An Efficient Flash-aware Spatial Index for Points

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

[1]Department of Computer Science – University of São Paulo – Brazil

accarniel@gmail.com, cdac@icmc.usp.br

[2]Department of Electrical and Computer Engineering – University of Thessaly – Greece

groumelis@uth.gr, mvasilako@uth.gr

[3]Department of Computing – Federal University of São Carlos – Brazil

ricardo@dc.ufscar.br

[4]Department on Informatics – University of Almeria – Spain

acorral@ual.es

Abstract. *Spatial database systems often employ spatial indices to speed up the processing of spatial queries. In addition, modern spatial database applications are interested in exploiting the positive characteristics of flash-based Solid State Drives (SSDs) like fast reads and writes. However, designing spatial indices for SSDs (i.e., flash-aware spatial indices) has been a challenging task because of the intrinsic characteristics of these devices. In this paper, we propose the eFIND xBR$^+$-tree, a novel flash-aware spatial index for points. The eFIND xBR$^+$-tree combines the efficient indexing method of the xBR$^+$-tree with the sophisticated data structures and algorithms of eFIND to handle points in SSDs efficiently. Experiments carried out considering real and synthetic spatial data showed that the eFIND xBR$^+$-tree overcame its closest competitor by reducing the elapsed time to construct the index from 28.4% to 83.5% and to execute spatial queries up to 34.6%.*

1. Introduction

The use of a spatial index is essential for processing spatial queries because the search space is greatly reduced [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 and the R*-tree, 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 [Gaede and Günther 1998]. 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 eval-

uations [Roumelis et al. 2017] showed that the xBR$^+$-tree outperforms variants of the R-tree (the R*-tree and the R$^+$-tree) when processing different types of spatial queries.

On the other hand, advanced database applications are interested in using modern storage devices like *flash-based Solid State Drives* (SSDs) [Mittal and Vetter 2016]. This includes spatial database systems that employ spatial indices to efficiently retrieve spatial objects stored in SSDs [Carniel et al. 2017a]. The main reason of 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 [Jung and Kandemir 2013, Mittal and Vetter 2016]. 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 [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, Carniel et al. 2018] 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 to recover its write buffer after a fatal problem (e.g., power failure). Comparing FAST to eFIND, the former does not fully exploit SSD performance because it does not consider several intrinsic characteristics of SSDs. On the other hand, eFIND contains managers based on a set of design goals that are developed to fully take into account the intrinsic characteristics of SSDs. Hence, we consider eFIND as the state-of-the-art method for porting disk-based spatial indices to SSDs.

Considering the aforementioned state-of-the-art methods, an open question is how to efficiently port the xBR$^+$-tree to SSDs using eFIND. In this paper, we answer this question by proposing the *eFIND xBR$^+$-tree*, a flash-aware spatial index for points. This novel index combines the efficient spatial organization of xBR$^+$-trees with the sophisticated managers of eFIND specifically designed for SSDs. That is, the eFIND xBR$^+$-tree is designed as an integration of the xBR$^+$-tree's hierarchical structure with the eFIND's data structures. We measure the efficiency of this porting by conducting experimental evaluations, considering real and synthetic datasets, against the FAST xBR$^+$-tree, the porting of the xBR$^+$-tree to SSDs using FAST. Our performance results show that the eFIND xBR$^+$-tree ports the xBR$^+$-tree to SSDs efficiently, guaranteeing smaller elapsed times to process insertions and intersection range queries.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 presents the eFIND xBR$^+$-tree. Section 4 discusses the conducted experiments. Finally, Section 5 concludes the paper and presents future work.

2. Related Work

A few flash-aware spatial indices 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. The main problem of RFTL is the flushing operation because it flushes all modifications stored in the write buffer, requiring high elapsed times. Another problem is related to the data durability. This means that the modifications stored in the write buffer are lost after a system crash or power failure.

FAST [Sarwat et al. 2013] distinguishes itself because it 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 picks only a set of nodes, termed *flushing unit*, to be written to the SSD instead of writing all modifications contained in the write buffer. FAST also provides support for data durability. However, 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. Finally, FAST does not improve the performance of reads.

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 disallowed us to create spatial indices over large and medium spatial datasets.

