

## 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. A. Ciferri  
Dept. of Computer Science, University of São Paulo, São Carlos, SP 13566-590, Brazil  
E-mail: cdac@icmc.usp.br

tions, the eFIND xBR<sup>+</sup>-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<sup>+</sup>-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

43 transforms a disk-based spatial index into a flash-aware spatial index. It is based  
44 on a distinct set of design goals that provides guidelines to deal with the intrinsic  
45 characteristics of SSDs. The effectiveness of these guidelines has been validated  
46 through experimental evaluations. Another advantage of eFIND is that its data  
47 structures do not change the structure of the index being ported, requiring a low-  
48 cost integration when implementing eFIND in spatial database systems and GIS.

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

58 In this article, our goal is to solve these problems by introducing a novel *sys-*  
59 *tematic approach* for porting disk-based spatial index structures to SSDs. The sys-  
60 *tematic approach* is based on the *characterization* of the types of operations that  
61 different indexing strategies (i.e., data partitioning and space partitioning) can per-  
62 form on index pages. In this sense, we focus on identifying when reads and writes  
63 are performed by index operations, such as insertions and queries. With this char-  
64 acterization, we leverage an extended and generalized version of the eFIND's data  
65 structures and algorithms to implement our systematic approach. We analyze and  
66 validate our systematic approach by porting an expressive set of disk-based spatial  
67 index structures to SSDs: (i) the R-tree, (ii) the R\*-tree, (iii) the Hilbert R-tree,  
68 and (iv) the xBR<sup>+</sup>-tree. Since they are hierarchical structures, in the remainder  
69 of this article, we use *node* as an equivalent term to *index page*.

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

- 71 – development of a *systematic approach* that provides the needed guidelines to  
72 port a disk-based spatial index to SSDs;
- 73 – application of the systematic approach using eFIND for porting the disk-based  
74 spatial indices R-tree, R\*-tree, Hilbert R-tree, and xBR<sup>+</sup>-tree to SSDs; thus,  
75 we show the *creation of the flash-aware spatial indices eFIND R-tree, eFIND R\**-  
76 *tree, eFIND Hilbert R-tree, and eFIND xBR<sup>+</sup>-tree*;
- 77 – analysis of an *extensive experimental evaluation* that compares the performance  
78 of the flash-aware spatial indices when inserting and querying points from  
79 synthetic and real datasets;
- 80 – identification of the eFIND R-tree as the best flash-aware spatial index to  
81 handle insertions, the eFIND xBR<sup>+</sup>-tree as an efficient structure to execute  
82 several types of spatial queries, and the eFIND Hilbert R-tree as an efficient  
83 indexing scheme for processing intersection range queries.

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

---

**90 2 Related Work**

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

99 **2.1 Generalized Search Trees**

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

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

121 **2.2 Approaches to Porting One-Dimensional Index Structures to SSDs**

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

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

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

### 154 2.3 Specific Approaches to Porting Spatial Index Structures to SSDs

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

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

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

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

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

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

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

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

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

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

223 Our previous work [14; 16] ports the xBR<sup>+</sup>-tree to SSDs using the generic  
 224 frameworks eFIND and FAST (Section 2.4); thus creating the flash-aware spatial  
 225 indices *eFIND xBR<sup>+</sup>-tree* and *FAST xBR<sup>+</sup>-tree*, respectively. The experiments show

226 that the eFIND xBR<sup>+</sup>-tree provides the best results because it fits well with the  
 227 properties and structural constraints of the xBR<sup>+</sup>-tree (see Section 3.4). However,  
 228 to accomplish this porting, some modifications in the eFIND's data structures are  
 229 performed. A limitation of the previous work is that these modifications are not  
 230 generalized in a form that can be applied to other disk-based spatial index struc-  
 231 tures. Other limitations are related to the use of eFIND, as detailed in Section 2.4.

## 232 2.4 General Approaches to Porting Spatial Index Structures to SSDs

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

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

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

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

- 269 – We propose a novel systematic approach for porting disk-based data-driven  
 270 and space-driven access methods to SSDs, in general. For this, we characterize  
 271 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<sup>+</sup>-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<sup>+</sup>-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

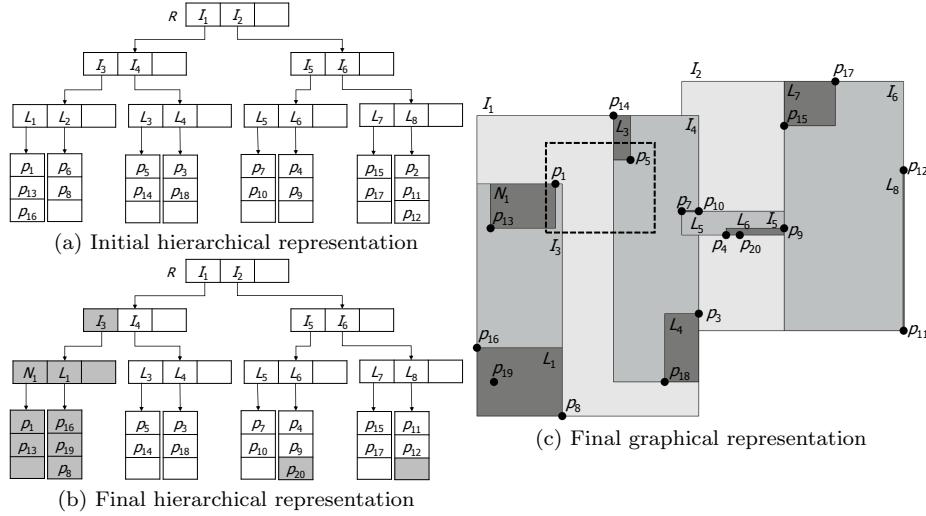
In this section we summarize four spatial index structures employed in this article. They are: (i) the R-tree (Section 3.1), (ii) the R\*-tree (Section 3.2), (iii) the Hilbert R-tree (Section 3.3), and (iv) the xBR<sup>+</sup>-tree (Section 3.4). For each spatial index, we provide its underlying structure 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



**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.

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

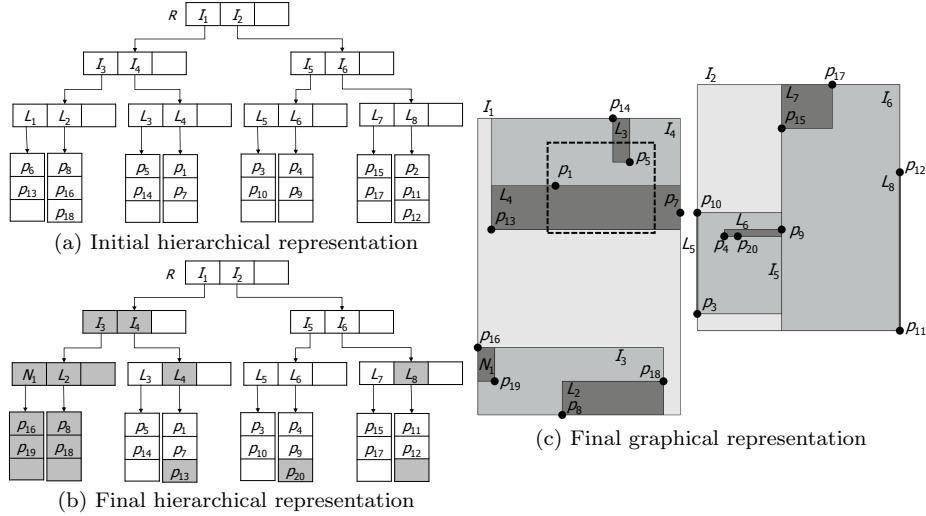
### 322 3.2 The R\*-tree

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

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

### 334 3.3 The Hilbert R-tree

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



**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.

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

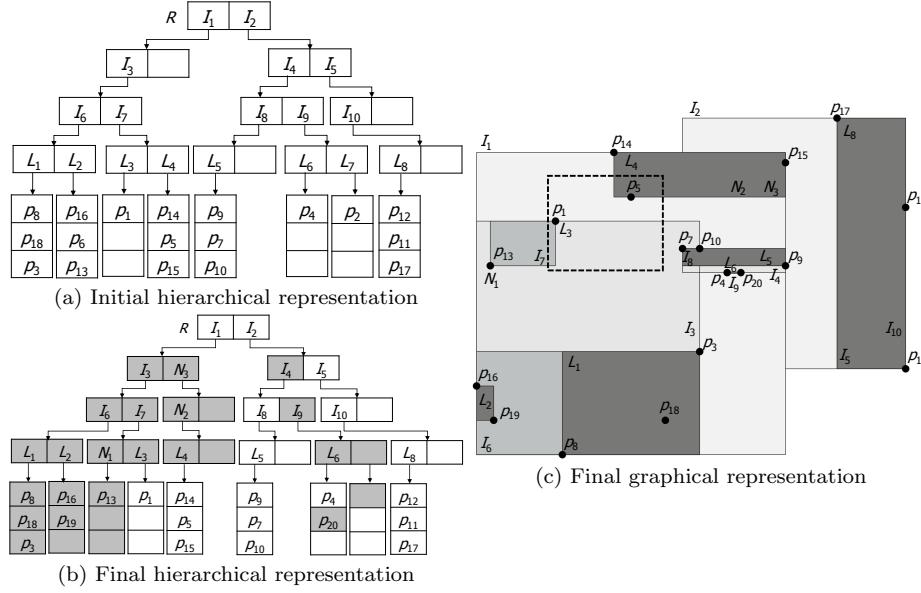
342 Figure 3 depicts the hierarchical and graphical representations of a Hilbert  
 343 R-tree in a similar way to the Figures 1 and 2. Because of the extra element in  
 344 internal nodes and considering that every node has a fixed number of bytes, the  
 345 maximum capacity of an internal node might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

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

### 353 3.4 The xBR<sup>+</sup>-tree

354 The xBR<sup>+</sup>-tree [54] is a hierarchical spatial index based on the regular decompo-  
 355 sition of space of Quadtrees [58] able to index multi-dimensional points. Hence,  
 356 it is a space-driven access method. For two-dimensional points, the xBR<sup>+</sup>-tree  
 357 decomposes recursively the space by 4 equal quadrants, called *sub-quadrants*.

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

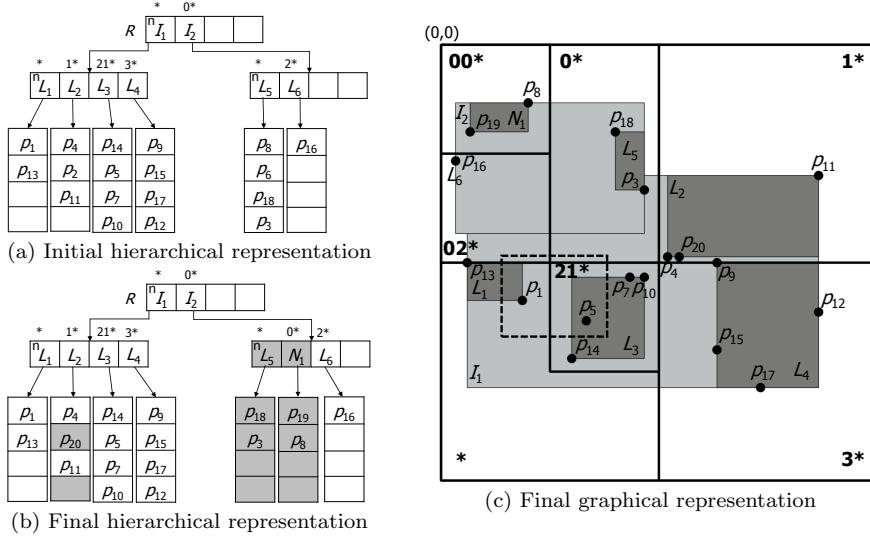


**Fig. 3** A Hilbert R-tree in hierarchical representation (a) and the Hilbert R-tree resulting after applying a set of modifications on it in hierarchical (b) and graphical (c) representations. The hierarchical representation highlights the performed modifications in gray.

