# A Generic and Efficient Framework for Flash-Aware Spatial Indexing

Anderson C. Carniel[a,*], Ricardo R. Ciferri[b], Cristina D. A. Ciferri[a]

[a]*Department of Computer Science, University of São Paulo, São Carlos, SP 13566-590, Brazil*

[b]*Department of Computing, Federal University of São Carlos, São Carlos, SP 13565-905, Brazil*

---

**Abstract**

Spatial indexing on *flash-based Solid State Drives* (SSDs) has become a core aspect in spatial database applications, and has been carried out by *flash-aware spatial indices.* Although there are some flash-aware spatial indices proposed in the literature, they do not exploit all the benefits of SSDs, leading to loss of efficiency and durability. In this article, we propose eFIND, a new generic and efficient framework for flash-aware spatial indexing. eFIND takes into account the intrinsic characteristics of SSDs by employing (i) a *write buffer* to avoid expensive random writes, (ii) a *flushing algorithm* that smartly picks modifications to be flushed in batch to the SSD, (iii) a *read buffer* to decrease the overhead of random reads, (iv) a *temporal control* to avoid interleaved reads and writes, and (v) a log-structured approach to provide *data durability.* Performance tests showed the efficiency of eFIND. Compared to the state of the art, eFIND improved the construction of spatial indices from 43% to 77%, and the spatial query processing from 4% to 23%.

*Keywords:* spatial indexing, spatial access methods, flash memory, SSDs, flash-aware spatial index, spatial database systems

---

*Corresponding author.

*Email addresses:* `accarniel@gmail.com` (Anderson C. Carniel), `ricardo@dc.ufscar.br` (Ricardo R. Ciferri), `cdac@icmc.usp.br` (Cristina D. A. Ciferri)

doi: https://doi.org/10.1016/j.is.2018.09.004

## 1. Introduction

Spatial indices are largely employed to improve spatial query processing since they reduce the search space by discarding portions of the dataset where the answer cannot be found [1, 2]. Nowadays, there is an increasing number of spatial database applications requiring the use of spatial indices to retrieve efficiently spatial objects stored in *flash-based Solid State Drives* (SSDs) [3, 4, 5, 6, 7]. In fact, SSDs have been widely used as secondary storage in notebooks, desktops, and database servers because of their improved characteristics compared to Hard Disk Drives (HDDs) [8]. These characteristics include smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

On the other hand, SSDs have intrinsic characteristics that introduce several system implications [9, 10]. A well-known characteristic is that a write requires more time and power consumption than a read. In addition, random writes can lead to high costly erase-before-update operations and thus, sequential writes are preferable. To deal with these characteristics, some *flash-aware spatial indices* have been proposed in the literature [11, 12, 13, 14, 15, 16, 17, 18]. Among them, FAST-based indices [14] distinguish themselves by providing efficiency and data durability.

Commonly, existing flash-aware spatial indices extend spatial indices originally designed for HDDs like the R-tree [19] (termed as *disk-based spatial indices*). Instead of directly performing random writes to the SSD, flash-aware spatial indices store index modifications in an in-memory buffer. When this buffer is full, a flushing policy picks a set of modified index pages to be sequentially written to the SSD.

However, current flash-aware spatial indices do not exploit all the benefits of SSDs. First, the management of the modifications contained in the buffer is based on inefficient data structures, such as lists with repeated elements. Second, these indices execute an excessive number of random reads, which can degenerate SSD performance [10]. Third, they perform interleaved reads and writes, also negatively impacting the SSD performance [9, 10]. Finally, they employ flushing algorithms that may lead to unnecessary writes to SSDs.

In this article, we solve the aforementioned problems by proposing the *efficient Framework for spatial INDexing on SSDs* (eFIND). It is a generic framework that transforms a disk-based spatial index into a flash-aware spatial index without requiring modifications in the structure and algorithms of the underlying index. Instead, eFIND efficiently changes the way in which reads and writes are performed on the SSD. This characteristic allows us to incorporate eFIND into existing spatial database systems with low implementation costs. eFIND is also efficient because it is based on a set of design goals specifically designated to take into account the intrinsic characteristics of SSDs.

