Indexing Points on Flash-based Solid State Drives using the xBR$^+$-tree

Anderson Chaves Carniel[1], George Roumelis[2], Ricardo Rodrigues Ciferri[3], Michael Vassilakopoulos[2], Antonio Corral[4], Cristina Dutra de Aguiar Ciferri[5]

[1] Federal University of Technology - Paraná, Brazil
accarniel@utfpr.edu.br
[2] University of Thessaly, Greece
groumelis@uth.gr, mvasilako@uth.gr
[3] Federal University of São Carlos, Brazil
ricardo@dc.ufscar.br
[4] University of Almeria, Spain
acorral@ual.es
[5] University of São Paulo, Brazil
cdac@icmc.usp.br

Abstract. Spatial database systems and Geographic Information Systems have focused on exploiting the positive characteristics of *flash-based Solid State Drives* (SSDs) like fast reads and writes. However, designing spatial indices for SSDs, termed *flash-aware spatial indices*, has been a challenging task because of the intrinsic characteristics of these devices. Among existing works in the literature, FAST and eFIND distinguish themselves by proposing general approaches that can be employed to design flash-aware spatial indices. In this article, we apply these approaches to the xBR$^+$-tree, an efficient spatial index for points. The goal is to understand how they should be adapted in order to deal with the particular properties of the xBR$^+$-tree. As a result, we specify two novel flash-aware spatial indices for points, the eFIND xBR$^+$-tree and the FAST xBR$^+$-tree. We also conduct a performance evaluation to analyze their performance. As main conclusion, we point out that eFIND fits very well with the xBR$^+$-tree, allowed the processing of the index construction to be reduced from 30.8% to 91.4%, and improved the execution of spatial queries from 22.5% to 46%.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Spatial databases and GIS

Keywords: flash-aware spatial index, flash memory, spatial databases, spatial indexing, xBR$^+$-trees

1. INTRODUCTION

The use of a spatial index is a usual strategy employed by spatial database systems and Geographic Information Systems (GIS) to speed up the processing of spatial queries. A common objective is to reduce the search space of the spatial query by avoiding the access of spatial objects that certainly do not belong to its final answer [Gaede and Günther 1998]. The main assumption of several spatial indices is that the spatial objects are stored in magnetic disks (i.e., *Hard Disk Drives* - HDDs). Hence, they often consider the slow mechanical access and the high cost of search and rotational delay of disks in their design. We term spatial indices designed for magnetic disks as *disk-based spatial indices*.

A wide range of disk-based spatial indices has been proposed in the literature [Gaede and Günther 1998]. The R-tree and its variants, such as the R$^+$-tree [Sellis et al. 1987] and the R*-tree [Beckmann et al. 1990], are well-known spatial indices. The efficient indexing of multidimensional points has been a main focus of several indices because of the use of points in real spatial database applications. For instance, spatial database systems and GIS represent specific locations in the Euclidean plane as point objects that are then retrieved by queries like intersection range queries (IRQs) [Gaede and Günther 1998; Rigaux et al. 2001; Oosterom 2005]. Other examples include spatial applications that implement spatial rankings [Hjaltason and Samet 1995] and location-based mobile services [Silva

et al. 2018]. Among the existing disk-based spatial indices, we highlight the *xBR$^+$-tree* [Roumelis et al. 2015], which provides data structures and algorithms for handling points efficiently. In fact, extensive experimental evaluations [Roumelis et al. 2017] showed that the xBR$^+$-tree has outperformed variants of the R-tree (i.e., the R*-tree and the R$^+$-tree) when processing different types of spatial queries.

On the other hand, advanced database and spatial applications are interested in using modern storage devices like *flash-based Solid State Drives* (SSDs) [Koltsidas and Viglas 2011a; Brayner and Monteiro Filho 2016; Mittal and Vetter 2016; Fevgas et al. 2019]. This includes spatial database systems and GIS that employ spatial indices to efficiently retrieve spatial objects stored in SSDs [Emrich et al. 2010; Koltsidas and Viglas 2011b; Carniel et al. 2016; 2017a; Carniel 2018]. The main reason for this interest is because SSDs, in contrast to HDDs, have 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 [Agrawal et al. 2008; Bouganim et al. 2009; Chen et al. 2009; Jung and Kandemir 2013; Mittal and Vetter 2016; Carniel et al. 2017b; 2019]. 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 on 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 the internal buffers and the read disturbance management [Chen et al. 2009; Jung and Kandemir 2013].

To deal with the intrinsic characteristics of SSDs, spatial indices specifically designed for SSDs have been proposed in the literature. However, designing spatial indices for SSDs, termed here as *flash-aware spatial indices*, has been a challenging task. A common strategy is to mitigate the poor performance of random writes by storing index modifications in a *write buffer*. Whenever this buffer is full, a *flushing operation* is performed. Among existing flash-aware spatial indices proposed in the literature (see Section 2), FAST-based indices [Sarwat et al. 2013] and eFIND-based indices [Carniel et al. 2017b; 2019] distinguish themselves. FAST and eFIND are generic frameworks that transform disk-based hierarchical indices into flash-aware hierarchical indices. They also provide support for data durability by using a log-structured approach that allows the framework's write buffer to be recovered after a fatal problem (e.g., power failure).

Considering the xBR$^+$-tree as the state-of-the-art when indexing points, an open question is how to efficiently deploy FAST and eFIND to make the xBR$^+$-tree also efficient in SSDs. In this article, we answer this question by proposing the *FAST xBR$^+$-tree* and the *eFIND xBR$^+$-tree*, two flash-aware spatial indices for points. For this, we adapt the data structures of FAST and eFIND to deal with the properties and structural constraints of the xBR$^+$-tree.

We validate the FAST xBR$^+$-tree and the eFIND xBR$^+$-tree by conducting an extensive experimental evaluation that consisted of constructing indices on real and synthetic datasets and of executing IRQs. The performance results allowed us to identify what is the best approach to make the xBR$^+$-tree efficient in SSDs. Our experiments showed that the eFIND xBR$^+$-tree yielded the best results in almost all cases, with execution time reductions from 30.8% to 91.4% when building indices and from 22.5% to 46% when processing IRQs.

The rest of this article is organized as follows. Section 2 surveys related work and details how this article extends our previous work [Carniel et al. 2018]. Section 3 summarizes the structure of the xBR$^+$-tree. Section 4 presents the FAST xBR$^+$-tree. Section 5 introduces the eFIND xBR$^+$-tree. Section 6 discusses the conducted experiments. Finally, Section 7 concludes the paper and presents future work.

2. RELATED WORK

There are few flash-aware spatial indices that have been proposed in the literature. In this section, we summarize the characteristics of the main flash-aware spatial indices as follows. The *RFTL* [Wu et al. 2003] ports the R-tree to SSDs using a write buffer to avoid random writes. Its main problem is the flushing operation that flushes all modifications stored in the write buffer, requiring high elapsed times. Another problem is related to lack of *data durability*. This means that the modifications stored in the write buffer are lost after a system crash or power failure.

MicroHash and *Micro Grid File* [Lin et al. 2006] are data structures for flash-based sensor devices. Due to the low processing capabilities of sensor devices, they deploy write buffers only. The *F-KDB* [Li et al. 2013] employs a write buffer that stores modified entries of the K-D-B-tree [Robinson 1981], called logging entries. Its main problem is that retrieving nodes is a complex operation because the entries of a node might be stored in different flash pages. Finally, the *Grid file for flash memory* [Fevgas and Bozanis 2015] employs a buffer strategy based on the Least Recently Used (LRU) replacement policy [Denning 1980] to maintain modifications of the grid file [Nievergelt et al. 1984]. A flushing operation writes to the SSD only those index pages that are classified as cold pages. However, the quantity of modifications is not considered, leading to a possibly high number of flushing operations.