Specific flash-aware spatial indices for points have also been developed. *Micro-Hash* and *MicroGF* [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, called logging entries. Its main problem is the complex operation to retrieve nodes 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 LRU to cache modifications of the grid file. 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.

eFIND [Carniel et al. 2017b, Carniel et al. 2018] is a generic framework that efficiently transforms any disk-based spatial index into a flash-aware spatial index. It is based

on distinct design goals that considers the intrinsic characteristics of SSDs. eFIND employs efficient in-memory data structures to handle index modifications and a specialized flushing operation that smartly picks a number of nodes to be written to the SSD. Further, eFIND prevents reads on frequent locations and avoids interleaved reads and writes. Due to its advantages, we consider eFIND as the state-of-the-art method for porting disk-based spatial indices to SSDs. Hence, we employ eFIND to port the xBR$^+$-tree to SSDs.

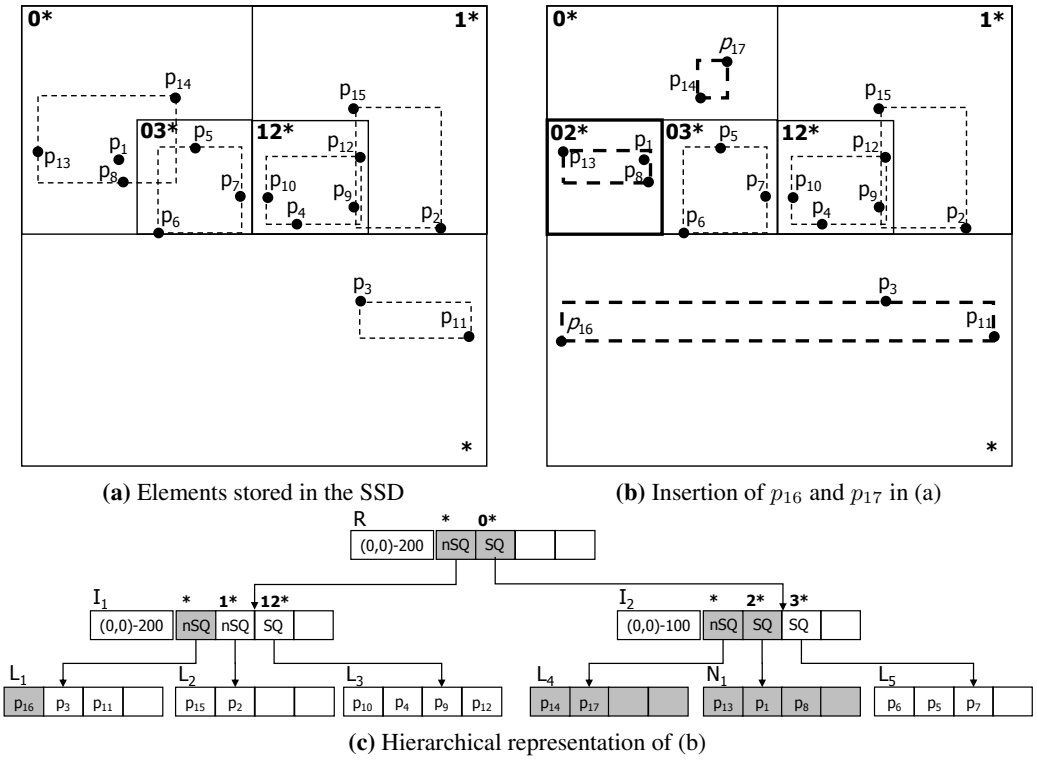
3. The eFIND xBR$^+$-tree: An Efficient Flash-Aware Spatial Index for Points

3.1. The Tree Structure

eFIND does not change the underlying tree structure of the ported index. Hence, the tree structure of the eFIND xBR$^+$-tree is the same as the xBR$^+$-tree. The xBR$^+$-tree is a hierarchical index based on the regular decomposition of space of Quadtrees [Gaede and Günther 1998] able to index multidimensional points. Hence, it is a *space-driven access method*. For bidimensional points, the xBR$^+$-tree decomposes recursively the space by 4 equal quadrants, called *sub-quadrants*. Figure 1a depicts an example of an eFIND xBR$^+$-tree that indexes 15 points (i.e., p_1 to p_{15}) and is whole stored in the SSD. Figure 1b shows the eFIND 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 are modifications stored in the main memory (Section 3.2) and are also highlighted in the hierarchical representation of Figure 1c. We detail the structure of this eFIND xBR$^+$-tree as follows.