Our experiments showed that eFIND is very efficient since it provides a consonance among the following elements:

- *write buffer* that leverages efficient data structures in the main memory to avoid random writes to the SSD;
- *flushing algorithm* that makes use of a *flushing policy* to pick modified index pages to be sequentially written to the SSD;
- *read buffer* that employs a *read buffer replacement policy* to cache index pages frequently accessed, decreasing the overhead of random reads;
- *temporal control* that stores identifiers of read and written index pages to avoid interleaved reads and writes;
- *log-structured approach* to guarantee data durability.

The rest of this article is organized as follows. Section 2 describes the intrinsic characteristics of SSDs and their impact on applications. Section 3 introduces our design goals that serve as a basis for eFIND. Section 4 proposes eFIND, which deploys specific data structures and algorithms to fulfill the design goals. Section 5 experimentally compares eFIND with the state of the art. Section 6 conducts a performance evaluation to study the effect of our design goals. Section 7 surveys related work and describes how this article extends our previous work [20]. Finally, Section 8 concludes the article and presents future work.

## 2. An Overview of Flash-based Solid State Drives

An SSD consists of several components, such as multiple cores, internal DRAM/SRAM buffer, and storage media [21, 10]. Our focus is on the storage media, which is an array of *flash memory packages* that delivers high storage capacity and internal parallelism of reads and writes. A flash memory package is made up of a set of dies (chips). A die is divided into multiple planes, where each plane consists of several blocks and a few registers. Each block is composed of a number of pages; thus, flash memory is *block-oriented*. A page stores data and metadata, such as the Error Correcting Code (ECC).

Flash memory supports erase, read and write (program) operations [21, 9, 10]. Erase is performed in block granularity, leading to the most expensive operation of the flash memory. Read and write are page level operations with asymmetric costs, where normally a read requires much less time and power consumption than a write. The write operation is only performed on cleaned pages since update operations are not supported. Hence, a *sequential write*, where pages in one block are written sequentially throughout the block, is preferable. Otherwise, an expensive *erase-before-update* operation is performed. This operation first saves the data of the block containing the page being updated, then erases the block, and finally writes the content of the modified page together with the non-modified content of the block. Further, flash memory has a limited endurance. After a number of writes and erases on a block, it can no longer store data.

There are a number of other factors that impact on SSD performance [22, 9, 10, 23]. The use of ECC, which introduces some overhead on reads and writes, can correct some bit errors on pages. However, bit error rate increases exponentially as writes and erases are performed on a block. Blocks with a high error rate, reaching to an error recovery coverage limit, are marked as bad blocks and need a *bad block management* [10] that degrades system performance.

The *read disturbance management* [10] is another management that requires extra computational time of SSDs. This management is needed to avoid read disturbances, which occur if multiple reads are issued on the same page without any erase. Because of a disturbance, reads can require long latencies similar to the write latency.

Another factor is that *reads and writes interfere with each other* [9, 10, 23]. A critical interference is when a write is performed on a page and this page is subsequently read. In this case, the read operation must wait for the write request because the write is internally buffered. Similarly to read disturbances, the interference makes reads require long latencies. Hence, mixing reads and writes negatively impact SSD performance.

The last factor is related to the *Flash Translation Layer* (FTL) [9, 24, 25, 26]. FTL is physically located inside of the SSD and deploys sophisticated algorithms to optimize SSD performance. It emulates the interface of HDDs to provide an easy integration of SSDs with existing computational systems. This interface maps the physical addresses of the array of flash memory packages into logical addresses that are used by applications. Further, FTL employs an out-of-place update algorithm to avoid erase-before-update operations, a garbage collector to reclaim pages and blocks, and a wear leveling to improve flash memory endurance. These complex algorithms can introduce additional computational costs if applications do not take into account the aforementioned intrinsic characteristics of SSDs.

## 3. Design Goals for Flash-Aware Spatial Indices

Despite the fact that intrinsic characteristics of SSDs have been well studied in the literature, it remains unclear how to deal with them to achieve good spatial indexing performance. In this section, we provide a conceptual discussion of the underlying ideas to solve this problem by introducing a set of *design goals for flash-aware spatial indices. They are inspired by existing flash-aware techniques (Section 7) and represent reasonable solutions that take into account the intrinsic characteristics of SSDs (Section 2).* Hence, our design goals can even serve as a foundation to implement other types of flash-aware algorithms, such as flash-aware unidimensional index structures (e.g., B-trees). In this article, we focus on spatial indexing on SSDs and our discussion is directed toward using these design goals as a basis to create efficient and robust flash-aware spatial indices. We detail each design goal as follows.