FAST [Sarwat et al. 2011; 2013] distinguishes itself as a generic framework that generalizes the write buffer to store modifications of any hierarchical index. Hence, it transforms any disk-based hierarchical index into a flash-aware index. Further, FAST provides a specialized *flushing algorithm* that employs a *flushing policy* for selecting a *flushing unit* to be written to the SSD instead of writing all modifications contained in the write buffer. A flushing unit consists of a set of nodes. Further, FAST defines that a flushing unit has only sequential nodes (based on its relative record number in the index file). Hence, FAST guarantees that the flushing operation is executed as a batch operation. The FAST's flushing policy then picks the flushing unit which has the best balance between the number of modifications and recency of the modifications. FAST also provides support for data durability by employing a log-structured approach.

The *FOR-tree* [Jin et al. 2015] improves the flushing algorithm of FAST by dynamically creating flushing units containing modified nodes only. It also abolishes splitting operations by allowing overflowed nodes. Whenever a specific number of accesses in an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting them into the parent node, growing up the tree if needed. However, the number of accesses of an overflowed root node is never incremented in an insertion operation. As a consequence, the construction of a FOR-tree, inserting one spatial object by time, forms an overflowed root node instead of a hierarchical structure. This critical problem prevented us from creating spatial indices over large and medium spatial datasets using the FOR-tree.

eFIND [Carniel et al. 2017b; 2019] is another generic framework that efficiently transforms any disk-based spatial index into a flash-aware spatial index. It is based on distinct design goals that consider the intrinsic characteristics of SSDs. eFIND leverages a specialized flushing algorithm that employs a flushing policy to pick the best flushing unit to be written to the SSD. The main flushing policy of eFIND takes into account the height of the node by prioritizing the nodes in the higher level of the tree. eFIND also deploys an in-memory *read buffer* in order to avoid reads on frequent locations. This read buffer deploys a read buffer replacement policy that manages the nodes to be cached in the main memory. Further, eFIND specifies a *temporal control* to mitigate the effects of interleaved reads and writes. Finally, eFIND applies a log-structure approach to guarantee data durability.

Although FAST and eFIND have the aforementioned advantages, they have been applied to the R-tree only [Carniel et al. 2019; Sarwat et al. 2013]. This limits the study on the applicability of these general approaches in other spatial indexing structures, such as the xBR$^+$-tree. In this article, we extend our previous work [Carniel et al. 2018] by detailing how FAST and eFIND should be adapted

to deal with the properties and structural constraints of the xBR$^+$-tree (Section 3). As a result, we present the FAST xBR$^+$-tree and the eFIND xBR$^+$-tree. We also extend the experiments of our previous work by considering other spatial datasets with different characteristics and volume. This allows us to identify the best approach to port the xBR$^+$-tree to SSDs.

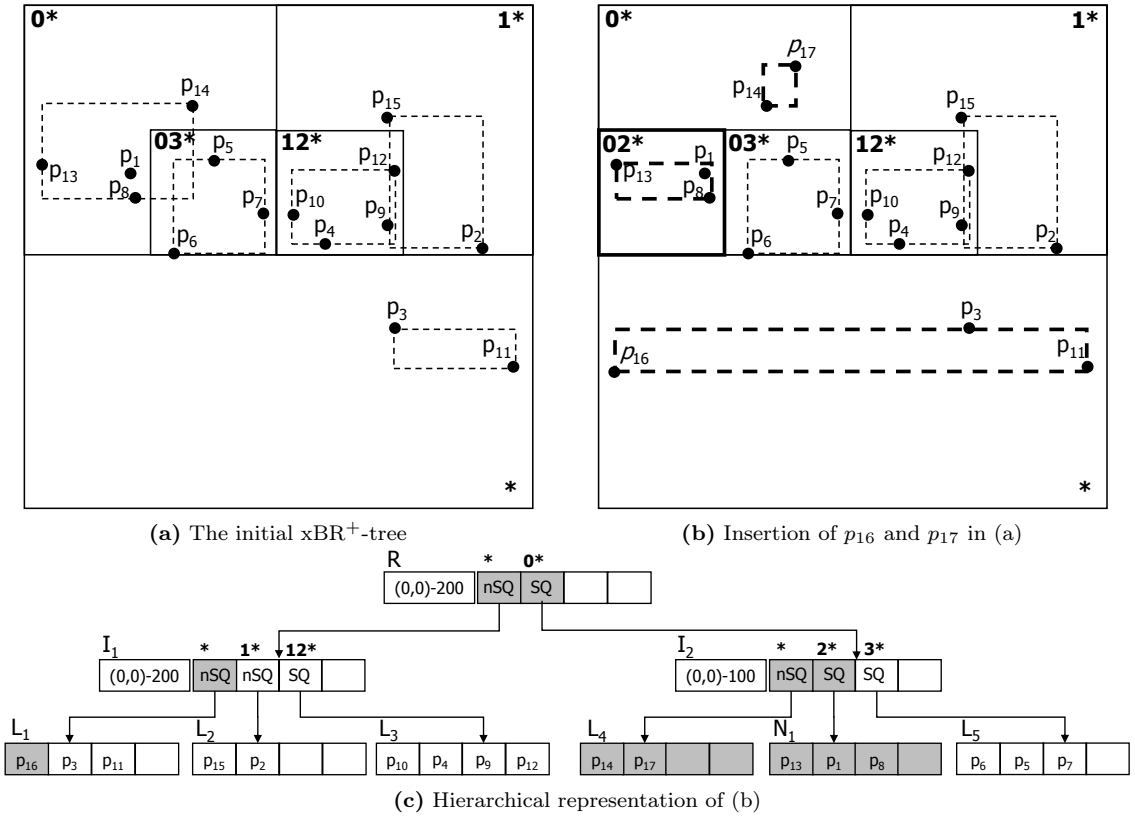
This article also advances on studies that aim to apply the xBR$^+$-tree to SSDs, such as algorithms for executing spatial batch-queries [Roumelis et al. 2018] and bulk-loading strategies [Roumelis et al. 2019]. The first study refers to the proposal of algorithms for processing sets of spatial queries only, whereas the second study proposes strategies to build xBR$^+$-trees from a set of points as a single and unique operation. Hence, these studies focus on specific types of algorithms involving the xBR$^+$-tree. Moreover, these studies do not deal with durability issues. On the other hand, in this article we focus on providing a general strategy to efficiently implement index operations on SSDs. That is, our solutions can be employed to process general transactions in spatial database systems, such as insertions, deletions, and spatial queries.

3. THE XBR$^+$-TREE

The xBR$^+$-tree is a hierarchical index based on the regular decomposition of space of Quadtrees [Samet 1984] able to index multidimensional points. Hence, it is a *space-driven access method*. For bidimensional points, the xBR$^+$-tree decomposes recursively the space to 4 equal quadrants, called *sub-quadrants*. Figure 1a depicts an example of an xBR$^+$-tree that indexes 15 points (i.e., p_1 to p_{15}). Figure 1b shows this xBR$^+$-tree with a set of adjustments, represented by thick lines, after the insertion of two new points, p_{16} and p_{17}. These points and the resulting adjustments should be handled by FAST (Section 4) and eFIND (Section 5) and are also highlighted in the hierarchical representation of Figure 1c. We detail the structure of the xBR$^+$-tree and its structural constraints and properties as follows.

There are two types of nodes, internal nodes and leaf nodes. Internal nodes consist of entries in the following format $(id, DBR, qside, shape)$. Each entry of an internal node refers to a child node that is pointed by *id* and represents a sub-quadrant of the original space. *DBR* refers to the data bounding rectangle that minimally encompasses the points stored in such a sub-quadrant. *qside* stores the side length of the sub-quadrant corresponding to the child node's entry. Finally, *shape* is a flag that indicates if the sub-quadrant is either a complete square or a non-complete square. The entries of an internal node are also sorted by their *addresses*. Each address is calculated by using *qside* and *DBR*, and consists of a sequence of *directional digits* representing a sub-quadrant. The directional digits 0, 1, 2, and 3 respectively symbolize the northwest (NW), northeast (NE), southwest (SW), and southeast (SE) sub-quadrants of a relative space. Hence, it follows the Z-order.