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

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

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



**Fig. 4** An xBR<sup>+</sup>-tree in hierarchical representation (a) and the xBR<sup>+</sup>-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.

### 385 3.5 Running Example

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

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

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

## 398 4 Generalizing and Adapting the eFIND for the Systematic Approach

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

---

 404 4.1 Data Structures

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

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

420  $\mathbf{M}$  consists of the attributes that store the metadata of the node required for  
 421 processing internal algorithms of the underlying index. Thus, the attributes may  
 422 vary. Considering the spatial indices detailed in Section 3,  $\mathbf{M}$  is empty if the un-  
 423 derlying index is the R-tree, the R\*-tree, and the Hilbert R-tree. If the underlying  
 424 index is the xBR<sup>+</sup>-tree,  $\mathbf{M}$  is an attribute named *header* that consists of the pair  
 425  $(o, s)$  corresponding to the metadata stored in internal nodes, where  $o$  is the origin  
 426 point and  $s$  is the side length of the sub-quadrant of the node, respectively. Since  
 427 this pair only applies to internal nodes,  $\mathbf{M}$  assumes NULL if the node is a leaf node  
 428 (see Figure 5d).

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

438  $\mathbf{E}$  refers to the *essential attributes to manage the modifications of the node*; it  
 439 consists of the pair  $(\text{status}, \text{mod\_tree})$ . *status* stores the type of modification made  
 440 on the node and can be NEW, MOD, or DEL for representing that the node is  
 441 a newly created node in the buffer, a node stored in the SSD but with modified  
 442 entries, or a deleted node, respectively. *mod\_tree* assumes NULL, if *status* is equal  
 443 to DEL. Otherwise, it is a red-black tree storing the most recent version of the  
 444 node's entries. Each element of this red-black tree is a pair  $(k, e)$ , where  $k$  is the  
 445 search key and corresponds to the unique identifier of the entry and  $e$  stores the  
 446 latest version of the entry, assuming NULL if it is removed from the node. We  
 447 employ red-black trees for storing the node's entries because of its amortized cost  
 448 of executing insertions, deletions, and searches. Further, it allows that only the  
 449 latest version of an entry be stored in the *Write Buffer Table*; thus, the space of  
 450 the write buffer is better managed with a low cost of retrieving the most recent

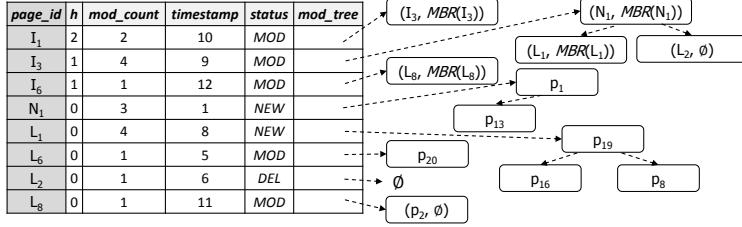
451 version of a node (see Section 5). More importantly, the red-black tree maintains  
 452 a specific order among the node’s entries, an essential aspect when dealing with  
 453 spatial indices that require a special sort property (e.g., the Hilbert R-tree and  
 454 the xBR<sup>+</sup>-tree). Hence, the design of the comparison function of the red-black  
 455 trees should accomplish the sort property of the underlying index. Considering  
 456 the spatial indices detailed in Section 3, we provide the following base ideas for  
 457 implementing their corresponding comparison functions as follows:

- 458 – **The R-tree and the R\*-tree.** Their comparison functions implement the  
 459 ascending order of *id*, which is an element that either gives direct access to the  
 460 indexed spatial object (if the node is a leaf node) or points to a child node (if  
 461 the node is an internal node).
- 462 – **The Hilbert R-tree.** If the node is a leaf node, its comparison function com-  
 463 putes the ascending order of the Hilbert values calculated from *r* (i.e., the  
 464 MBR). Otherwise, its comparison function implements the ascending order of  
 465 *lhv*, which is an element of internal nodes that stores the largest Hilbert value  
 466 of a child node. In both cases, ties are resolved by sorting the entries by *id*.
- 467 – **The xBR<sup>+</sup>-tree.** If the node is a leaf node, its comparison function imple-  
 468 ments the ascending order of the *x*-axis coordinates of the points where ties  
 469 are resolved by sorting the entries by their *y*-axis coordinates and then by  
 470 their *id*. Otherwise, its comparison function implements the ascending order  
 471 of the directional digits of the entries (using the *qside* and *DBR*), considering  
 472 the metadata of the internal node (i.e., the pair  $(o, s)$ ).

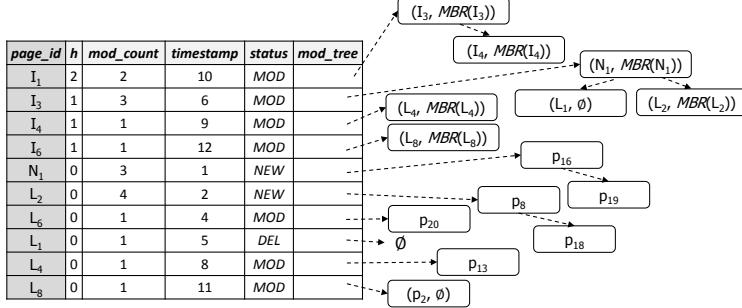
473 It is important to emphasize the role of the comparison function in the cost of  
 474 performing operations in red-black trees. In our running example, the comparison  
 475 functions for the R-tree and R\*-tree have a constant cost. On the other hand, the  
 476 Hilbert R-tree and the xBR<sup>+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

479 Figure 5 shows the *Write Buffer Tables* for each spatial index of our running  
 480 example. In this figure, *MBR* is a function for computing the rectangle that encom-  
 481 passes all entries of a node by considering current modifications in the write buffer.  
 482 For instance, the first line of the hash table in Figure 5a shows that  $I_1$ , located in  
 483 the *height* 2, has the *status* MOD to store the entry  $(I_3, MBR(I_3))$ . Note that this  
 484 entry now corresponds to the most recent version of the first entry of  $I_1$  in the  
 485 eFIND R-tree depicted in Figure 1. This modification occurred in the *timestamp*  
 486 10 and is derived from the adjustment of the node  $I_3$  after the reinsertion of the  
 487 point  $p_8$ . The other write buffers (Figures 5b to d) store the needed modifications  
 488 performed on their corresponding spatial indices to process the index operations  
 489 of our running example, which are further detailed in Section 5.

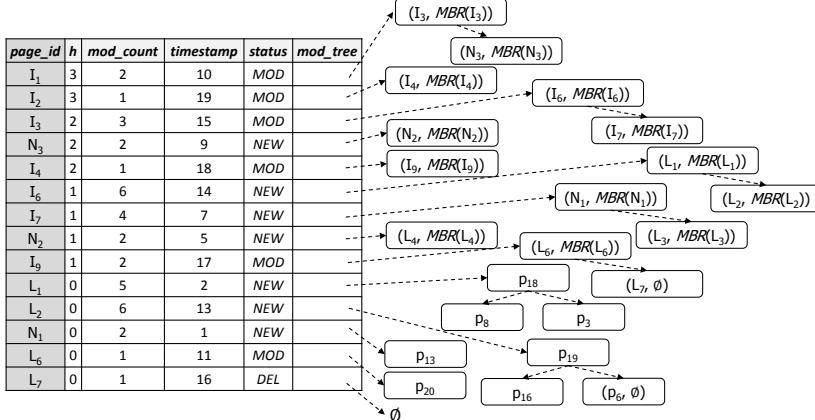
490 **Read buffer.** Its main goal is to avoid excessive random reads by caching the nodes  
 491 stored in the SSD (design goal 3). eFIND leverages another hash table named *Read*  
 492 *Buffer Table* to implement the read buffer. It does not employ the same hash table  
 493 of the write buffer because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



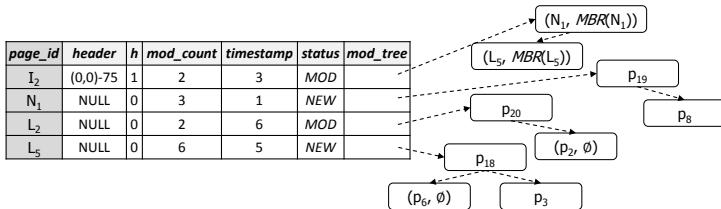
(a) The write buffer for the eFIND R-tree (Figure 1)



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



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



(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<sup>+</sup>-tree transforming them to the eFIND R-tree (a), the eFIND R\*-tree (b), the eFIND Hilbert R-tree (c), and the eFIND xBR<sup>+</sup>-tree (d).

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

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

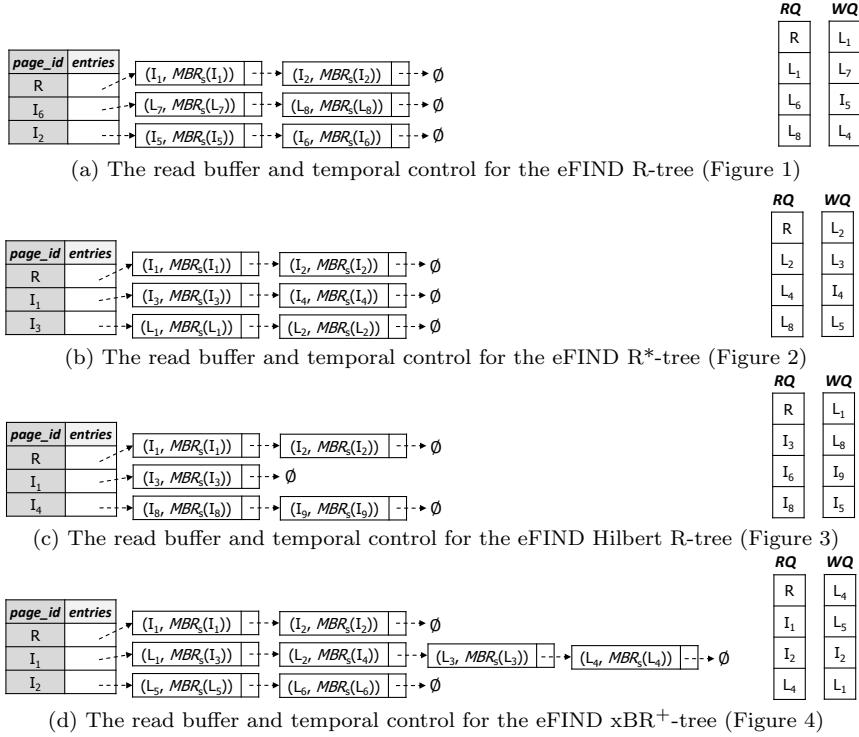
513  $\text{entries}$  refers to a list storing the node's entries. Since the *Read Buffer Table*  
 514 caches nodes stores in the SSD, this list does not consider the modifications stored  
 515 in the write buffer. An element of the  $\text{entries}$  has the same format as an entry  
 516 of the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

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

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

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

543  $\mathbf{M}$  is the same  $\mathbf{M}$  from that used in each hash entry of the *Write Buffer Table*.  $\mathbf{P}$   
 544 is a subset of  $\mathbf{F}$ . In this article, it consists of a single attribute named  $h$  that stores  
 545 the height of the node. The other attributes of  $\mathbf{F}$  (i.e., *timestamp* and *mod\_count*)



**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<sup>+</sup>-tree (d).

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

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

558 Figure 7 shows the log file for each spatial index of our running example. In this  
559 figure, the first column (*log#*) refers to the sequence of the processed modification.  
560 Thus, we can follow the sequence of modifications performed to process the index  
561 operations of our running example. For instance, the first modification of the  
562 eFIND R-tree is the creation of the node N<sub>1</sub> (*timestamp* equal to 1 in Figure 5a)  
563 that contains the points p<sub>1</sub> and p<sub>13</sub>. This sequence is stored in the first three log  
564 entries in Figure 7a. Section 5 further details how the modifications are appended  
565 in the log files of each spatial index.