**Goal 1 - Avoid random writes.** Random writes are expensive and can lead to erase-before-update operations, bad block management, and poor performance of FTL algorithms. To achieve Goal 1, a flash-aware spatial index should employ a buffer in the main memory, called *write buffer*, to store the most recent modifications of the index. Hence, it should avoid random writes from being directly performed on the SSD. Further, the write buffer should leverage efficient in-memory data structures since retrieving an index page with modifications involves the integration of its modifications stored in the write buffer with its version stored in the SSD. Whenever the write buffer is full, a *flushing algorithm* should be executed to give space for storing new modifications. This algorithm should write sequentially a set of modifications to the SSD as specified in Goal 2.

**Goal 2 - Dynamically pick modifications to be sequentially flushed.** A flushing operation that writes all modifications contained in the write buffer leads to a big write to the SSD, degenerating its performance. Further, it writes index pages that would be potentially modified soon [14]. To achieve Goal 2, a flash-aware spatial index should include a *specialized flushing algorithm* consisting of a *flushing policy* and a *flushing unit creator*. The flushing policy should pick the modified index pages to be written, according to distinct criteria. For instance, a modified index page can be picked based on its number of modifications, the moment of its last modification, and/or internal characteristics (e.g., its height). The flushing unit creator should organize index pages in flushing units, following the flushing policy, and determine the size of data that is written to the SSD in each flushing operation. Ideally, a flushing unit should contain only sequential index pages. But, when it is not possible, a flushing unit should be composed of (i) almost sequential index pages, or (ii) distant index pages (e.g., at least 100 index pages of distance). The former leads to a performance as similar as a sequential write, while the latter leads to a better performance compared to random writes [9, 10, 23]. Finally, the flushing operation should also take into account Goal 4.

**Goal 3 - Avoid excessive random reads in frequent locations.** The common assumption that the random read is the fastest operation of SSDs is not always valid because of the read disturbance management. As a result, this operation can take as long as a write or erase. To achieve Goal 3, a flash-aware spatial index should use an in-memory buffer dedicated to the management of reads, called *read buffer*. Thus, instead of performing a random read directly from the SSD to obtain a frequently accessed index page, the index page can be obtained from the read buffer. For instance, in hierarchical indices like the R-tree, nodes (i.e., index pages) located near to the root are frequently accessed in search operations and are reasonable candidates to be cached in the read buffer. Further, the management of the read buffer should include a *read buffer replacement policy*. Examples are the well-known Least Recently Used (LRU) replacement algorithm [27], the two versions of the 2Q replacement algorithm [28], and the Adaptive Replacement Cache (ARC) [29].

**Goal 4 - Avoid interleaved reads and writes.** Mixing reads and writes negatively affects SSD performance because of the interference between these operations. More importantly, this interference severely degrades read performance. To achieve Goal 4, a flash-aware spatial index should use read and write buffers together with a *temporal control*, which temporally stores the identifiers of the last read and written index pages. The temporal control

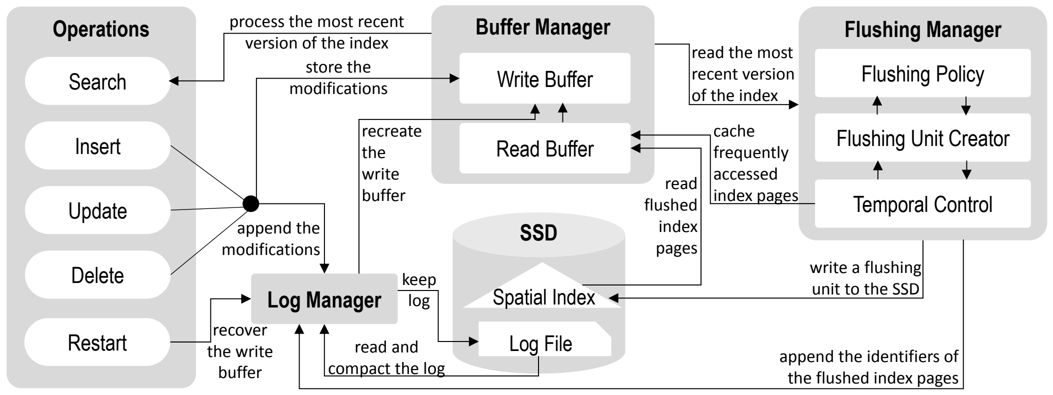


Figure 1: The architecture of eFIND.

of reads should determine whether a flushed index page must be cached to avoid a read after a write. The temporal control of writes should help the flushing operation by discarding index pages that do not lead to the desired organization, such as a sequential write.