Figure 1c depicts a tree with 3 internal nodes, R, I_1, and I_2, that corresponds to the graphical representation of Figure 1b. In Figure 1c, the entries and nodes updated by the insertion of p_{16} and p_{17} appear shaded; in fact, N_1 is a newly created node resulting from dividing L_4 which previously hosted the points p_{13}, p_1, p_8, and p_{14} (Figure 1a). Each internal node has also a header containing data about its sub-quadrant. For instance, the origin point of the sub-quadrant of R is $(0, 0)$ with a side length of 200. The address of each entry of an internal node is shown in bold (but, this is not actually stored). For instance, the right child of R that points to I_2 is the NW quadrant of the original space, denoted as *0** (* is used to mark the end of the address). Further, it represents a complete square (i.e., *SQ*). Its *DBR* consists of a minimum bounding rectangle containing the points p_1, p_5, p_6, p_7, p_8, p_{13}, p_{14}, and p_{17}. The left child of R represents a region derived from the spatial difference between the original space and the region of the NW quadrant, that is, the union of the NE, SW, and SE quadrants of the original space. Hence, it has the address * (i.e., empty) and represents a non-complete square (i.e., *nSQ*). Its DBR consists of a minimum bounding rectangle containing the points p_2, p_3, p_4, p_9, p_{10}, p_{11}, p_{12}, p_{15}, p_{16}. Finally, addresses of entries of internal nodes determine a sub-quadrant in relation to the region of their node. For instance, the address *3** (in node I_2 of



(a) The initial xBR$^+$-tree

(b) Insertion of p_{16} and p_{17} in (a)

(c) Hierarchical representation of (b)

Fig. 1. An example of an xBR$^+$-tree.

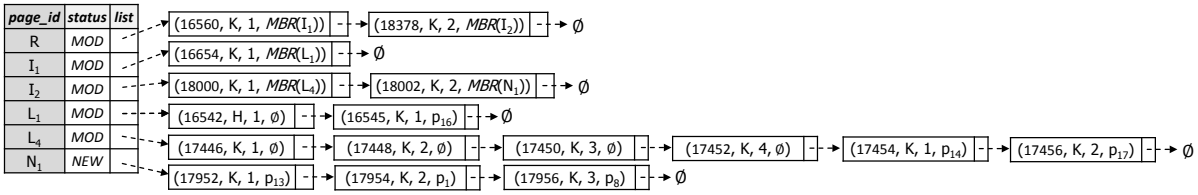
Figure 1c) represents the SE sub-quadrant of the NW sub-quadrant of the original space (the region of I_2, denoted by *0** in R of Figure 1c).

Leaf nodes contain entries in the format (id, p), where p is the multidimensional point and id is a pointer to the register of p. These entries are sorted by X-axis coordinates of the points, allowing the use of the *plane sweep technique* in specific spatial query types. For instance, the leaf node L_1 in Figure 1c contains the points p_{16}, p_3, and p_{11}, which are sorted by their X-axis coordinates depicted in Figure 1b. The pointers to the registers of these points are omitted.

Since the entries of internal and leaf nodes have fixed sizes, each type of nodes is characterized by a specific capacity of its type of entries. Whenever the capacity of a leaf or internal node is exceeded, the quadrant encompassing the overflowed node is partitioned into two sub-quadrants according to a Quadtree-like hierarchical decomposition. Different criteria for this partitioning are conceivable [Roumelis et al. 2017]. For instance, Figure 1b depicts the creation of a new sub-quadrant with address *02** (i.e., node N_1 in Figure 1c) resulting from a splitting operation after inserting p_{17}.

4. THE FAST XBR$^+$-TREE

FAST provides two main data structures to handle index modifications [Sarwat et al. 2013]; they are: a *write buffer*, and a *log file*. To deal with the xBR$^+$-tree, we extend the FAST's data structures as follows: (i) we adapt the write buffer to store specific data related to internal nodes, and (ii) we also apply similar changes to the log file, making possible the recovery of the write buffer after a system crash. Further, FAST provides generic algorithms that handle its data structures when executing the index operations of the underlying index. To deal with the xBR$^+$-tree, we also adapt these algorithms.



(a) *Tree Modifications Table*

log#	operation	node_list	modification
1	MOD	L_1	(H, 1, ∅)
2	MOD	L_1	(K, 1, $_{16}$)
3	MOD	I_1	(K, 1, *MBR*(L_1))
4	MOD	R	(K, 1, *MBR*(I_1))
5	MOD	L_4	(K, 1, ∅)
6	MOD	L_4	(K, 2, ∅)
7	MOD	L_4	(K, 3, ∅)
8	MOD	L_4	(K, 4, ∅)
9	MOD	L_4	(K, 1, p_{14})
10	MOD	L_4	(K, 2, p_{17})
11	NEW	N_1	-
12	MOD	N_1	(K, 1, p_{13})
13	MOD	N_1	(K, 2, p_1)
14	MOD	N_1	(K, 3, p_8)
15	MOD	I_2	(K, 2, *MBR*(N_1))
16	MOD	R	(K, 2, *MBR*(I_2))

(b) *Log file*

Fig. 2. The *Tree Modifications Table* and the *log file* of FAST to handle the modifications of the xBR$^+$-tree of Figure 1c. As a result, the FAST xBR$^+$-tree is created.

We detail the extensions performed to propose the FAST xBR$^+$-tree as follows.

Adapting the write buffer. The write buffer is implemented as a hash table named *Tree Modifications Table*. The key of this hash table is the identifier of the node (i.e., *page_id*) and its value is a pair (*status*, *list*). *status* is a flag indicating the type of the node modification and assumes the value NEW, DEL, or MOD if the node is newly created, deleted, or modified, respectively. For *status* equal to NEW and DEL, *list* points to the newly created node stored in the main memory and to null, respectively. For *status* equal to MOD, *list* is a modification list storing quadruples (*timestamp*, *type*, *index*, *value*), where *timestamp* stores the moment of the modification, *type* informs what is modified (*K* for a modified entry), *index* is the position of the modification, and *value* is the result of the modification. We have also employed *type* to indicate if an entry should be inserted in a specific position of a node to satisfy the sorting property of the xBR$^+$-tree. Before indicating that a new entry will be inserted in such a position, an element of the modification list should store the position that will accommodate the new entry. This is indicated in the form of a *hole* (i.e., *type* equal to *H*).

Figure 2a depicts the *Tree Modifications Table* for handling the modifications of the xBR$^+$-tree of Figure 1b. In this figure, *MBR* represents the rectangle that encompasses all points of a sub-quadrant considering the modifications stored in the write buffer in a particular moment (i.e., indicated by *timestamp*). Each hash entry has a modification list indicating the final result of the modified entries. For instance, the first line of the hash table in Figure 2a shows that *R* has the *status* MOD, and stores a modification list with size equal to 2. The first modification refers to the first entry of *R* in order to handle the adjustment of its *DBR* after inserting p_{16}, whereas the second modification indicates that the *DBR* is adjusted in order to deal with the insertion of p_{17}.

Another example of using the modification list is when dealing with splits. Let *A* be an overflowed

node. First, the splitting operation of the underlying index is executed, distributing the entries of A between itself and a newly created node, called B. To save these modifications in the write buffer, all entries of A are deleted, then its new entries are added to its modification list, and finally the newly created node is stored as a new hash entry in the *Tree Modifications Table*. In our example, this procedure is performed to handle the splitting operation after inserting p_{17} in the node L_4, as depicted in the two last lines of the *Tree Modifications Table* in Figure 2a. That is, the first four elements of the modification list of L_4 refer to the deletion of the old entries, and the remaining elements are related to the new entries of L_4 after executing the splitting operation. In addition, the new node N_1 is created, containing three elements that were previously stored in L_4 (Figure 2a).