<i>log#</i>	<i>page_id</i>	<i>h</i>	<i>type_mod</i>	<i>result</i>
1	N <sub>1</sub>	0	NEW	-
2	N <sub>1</sub>	0	MOD	p <sub>1</sub>
3	N <sub>1</sub>	0	MOD	p <sub>13</sub>
4	L <sub>1</sub>	0	DEL	-
5	L <sub>1</sub>	0	NEW	-
6	L <sub>1</sub>	0	MOD	p <sub>16</sub>
7	L <sub>1</sub>	0	MOD	p <sub>19</sub>
8	I <sub>3</sub>	1	MOD	(N <sub>1</sub> , MBR(N <sub>1</sub> ))
9	I <sub>3</sub>	1	MOD	(L <sub>1</sub> , MBR(L <sub>1</sub> ))
10	L <sub>6</sub>	0	MOD	p <sub>20</sub>
11	L <sub>2</sub>	0	DEL	-
12	I <sub>3</sub>	1	MOD	(L <sub>2</sub> , ∅)
13	I <sub>1</sub>	2	MOD	(I <sub>3</sub> , MBR(I <sub>3</sub> ))
14	L <sub>1</sub>	0	MOD	p <sub>8</sub>
15	I <sub>3</sub>	1	MOD	(L <sub>1</sub> , MBR(L <sub>1</sub> ))
16	I <sub>1</sub>	2	MOD	(I <sub>3</sub> , MBR(I <sub>3</sub> ))
17	L <sub>8</sub>	0	MOD	(p <sub>2</sub> , ∅)
18	I <sub>6</sub>	1	MOD	(L <sub>8</sub> , MBR(L <sub>8</sub> ))

(a) The log file for the eFIND R-tree  
(Figure 1) (b) The log file for the eFIND R\*-tree  
(Figure 2)

<i>log#</i>	<i>page_id</i>	<i>h</i>	<i>type_mod</i>	<i>result</i>
1	N <sub>1</sub>	0	NEW	-
2	N <sub>1</sub>	0	MOD	p <sub>13</sub>
3	L <sub>1</sub>	0	DEL	-
4	L <sub>1</sub>	0	NEW	-
5	L <sub>1</sub>	0	MOD	p <sub>8</sub>
6	L <sub>1</sub>	0	MOD	p <sub>18</sub>
7	L <sub>1</sub>	0	MOD	p <sub>3</sub>
8	L <sub>2</sub>	0	DEL	-
9	L <sub>2</sub>	0	NEW	-
10	L <sub>2</sub>	0	MOD	p <sub>16</sub>
11	L <sub>2</sub>	0	MOD	p <sub>19</sub>
12	L <sub>2</sub>	0	MOD	p <sub>6</sub>
13	I <sub>6</sub>	1	MOD	(L <sub>2</sub> , MBR(L <sub>2</sub> ))
14	N <sub>2</sub>	1	NEW	-
15	N <sub>2</sub>	1	MOD	(L <sub>4</sub> , MBR(L <sub>4</sub> ))
16	I <sub>6</sub>	1	DEL	-
17	I <sub>6</sub>	1	NEW	-
18	I <sub>6</sub>	1	MOD	(L <sub>1</sub> , MBR(L <sub>1</sub> ))
19	I <sub>6</sub>	1	MOD	(L <sub>3</sub> , MBR(L <sub>3</sub> ))

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

<i>log#</i>	<i>page_id</i>	<i>header</i>	<i>h</i>	<i>type_mod</i>	<i>result</i>
1	N <sub>1</sub>	-	0	NEW	-
2	N <sub>1</sub>	-	0	MOD	p <sub>19</sub>
3	N <sub>1</sub>	-	0	MOD	p <sub>8</sub>
4	L <sub>5</sub>	-	0	DEL	-
5	L <sub>5</sub>	-	0	NEW	-
6	L <sub>5</sub>	-	0	MOD	p <sub>18</sub>
7	L <sub>5</sub>	-	0	MOD	p <sub>3</sub>
8	L <sub>5</sub>	-	0	MOD	p <sub>6</sub>
9	I <sub>2</sub>	(0,0)-75	1	MOD	(N <sub>1</sub> , MBR(N <sub>1</sub> ))
10	I <sub>2</sub>	(0,0)-75	1	MOD	(L <sub>5</sub> , MBR(L <sub>5</sub> ))
11	L <sub>2</sub>	-	0	MOD	p <sub>20</sub>
12	L <sub>5</sub>	-	0	MOD	(p <sub>6</sub> , ∅)
13	L <sub>2</sub>	-	0	MOD	(p <sub>2</sub> , ∅)

(d) The log file for the eFIND xBR<sup>+</sup>-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<sup>+</sup>-tree (d).

---

**566 4.2 General Algorithms**

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

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

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

---

**601 5 Porting Disk-based Spatial Indices to SSDs**

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

---

610    5.1 Insert Operations

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

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

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

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

---

**Algorithm 1:** Inserting an entry into a spatial index

---

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

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

- 1 let  $E'$  be an entry;
- 2 **if**  $O$  is a delete operation **then**
  - 3    let  $E'$  become NULL;
- 4 **else**
  - 5    let  $E'$  point to  $E$ ;
- 6 append the new log entry  $\langle N_{id}, (\text{metadata}(N), \text{height}(N), (\text{MOD}, (\text{key}(E), E'))) \rangle$  into the log file;
- 7 let  $WBEntry$  be the hash entry of  $N$  in the *Write Buffer Table*;
- 8 **if**  $WBEntry$  is not NULL **then**
  - 9    **if** the *mod-tree* of  $WBEntry$  contains an element with key equal to  $\text{key}(E)$  **then**
    - 10      replace it by the element  $(\text{key}(E), E')$ ;
  - 11    **else**
    - 12      insert the element  $(\text{key}(E), E')$  into *mod-tree* of  $WBEntry$ ;
  - 13    update the value of *timestamp* of  $WBEntry$  to  $\text{now}()$ ;
  - 14    increase the value of *mod\_count* of  $WBEntry$  by 1;
- 15 **else**
  - 16    set  $WBEntry$  to the hash entry  $\langle N_{id}, (\text{metadata}(N), (\text{now}(), \text{height}(N), 1), (\text{MOD}, \text{emptyRBTree}())) \rangle$ ;
  - 17    store  $WBEntry$  in the *Write Buffer Table*;
  - 18    insert the element  $(\text{key}(E), E')$  into *mod-tree* of  $WBEntry$ ;
- 19 **if** *Write Buffer Table* is full **then**
  - 20      execute a flushing operation (as detailed in [15]);

---

657       Algorithm 2 shows how the extended eFIND processes a node modification.  
 658       Its inputs are the type of modification to be handled ( $O$ ), the entry ( $E$ ) being  
 659       manipulated, and its node ( $N$ ). This algorithm is employed to execute the cases  
 660       (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 accommodation 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  $\langle N_{id}, (\text{metadata}(N), \text{height}(N), (\text{NEW}, \text{NULL})) \rangle$  into the
   log file;
2 let  $WBEntry$  be the hash entry of  $N$  in the Write Buffer Table;
3 if  $WBEntry$  is not  $\text{NULL}$  then
4   if the status of  $WBEntry$  is equal to  $\text{DEL}$  then
5     set the status and  $\text{mod\_tree}$  of  $WBEntry$  to  $\text{NEW}$  and  $\text{emptyRBTee}()$ ,
       respectively;
6 else
7   let  $WBEntry$  become the hash entry
       $\langle N_{id}, (\text{metadata}(N), (\text{now}(), \text{height}(N), 1), (\text{NEW}, \text{emptyRBTee}())) \rangle$ 
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
         $\langle N_{id}, (\text{metadata}(N), \text{height}(N), (\text{MOD}, (\text{key}(E), E))) \rangle$  into the log file;
12     insert the element  $(\text{key}(E), E)$  into  $\text{mod\_tree}$  of  $WBEntry$ ;
13     increase the value of  $\text{mod\_count}$  of  $WBEntry$  by 1;
14   update the value of  $\text{timestamp}$  of  $WBEntry$  to  $\text{now}()$ ;
15 if Write Buffer Table is full then
16   execute a flushing operation (as detailed in [15]);

```

---

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

713 **Complexity Analysis.** Our goal is not to analyze the complexity of algorithms  
714 belonging to the underlying spatial index since it goes beyond the scope of this  
715 article (see [51; 4] for complexity analysis of R-trees). In this sense, we analyze the  
716 complexity of Algorithms 2 to 4 as follows. The time complexity of Algorithm 2  
717 can be determined by  $C_{alg2} = \mathcal{W}_s + \mathcal{H} + \mathcal{O}(\log n)$ , where  $\mathcal{W}_s$  is the average cost  
718 of one sequential write to the SSD in order to log the modification,  $\mathcal{H}$  refers to  
719 the cost of accessing an element from the hash table that implements the write  
720 buffer (i.e., which is usually  $\mathcal{O}(1)$ ), and  $\mathcal{O}(\log m)$  is the average cost of updating  
721 an element of the red-black tree with  $m$  elements. Note that red-black trees have  
722 an amortized update cost, as discussed in [46], which is particularly useful for  
723 implementing the write buffer. In addition, the time complexity of Algorithm 2  
724 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*<sub>2</sub> (*log#* 13 in Figure 7c and the sixth line in Figure 5c). The second 2-to-3 split occurs when inserting the node *N*<sub>1</sub> into the node *I*<sub>6</sub>. Thus, it creates the new node *N*<sub>2</sub> containing the entry pointing to *L*<sub>4</sub> (*log#* 14 and 15 in Figure 7c and the eighth line in Figure 5c), and then redistributes the entries among the nodes *I*<sub>6</sub> and *I*<sub>7</sub> (*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*<sub>2</sub>, another new node is created, named *N*<sub>3</sub> (*log#* 26 and 27 in Figure 7c and the fourth line in Figure 5c). Then, two entries of the node *I*<sub>1</sub> are adjusted accordingly (*log#* 28 and 29 in Figure 7c and the first line in Figure 5c), concluding the insertion of the point *p*<sub>19</sub>. The insertion of the point *p*<sub>20</sub> requires the creation of a new corresponding entry in the node *L*<sub>6</sub> (*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*<sub>9</sub> (*log#* 31 in Figure 7c and the ninth line in Figure 5c).

- **The xBR<sup>+</sup>-tree (Figure 4a).** To insert the point *p*<sub>19</sub>, the new sub-quadrant 00\* that also accommodates the point *p*<sub>8</sub> 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*<sub>5</sub>, which then stores the points *p*<sub>18</sub>, *p*<sub>3</sub>, and *p*<sub>6</sub> (*log#* 4 to 8 in Figure 7d and the fourth line in Figure 5d). The node *I*<sub>2</sub> is modified to accommodate the newly created node and to store the adjusted DBR of the node *L*<sub>5</sub> (*log#* 9 and 10 in Figure 7d and the first line in Figure 5d). The point *p*<sub>20</sub> is directly inserted into the node *L*<sub>2</sub> (*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<sup>+</sup>-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}*, 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  $(\{L\}, \text{NULL})$ ;
- 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;

---

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

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

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

843 **Handling nodes with eFIND.** The execution of Algorithm 5 can invoke four  
 844 specialized algorithms of eFIND to manipulate nodes of the underlying index in  
 845 the following cases: (i) the retrieval of nodes (line 1), (ii) the direct deletion of  
 846 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;

---

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

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

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

874 **Complexity Analysis.** The complexity analysis of Algorithm 5 depends on the  
 875 underlying index being ported. Hence, we focus on understanding the complexity  
 876 of Algorithms 6 and 7. The time complexity of Algorithm 6 is similar to the  
 877 complexity of Algorithm 3 (Section 5.1). In the worst case, its complexity is given  
 878 by  $C_{alg6} = C_{alg7} + pC_{alg7} + pC_{alg4}$ , where  $p$  is the number of nodes in  $D$ . The time  
 879 complexity of Algorithm 7 is given by  $C_{alg7} = \mathcal{W}_s + \mathcal{H} + \mathcal{F}$ , where  $\mathcal{F}$  refers to the  
 880 cost of freeing the red-black tree of the deleted node, if any. In addition, the time  
 881 complexity of Algorithm 7 can also include the cost of a flushing operation, as  
 882 detailed in [15]. As for the space complexity, Algorithm 6 does not require extra  
 883 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

- 1 append the new log entry  $\langle N_{id}, (\text{metadata}(N), \text{height}(N), (\text{DEL}, \text{NULL})) \rangle$  into the log file;
- 2 let  $WBEntry$  be the hash entry of  $N$  in the *Write Buffer Table*;
- 3 **if**  $WBEntry$  is not  $\text{NULL}$  **then**
- 4     free its  $\text{mod\_tree}$ , if any;
- 5     set the status of  $WBEntry$  to  $\text{DEL}$ ;
- 6     increase the value of  $\text{mod\_count}$  of  $WBEntry$  by 1;
- 7     update the value of  $\text{timestamp}$  of  $WBEntry$  to  $\text{now}()$ ;
- 8 **else**
- 9     set  $WBEntry$  to the hash entry  
     $\langle N_{id}, (\text{metadata}(N), (\text{now}(), \text{height}(N), 1), (\text{DEL}, \text{NULL})) \rangle$ ;
- 10    store  $WBEntry$  in the *Write Buffer Table*;
- 11 **if** *Write Buffer Table* is full **then**
- 12    execute a flushing operation (as detailed in [15]);

---

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

- 889 – **The R-tree.** To delete the point  $p_6$ , it processes an underflow operation on the  
 890 node  $L_2$ , deleting it ( $\log\# 11$  and  $12$  in Figure 7a and the seventh and second  
 891 lines in Figure 5a) and adjusting the MBR of the entry pointing to the node  
 892  $I_3$  ( $\log\# 13$  in Figure 7a and the first line in Figure 5a). Then, the point  $p_8$  is  
 893 reinserted into the R-tree in the node  $L_1$  ( $\log\# 14$  in Figure 7a and the fifth  
 894 line in Figure 5a). This reinsertion provokes one adjustment in its parent entry  
 895 ( $\log\# 15$  in Figure 7a and the second line in Figure 5a) and another adjustment  
 896 in an entry of the node  $I_1$  ( $\log\# 16$  in Figure 7a and the first line in Figure 5a).  
 897 The point  $p_2$  is directly removed from the node  $L_8$  that has its MBR adjusted  
 898 ( $\log\# 17$  and  $18$  in Figure 7a and the last and third lines in Figure 5a).
- 899 – **The R\*-tree.** Similarly to the R-tree, it processes an underflow operation on  
 900 the node  $L_1$  to delete the point  $p_6$  ( $\log\# 11$  and  $12$  in Figure 7b and the  
 901 eighth and second lines in Figure 5b), adjusting its parent entry ( $\log\# 13$  in  
 902 Figure 7b and the first line in Figure 5b). Next, it reinserts the point  $p_{13}$  into  
 903 the R\*-tree ( $\log\# 14$  in Figure 7b and the ninth line in Figure 5b), requiring  
 904 two adjustments in the upper levels of the tree ( $\log\# 15$  and  $16$  in Figure 7b  
 905 and the third and first lines in Figure 5b). The deletion of the point  $p_2$  is  
 906 directly performed on the node  $L_8$  ( $\log\# 17$  in Figure 7b and the last line in  
 907 Figure 5a), which has its corresponding parent entry adjusted afterwards ( $\log\#$   
 908  $18$  in Figure 7b and the fourth line in Figure 5b).
- 909 – **The Hilbert R-tree.** It deletes the point  $p_6$  from the node  $L_2$  ( $\log\# 32$  in  
 910 Figure 7c and the eleventh line in Figure 5c), adjusting the MBR of entries in  
 911 the two levels upwards ( $\log\# 33$  and  $34$  in Figure 7c and the sixth and third  
 912 lines in Figure 5c). Then, it deletes the node  $L_7$  when removing the point  $p_2$   
 913 ( $\log\# 35$  and  $36$  in Figure 7c and the last line in Figure 5c). This consequently  
 914 provokes the adjustment of entries in the nodes  $I_4$  and  $I_2$  ( $\log\# 37$  and  $38$  in  
 915 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  $R$ ;