**Goal 5 - Provide data durability.** System crashes and power failures impact the consistency of the index since modifications stored in the write buffer are lost. To achieve Goal 5, a flash-aware spatial index should use a *log-structured approach* that sequentially saves modifications in a log file. By using this log file, it is possible to rebuild the write buffer after a system crash. To improve the space utilization of the SSD, the log file should be also compacted when it reaches a specific size.

## 4. The Efficient Framework for Spatial Indexing on SSDs

In this section, we detail how to achieve each design goal conceptually discussed in Section 3. This is done by proposing eFIND, a generic and efficient framework that transforms a disk-based spatial index into a flash-aware spatial index. Figure 1 depicts the eFIND's architecture, which consists of three sophisticated managers to meet the requirements of the design goals.

**Buffer Manager.** It leverages two in-memory buffers to decrease the overhead of random writes and reads. The first one is the *write buffer*, which stores the most recent index modifications generated from *insert*, *update*, and *delete* operations (Goal 1). The second one is the *read buffer*, which caches index pages frequently accessed in *search* operations (Goal 3).

**Flushing Manager.** It contains three interacting components to perform a flushing operation. The first component is the *flushing unit creator*, which builds flushing units by grouping sequential index pages. The second component is the *flushing policy*, which ranks flushing units according to different criteria (Goal 2). The last component is the *temporal control of reads and writes*, which avoids interleaved reads and writes (Goal 4).

**Log Manager.** It is responsible for keeping a log of all modifications stored in the write buffer and of flushing operations; thus, this manager guarantees data durability (Goal 5). Modifications lost after a system crash can be recovered by dispatching the *restart operation.* This manager also compacts the log file to decrease the cost of the space utilization.

Before describing how eFIND works in Sections 4.2 to 4.6, we introduce in Section 4.1 a running example used throughout this article.

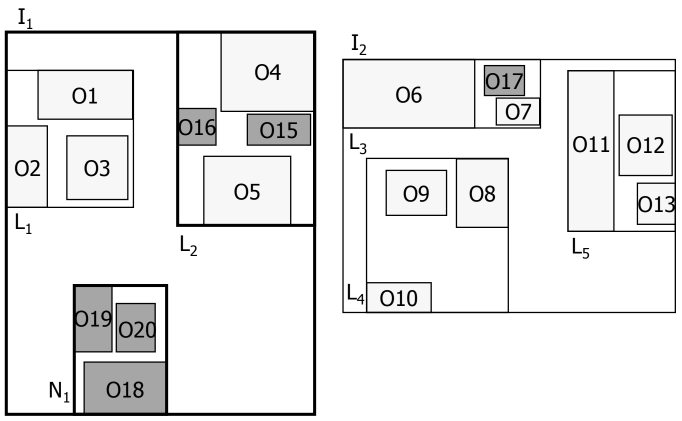
## *4.1. Running Example*

In this running example, we apply eFIND to an R-tree resulting in the *eFIND R-tree* shown in Figure 2. The objects *O1* to *O13* depicted in light gray represent the spatial objects stored in the SSD. The rectangles $L_1$, $L_3$, $L_4$, $L_5$, and $I_2$ represent entries that are also stored in the SSD. The other objects shown in dark gray and rectangles with thick lines are not stored in the SSD. They are derived from the following modifications: (i) insertion of 6 new spatial objects (i.e., *O15* to *O20*), which leads to the creation of the new node $N_1$, and (ii) deletion of the leaf node $L_6$ that contained one spatial object and was stored in the last entry of $I_2$. These modifications should be handled by the eFIND's data structures.