Adapting the log file. FAST guarantees data durability by storing all modifications contained in the *Tree Modifications Table* in a log file. Each log entry is a triple (*operation*, *node_list*, *modification*), where *operation* extends the possible values of *status* by adding the flag FLUSH for flushing operations, *node_list* is the list of affected nodes, and *modification* is the subset (*type*, *index*, *value*) of the *value* used in the *Tree Modifications Table*. Figure 2b depicts the log file for storing all the modifications of the *Tree Modifications Table* of Figure 2a.

Adapting the generic algorithms. FAST provides generic algorithms for executing the following operations: (i) *insert operation*, which specifies how the index modifications are stored in the write buffer and in the log file, (ii) *search operation*, which is responsible for retrieving nodes, (iii) *flushing operation*, which selects a set of modifications stored in the write buffer to be written to the SSD according to a flushing policy, and (iv) *restart operation*, which rebuilds the write buffer after a fatal problem and compacts the log file.

To correctly retrieve a node of the xBR$^+$-tree, we mainly adapt FAST as follows. There are three possible cases when retrieving a node N: (i) N is read from the SSD if it has no modifications, (ii) N is directly returned from the *Tree Modifications Table* if its status is equal to DEL or NEW, and (iii) the modifications of N are applied to it if its status is equal to MOD. In the third case, which involves our extensions, the old version of N is read from the SSD and then for each element in the modification list, we apply the modification to the corresponding position (i.e., *index*) of N. Note that if the modification is a hole, we shift all elements after the position in order to create free space for the new element that is inserted into the sequence. This guarantees that the elements in N fulfill the sorting property of the xBR$^+$-tree.

5. THE EFIND XBR$^+$-TREE

eFIND provides specific data structures to handle index modifications and exploit SSD performance according to its design goals [Carniel et al. 2019]. They are: a *write buffer*, a *read buffer*, a *log file*, and *read and write queues*. To deal with the xBR$^+$-tree, we extend the eFIND's data structures as follows: (i) we adapt the write and read buffers to store specific data related to internal nodes, (ii) we generalize the storage of index modifications according to the sorting properties of internal and leaf nodes, and (iii) we adjust the structure of log entries to recover the write buffer after a system crash. Further, eFIND provides generic algorithms that handle its data structures. To deal with the xBR$^+$-tree, we also adapt these algorithms. We detail the extensions performed to propose the eFIND xBR$^+$-tree as follows.

Adapting the write and read buffers. The write buffer is implemented as a hash table named *Write Buffer Table* and stores the modifications of nodes that were not applied to the SSD yet. Its main goal is to avoid random writes to the SSD. The key of this hash table is the identifier of a node (*page_id*) and its value stores modifications in the format (h, *mod_count*, *timestamp*, *reg*, *status*, *mod_tree*). Here, h stores the height of the modified node, *mod_count* is the quantity of in-memory modifications, *timestamp* informs when the last modification was made, *reg* is the sub-quadrant of a

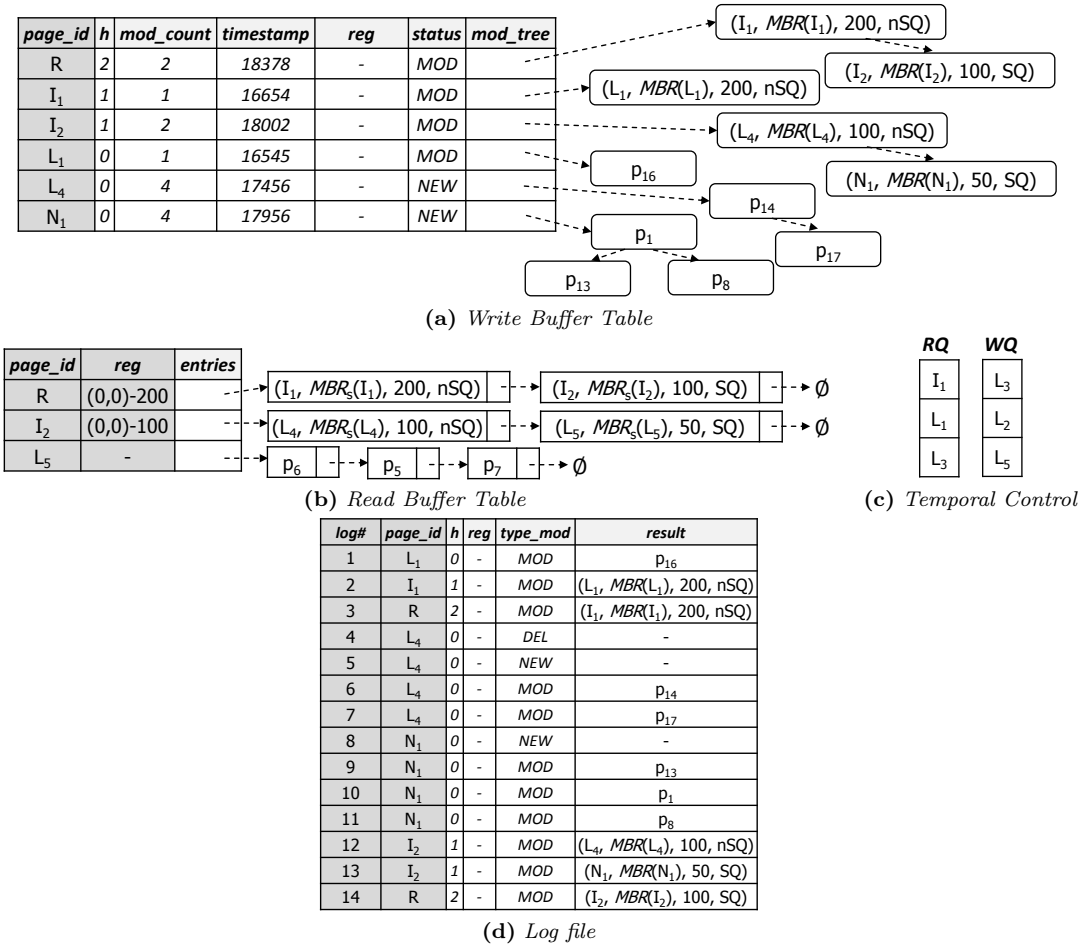
newly created internal node, and *status* is the type of modification made and can be NEW, MOD, or DEL for representing newly created nodes in the buffer, nodes stored in the SSD but with modified entries, and deleted nodes, respectively. If *status* is equal to DEL, *mod_ tree* is null. Otherwise, it is a red-black tree containing the most recent version of modified entries. Each element of this red-black tree has the format (*e*, *mod_ result*), where *e* is the key and corresponds to the unique identifier of an entry and *mod_ result* stores the latest version of an entry, assuming null if *e* was removed. To deal with the xBR$^+$-tree, we extend the standard comparison function of eFIND to guarantee the sorting property of internal and leaf nodes. That is, for internal nodes, the comparison function deploys the ascending order of the directional digits of the entries by using *reg* as a basis for the calculation; for leaf nodes, the comparison function considers the ascending order of the X-axis coordinates.

Figure 3a shows the *Write Buffer Table* for handling the modifications of the xBR$^+$-tree of Figure 1b. In this figure, *MBR* has the same representation as in Figure 2a, that is, the rectangle that encompasses all points of a sub-quadrant. The elements of the *mod_ tree* employ the same format as an entry of the underlying index. For instance, the first line of the hash table in Figure 3a shows that R, located in the *height* 2, has the *status* MOD, and stores 2 in-memory modifications in the *mod_ tree*. Hence, the most recent version of the two entries of R are now the entries of the red-black tree, that is, $(I_1, MBR(I_1), 200, nSQ)$ and $(I_2, MBR(I_2), 100, SQ)$.