```

---

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

## 919 5.3 Search Operations

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

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

938 Algorithm 9 specifies the procedure employed by eFIND to retrieve a node  
 939 and is equivalent to the algorithm presented in [15]. We included this algorithm  
 940 in the article for completeness purposes. First, the algorithm takes the identifier  
 941 of a node as input and returns the most recent version of this node. There are  
 942 three alternative cases. The first one is whether the node is stored in the *Write*  
 943 *Buffer Table* with status equal to NEW or DEL (lines 1 and 2); thus, it is directly  
 944 returned by using the pointer stored in the write buffer since it does not contain

945 further modifications (line 3). The second case refers to a not modified node; the  
 946 algorithm verifies if this node contains a cached version in the *Read Buffer Table*  
 947 (lines 4 to 7), avoiding a read operation to be performed on the SSD (returning  
 948 the node in line 14). Otherwise, the node is read from the SSD and inserted in  
 949 the *Read Buffer Table* (lines 8 to 10, and returning the node in line 14). In both  
 950 cases, the *Read Buffer Table* is possibly reorganized by the read buffer replacement  
 951 policy (line 11). The last case is if the node has modifications stored in the write  
 952 buffer. Here, a merge operation is needed in order to combine the entries stored in  
 953 the modification tree and the existing entries of the node (lines 12 and 13). After  
 954 applying this merging, the algorithm returns the most recent version of the node  
 955 (line 14).

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

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

989 **Complexity Analysis.** Since the complexity of Algorithm 8 depends on the un-  
 990 derlying index, we focus on analyzing the complexity of Algorithms 9 and 10. The  
 991 time complexity of Algorithm 9, in the best case, is the cost of accessing the hash  
 992 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  $I$

- 1 let  $WBEntry$  be the hash entry in the *Write Buffer Table* with key  $I$ ;
- 2 if  $WBEntry$  has status equal to NEW or DEL then
  - 3 return the node pointed by  $WBEntry$ ;
- 4 let  $N$  be an empty node;
- 5 let  $RBEntry$  be the hash entry in the *Read Buffer Table* with key  $I$ ;
- 6 if  $RBEntry$  is not NULL then
  - 7 let  $N$  become the node pointed by  $RBEntry$ ;
- 8 else
  - 9 let  $N$  become its version read from the SSD;
  - 10 insert  $N$  into the *Read Buffer Table* and its identifier in the *RQ*;
- 11 apply the read buffer replacement policy;
- 12 if  $WBEntry$  has status equal to MOD then
  - 13 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;
- 14 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

- 1 let  $i$  and  $j$  be two integers equal to 0;
- 2 let  $N$  be an empty node;
- 3 while  $i < \text{length}(L_1)$  and  $j < \text{length}(L_2)$  do
  - 4 let  $r$  become the result of the comparison function between  $L_1[i]$  and  $L_2[j]$  according to structural constraints and properties of  $SI$ ;
  - 5 if  $r < 0$  then
    - 6 append  $L_1[i]$  into  $N$  and increment  $i$  by 1;
  - 7 else if  $r > 0$  then
    - 8 append  $L_2[j]$  into  $N$  and increment  $j$  by 1;
  - 9 else
    - 10 append  $L_1[i]$  into  $N$  and increment both  $i$  and  $j$  by 1;
- 11 for  $i$  to  $\text{length}(L_1)$  by 1 do
  - 12 append  $L_1[i]$  into  $N$ ;
- 13 for  $j$  to  $\text{length}(L_2)$  by 1 do
  - 14 append  $L_2[j]$  into  $N$ ;
- 15 return  $N$ ;

---

993 time complexity of Algorithm 9 is given by  $C_{alg9} = 2\mathcal{H} + \mathcal{R} + C_{alg10}$ , where  $\mathcal{R}$  refers  
 994 to the average cost of a read operation to the SSD. Note that Algorithm 9 may  
 995 have the time complexity of  $2\mathcal{H}$  or  $2\mathcal{H} + \mathcal{R}$  (i.e., they occur if the node has not  
 996 modification). The time complexity of  $C_{alg10}$  can be determined by  $\mathcal{O}(l_1 + l_2)$ , where  
 997  $l_1$  and  $l_2$  represent the number of entries stored in the main memory and in the  
 998 SSD, respectively. Recall that the use of the comparison function defined by the  
 999 underlying index (Section 4.1), which checks the order of entries, also impacts the  
 1000 complexity of Algorithm 10.

1001 As for the space complexity, Algorithm 9 does not require extra space. On the  
 1002 other hand, Algorithm 10 requires additional memory to keep the merged  $c$  entries  
 1003 of the node. Thus, it can assume the space complexity  $\mathcal{O}(c)$ .

1004 **Examples of Execution.** Our running example executes one IRQ in each ported  
 1005 spatial index, after applying the insertions (Section 5.1) and deletions (Section 5.2).  
 1006 Algorithm 8 is employed to execute this IRQ in each spatial index, resulting in  
 1007 the following sequence of operations:

- 1008 – **The R-tree.** It starts reading the its root node  $R$  from the *Read Buffer Table*  
 1009 (first line in Figure 6a). Then, it descends the tree by accessing the node  $I_1$   
 1010 since the IRQ intersects its MBR. For this, a merging operation (Algorithm 10)  
 1011 between the entries stored in the *mod\_tree* of the  $I_1$  and the entries stored in  
 1012 the SSD is performed, resulting in the most recent version of this node. That  
 1013 is, this merge operation returns the node containing the modified version of  
 1014 the entry  $I_3$  (stored in the first line in Figure 5a) and the stored version of the  
 1015 entry  $I_4$ . Next, the node  $I_3$  is read from the SSD, which has also modifications  
 1016 stored in its corresponding *mod\_tree* to be merged (Algorithm 10). Afterward,  
 1017 the leaf node  $N_1$  is directly accessed from the *Write Buffer Table* since it is a  
 1018 newly created node. It stores the point  $p_1$  in the result of the spatial query.  
 1019 Then, recursively the node  $I_4$  is read from the SSD because its MBR also  
 1020 intersects the query window of the IRQ. The last accessed node is  $L_4$ , read  
 1021 from the SSD. Then, the point  $p_5$  is appended to the final result of the query.
- 1022 – **The R\*-tree.** It firstly reads the root node  $R$  and then its child node  $I_1$ , both  
 1023 stored in the *Read Buffer Table* (first two lines in Figure 6b). Next, it accesses  
 1024 the node  $I_4$ . For retrieving this node, a merging operation (Algorithm 10) is  
 1025 performed to integrate the modified entries stored in the *Write Buffer Table*  
 1026 (third line in Figure 5b) and the stored entries. Then, the node  $L_3$  is read  
 1027 from the SSD since it does not contain modifications. From this node, the  
 1028 point  $p_5$  is added to the result. Afterward, its sibling node  $L_4$  is retrieved by  
 1029 performing the merging operation (considering the modified entry in the ninth  
 1030 line in Figure 5b), adding the point  $p_1$  to the result.
- 1031 – **The Hilbert R-tree.** Starting from the root node  $R$ , it descends the tree by  
 1032 accessing the node  $I_1$ . These nodes are retrieved from the *Read Buffer Table*  
 1033 (first two lines in Figure 6c). Then, two paths are followed. The first path  
 1034 descends the tree by retrieving the nodes  $I_3$ ,  $I_7$ , and  $L_3$ . Expect for the node  
 1035  $L_3$  that is read from the SSD, the remaining nodes have modifications merged  
 1036 (Algorithm 10) to their stored versions (using the third and seventh lines in  
 1037 Figure 5c). After reading the leaf node of this path, it adds the point  $p_1$  to the  
 1038 result of the spatial query. The second path accesses the newly created nodes  
 1039  $N_3$  and  $N_4$  directly from the *Write Buffer Table* (fourth and eighth lines in  
 1040 Figure 5c), and then retrieve the node  $L_4$  that is read from the SSD. It finishes  
 1041 by adding the point  $p_5$  to the result.
- 1042 – **The xBR<sup>+</sup>-tree.** It follows a single path to solve the spatial query. It starts  
 1043 from the root node  $R$  and then reads the node  $I_1$ , both cached in the *Read*  
 1044 *Buffer Table* (first two lines in Figure 6d). Next, the leaf nodes  $L_1$  and  $L_3$  are  
 1045 accessed because their data bounding rectangles intersect the query window of  
 1046 the IRQ. Since they do not have modifications and are not cached in the read  
 1047 buffer, they are directly read from the SSD. After accessing each leaf node, the  
 1048 points  $p_1$  and  $p_5$  returned as result.