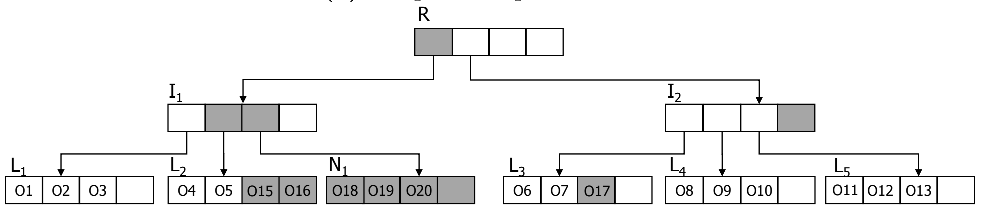
This eFIND R-tree has a height equal to 2 and indexes 19 spatial objects. Each node of the tree represents an *index page* and consists of a fixed number of entries (four, in the current example). Each entry has the format $(p, r)$. If the entry is contained in a leaf node, then $p$ is a unique identifier that provides direct access to the indexed spatial object represented by its *Minimum Bounding Rectangle* (MBR) $r$. Otherwise, $p$ is the index page identifier that supplies the direct access to a child node, and $r$ corresponds to the MBR that covers all MBRs in the child node's entries.

## *4.2. Data Structures*

eFIND leverages specific data structures to deal with the design goals of Section 3. To implement the write buffer (Goal 1), a hash table named



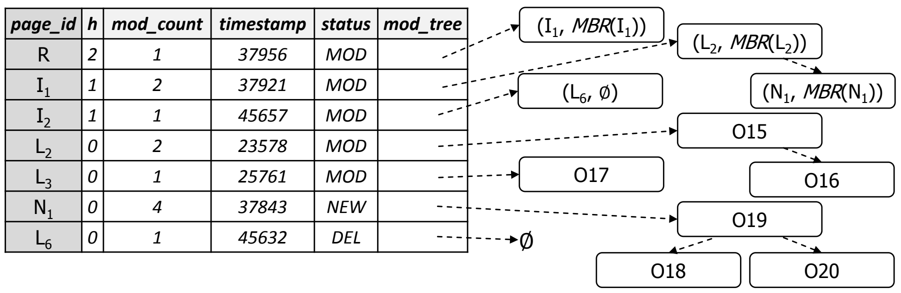
**(a)** Graphical representation



**(b)** Hierarchical representation

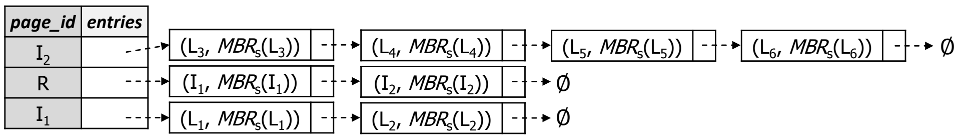
Figure 2: The eFIND R-tree of our running example.

*Write Buffer Table* is employed. It stores the modifications of index pages that were not applied to the SSD yet. We use a hash table as the structure for the write buffer because we can access modified entries of index pages usually in constant time. The key of this hash table is the identifier of an index page (*page_id*) and its value stores modifications in the format ($h$, *mod_count*, *timestamp*, *status*, *mod_tree*). Here, $h$ refers to a specific parameter for hierarchical indices and stores the height of the modified index page; *mod_count* is the quantity of in-memory modifications; *timestamp* informs when the last modification was made; and *status* is the type of modification made and can be NEW, MOD, or DEL for representing newly created index pages in the buffer, index pages stored in the SSD but with modified entries, and deleted index pages, respectively. This information is mainly used in flushing operations (see Section 4.4). For *status* equal to NEW or MOD, *mod_tree* is a red-black tree containing the result of modified entries; otherwise, it is null. An element of a red-black tree has the format ($e$, *mod_result*), where $e$ is the key and corresponds to the unique identifier of an entry of the index page



| page_id | h | mod_count | timestamp | status | mod_tree |
|---|---|---|---|---|---|
| R | 2 | 1 | 37956 | MOD | |
| $I_1$ | 1 | 2 | 37921 | MOD | |
| $I_2$ | 1 | 1 | 45657 | MOD | |
| $L_2$ | 0 | 2 | 23578 | MOD | |
| $L_3$ | 0 | 1 | 25761 | MOD | |
| $N_1$ | 0 | 4 | 37843 | NEW | |
| $L_6$ | 0 | 1 | 45632 | DEL | |