To improve the space utilization of the write buffer, splits are handled as follows. If the overflowed node, called A, has a hash entry in the *Write Buffer Table*, it first assumes *status* equal to DEL, deleting its previous modifications and thus freeing some space in the write buffer. Otherwise, a new hash entry with *status* equal to DEL is created in the *Write Buffer Table*. Then, after completing the splitting operation in the main memory, A has a new set of entries and a new node, called B, is created. Hence, the hash entry of A in the *Write Buffer Table* becomes NEW and the entries of A are added to its corresponding *mod_ tree*. A similar procedure for B is employed. An example of splitting operation is after inserting p_{17} (Figure 1b). As a result, L_4 has 4 modifications (fifth line in the *Write Buffer Table* of Figure 3a), where one modification is related to its deletion, another modification is related to its creation, and then two modifications for inserting its two entries (i.e., p_{14} and p_{17}). Further, N_1 is newly created in the write buffer (last line in the *Write Buffer Table* of Figure 3a), containing 4 modifications: one for its creation and 3 for inserting p_1, p_{13} and p_8.

The read buffer is implemented as another hash table named *Read Buffer Table* and caches nodes stored in the SSD that are frequently accessed. The key of this hash table is the unique node identifier (*page_ id*) and its value stores a list of entries of the node (*entries*) and its sub-quadrant, if it is an internal node (*reg*). A read buffer replacement policy is employed to decide which hash entries should be replaced if the read buffer is full and another hash entry must be stored in the read buffer. Experiments conducted in [Carniel et al. 2019] have shown that the Simplified 2Q [Johnson and Shasha 1994] as replacement policy guarantees good performance results for the read buffer. Figure 3b depicts that R, I_2, and L_5 are cached in the *Read Buffer Table*. In this figure, MBR_S refers to the stored data bounding rectangle of a child node. For instance, the entries of the cached version of I_2 consist of two entries, even after the creation of N_1.

Adapting the log file. To provide data durability, all modifications are also stored in a log file. The format of a log entry is very similar to the format of a hash entry in the *Write Buffer Table*. More specifically, each log entry consists of a tuple (*page_ id*, h, *reg*, *type_ mod*, *result*), where *page_ id* stores the identifier of the node, h is the height of the node, *reg* corresponds to the sub-quadrant of a newly created internal node and assumes *null* otherwise, *type_ mod* assumes NEW for newly created nodes, DEL for deleting nodes, FLUSH for flushing operations, and MOD otherwise. Finally, *result* is equivalent to an element of the red-black tree of the node in the *mod_ tree*. By storing this data, we are able to recover the write buffer after a fatal problem. The cost of keeping the log of the modifications is very low because it requires sequential writes only [Sarwat et al. 2013; Carniel et al. 2019]. Figure 3d depicts the log file for storing all the modifications of the *Write Buffer Table*



(a) *Write Buffer Table*

(b) *Read Buffer Table*

(c) *Temporal Control*

log#	page_id	h	reg	type_mod	result
1	L_1	0	-	MOD	p_{16}
2	I_1	1	-	MOD	(L_1, *MBR*(L_1), 200, nSQ)
3	R	2	-	MOD	(I_1, *MBR*(I_1), 200, nSQ)
4	L_4	0	-	DEL	-
5	L_4	0	-	NEW	-
6	L_4	0	-	MOD	p_{14}
7	L_4	0	-	MOD	p_{17}
8	N_1	0	-	NEW	-
9	N_1	0	-	MOD	p_{13}
10	N_1	0	-	MOD	p_1
11	N_1	0	-	MOD	p_8
12	I_2	1	-	MOD	(L_4, *MBR*(L_4), 100, nSQ)
13	I_2	1	-	MOD	(N_1, *MBR*(N_1), 50, SQ)
14	R	2	-	MOD	(I_2, *MBR*(I_2), 100, SQ)

(d) *Log file*

Fig. 3. Data structures to handle the modifications of the xBR$^+$-tree of Figure 1c. As a result, the eFIND xBR$^+$-tree is created.

of Figure 3a.

Unchanged structures. The temporal control of eFIND remains unchanged. The read and writes queues, named *RQ* and *WQ*, are employed to provide temporal control of eFIND. Each queue is a First-In-First-Out data structure. *RQ* stores identifiers of the nodes read from the SSD, while *WQ* keeps the identifiers of the last nodes written to the SSD. Figure 3c shows that the last read nodes are I_1, L_1, and L_3, and the last flushed nodes are L_3, L_2, and L_5.

Adapting the generic algorithms. The generic algorithms provided by eFIND can be applied to any eFIND-based spatial index to execute the following operations: (i) *maintenance operation*, which is responsible for reorganizing the index whenever modifications are made on the underlying spatial dataset (i.e., insertions, deletions, and updates), (ii) *search operation*, which is responsible for executing spatial queries, (iii) *flushing operation*, which selects a set of modifications stored in the write buffer to be written to the SSD according to a flushing policy, and (iv) *restart operation*, which rebuilds the write buffer after a fatal problem and compacts the log file.

To deal with the sorting property of the xBR$^+$-tree, we extend eFIND when retrieving nodes in search operations as follows. To retrieve a node N, the eFIND xBR$^+$-tree returns the modified entries stored in the *Write Buffer Table* or the entries stored in the SSD, if one of them is empty. The former

is empty if N has no modifications, while the latter is empty if there exists a hash entry of N in the *Write Buffer Table* with *status* equal to NEW. If one list is empty, the other non-empty list is directly returned. The second list is always sorted because its first flushing happens when its status in the *Write Buffer Table* is equal to NEW. If these lists are not empty, a classical merge operation between two sorted lists is performed [Folk et al. 1997]. That is, the lists are merged into a unique list, which represents the entries of N, by considering the same comparison function of the write buffer. During the merging, if two elements correspond to the same entry, the preference is for the element that is stored in the main memory since it represents the most recent version of the entry.

6. EXPERIMENTAL EVALUATION

This section presents our experiments conducted to measure the efficiency of the FAST xBR$^+$-tree and the eFIND xBR$^+$-tree. Section 6.1 presents the experimental setup. Performance results are presented in Sections 6.2 and 6.3.

6.1 Experimental Setup

Datasets. We used four spatial datasets. Two of them are synthetic datasets, called *synthetic1* and *synthetic2*, containing respectively 500,000 points and 1,000,000 points. Each synthetic dataset stores points equally distributed in 125 clusters uniformly distributed in the range $[0, 1]^2$. The points in each cluster (i.e., 4,000 points for *synthetic1* and 8,000 points for *synthetic2*) were located around the center of each cluster, according to Gaussian distribution. The methodology for creating these synthetic datasets is the same as the experiments conducted in [Roumelis et al. 2017]. The remaining two spatial datasets contain real data collected from OpenStreetMaps. The first one is a real spatial dataset, called *brazil_points2017*, containing 770,842 points inside Brazil. The second one, called *us_midwest_points2017*, contains 1,720,357 points inside the Midwest of the USA. These real spatial datasets were extracted using the methodology in [Carniel et al. 2017c] and represent geographical locations like public telephones, ATMs, and towers.

Configurations. We compared two configurations: (i) the *FAST xBR$^+$-tree* (Section 4), and (ii) the *eFIND xBR$^+$-tree* (Section 5). They had a buffer of 512KB, log capacity of 10MB, and employed index page sizes (i.e., node sizes) from 4KB to 32KB. For the FAST xBR$^+$-tree, we used the FAST* flushing policy, which provided the best results according to [Sarwat et al. 2013]. For the eFIND xBR$^+$-tree, we employed the best parameter values according to our experiments [Carniel et al. 2019]: the use of 60% of the oldest modified nodes to create flushing units, a flushing policy using the height of nodes as weight to choose one flushing unit to be written, and the allocation of 20% of the buffer for the read buffer. Finally, both configurations employed a flushing unit size equal to 5 since this value commonly provides good results for FAST and eFIND [Carniel et al. 2017b; 2019].