There are two types of nodes, internal nodes and leaf nodes. Internal nodes consist of entries in the following format $(p, DBR, qside, shape)$. Each entry of an internal node refers to a child node that is pointed by p and represents a sub-quadrant of the original space. DBR refers to the data bounding rectangle that minimally encompasses the points stored in such 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 NW, NE, SW, and 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. 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 showed 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_5 to p_8, p_{14}, p_{17}, p_{13}, and p_1. The left child of R represents a region derived from the spatial difference between the original space and the region of the NW quadrant. Hence, it has address equal to * (i.e., empty) and represents a non-complete square (i.e., *nSQ*). 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 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).



(a) Elements stored in the SSD

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

(c) Hierarchical representation of (b)

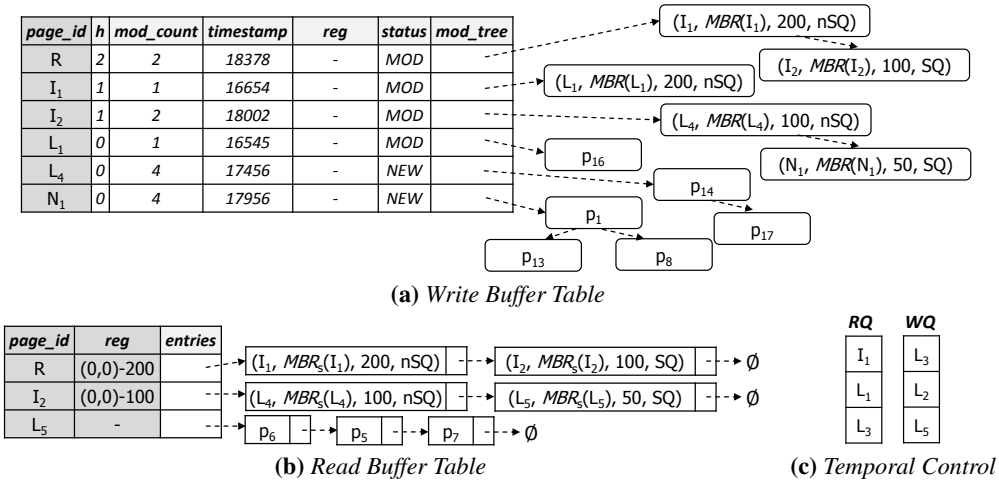
Figure 1. An example of an eFIND xBR$^+$-tree.

Leaf nodes contain entries in the format (p, id), 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.

When the capacity of a leaf or internal node is achieved, 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, as discussed in 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}.

3.2. Employed Data Structures

eFIND provides specific data structures to fulfill its design goals [Carniel et al. 2018]; they are: (i) a write buffer, (ii) a read buffer, (iii) a log file, and (iii) 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 (Section 3.1), and (iii) we adjust the structure of log entries to recover the write buffer after a system crash. We detail these extensions as follows.



(a) *Write Buffer Table*

(b) *Read Buffer Table*

(c) *Temporal Control*

Figure 2. Data structures to handle the eFIND xBR$^+$-tree of Figure 1.

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. The comparison function to determine the order of the elements in the red-black tree is defined to deal with the specific sorting of entries of internal and leaf nodes (Section 3.1). This is important when retrieving nodes (Section 3.3).

Figure 2a shows the *Write Buffer Table* for the eFIND xBR$^+$-tree of Figure 1b. In this figure, MBR means the rectangle that encompasses all points of a sub-quadrant considering the modifications stored in the write buffer. 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 2a 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, i.e., $(I_1, MBR(I_1), 200, nSQ)$ and $(I_2, MBR(I_2), 100, SQ)$.

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*). Figure 2b 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 consists of two entries, even after the creation of N_1.

To provide data durability, all modifications are also stored in a log file. The format of a log entry is the same as a hash entry in the *Write Buffer Table* to rebuild the write buffer after a system crash. 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. 2018].

The temporal control of eFIND remains unchanged. The read and writes queues, named *RQ* and *WQ*, are employed to provide the 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 2c 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.