**(a)** *Write Buffer Table*



| page_id | entries |
|---|---|
| $I_2$ | |
| R | |
| $I_1$ | |

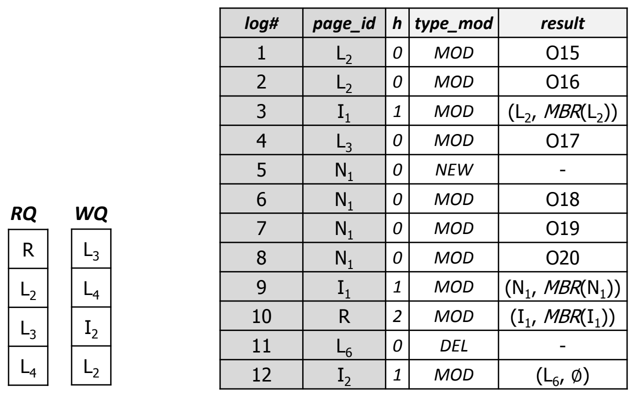
**(b)** *Read Buffer Table*

Figure 3: Graphical representation of the buffers managed by eFIND for applying the modifications of Figure 2.

and *mod_result* stores the most recent version of this entry, assuming null if $e$ was removed. The main benefit of this strategy is that only the latest version of a modified entry is stored in the write buffer. This is achieved by the use of a red-black tree to store the modified entries of an index page instead of using a list with repeated elements. Consequently, the space of the write buffer is better managed with a low cost of retrieving the most recent version of an index page (see Section 4.3). Further, the cost of updating *mod_tree* is amortized by the use of a red-black tree.

Figure 3a shows the *Write Buffer Table* for our running example. In this figure, *MBR* is a function for computing the rectangle that encompasses all entries of a node by considering current modifications in the write buffer. Hence, the same format of an entry of the underlying index is used for each element in *mod_tree*. That is, $(e,\ mod\_result)$ is equivalent to $(p, r)$ (Section 4.1). For instance, the first line of the hash table in Figure 3a shows that $R$, located in the *height* 2, has the *status* MOD to store the entry $(I_1, MBR(I_1))$. Note that this entry now corresponds to the most recent version of the first entry of $R$. This modification occurred in the *timestamp* 37956 and is derived from the adjustment of $I_1$ after the creation of $N_1$.

To implement the read buffer (Goal 3), another hash table named *Read Buffer Table* is employed. It caches the index pages that are already stored



| *log#* | *page_id* | *h* | *type_mod* | *result* |
|---|---|---|---|---|
| 1 | $L_2$ | *0* | *MOD* | O15 |
| 2 | $L_2$ | *0* | *MOD* | O16 |
| 3 | $I_1$ | *1* | *MOD* | ($L_2$, *MBR*($L_2$)) |
| 4 | $L_3$ | *0* | *MOD* | O17 |
| 5 | $N_1$ | *0* | *NEW* | - |
| 6 | $N_1$ | *0* | *MOD* | O18 |
| 7 | $N_1$ | *0* | *MOD* | O19 |
| 8 | $N_1$ | *0* | *MOD* | O20 |
| 9 | $I_1$ | *1* | *MOD* | ($N_1$, *MBR*($N_1$)) |
| 10 | R | *2* | *MOD* | ($I_1$, *MBR*($I_1$)) |
| 11 | $L_6$ | *0* | *DEL* | - |
| 12 | $I_2$ | *1* | *MOD* | ($L_6$, ∅) |

**(a)** Temporal control **(b)** Log file

Figure 4: The queues of the temporal control and the log file after applying the modifications of Figure 2.

in the SSD, prioritizing pages frequently accessed. We employ a hash table because of its usually constant time to access cached index pages. Further, we do not use the same hash table of the write buffer because the read buffer has a different purpose and requires a read buffer replacement policy like LRU and 2Q to decide which index pages should be cached. The key of *Read Buffer Table* is the unique index page identifier (*page_id*) and its value (*entries*) stores a list of entries of the index page. For the R-tree, *entries* stores a set of $(p, r)$ values. Figure 3b depicts that $I_2$, $R$, and $I_1$ are cached in the *Read Buffer Table* by considering only entries stored in the SSD. In this figure, $MBR_S$ is a function for extracting the stored MBR. For instance, the entries of $I_2$ includes $L_6$ even after its deletion, which is indicated in the third line of the hash table in Figure 3a.

To deal with the temporal control (Goal 4), eFIND employs two queues named *RQ* and *WQ*. Each queue is a First-In-First-Out (FIFO) data structure. *RQ* stores identifiers of the index pages read from the SSD, while *WQ* keeps the identifiers of the last index pages written to the SSD. Figure 4a shows that the last read nodes are $R$, $L_2$, $L_3$, and $L_4$, and the last flushed nodes are $L_3$, $L_4$, $I_2$, and $L_2$.