Workloads. We executed two workloads on each spatial dataset: (i) index construction by inserting points one-by-one, and (ii) execution of 300 intersection range queries (IRQs). An IRQ retrieves the points contained in a given rectangular query window, including its borders. Three different sets of query windows were used, representing respectively 100 rectangles with 0.001%, 0.01%, and 0.1% of the area of the total extent of the dataset being used by the workload. We generated different query windows for each dataset using the algorithms described in [Carniel et al. 2017c]. This method allows us to measure the performance of spatial queries with distinct selectivity levels. We consider the selectivity of a spatial query as the ratio of the number of returned objects and the total objects; thus, the three sets of query windows built IRQs with low, medium, and high selectivity, respectively. We executed the workloads as a sequence, that is, the index construction followed by the execution of IRQs. For each configuration and dataset, this sequence was executed 5 times. We avoided the page caching of the system by using direct I/O. For the first workload, we collected the average elapsed

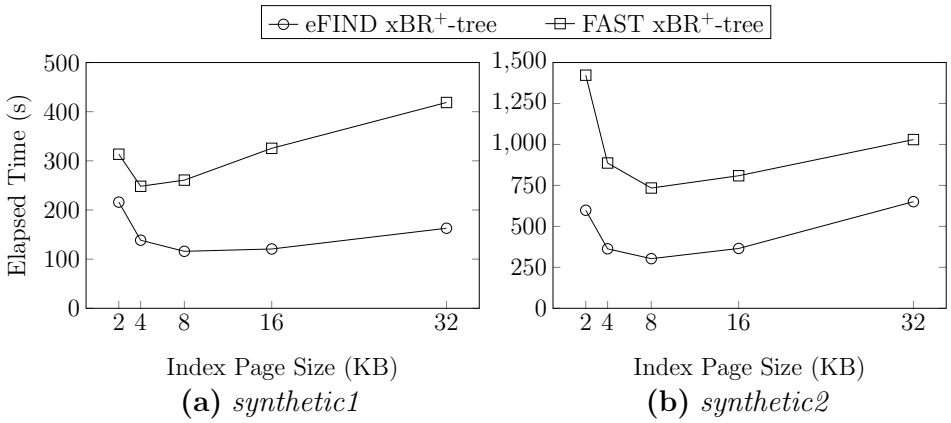


Fig. 4. The eFIND xBR$^+$-tree showed the best elapsed times when building spatial indices on the synthetic datasets.

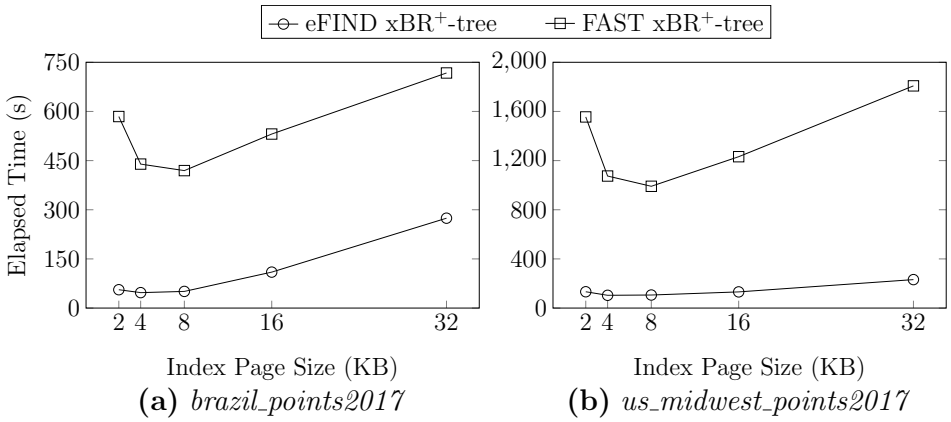


Fig. 5. The eFIND xBR$^+$-tree also showed the fastest elapsed times when building spatial indices on the real datasets. Its reductions were more expressive than those reported in Figure 4.

time. For the second workload, we calculated the average elapsed time to execute each set of query windows.

Running Environment. We employed a server equipped with an Intel Core® i7-4770 with a frequency of 3.40GHz, 32GB of main memory, and the SSD Kingston V300 of 480GB. The operating system used was Ubuntu Server 14.04 64 bits.

6.2 Index Construction

Figures 4 and 5 show that the eFIND xBR$^+$-tree always outperformed the FAST xBR$^+$-tree for all used spatial datasets. The performance gains of the eFIND xBR$^+$-tree against the FAST xBR$^+$-tree ranged from 30.8% to 62.8% for the synthetic spatial datasets (Figure 4) and from 61.7% to 91.4% for the real spatial datasets (Figure 5). A performance gain shows how much a configuration reduced the elapsed time from another configuration.

The eFIND xBR$^+$-tree exploited the benefits of the SSD because it exploits specific data structures and sophisticated methods that take into account the intrinsic characteristics of SSDs. That is, the adaptations performed on the eFIND fitted well with the structure of the xBR$^+$-tree. We highlight

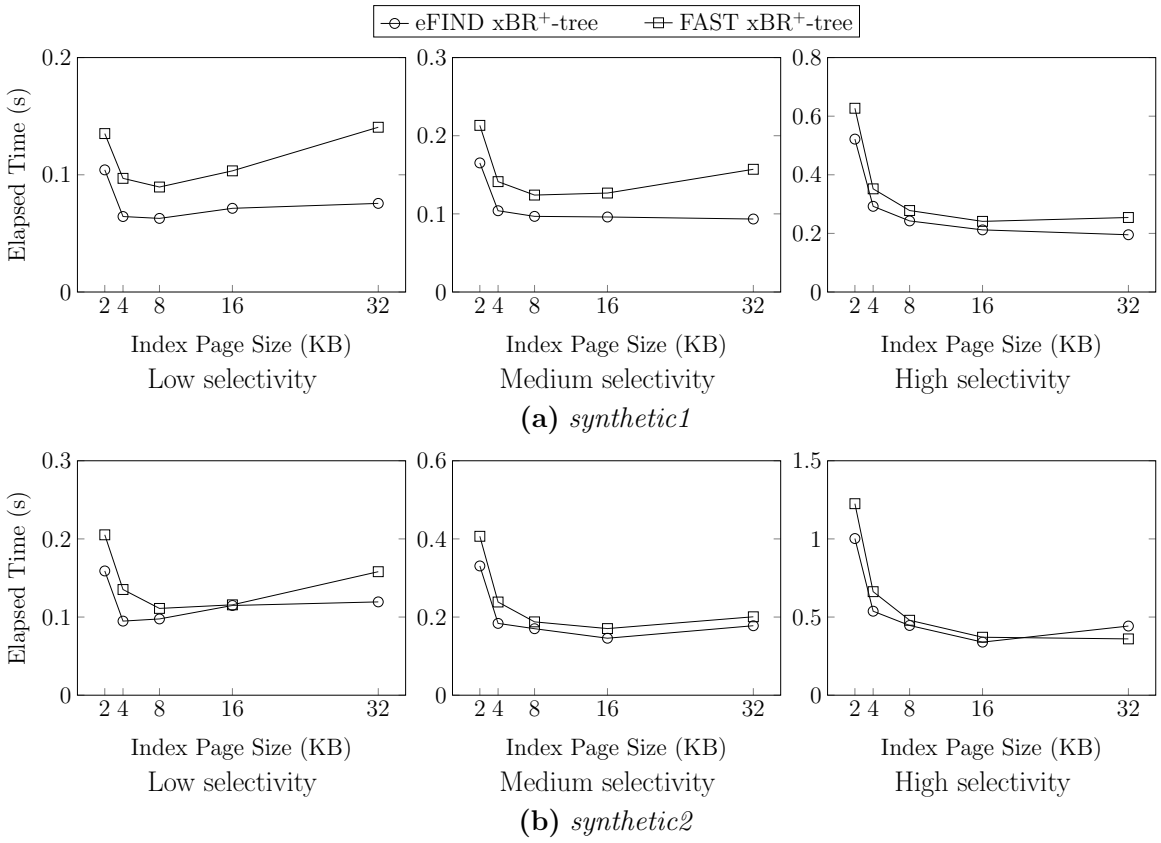


Fig. 6. Performance results when processing IRQs on the synthetic spatial datasets. For the *synthetic1*, the eFIND xBR$^+$-tree outperformed the FAST xBR$^+$-tree for all selectivity levels. For the *synthetic2*, the eFIND xBR$^+$-tree showed to be the best configuration since it reduced the elapsed time for almost all cases.

two main contributions. First, the use of the read buffer avoided several reads on frequent locations of the SSD, even using a small portion of the whole buffer size. Recall that such read buffer has been adapted to correctly manipulate a node of the xBR$^+$-tree. Second, the adapted write buffer of the eFIND xBR$^+$-tree naturally ensures the order of node entries. This aspect accelerates the retrieval of the most recent version of modified nodes. Other intrinsic aspects of eFIND also contributed to the performance results, such as the treatment of reads and writes.

