
10. Carniel AC, Ciferri RR, Ciferri CDA (2017) Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives. Journal of Information and Data Management 8(1):34–49
11. Carniel AC, Ciferri RR, Ciferri CDA (2017) A generic and efficient framework for spatial indexing on flash-based solid state drives. In: European Conf. on Advances in Databases and Information Systems, pp 229–243

12. Carniel AC, Ciferri RR, Ciferri CDA (2017) Spatial datasets for conducting experimental evaluations of spatial indices. In: Satellite Events of the Brazilian Symp. on Databases - Dataset Showcase Workshop, pp 286–295
13. Carniel AC, Silva TB, Bonicenha KLS, Ciferri RR, Ciferri CDA (2017) Analyzing the performance of spatial indices on flash memories using a flash simulator. In: Brazilian Symp. on Databases, pp 40–51
14. Carniel AC, Roumelis G, Ciferri RR, Vassilakopoulos M, Corral A, Ciferri CDA (2018) An efficient flash-aware spatial index for points. In: Brazilian Symp. on GeoInformatics, pp 68–79
15. Carniel AC, Ciferri RR, Ciferri CDA (2019) A generic and efficient framework for flash-aware spatial indexing. Information Systems 82:102–120
16. Carniel AC, Roumelis G, Ciferri RR, Vassilakopoulos M, Corral A, Ciferri CDA (2019) Indexing points on flash-based solid state drives using the xBR$^+$-tree. Journal of Information and Data Management 10(1):35–48
17. Carniel AC, Ciferri RR, Ciferri CDA (2020) FESTIval: A versatile framework for conducting experimental evaluations of spatial indices. MethodsX 7:1–19
18. Chen F, Koufaty DA, Zhang X (2009) Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, pp 181–192
19. Chen F, Hou B, Lee R (2016) Internal parallelism of flash memory-based solid-state drives. ACM Trans on Storage 12(3):13:1–13:39
20. Cormer D (1979) Ubiquitous B-tree. ACM Comp Surveys 11(2):121–137
21. Denning PJ (1980) Working sets past and present. IEEE Trans on Software Engineering SE-6(1):64–84
22. Effelsberg W, Haerder T (1984) Principles of database buffer management. ACM Trans on Database Systems 9(4):560–595
23. Emrich T, Graf F, Kriegel HP, Schubert M, Thoma M (2010) On the impact of flash SSDs on spatial indexing. In: Int. Workshop on Data Management on New Hardware, pp 3–8
24. Fevgas A, Bozanis P (2015) Grid-file: Towards to a flash efficient multi-dimensional index. In: Int. Conf. on Database and Expert Systems Applications, pp 285–294
25. Fevgas A, Bozanis P (2019) LB-Grid: An SSD efficient grid file. Data & Knowledge Engineering 121:18–41
26. Fevgas A, Akritidis L, Bozanis P, Manolopoulos Y (2019) Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. The VLDB Journal pp 1–39
27. Folk MJ, Zoellick B, Riccardi G (1997) File Structures: An Object-Oriented Approach with C++, 3rd edn. Addison Wesley
28. Gaede V, Günther O (1998) Multidimensional access methods. ACM Comp Surveys 30(2):170–231
29. Graefe G (2012) A survey of b-tree logging and recovery techniques. ACM Trans on Database Systems 37(1)
30. Guttman A (1984) R-trees: A dynamic index structure for spatial searching. In: ACM SIGMOD Int. Conf. on Management of Data, pp 47–57
31. Harder T, Reuter A (1993) Principles of transaction-oriented database recovery. ACM Comp Surveys 15(4):287–317