To guarantee data durability (Goal 5), eFIND sequentially writes to a log file the modifications contained in the *Write Buffer Table*. The format (*page_id*, *h*, *type_mod*, *result*) is employed to store each log entry. It has a very similar format to that of the *Write Buffer Table* because we need to

Table 1: Employed notations to measure the cost of the eFIND's algorithms.

| *Notation* | *Description* |
|---|---|
| $\mathcal{R}$ | The average cost of performing a read from the SSD. |
| $\mathcal{W}$ | The average cost of performing a write to the SSD. |
| $\mathcal{H}$ | The average cost of processing the operations put and get on the *Write Buffer Table* and the *Read Buffer Table*. It is *usually* a constant cost. |
| $\mathcal{T}(P)$ | The average cost of executing an operation in *mod_tree* of the corresponding hash entry of the index page $P$ in the *Write Buffer Table*. It corresponds to $\mathcal{O}(\log n)$, where $n$ is the number of modifications stored in the *mod_tree*. |
| $\mathcal{B}$ | The average cost of applying the read buffer replacement policy. |
| $\mathcal{Q}$ | The average cost of manipulating the *RQ* and the *WQ* queues. It is usually a constant cost. |

be able to recover the write buffer after a fatal problem. That is, *page_id* is the identifier of an index page, $h$ is the height of the modified page, and *type_mod* corresponds to the *status*, and *result* is an element of the corresponding *mod_tree*. Hence, different modifications made on the same index page are stored in sequential log entries. Further, index pages written to the SSD are also stored in the log file to allow log compaction, as explained in Section 4.6. In this case, *type_mod* assumes the value FLUSH, *result* is the set of flushed index pages, and the value null is stored for the remaining attributes. Figure 4b shows the log file that follows the chronological order of the modifications stored in the *Write Buffer Table* (Figure 3a). For instance, the first log entry corresponds to the first needed modification to insert *O16*, that is, the accommodation of this object in $L_2$.

The main advantage of the eFIND's data structures is their low-cost integration into existing spatial database systems since they do not require the changing of the underlying spatial index. Thus, to estimate the average time to process the operations of Figure 1, we consider the eFIND's algorithms only. For this, we make use of the notations in Table 1 in our cost analyses. Note that the employed data structures have also a low maintenance cost ($\mathcal{H}$, $\mathcal{T}(P)$, and $\mathcal{Q}$).

**Algorithm 1:** Execution of a maintenance operation by using eFIND

```
   Input: $MO$ as a maintenance operation
1  let MPages be a list of modified index pages resulted from the in-memory
     execution of $MO$;
2  foreach $P_i$ in MPages do
3      let WBEntry be the hash entry of $P_i$ in the Write Buffer Table;
4      if WBEntry is not NULL then
5          if $P_i$ is a deleted index page then
6              free all modifications contained in the WBEntry;
7              set the status of WBEntry to DEL;
8          else
9              store the changes of $P_i$ in the mod_tree of WBEntry;
10     else
11         set WBEntry to be a new hash entry in the Write Buffer Table
             with key $P_{i_{id}}$;
12         if $P_i$ is a newly created node then
13             store the entries of $P_i$ in the mod_tree of WBEntry;
14             set the status of WBEntry to NEW;
15         else if $P_i$ is a deleted index page then
16             set the status of WBEntry to DEL;
17         else
18             store the changes of $P_i$ in the mod_tree of WBEntry;
19             set the status of WBEntry to MOD;
20     set the mod_count and timestamp of WBEntry accordingly;
21     call Log Manager to append the changes of $P_i$ to the log file;
22 if Write Buffer Table is full then
23     call Flushing Manager to execute a flushing operation (Algorithm 3);
```

## *4.3. Maintenance Operations*

**Algorithm Descriptions.** Algorithm 1 shows how eFIND executes *insert*, *update*, and *delete* operations. Its input is a maintenance operation, which is responsible for reorganizing the index whenever modifications are made on the underlying spatial dataset. The first step of the algorithm executes this maintenance operation as an *in-memory operation* (line 1). It returns a list of modified index pages, which can include the creation of new index pages, the adjustment of entries, and the deletion of index pages. After that, the modifications of these index pages are stored in the *Write Buffer Table* (lines 2 to 21). If a modified index page has an entry in this hash table, the corresponding entry is modified accordingly (lines 4 to 9). Otherwise, Algorithm 1 creates a new hash entry (line 11), which can be either a newly created index page (lines 13 and 14), a deleted index page (line 16), or an index page with in-memory modifications (lines 18 and 19). Next, the number of modifications and the time stamp of the modified index page are set accordingly (line 20).