The experiments also showed that FAST faces several problems. First, its flushing algorithm might pick nodes without modifications, resulting in unnecessary writes to the SSD. This is due to the static creation of flushing units as soon as nodes are created in the index. Second, its write buffer stores the modifications in a list possibly containing repeated entries, impacting negatively the performance of retrieving modified nodes during an index construction. Finally, FAST does not improve the performance of reads since it does not store nodes that are frequently read.

Building spatial indices on the voluminous datasets (i.e., *synthetic2* and *us_midwest_points2017*) required more time because they are larger than the other datasets (i.e., *synthetic1* and *brazil_points2017*). In several cases, the eFIND xBR$^+$-tree provided the best elapsed time by using the page size equal to 8KB. The use of larger page sizes brought the problem of writing big flushing units [Sarwat et al. 2013; Carniel et al. 2019], while the use of smaller page sizes introduced the management of a high number of nodes.

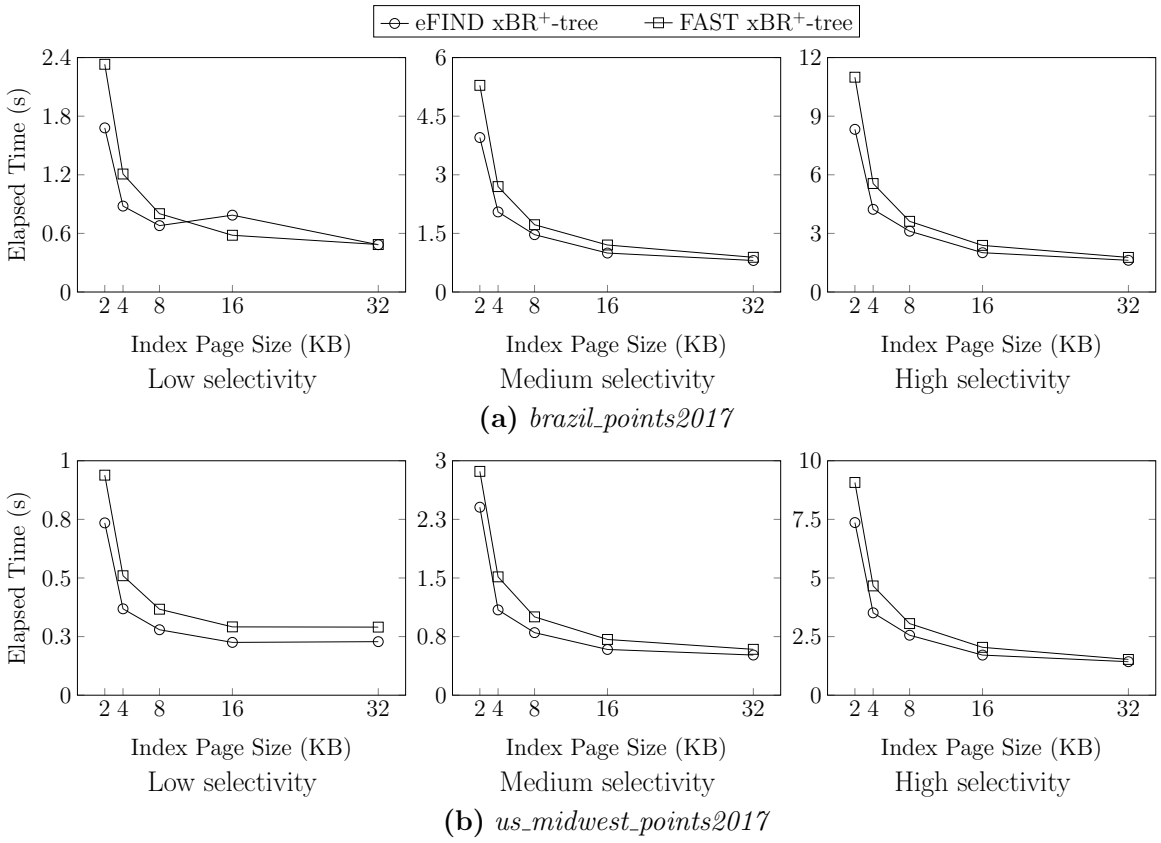


Fig. 7. Performance results when processing IRQs on the real spatial datasets. For the *brazil_points2017*, the eFIND xBR$^+$-tree reduced the elapsed time in almost all cases. For the *us_midwest_points2017*, the eFIND xBR$^+$-tree outperformed the FAST xBR$^+$-tree for all selectivity levels.

6.3 Spatial Query Processing

Figures 6 and 7 show that the eFIND xBR$^+$-tree was the best configuration for executing IRQs since it provided the best performance results in almost all cases. For the synthetic spatial datasets (Figure 6), it showed performance gains of up to 46%, 40.2%, and 22.5% for the low, medium, and high selectivity levels, respectively. For the real spatial datasets (Figure 7), the eFIND xBR$^+$-tree showed performance gains of up to 27.7%, 27.5%, and 24.2% for the low, medium, and high selectivity levels, respectively. Similar to our previous discussions (Section 6.2), these performance gains were obtained thanks to the effective use of the merge operation and read buffer.

Processing IRQs on the synthetic datasets required much less time than processing IRQs on the real datasets because of their specific spatial distribution. Another observation is related to the real datasets, where the execution of the IRQs on *brazil_points2017* was slower than the execution of the IRQs on *us_midwest_points2017*, although the latter dataset is more voluminous than the former. We believe that the main reason for this behavior is also the spatial distribution. That is, the spatial distribution of *us_midwest_points2017* is quite similar to the spatial distribution of the synthetic datasets, allowing a better internal organization of the xBR$^+$-tree.

In most of the cases, better elapsed times were obtained by using large page sizes (i.e., 16KB and 32KB) because more entries are loaded into the main memory with a few reads. IRQs returning more points (i.e., with high selectivity) exhibited higher elapsed times. This is due to the traversal of

multiple large nodes in the main memory, requiring more CPU time than queries with low selectivity. This fact also contributed to a similar time among the configurations when processing IRQs with high selectivity using the page size of 32KB.

7. CONCLUSIONS AND FUTURE WORK

This paper analyzes strategies to adapt the xBR$^+$-tree to SSDs by using two general frameworks, FAST and eFIND. As a result, two novel flash-aware spatial indices for points have been proposed, the FAST xBR$^+$-tree and the eFIND xBR$^+$-tree. To design these indices, we have mainly adapted the data structures of FAST and eFIND to deal with the structural constraints and properties of the xBR$^+$-tree. That is, our adaptations ensured the sorting property of entries stored in internal and leaf nodes. Since we have adapted the data structures of these frameworks, we also have slightly adapted their general algorithms.

We empirically analyzed the efficiency of the FAST xBR$^+$-tree and the eFIND xBR$^+$-tree by means of extensive experimental evaluations that considered two real spatial datasets and two synthetic spatial datasets. In general, the eFIND xBR$^+$-tree showed the best performance results. It provided performance gains ranging from 30.8% to 62.8% and from 61.7% to 91.4% when building indices on the synthetic datasets and on the real datasets, respectively. As for spatial query processing, the eFIND xBR$^+$-tree showed performance gains ranging from 22.5% to 46% and from 24.2% and 27.7% for the synthetic datasets and the real datasets, respectively.