---

**1049 6 Experimental Evaluation**

1050 In this section, we empirically measure the efficiency of porting disk-based spatial  
1051 index structures by using our systematic approach. For this, we port the R-tree,  
1052 the R\*-tree, the Hilbert R-tree, and the xBR<sup>+</sup>-tree by using eFIND and FAST. It  
1053 shows that our systematic approach can be deployed by using different frameworks.  
1054 In particular, FAST-based spatial indices are considered the main competitors of  
1055 the eFIND-based spatial indices, which were discussed in this article. To create the  
1056 FAST-based spatial indices, we adapted the FAST's data structures and algorithms  
1057 in a similar way to the adaptations performed on eFIND. Section 6.1 shows the  
1058 experimental setup. Performance results when building spatial indices, performing  
1059 spatial queries, and computing mixed operations are discussed in Sections 6.2, 6.3,  
1060 and 6.4, respectively.

1061 **6.1 Experimental Setup**

1062 **Datasets.** We used four spatial datasets, stored in PostGIS/PostgreSQL [50]. Two  
1063 of them contain real data collected from OpenStreetMaps following the methodology  
1064 in [12]. The first one is a real spatial dataset, called *brazil\_points2019*, con-  
1065 taining 2,139,087 points inside Brazil (approximately, 156MB). The second one,  
1066 called *us\_midwest\_points2019*, contains 2,460,597 points inside the Midwest of the  
1067 USA (approximately, 180MB). The other two spatial datasets are synthetic, called  
1068 *synthetic1* and *synthetic2*, containing respectively 5 and 10 million points (approx-  
1069 imately, 326MB and 651MB, respectively). Each synthetic dataset stores points  
1070 equally distributed in 125 clusters uniformly distributed in the range  $[0, 1]^2$ . The  
1071 points in each cluster (i.e., 40,000 points for *synthetic1* and 80,000 points for *syn-*  
1072 *thetic2*) were located around the center of each cluster, according to Gaussian dis-  
1073 tribution. It follows the same methodology as the experiments conducted in [55].  
1074 The use of spatial datasets with different characteristics and volume allows us to  
1075 analyze the spatial indices under distinct scenarios.

1076 **Configurations.** We employed our systematic approach to creating different con-  
1077 figurations of the ported spatial index structures based on the frameworks eFIND  
1078 and FAST. As a result, we evaluated the following flash-aware spatial indices:  
1079 (i) the *eFIND R-tree*, (ii) the *eFIND R\*-tree*, (iii) the *eFIND Hilbert R-tree*, (iv)  
1080 the *eFIND xBR<sup>+</sup>-tree*, (v) the *FAST R-tree*, (vi) the *FAST R\*-tree*, (vii) the *FAST*  
1081 *Hilbert R-tree*, and (viii) the *FAST xBR<sup>+</sup>-tree*. The R-tree used the quadratic split  
1082 algorithm, the R\*-tree employed the reinsertion policy of 30%, and the Hilbert  
1083 R-tree leveraged the 2-to-3 split policy. We varied the employed node (i.e., page)  
1084 sizes from 2KB to 16KB. The buffer and log sizes were 512KB and 10MB, respec-  
1085 tively. We employed the best parameter values of FAST, as reported in [59]: the  
1086 FAST\* flushing policy. We also employed the best parameter values of eFIND,  
1087 as reported in [15]: the use of 60% of the oldest modified nodes to create flushing  
1088 units, the flushing policy using the height of nodes as weight, the allocation of 20%  
1089 of the buffer for the read buffer, and flushing unit size equal to 5. Hence, we built  
1090 and evaluated 32 different configurations. We did not include non-ported spatial  
1091 indices (e.g., original R-tree) since other works in the literature have shown that  
1092 the number of reads and writes of such indices is high and negatively impact on  
1093 the SSD performance [63; 59; 34; 13].

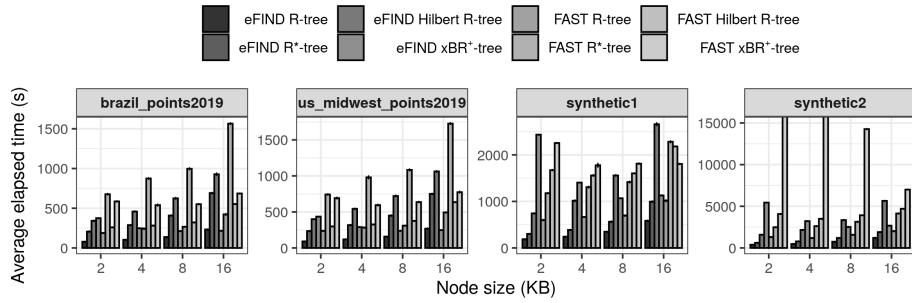
**1094 Workloads.** We executed three types of workloads on each spatial dataset: (i)  
**1095** index construction by inserting point objects one-by-one, (ii) execution of 1,000  
**1096** point queries and 3,000 intersection range queries (IRQs), and (iii) execution of  
**1097** insertions and queries. A point query returns the points that are equal to a given  
**1098** point. An IRQ retrieves the points contained in a given rectangular query window,  
**1099** including its borders. Three different sets of query windows were used, represent-  
**1100** ing respectively 1,000 rectangles with 0.001%, 0.01%, and 0.1% of the area of  
**1101** the total extent of the dataset being used by the workload. We generated differ-  
**1102** ent query windows for each dataset using the algorithms described in [12]. This  
**1103** method allows us to measure the performance of spatial queries with distinct se-  
**1104** lectivity levels. We consider the selectivity of a spatial query as the ratio of the  
**1105** number of returned objects and the total objects; thus, the three sets of query  
**1106** windows built IRQs with low, medium, and high selectivity, respectively. For each  
**1107** configuration and dataset, the workloads were executed 5 times. We avoided the  
**1108** page caching of the system by using direct I/O. For computing statistical values of  
**1109** insertions, we collected the average elapsed time. For computing statistical values  
**1110** of spatial queries, we calculated the average elapsed time to execute each set of  
**1111** query windows.

**1112 Running Environment.** We employed a server equipped with an Intel Core® i7-  
**1113** 4770 with a frequency of 3.40GHz and 32GB of main memory. We made use of  
**1114** two SSDs: (i) Kingston V300 of 480GB, and (ii) Intel Series 535 of 240GB. The  
**1115** Intel SSD is a high-end SSD that provides faster reads and writes than the low-  
**1116** end Kingston SSD. We employed the Intel SSD to execute all the workloads and  
**1117** configurations. This provided us an overview of the performance behavior of the  
**1118** underlying framework implementing our systematic approach. Next, we used the  
**1119** Kingston SSD to compare eFIND-based configurations, allowing us to analyze the  
**1120** performance of eFIND-based spatial indices by considering different architectures  
**1121** of SSDs. The operating system used was Ubuntu Server 14.04 64 bits. We also  
**1122** used FESTIval [17] to execute the workloads.

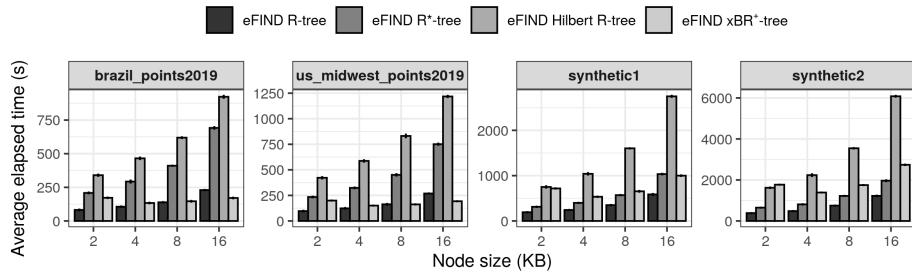
## 1123 6.2 Building Spatial Indices

**1124** Figure 8 shows that eFIND fits well in our systematic approach since a particu-  
**1125** lar disk-based spatial index ported by eFIND provided better performance than  
**1126** the same disk-based spatial index ported by FAST. The eFIND R-tree delivered  
**1127** the best results in most cases, followed by the eFIND xBR<sup>+</sup>-tree, which provided  
**1128** the second-best results. Compared to the FAST R-tree, the eFIND R-tree showed  
**1129** performance gains ranging from 40% to 70.3%. A performance gain shows how  
**1130** much a configuration reduced the elapsed time from another configuration. We  
**1131** highlight the long processing times of the FAST xBR<sup>+</sup>-tree (mainly in the dataset  
**1132** *synthetic2*) due to the complexity of adapting FAST to deal with the special con-  
**1133** straints of the xBR<sup>+</sup>-tree, as discussed in [16]. Since the eFIND-based spatial in-  
**1134** dices provided the best results, our analysis focuses on detailing their performance  
**1135** behavior, including experiments conducted in the Kingston SSD.

**1136** Figure 9 depicts the performance results obtained in the Kingston SSD. We  
**1137** can note that the underlying characteristics of the ported index structures (Sec-  
**1138** tion 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<sup>+</sup>-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<sup>+</sup>-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.

1139 two different behaviors were observed. Compared to the other eFIND-based spa-  
 1140 tial indices and considering the node sizes from 2KB to 8KB, the eFIND R-tree  
 1141 provided performance gains from 33.8% to 79.1% for the Intel SSD and from 5.2%  
 1142 to 80.4% for the Kingston SSD. On the other hand, for the node size equal to  
 1143 16KB, the eFIND xBR<sup>+</sup>-tree overcame the eFIND R-tree with reductions up to  
 1144 7.6% for the Intel SSD and up to 28.3% for the Kingston SSD. Analyzing the cost  
 1145 of building spatial indices using this size is particularly useful when considering  
 1146 the spatial query processing (see Section 6.3).

1147 As for the synthetic spatial datasets, the eFIND R-tree was the fastest spatial  
 1148 index in both SSDs. Its performance gains against the other eFIND-based spatial  
 1149 indices were very expressive. It ranged from 36.4% to 92.9% for the Intel SSD  
 1150 (Figure 8), and from 37.6% to 79.9% for the Kingston SSD (Figure 9).