32. Hellerstein JM, Naughton JF, Pfeffer A (1995) Generalized search trees for database systems. In: Int. Conf. on Very Large Databases, pp 562–573
33. Jenkins B (2006) Hash functions for hash table lookup. `http://burtleburtle.net/bob/hash/index.html`
34. Jin P, Xie X, Wang N, Yue L (2015) Optimizing R-tree for flash memory. Expert Systems with Applications 42(10):4676–4686
35. Jin P, Yang C, Jensen CS, Yang P, Yue L (2016) Read/write-optimized tree indexing for solid-state drives. The VLDB Journal 25(5):695–717
36. Johnson T, Shasha D (1994) 2Q: A low overhead high performance buffer management replacement algorithm. In: Int. Conf. on Very Large Databases, pp 439–450
37. Jung M, Kandemir M (2013) Revisiting widely held SSD expectations and rethinking system-level implications. In: ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, pp 203–216
38. Kamel I, Faloutsos C (1994) Hilbert R-tree: An improved R-tree using fractals. In: Int. Conf. on Very Large Databases, pp 500–509
39. Koltsidas I, Viglas SD (2011) Spatial data management over flash memory. In: Int. Conf. on Advances in Spatial and Temporal Databases, pp 449–453
40. Kornacker M (1999) High-performance extensible indexing. In: Int. Conf. on Very Large Databases, pp 699–708
41. Kwon SJ, Ranjitkar A, Ko YB, Chung TS (2011) FTL algorithms for NAND-type flash memories. Design Automation for Embedded Systems 15(3-4):191–224
42. Li G, Zhao P, Yuan L, Gao S (2013) Efficient implementation of a multi-dimensional index structure over flash memory storage systems. The Journal of Supercomputing 64(3):1055–1074
43. Li Y, He B, Yang RJ, Luo Q, Yi K (2010) Tree indexing on solid state drives. VLDB Endowment 3(1-2):1195–1206
44. Lin S, Zeinalipour-Yazti D, Kalogeraki V, Gunopulos D, Najjar WA (2006) Efficient indexing data structures for flash-based sensor devices. ACM Trans on Storage 2(4):468–503
45. Lv Y, Li J, Cui B, Chen X (2011) Log-Compact R-tree: An efficient spatial index for SSD. In: Int. Conf. on Database Systems for Advanced Applications, pp 202–213
46. Mehlhorn K, Sanders P (2008) Algorithms and Data Structures: The Basic Toolbox. Springer
47. Mittal S, Vetter JS (2016) A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans on Parallel and Distributed Systems 27(5):1537–1550
48. Nievergelt J, Hinterberger H, Sevcik KC (1984) The grid file: An adaptable, symmetric multikey file structure. ACM Trans on Database Systems 9(1):38–71
49. Oosterom PVaN (2005) Spatial Access Methods. In: Longley PA, Goodchild MF, Maguire DJ, Rhind DW (eds) Geographical Information Systems: Principles, Techniques, Management and Applications, 2nd edn, pp 385–400
50. PostGIS (2020) Spatial and geographic objects for postgresql. `https://postgis.net/`
51. Proietti G, Faloutsos C (1999) I/O complexity for range queries on region data stored using an r-tree. In: Int. Conf. on Data Engineering, pp 628–635

52. Rigaux P, Scholl M, Voisard A (2001) Spatial databases: with application to GIS, 1st edn. Morgan Kaufmann
53. Robinson JT (1981) The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: ACM SIGMOD Int. Conf. on Management of Data, pp 10–18
54. Roumelis G, Vassilakopoulos M, Loukopoulos T, Corral A, Manolopoulos Y (2015) The xBR+-tree: an efficient access method for points. In: Int. Conf. on Database and Expert Systems Applications, pp 43–58
55. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2017) Efficient query processing on large spatial databases: A performance study. Journal of Systems and Software 132:165–185
56. Roumelis G, Vassilakopoulos M, Corral A, Fevgas A, Manolopoulos Y (2018) Spatial batch-queries processing using xbr^{+}-trees in solid-state drives. In: Int. Conf. on Model and Data Engineering, pp 301–317
57. Roumelis G, Fevgas A, Vassilakopoulos M, Corral A, Bozanis P, Manolopoulos Y (2019) Bulk-loading and bulk-insertion algorithms for xBR^{+}-trees in solid state drives. Computing pp 1–25
58. Samet H (1984) The quadtree and related hierarchical data structures. ACM Comp Surveys 16(2):187–260
59. Sarwat M, Mokbel MF, Zhou X, Nath S (2013) Generic and efficient framework for search trees on flash memory storage systems. GeoInformatica 17(3):417–448
60. Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-tree: A dynamic index for multi-dimensional objects. In: Int. Conf. on Very Large Databases, pp 507–518
61. Su X, Jin P, Xiang X, Cui K, Yue L (2009) Flash-DBSim: A simulation tool for evaluating flash-based database algorithms. In: IEEE Int. Conf. on Computer Science and Information Technology, pp 185–189
62. Thonangi R, Babu S, Yang J (2012) A practical concurrent index for solid-state drives. In: ACM Int. Conf. on Information and Knowledge Management, p 13321341
63. Wu CH, Chang LP, Kuo TW (2003) An efficient R-tree implementation over flash-memory storage systems. In: ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems, pp 17–24
64. Wu CH, Kuo TW, Chang LP (2007) An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans on Embedded Computing Systems 6(3)
65. Zhang Y, Swanson S (2015) A study of application performance with non-volatile main memory. In: Symp. on Mass Storage Systems and Technologies, pp 1–10