In our experiments, the use of the page size equal to 16KB was the best solution for the employed configurations. Although this page size required more time to build an index compared to smaller page sizes, it provided good results to execute the IRQs. Hence, the cost of its construction can be counterbalanced by its efficiency when processing spatial queries.

The efficiency of the eFIND xBR$^+$-tree is obtained mainly because of two reasons. First, the internal structure of the xBR$^+$-tree was completely integrated to eFIND, guaranteeing all the properties of the xBR$^+$-tree that offer good spatial indexing performance. Second, eFIND is based on distinct design goals that fully exploit SSD performance. Thus, the eFIND xBR$^+$-tree takes into account many intrinsic characteristics of SSDs by providing write and read buffers to improve the performance of writes and reads, by specifying a temporal control to deal with interleaved reads and writes, by employing a specialized flushing algorithm, and by guaranteeing data durability.

Our future work will include an evaluation of the eFIND xBR$^+$-tree against other spatial organizations ported by eFIND, such as the data partitioning strategy of eFIND R-trees [Carniel et al. 2019]. Another future work is to extend our experiments to consider workloads that mix insertions and other types of queries, such as point queries.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work has also been supported by CNPq and by the São Paulo Research Foundation (FAPESP). Anderson C. Carniel was supported by the grants #2015/26687-8 and #2018/10687-7, FAPESP. Ricardo R. Ciferri has been supported by the grant #311868/2015-0, CNPq. Cristina D. A. Ciferri has been supported by the grant #2018/22277-8, FAPESP.

REFERENCES

Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference*. pp. 57–70, 2008.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD International Conference on Management of Data*. pp. 322–331, 1990.

Bouganim, L., Jónsson, B., and Bonnet, P. uFLIP: Understanding flash IO patterns. In *Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

Brayner, A. and Monteiro Filho, J. M. Hardware-aware database systems: A new era for database technology is coming - vision paper. In *Brazilian Symposium on Databases*. pp. 187–192, 2016.

Carniel, A. C. Spatial indexing on flash-based solid state drives. In *Proceedings of the VLDB 2018 PhD Workshop*. pp. 1–4, 2018.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. The performance relation of spatial indexing on hard disk drives and solid state drives. In *Brazilian Symposium on GeoInformatics*. pp. 263–274, 2016.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives. *Journal of Information and Data Management* 8 (1): 34–49, 2017a.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. A generic and efficient framework for spatial indexing on flash-based solid state drives. In *European Conference on Advances in Databases and Information Systems*. pp. 229–243, 2017b.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. Spatial datasets for conducting experimental evaluations of spatial indices. In *Satellite Events of the Brazilian Symposium on Databases - Dataset Showcase Workshop*. pp. 286–295, 2017c.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. A generic and efficient framework for flash-aware spatial indexing. *Information Systems* vol. 82, pp. 102–120, 2019.

Carniel, A. C., Roumelis, G., Ciferri, R. R., Vassilakopoulos, M., Corral, A., and Ciferri, C. D. A. An efficient flash-aware spatial index for points. In *Brazilian Symposium on GeoInformatics*. pp. 68–79, 2018.

Chen, F., Koufaty, D. A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. pp. 181–192, 2009.

Denning, P. J. Working sets past and present. *IEEE Transactions on Software Engineering* SE-6 (1): 64–84, 1980.

Emrich, T., Graf, F., Kriegel, H.-P., Schubert, M., and Thoma, M. On the impact of flash SSDs on spatial indexing. In *International Workshop on Data Management on New Hardware*. pp. 3–8, 2010.

Fevgas, A., Akritidis, L., Bozanis, P., and Manolopoulos, Y. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal*, 2019.

Fevgas, A. and Bozanis, P. Grid-file: Towards to a flash efficient multi-dimensional index. In *International Conference on Database and Expert Systems Applications*. pp. 285–294, 2015.

Folk, M. J., Zoellick, B., and Riccardi, G. *File Structures: An Object-Oriented Approach with C++*. Addison Wesley, 1997.

Gaede, V. and Günther, O. Multidimensional access methods. *ACM Computing Surveys* 30 (2): 170–231, 1998.

Hjaltason, G. R. and Samet, H. Ranking in spatial databases. In *International Symposium on Spatial Databases*. pp. 83–95, 1995.

Jin, P., Xie, X., Wang, N., and Yue, L. Optimizing R-tree for flash memory. *Expert Systems with Applications* 42 (10): 4676–4686, 2015.

Johnson, T. and Shasha, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Databases*. pp. 439–450, 1994.

Jung, M. and Kandemir, M. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. pp. 203–216, 2013.

Koltsidas, I. and Viglas, S. D. Data management over flash memory. In *ACM SIGMOD International Conference on Management of Data*. pp. 1209–1212, 2011a.

Koltsidas, I. and Viglas, S. D. Spatial data management over flash memory. In *International Conference on Advances in Spatial and Temporal Databases*. pp. 449–453, 2011b.

Li, G., Zhao, P., Yuan, L., and Gao, S. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. *The Journal of Supercomputing* 64 (3): 1055–1074, 2013.

Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D., and Najjar, W. A. Efficient indexing data structures for flash-based sensor devices. *ACM Transactions on Storage* 2 (4): 468–503, 2006.

Mittal, S. and Vetter, J. S. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems* 27 (5): 1537–1550, 2016.

Nievergelt, J., Hinterberger, H., and Sevcik, K. C. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems* 9 (1): 38–71, 1984.

Oosterom, P. V. a. N. Spatial Access Methods. In *Geographical Information Systems: Principles, Techniques, Management and Applications*, 2nd Edition ed., P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind (Eds.). pp. 385–400, 2005.

Rigaux, P., Scholl, M., and Voisard, A. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

Robinson, J. T. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *ACM SIGMOD International Conference on Management of Data*. pp. 10–18, 1981.

Roumelis, G., Fevgas, A., Vassilakopoulos, M., Corral, A., Bozanis, P., and Manolopoulos, Y. Bulk-loading and bulk-insertion algorithms for xBR$^+$-trees in solid state drives. *Computing*, 2019.

Roumelis, G., Vassilakopoulos, M., Corral, A., Fevgas, A., and Manolopoulos, Y. Spatial batch-queries processing using xBR$^+$-trees in solid-state drives. In *International Conference on Model and Data Engineering*. pp. 301–317, 2018.

Roumelis, G., Vassilakopoulos, M., Corral, A., and Manolopoulos, Y. Efficient query processing on large spatial databases: A performance study. *Journal of Systems and Software* vol. 132, pp. 165–185, 2017.

Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., and Manolopoulos, Y. The xBR+-tree: an efficient access method for points. In *International Conference on Database and Expert Systems Applications*. pp. 43–58, 2015.

Samet, H. The quadtree and related hierarchical data structures. *ACM Computing Surveys* 16 (2): 187–260, 1984.

Sarwat, M., Mokbel, M. F., Zhou, X., and Nath, S. FAST: A generic framework for flash-aware spatial trees. In *International Conference on Advances in Spatial and Temporal Databases*. pp. 149–167, 2011.

Sarwat, M., Mokbel, M. F., Zhou, X., and Nath, S. Generic and efficient framework for search trees on flash memory storage systems. *GeoInformatica* 17 (3): 417–448, 2013.

Sellis, T. K., Roussopoulos, N., and Faloutsos, C. The R+-Tree: A dynamic index for multi-dimensional objects. In *International Conference on Very Large Databases*. pp. 507–518, 1987.

Silva, F. A., Domingues, A. C. S. A., and Silva, T. R. M. B. Discovering mobile application usage patterns from a large-scale dataset. *ACM Transactions on Knowledge Discovery from Data* 12 (5): 59:1–59:36, 2018.

Wu, C.-H., Chang, L.-P., and Kuo, T.-W. An efficient R-tree implementation over flash-memory storage systems. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. pp. 17–24, 2003.