1151 The poor performance of the eFIND R\*-tree and the eFIND Hilbert R-tree  
 1152 is related to the management of overflowed nodes. The overhead of the eFIND  
 1153 R\*-tree is due to its reinsertion policy, requiring more reads in insert operations  
 1154 compared to the R-tree. As discussed in the literature (see Section 2), the excessive  
 1155 number of reads impairs the performance of applications in SSDs. Concerning the  
 1156 eFIND Hilbert R-tree, its bad performance is because of the redistribution policy.

1157 It is comparable to the cost of a split operation of the R-tree since  $s$  sibling  
 1158 nodes should be written together with a possible adjustment of their parent node.  
 1159 Further, the split operation of the eFIND Hilbert R-tree possibly requires four  
 1160 writes because of the 2-to-3 split policy. Thus, the eFIND Hilbert R-tree required  
 1161 long processing times to build spatial indices in both SSDs.

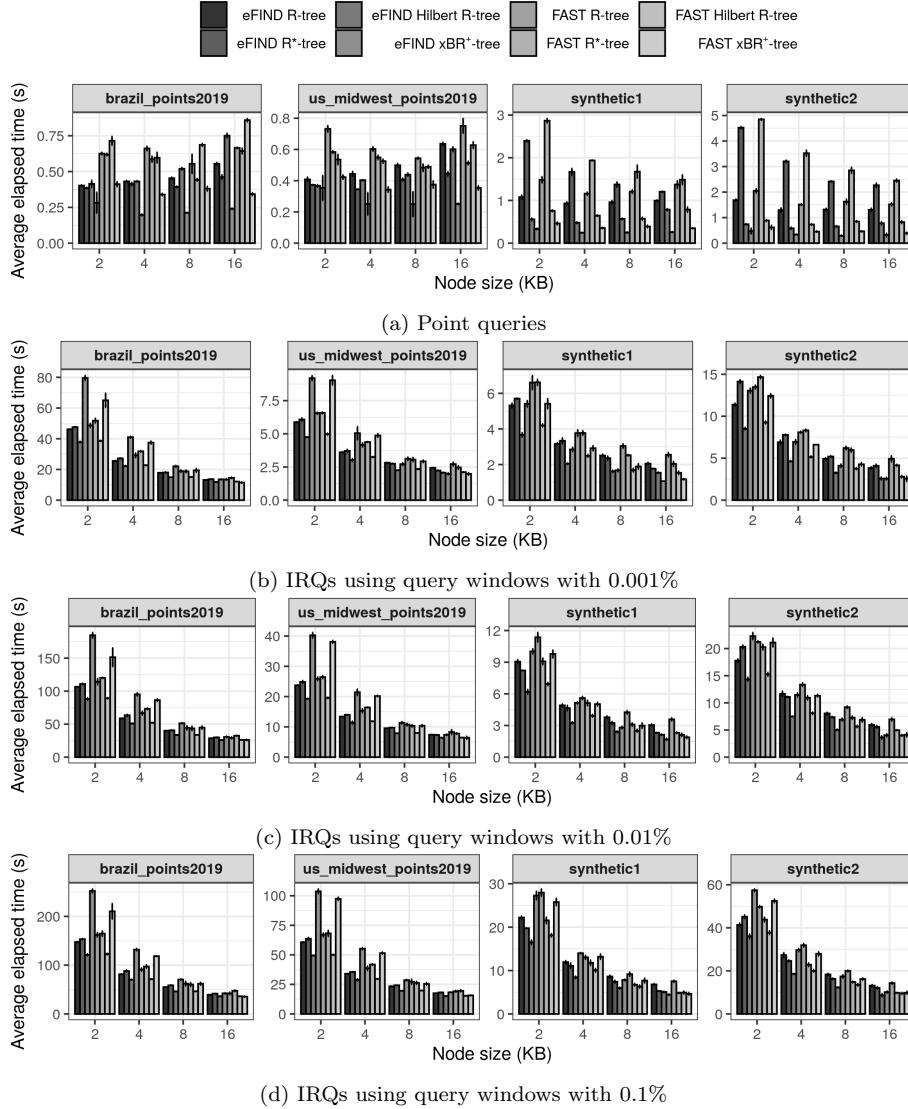
1162 Another important observation is that the special constraints of the underlying  
 1163 index may impair the performance when retrieving nodes by using the eFIND's  
 1164 algorithms and data structures. For instance, the requirement of a sophisticated  
 1165 comparison function to guarantee the sorting property among entries of internal  
 1166 and leaf nodes. We note this influence when analyzing the experimental results of  
 1167 the Hilbert R-tree and xBR<sup>+</sup>-tree. They require that nodes' entries are sorted by  
 1168 their Hilbert values and directional digits, respectively. eFIND makes use of this  
 1169 comparison function every time that a modified node is recovered by the index  
 1170 (Algorithm 9). Hence, it mainly impacts the performance of the insertions. To  
 1171 improve it, there are efforts in the literature that propose specific bulk-insertions  
 1172 and bulk-loading algorithms. For xBR<sup>+</sup>-trees, examples of such algorithms are  
 1173 given in [57].

1174 Several configurations presented the best results by employing the node size  
 1175 of 2KB. This is due to the high cost of writing flushing units with larger index  
 1176 pages (e.g., 16KB) since a write made on the application layer can be split into  
 1177 several internal writes to the SSD. Further, the data volume also impacted the  
 1178 construction time, as expected. Hence, building flash-aware spatial indices required  
 1179 more time as the node size and the data volume also increased.

### 1180 6.3 Query Processing

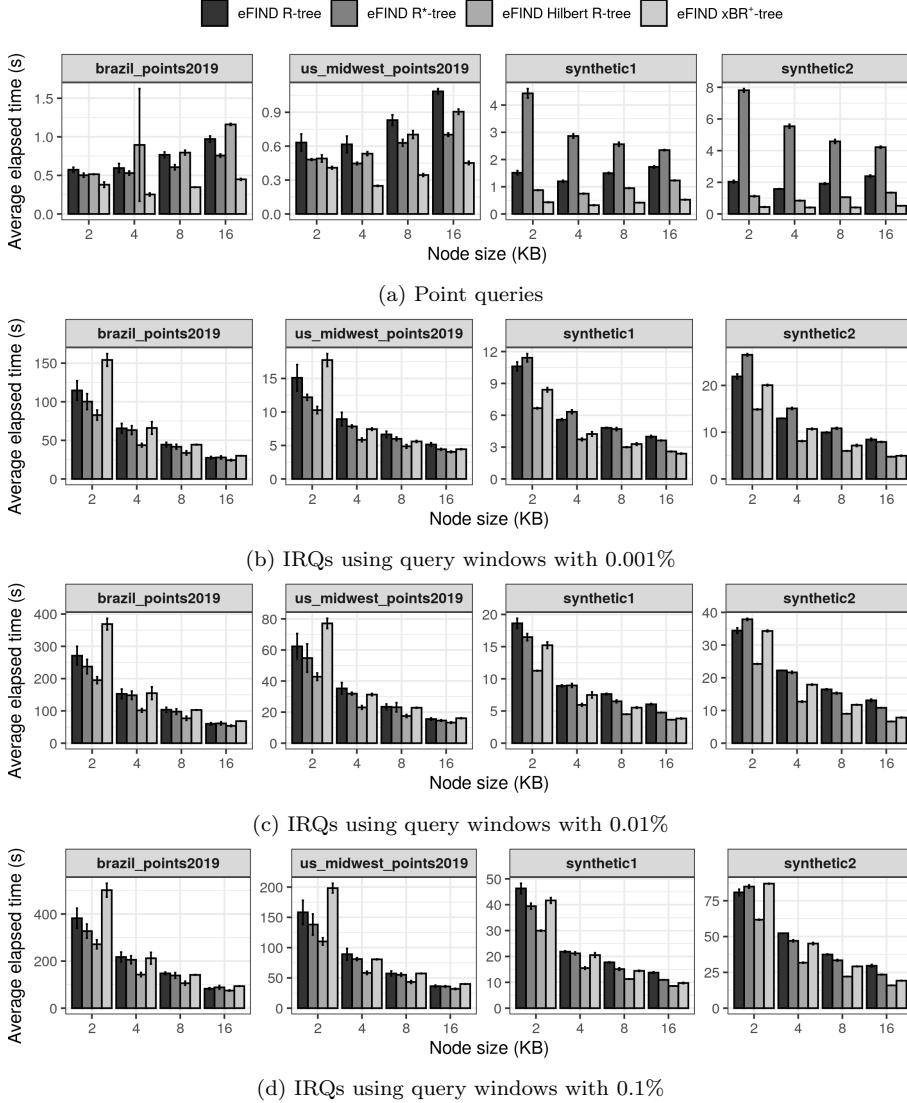
1181 Figure 10 shows that eFIND-based spatial indices outperformed their correspond-  
 1182 ing FAST-based spatial indices. The eFIND xBR<sup>+</sup>-tree delivered the best results  
 1183 when processing the point queries, whereas the eFIND Hilbert R-tree, in most  
 1184 cases, provided the best results when processing the IRQs. Note that the FAST  
 1185 xBR<sup>+</sup>-tree delivered the best performance results among the FAST-based spa-  
 1186 tial indices to process the point queries. This reveals that the space partitioning  
 1187 strategy of the xBR<sup>+</sup>-tree distinguishes itself by delivering lesser elapsed times for  
 1188 computing point queries on SSDs. To process the point queries, the eFIND xBR<sup>+</sup>-  
 1189 tree showed performance gains ranging from 16.4% to 44.2%, if compared to the  
 1190 FAST xBR<sup>+</sup>-tree. As for the IRQs, the eFIND Hilbert R-tree showed reductions  
 1191 up to 17.6%, 17%, and 16.3% for the low, medium, and high selectivity levels,  
 1192 respectively, if compared to the FAST Hilbert R-tree. Due to the superior per-  
 1193 formance of the eFIND-based spatial indices, our next analysis focuses on detailing  
 1194 their performance results, including experiments conducted in the Kingston SSD.

1195 Figure 11 shows the performance results when processing the spatial queries  
 1196 in the Kingston SSD. As for the point queries, the eFIND xBR<sup>+</sup>-tree showed  
 1197 performance gains from 3.6% to 89.5% for the Intel SSD and from 15% to 94.4%  
 1198 for the Kingston SSD, if compared to the other eFIND-based spatial indices. In  
 1199 general, a point query requires the traversal of a small number of paths in the tree.  
 1200 Thus, processing point queries using node sizes equal to 4KB and 8KB provided  
 1201 better results.



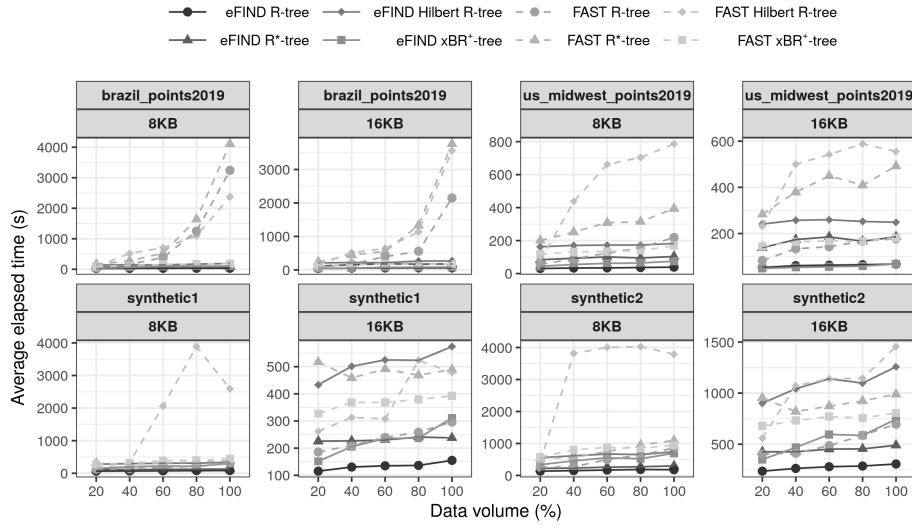
**Fig. 10** Performance results when executing the point queries and IRQs in the Intel SSD. It showed that the best results were delivered by the eFIND-based spatial indices.