3.3. Methods for Handling the Index Operations

eFIND provides generic algorithms to execute the following operations: (i) maintenance operation, which is responsible for reorganizing the index whenever modifications are made on the underlying spatial dataset (i.e., insertions, deletions, and updates); (ii) search operation, which is responsible for executing spatial queries; (iii) flushing operation, which selects a set of modifications stored in the write buffer to be written to the SSD according to a flushing policy; and (iii) restart operation, which rebuilds the write buffer after a fatal problem and compacts the log file. To deal with the xBR$^+$-tree, we extend eFIND as follows: (i) we generalize the retrieval algorithm of eFIND to return valid internal and leaf nodes, respecting their sorting properties (Section 3.1), and (ii) we detail the management of splits, which improves the space utilization of the write buffer.

To retrieve a node N, the eFIND xBR$^+$-tree takes two sorted lists as input: (i) the modified entries stored in the *Write Buffer Table*, and (ii) the entries stored in the SSD. The former is empty if N has not 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.

The following merge operation should be performed if these lists are not empty. It is based on the classical merge operation between sorted files [Folk et al. 1997]. Let i, j be two integer values, where i indicates the position in the first list and j indicates the position in the second list. A loop is then processed, starting with $i = j = 0$. If the element in the position i on the first list, called E_a, goes before the element in the position j on the second list, called E_b, this means that the merge operation appends E_a to N and increments i by 1 since an element of the first list has been processed. If the inverse happens, i.e., E_b goes before E_a, the merge operation appends E_b to N and increments j by 1. Evaluating the order of two node entries requires the execution of the same comparison function employed by the red-black trees (Section 3.1). If E_a and E_b point to the same entry (i.e., their unique identifier are equal), the merge operation appends only E_a to N if its value (i.e., *mod_result* in the *mod_tree*) is different to null and increment both i and j by 1. This is done because the result should only maintain the latest version of the entry and non-null entries. The loop is finished if i (j) is equal to the number of entries in the first (second) list. Finally, the entries that were not evaluated by the loop are appended to N, which is returned as the final step of the merge operation.

The merge operation requires a cost of $\mathcal{O}(n + m)$, where n is the number of elements in the first list and m is the number of elements in the second list. The use of a red-black tree for storing modified entries is essential for the merge operation and represents a main advantage compared to FAST. First, it guarantees the order between the entries stored in the write buffer. Hence, the resulting node is valid. Second, it has an amortized cost of inserting and updating entries stored in the main memory. Finally, the space allocated in the main memory is better managed because it does not allow repeated elements. All these factors combined to the other eFIND's managers lead to a better performance compared to porting the xBR$^+$-tree using FAST, as reported in our experiments (Section 4).

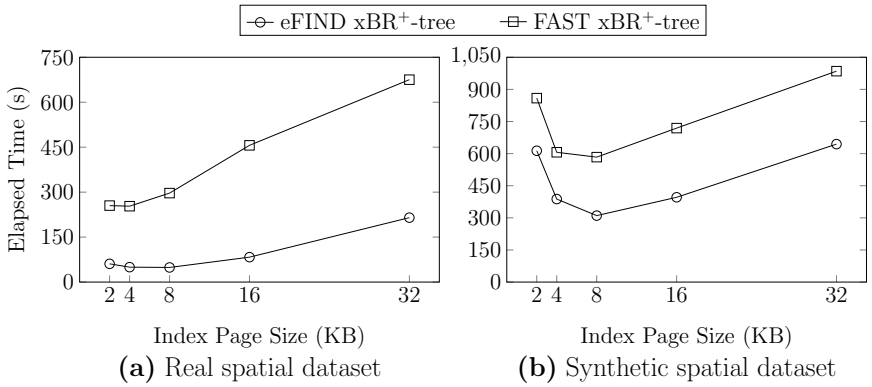
Handling splitting operations in the write buffer is performed as follows. Let A be an overflowed node. First, if A has a hash entry in the *Write Buffer Table*, it assumes *status* equal to DEL, deleting previous modifications of A and thus freeing some space in the write buffer. Otherwise, a new hash entry, with *status* equal to DEL, in the *Write Buffer Table* is created. 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 in its corresponding *mod_tree*. A similar procedure for B is employed. This strategy for handling splitting operations is important because of the management of the write buffer space. An example of handling of a splitting operation is depicted in Figure 1c, after inserting p_{17}. As a result, L_4 has 4 modifications (fifth line in the *Write Buffer Table* of Figure 2a), where one modification is related to its deletion, another modification for its creation, and then two modifications for inserting its two entries. Further, N_1 is newly created in the write buffer (last line in the *Write Buffer Table* of Figure 2a).

4. Experimental Evaluation

4.1. Experimental Setup

Datasets. We used two spatial datasets. The first one is a real spatial dataset, called *brazil_points2017*, containing 770,842 points that represent geographical locations of Brazil like public telephones, ATMs, and towers. This dataset was extracted from the OpenStreetMap and its statistical description can be found in Carniel et al. 2017c. The second one is a synthetic dataset containing 1,000,000 points equally distributed in 125 clusters uniformly distributed in the range $[0, 1]^2$. The points in each cluster (i.e., 8,000 points) were located around the center of each cluster, according to Gaussian distribution.

Configurations. We compared two configurations: (i) the *FAST xBR$^+$-tree*, which is our closest competitor (Section 2), and (ii) the *eFIND xBR$^+$-tree*, which is our proposed index. We created the FAST xBR$^+$-tree by extending FAST in an analogous way to the extensions we performed to eFIND. However, due to space limitations, this extension is not presented here. Both configurations 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. 2018]: 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.



(a) Real spatial dataset (b) Synthetic spatial dataset

Figure 3. The eFIND xBR$^+$-tree showed the fastest elapsed times when building spatial indices over both spatial datasets.

Finally, both configurations employed the flushing unit size equal to 5 since this value commonly provide good results for FAST and eFIND [Carniel et al. 2018].

Workloads. We executed two workloads: (i) index construction, 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 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.

4.2. Performance Results

Index Construction. Figure 3 depicts that the eFIND xBR$^+$-tree overcame the FAST xBR$^+$-tree for both spatial datasets. The performance gains of the eFIND xBR$^+$-tree ranged from 68.1% to 83.5% for the real spatial dataset (Figure 3a) and from 28.4% to 46.5% for the synthetic spatial dataset (Figure 3b). 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 leverages specific data structures and sophisticated methods that take into account the intrinsic characteristics of SSDs. We highlight three main contributions. First, the use of the read buffer

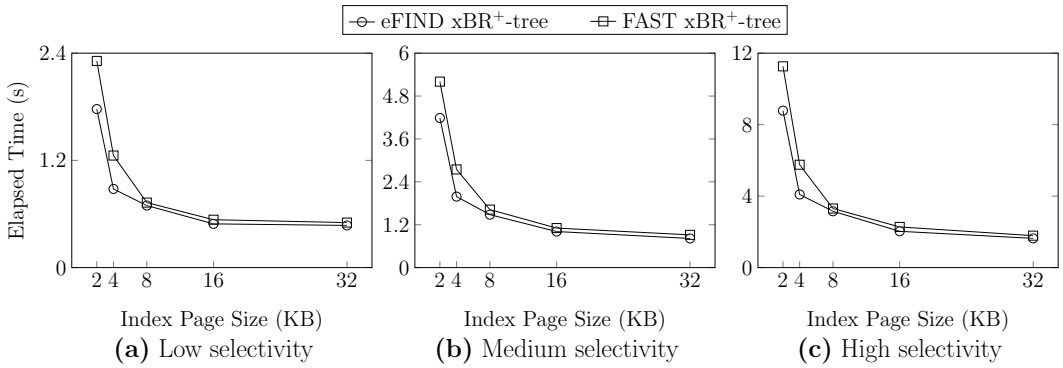


Figure 4. Performance results when processing IRQs on the real spatial dataset. The eFIND xBR$^+$-tree outperformed the FAST xBR$^+$-tree for all selectivity levels, showing expressive performance gains.