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<sup>+</sup>-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<sup>+</sup>-tree outperformed the eFIND Hilbert R-tree.



**Fig. 11** Performance results when executing the point queries and IRQs in the Kingston SSD. As for the point queries, the eFIND xBR<sup>+</sup>-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<sup>+</sup>-tree delivered the best results.

1209 to process IRQs with low selectivity in most cases (Figure 10b), with performance  
 1210 gains up to 30.9%. On the other hand, the eFIND Hilbert R-tree imposed reduc-  
 1211 tions between 10.1% and 17.4% for the other selectivity levels (Figure 10c and d).  
 1212 For the Kingston SSD (Figure 11), the eFIND Hilbert R-tree was better than the  
 1213 eFIND xBR<sup>+</sup>-tree in the majority of cases by gathering reductions up to 18.9%,  
 1214 21.1%, and 20.2% for the low, medium, and high selectivity levels, respectively.



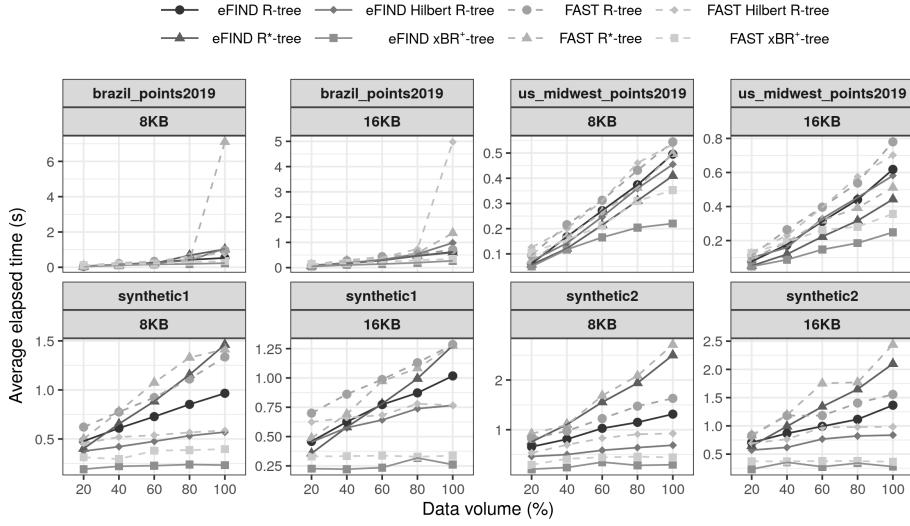
**Fig. 12** Performance results for inserting point objects by gradually increasing the data volume. Each spatial dataset and node size are showed in the header of each chart. In most cases, the eFIND R-tree provided the fastest processing time.

1215 In most cases, processing IRQs on the synthetic datasets required much less  
 1216 time than on the real datasets because of their specific spatial distribution. IRQs  
 1217 returning more points (i.e., with high selectivity) exhibited higher elapsed times.  
 1218 This is due to the traversal of multiple large nodes in the main memory, requiring  
 1219 more CPU time than queries with low selectivity. Hence, the performance behavior  
 1220 of IRQs is quite different from the performance behavior of the point queries.

#### 1221 6.4 Mixing Insertions and Queries

1222 In this section, we analyze the performance of the configurations to handle inser-  
 1223 tions and queries by gradually increasing the volume of the spatial dataset. To  
 1224 this end, we executed a workload that has three sequential steps; the workload  
 1225 sequentially (i) indexes 20% of the point objects stored in the spatial dataset, (ii)  
 1226 computes the point queries, and (iii) executes the IRQs. This sequence is repeated  
 1227 until all the point objects of the corresponding dataset are indexed. Thus, the  
 1228 workload has 5 phases of insertions and queries, where each phase means that the  
 1229 data volume increases 20%. We executed this workload by using the ported spatial  
 1230 indices with eFIND and FAST in the Intel SSD.

1231 Figures 12 to 14 depict the performance results considering the node sizes equal  
 1232 to 8KB and 16KB only. Thus, we can analyze the performance of the flash-aware  
 1233 spatial indices in each step of the workload, that is, the execution of insertions  
 1234 (Figure 12), point queries (Figure 13), and IRQs (Figure 14). The use of the node  
 1235 size equal to 8KB allows us to deliver a good balance between the performance of  
 1236 insertions and queries, whereas the node size equal to 16KB shows better perfor-  
 1237 mance when executing queries, such as discussed in Section 6.3.

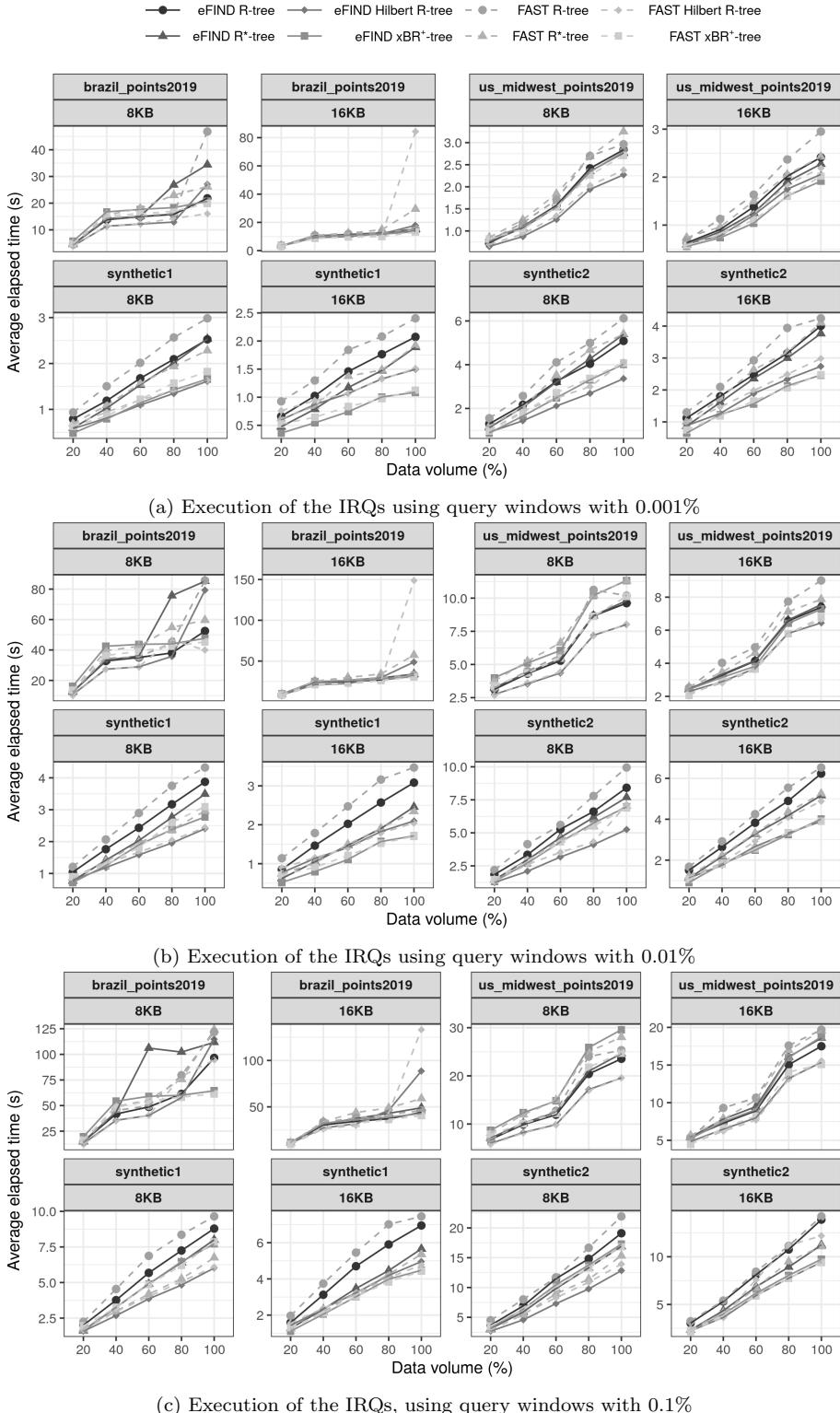


**Fig. 13** Performance results for executing point queries by gradually increasing the data volume. Each spatial dataset and node size are showed in the header of each chart. The point queries were executed after inserting the point objects (Figure 14). The eFIND xBR<sup>+</sup>-tree delivered the best 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<sup>+</sup>-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<sup>+</sup>-tree. A similar behavior indicates that the proposed approach to porting spatial index structures to SSDs is consistent when increasing the handled data volume.

## 1253 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<sup>+</sup>-tree.



**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<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-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<sup>+</sup>-tree was very efficient when processing point queries;
- The eFIND Hilbert R-tree, followed by the eFIND xBR<sup>+</sup>-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

1308 the behavior of real SSDs in the main memory. Future work also includes the  
1309 extension of our systematic approach to port spatial index structures to *non-volatile*  
1310 *main memories* (NVMM) like ReRAM, STT-RAM, and PCM [65]. These memories  
1311 are byte-addressable, allowing us to access persistent data with CPU load and  
1312 store instructions. Finally, the last topic of future work is to apply the proposed  
1313 systematic approach, with its integration with eFIND, to port one-dimensional  
1314 index structures to SSDs and NVMMs. This includes the generalization of data  
1315 structures and algorithms to deal with one- and multi-dimensional data.

1316 **Acknowledgements** This study was financed in part by the Coordenação de Aperfeiçoamento  
1317 de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work has also been  
1318 supported by CNPq and by the São Paulo Research Foundation (FAPESP). Cristina D. A.  
1319 Ciferri has been supported by the grant #2018/22277-8, FAPESP. The work of Michael Vassi-  
1320 lakopoulos and Antonio Corral is funded by the MINECO research project [TIN2017-83964-R].

## 1321 References

- 1322 1. Agrawal D, Ganesan D, Sitaraman R, Diao Y, Singh S (2009) Lazy-adaptive  
1323 tree: An optimized index structure for flash devices. VLDB Endowment  
1324 2(1):361–372
- 1325 2. Agrawal N, Prabhakaran V, Wobber T, Davis JD, Manasse M, Panigrahy  
1326 R (2008) Design tradeoffs for SSD performance. In: USENIX 2008 Annual  
1327 Technical Conf., pp 57–70
- 1328 3. Aref WG, Ilyas IF (2001) SP-GiST: An extensible database index for sup-  
1329 porting space partitioning trees. Journal of Intelligent Information Systems  
1330 17:215–240
- 1331 4. Arge L, Berg MD, Haverkort H, Yi K (2008) The Priority R-tree: A practically  
1332 efficient and worst-case optimal r-tree. ACM Trans Algorithms 4(1)
- 1333 5. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R\*-tree: An effi-  
1334 cient and robust access method for points and rectangles. In: ACM SIGMOD  
1335 Int. Conf. on Management of Data, pp 322–331
- 1336 6. Berchtold S, Keim DA, Kriegel HP (1996) The X-Tree: An index structure for  
1337 high-dimensional data. In: Int. Conf. on Very Large Databases, pp 28–39
- 1338 7. Bouganis L, Jónsson B, Bonnet P (2009) uFLIP: Understanding flash IO  
1339 patterns. In: Fourth Biennial Conf. on Innovative Data Systems Research
- 1340 8. Brayner A, Monteiro Filho JM (2016) Hardware-aware database systems: A  
1341 new era for database technology is coming - vision paper. In: Brazilian Symp.  
1342 on Databases, pp 187–192
- 1343 9. Carniel AC, Ciferri RR, Ciferri CDA (2016) The performance relation of spa-  
1344 tial indexing on hard disk drives and solid state drives. In: Brazilian Symp.  
1345 on GeoInformatics, pp 263–274
- 1346 10. Carniel AC, Ciferri RR, Ciferri CDA (2017) Analyzing the performance of  
1347 spatial indices on hard disk drives and flash-based solid state drives. Journal of  
1348 Information and Data Management 8(1):34–49
- 1349 11. Carniel AC, Ciferri RR, Ciferri CDA (2017) A generic and efficient framework  
1350 for spatial indexing on flash-based solid state drives. In: European Conf. on  
1351 Advances in Databases and Information Systems, pp 229–243