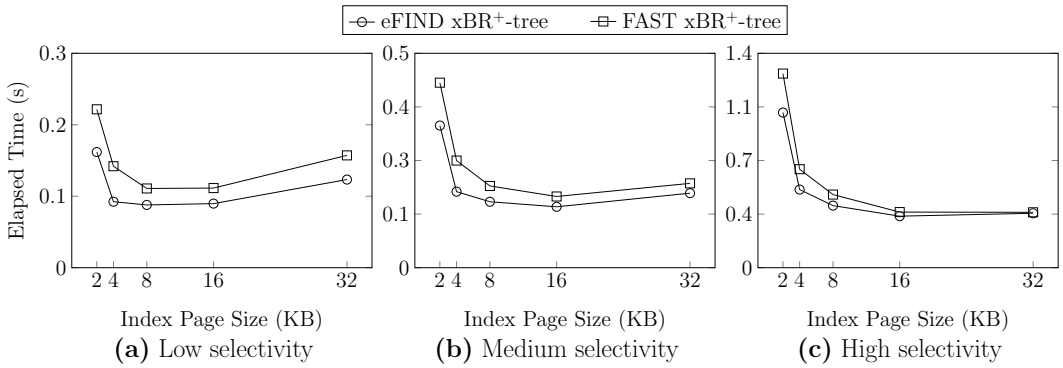


Figure 5. Performance results when processing IRQs on the synthetic spatial dataset. The eFIND xBR$^+$-tree showed better elapsed time than the FAST xBR$^+$-tree for all selectivity levels.

avoided several reads on frequent locations of the SSD, even using a small portion of the whole buffer size. Second, the merge operation accelerated the retrieval of the most recent version of modified nodes. This operation also naturally guaranteed the order of node entries. Finally, the eFIND xBR$^+$-tree avoided interleaved reads and writes.

Building spatial indices over the synthetic spatial dataset required more time because it is larger than the real spatial dataset. In both spatial datasets, the eFIND xBR$^+$-tree provided the best elapsed time by using the page size equal to 8KB. The use of larger page sizes faced the problem of writing big flushing units [Sarwat et al. 2013, Carniel et al. 2018], while the use of smaller page sizes introduced the management of a high number of nodes.

Spatial Query Processing. Figures 4 and 5 depict that the eFIND xBR$^+$-tree always provided the best performance results when processing all selectivity levels of IRQs. For the real spatial dataset (Figure 4), the eFIND xBR$^+$-tree showed performance gains up to 29.4%, 27.2%, and 28.5% for the high, medium, and high selectivity levels, respectively. For the synthetic spatial dataset (Figure 5), it showed performance gains up to 34.6%, 28.6%, and 20.2% for the high, medium, and high selectivity levels, respectively. Sim-

ilarly to our previous discussions, these performance gains were obtained thanks to the effective use of the merge operation and read buffer.

Processing IRQs over the synthetic dataset required much less time than processing IRQs over the real dataset because of its specific spatial distribution. 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.

5. Conclusions and Future Work

This paper proposes the eFIND xBR$^+$-tree, a novel flash-aware spatial index for points. eFIND allowed to efficiently port the xBR$^+$-tree to SSDs because its data structures fit well the properties and spatial organization of the xBR$^+$-tree. To accomplish this porting, eFIND has been generalized to deal with the sorting properties of nodes and to efficiently handle modifications produced by the xBR$^+$-tree.

The eFIND xBR$^+$-tree has empirically evaluated against the FAST xBR$^+$-tree, which employed FAST to port the xBR$^+$-tree to SSDs. The eFIND xBR$^+$-tree provided performance gains from 28.4% to 83.5% when building spatial indices and up to 34.6% when processing IRQs. In general, the page size of 16KB was the best configuration. Although this page size required more time to build an index compared to smaller page sizes, it provided the best results to execute the IRQs. Hence, the cost of its construction can be suppressed 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.

Our future work includes to evaluate the eFIND xBR$^+$-tree against other spatial organizations, such as the data partitioning strategy of eFIND R-trees [Carniel et al. 2018]. 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 has been 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

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. (2017a). 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.

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

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

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. (2018). A generic and efficient framework for flash-aware spatial indexing. *Information Systems*. https://doi.org/10.1016/j.is.2018.09.004.

Fevgas, A. and Bozanis, P. (2015). Grid-file: Towards to a flash efficient multi-dimensional index. In *Int. Conf. on Database and Expert Systems Applications*, pages 285–294.

Folk, M. J., Zoellick, B., and Riccardi, G. (1997). *File Structures: An Object-Oriented Approach with C++*. Addison Wesley, 3rd edition.

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

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

Jung, M. and Kandemir, M. (2013). Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, pages 203–216.

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

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

Mittal, S. and Vetter, J. S. (2016). 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.

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

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

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

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