- 1352 12. Carniel AC, Ciferri RR, Ciferri CDA (2017) Spatial datasets for conducting  
1353 experimental evaluations of spatial indices. In: Satellite Events of the Brazilian  
1354 Symp. on Databases - Dataset Showcase Workshop, pp 286–295
- 1355 13. Carniel AC, Silva TB, Bonicenha KLS, Ciferri RR, Ciferri CDA (2017) An-  
1356 alyzing the performance of spatial indices on flash memories using a flash  
1357 simulator. In: Brazilian Symp. on Databases, pp 40–51
- 1358 14. Carniel AC, Roumelis G, Ciferri RR, Vassilakopoulos M, Corral A, Ciferri  
1359 CDA (2018) An efficient flash-aware spatial index for points. In: Brazilian  
1360 Symp. on GeoInformatics, pp 68–79
- 1361 15. Carniel AC, Ciferri RR, Ciferri CDA (2019) A generic and efficient framework  
1362 for flash-aware spatial indexing. *Information Systems* 82:102–120
- 1363 16. Carniel AC, Roumelis G, Ciferri RR, Vassilakopoulos M, Corral A, Ciferri  
1364 CDA (2019) Indexing points on flash-based solid state drives using the xBR<sup>+</sup>-  
1365 tree. *Journal of Information and Data Management* 10(1):35–48
- 1366 17. Carniel AC, Ciferri RR, Ciferri CDA (2020) FESTIval: A versatile framework  
1367 for conducting experimental evaluations of spatial indices. *MethodsX* 7:1–19
- 1368 18. Chen F, Koufaty DA, Zhang X (2009) Understanding intrinsic characteristics  
1369 and system implications of flash memory based solid state drives. In: ACM  
1370 SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Sys-  
1371 tems, pp 181–192
- 1372 19. Chen F, Hou B, Lee R (2016) Internal parallelism of flash memory-based solid-  
1373 state drives. *ACM Trans on Storage* 12(3):13:1–13:39
- 1374 20. Cormer D (1979) Ubiquitous B-tree. *ACM Comp Surveys* 11(2):121–137
- 1375 21. Denning PJ (1980) Working sets past and present. *IEEE Trans on Software  
1376 Engineering* SE-6(1):64–84
- 1377 22. Effelsberg W, Haerder T (1984) Principles of database buffer management.  
1378 *ACM Trans on Database Systems* 9(4):560–595
- 1379 23. Emrich T, Graf F, Kriegel HP, Schubert M, Thoma M (2010) On the impact  
1380 of flash SSDs on spatial indexing. In: Int. Workshop on Data Management on  
1381 New Hardware, pp 3–8
- 1382 24. Fevgas A, Bozanis P (2015) Grid-file: Towards to a flash efficient multi-  
1383 dimensional index. In: Int. Conf. on Database and Expert Systems Appli-  
1384 cations, pp 285–294
- 1385 25. Fevgas A, Bozanis P (2019) LB-Grid: An SSD efficient grid file. *Data & Knowl-  
1386 edge Engineering* 121:18–41
- 1387 26. Fevgas A, Akritidis L, Bozanis P, Manolopoulos Y (2019) Indexing in flash  
1388 storage devices: a survey on challenges, current approaches, and future trends.  
1389 *The VLDB Journal* pp 1–39
- 1390 27. Folk MJ, Zoellick B, Riccardi G (1997) *File Structures: An Object-Oriented  
1391 Approach with C++*, 3rd edn. Addison Wesley
- 1392 28. Gaede V, Günther O (1998) Multidimensional access methods. *ACM Comp  
1393 Surveys* 30(2):170–231
- 1394 29. Graefe G (2012) A survey of b-tree logging and recovery techniques. *ACM  
1395 Trans on Database Systems* 37(1)
- 1396 30. Guttman A (1984) R-trees: A dynamic index structure for spatial searching.  
1397 In: ACM SIGMOD Int. Conf. on Management of Data, pp 47–57
- 1398 31. Harder T, Reuter A (1993) Principles of transaction-oriented database recov-  
1399 ery. *ACM Comp Surveys* 15(4):287–317

- 1400 32. Hellerstein JM, Naughton JF, Pfeffer A (1995) Generalized search trees for  
1401 database systems. In: Int. Conf. on Very Large Databases, pp 562–573
- 1402 33. Jenkins B (2006) Hash functions for hash table lookup. [http://burtleburtle.  
1403 net/bob/hash/index.html](http://burtleburtle.net/bob/hash/index.html)
- 1404 34. Jin P, Xie X, Wang N, Yue L (2015) Optimizing R-tree for flash memory.  
1405 Expert Systems with Applications 42(10):4676–4686
- 1406 35. Jin P, Yang C, Jensen CS, Yang P, Yue L (2016) Read/write-optimized tree  
1407 indexing for solid-state drives. The VLDB Journal 25(5):695–717
- 1408 36. Johnson T, Shasha D (1994) 2Q: A low overhead high performance buffer  
1409 management replacement algorithm. In: Int. Conf. on Very Large Databases,  
1410 pp 439–450
- 1411 37. Jung M, Kandemir M (2013) Revisiting widely held SSD expectations and  
1412 rethinking system-level implications. In: ACM SIGMETRICS Int. Conf. on  
1413 Measurement and Modeling of Computer Systems, pp 203–216
- 1414 38. Kamel I, Faloutsos C (1994) Hilbert R-tree: An improved R-tree using fractals.  
1415 In: Int. Conf. on Very Large Databases, pp 500–509
- 1416 39. Koltsidas I, Viglas SD (2011) Spatial data management over flash memory.  
1417 In: Int. Conf. on Advances in Spatial and Temporal Databases, pp 449–453
- 1418 40. Kornacker M (1999) High-performance extensible indexing. In: Int. Conf. on  
1419 Very Large Databases, pp 699–708
- 1420 41. Kwon SJ, Ranjitkar A, Ko YB, Chung TS (2011) FTL algorithms for NAND-  
1421 type flash memories. Design Automation for Embedded Systems 15(3-4):191–  
1422 224
- 1423 42. Li G, Zhao P, Yuan L, Gao S (2013) Efficient implementation of a multi-  
1424 dimensional index structure over flash memory storage systems. The Journal  
1425 of Supercomputing 64(3):1055–1074
- 1426 43. Li Y, He B, Yang RJ, Luo Q, Yi K (2010) Tree indexing on solid state drives.  
1427 VLDB Endowment 3(1-2):1195–1206
- 1428 44. Lin S, Zeinalipour-Yazti D, Kalogeraki V, Gunopulos D, Najjar WA (2006)  
1429 Efficient indexing data structures for flash-based sensor devices. ACM Trans  
1430 on Storage 2(4):468–503
- 1431 45. Lv Y, Li J, Cui B, Chen X (2011) Log-Compact R-tree: An efficient spatial  
1432 index for SSD. In: Int. Conf. on Database Systems for Advanced Applications,  
1433 pp 202–213
- 1434 46. Mehlhorn K, Sanders P (2008) Algorithms and Data Structures: The Basic  
1435 Toolbox. Springer
- 1436 47. Mittal S, Vetter JS (2016) A survey of software techniques for using non-  
1437 volatile memories for storage and main memory systems. IEEE Trans on Par-  
1438 allel and Distributed Systems 27(5):1537–1550
- 1439 48. Nievergelt J, Hinterberger H, Sevcik KC (1984) The grid file: An adaptable,  
1440 symmetric multikey file structure. ACM Trans on Database Systems 9(1):38–  
1441 71
- 1442 49. Oosterom PVaN (2005) Spatial Access Methods. In: Longley PA, Goodchild  
1443 MF, Maguire DJ, Rhind DW (eds) Geographical Information Systems: Prin-  
1444 ciples, Techniques, Management and Applications, 2nd edn, pp 385–400
- 1445 50. PostGIS (2020) Spatial and geographic objects for postgresql. [https://  
1446 postgis.net/](https://postgis.net/)
- 1447 51. Proietti G, Faloutsos C (1999) I/O complexity for range queries on region data  
1448 stored using an r-tree. In: Int. Conf. on Data Engineering, pp 628–635

- 1449 52. Rigaux P, Scholl M, Voisard A (2001) Spatial databases: with application to  
1450 GIS, 1st edn. Morgan Kaufmann
- 1451 53. Robinson JT (1981) The K-D-B-tree: a search structure for large multidimensional  
1452 dynamic indexes. In: ACM SIGMOD Int. Conf. on Management of  
1453 Data, pp 10–18
- 1454 54. Roumelis G, Vassilakopoulos M, Loukopoulos T, Corral A, Manolopoulos Y  
1455 (2015) The xBR<sup>+</sup>-tree: an efficient access method for points. In: Int. Conf. on  
1456 Database and Expert Systems Applications, pp 43–58
- 1457 55. Roumelis G, Vassilakopoulos M, Corral A, Manolopoulos Y (2017) Efficient  
1458 query processing on large spatial databases: A performance study. Journal of  
1459 Systems and Software 132:165–185
- 1460 56. Roumelis G, Vassilakopoulos M, Corral A, Fevgas A, Manolopoulos Y (2018)  
1461 Spatial batch-queries processing using xBR<sup>+</sup>-trees in solid-state drives. In: Int.  
1462 Conf. on Model and Data Engineering, pp 301–317
- 1463 57. Roumelis G, Fevgas A, Vassilakopoulos M, Corral A, Bozanis P, Manolopoulos  
1464 Y (2019) Bulk-loading and bulk-insertion algorithms for xBR<sup>+</sup>-trees in solid  
1465 state drives. Computing pp 1–25
- 1466 58. Samet H (1984) The quadtree and related hierarchical data structures. ACM  
1467 Comp Surveys 16(2):187–260
- 1468 59. Sarwat M, Mokbel MF, Zhou X, Nath S (2013) Generic and efficient framework  
1469 for search trees on flash memory storage systems. GeoInformatica 17(3):417–  
1470 448
- 1471 60. Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-tree: A dynamic index  
1472 for multi-dimensional objects. In: Int. Conf. on Very Large Databases, pp  
1473 507–518
- 1474 61. Su X, Jin P, Xiang X, Cui K, Yue L (2009) Flash-DBSim: A simulation tool for  
1475 evaluating flash-based database algorithms. In: IEEE Int. Conf. on Computer  
1476 Science and Information Technology, pp 185–189
- 1477 62. Thonangi R, Babu S, Yang J (2012) A practical concurrent index for solid-  
1478 state drives. In: ACM Int. Conf. on Information and Knowledge Management,  
1479 p 13321341
- 1480 63. Wu CH, Chang LP, Kuo TW (2003) An efficient R-tree implementation over  
1481 flash-memory storage systems. In: ACM SIGSPATIAL Int. Conf. on Advances  
1482 in Geographic Information Systems, pp 17–24
- 1483 64. Wu CH, Kuo TW, Chang LP (2007) An efficient B-tree layer implementa-  
1484 tion for flash-memory storage systems. ACM Trans on Embedded Computing  
1485 Systems 6(3)
- 1486 65. Zhang Y, Swanson S (2015) A study of application performance with non-  
1487 volatile main memory. In: Symp. on Mass Storage Systems and Technologies,  
1488 pp 1–10