To guarantee data durability, Algorithm 1 calls the *Log Manager* to keep the log of all modifications stored in the *Write Buffer Table* (line 21). This is a low latency operation since it involves only sequential writes. The final step of Algorithm 1 is the execution of a flushing operation (Section 4.4) whenever the write buffer is full (line 23).

The execution of an in-memory maintenance operation (line 1 in Algorithm 1) involves retrieving index pages. For instance, to choose an index page to accommodate or delete a spatial object. Because parts of the underlying spatial index can be stored in three different locations, the SSD, the *Read Buffer Table*, and the *Write Buffer Table*, we specify Algorithm 2 to retrieve index pages by considering all these locations.

Algorithm 2 has the identifier $P_{id}$ of an index page as input. If this page is a newly created index page or a deleted index page, the algorithm returns the index page pointed by its corresponding entry in the *Write Buffer Table* (line 3). Otherwise, it gets the last stored version of the index page that is either buffered in the *Read Buffer Table* (lines 6 and 7) or stored in the SSD (lines 9 to 11). In the former case, we avoid a read operation. In the latter case, the index page is read from the SSD and stored in the *Read Buffer Table* (lines 9 and 10). Further, the read operation is added to the FIFO queue *RQ* (line 11) designated for handling the temporal control of reads. Afterwards, the read buffer replacement policy is applied (line 12). Finally, Algorithm 2 returns either the index page read from the SSD or the index page cached in the *Read Buffer Table*, if it does not have modifications in the *Write Buffer Table* (line 15). If $P_{id}$ has modifications (line 13), a merge operation is executed to return the most recent version of the index page (line 14). This merge operation replaces the old version of modified entries by the entries stored in the *Write Buffer Table*, maintains the entries that were not modified, and adds newly created entries if any.

**Example of Execution.** To illustrate the execution of a maintenance oper-

**Algorithm 2:** Retrieving an index page by using eFIND

```
Input: $P_{id}$ as the identifier of an index page to be returned
Output: The current version of the index page identified by $P_{id}$
1  let WBEntry be the hash entry in the Write Buffer Table with key $P_{id}$;
2  if WBEntry has status equal to NEW or DEL then
3      return the index page pointed by WBEntry;
4  let $P$ be an empty index page;
5  let RBEntry be the hash entry in the Read Buffer Table with key $P_{id}$;
6  if RBEntry is not NULL then
7      let $P$ become the index page pointed by RBEntry;
8  else
9      let $P$ become the index page $P$ read from the SSD;
10     insert $P$ into the Read Buffer Table;
11     insert the identifier of $P$ in the RQ;
12 apply the read buffer replacement policy;
13 if WBEntry has status equal to MOD then
14     return the result of a merge operation between the entries contained in
         the mod_tree of WBEntry and the entries of $P$;
15 return $P$;
```

ation, let us consider the second modification on the eFIND R-tree (Figure 2), that is, the insertion of *O16*. First, a leaf node should be chosen to accommodate this object. Since the MBR of the internal node $I_1$ intersects *O16*, Algorithm 2 gets the cached version of $I_1$ from the *Read Buffer Table* (last line of Figure 3b) without requiring reads from the SSD. Then, the leaf node $L_2$ is chosen to accommodate *O16*. This node is read from the SSD since it is not cached in the *Read Buffer Table*. This read operation is registered in the *RQ* (second line of Figure 4a).

Accommodating *O16* in $L_2$ results in two modified pages to be traversed. The first modification is stored in the *Write Buffer Table* (fourth line of Figure 3a). It indicates that $L_2$, with height 0, has 2 modifications. The last of them is the insertion of *O16* at the time 23578. At the same time, this modification is also stored in the log file (second line of Figure 4b). The second modification corresponds to the adjustment of $I_1$. It is stored in its *mod_tree* as ($L_2$, $MBR(L_2)$) (second line of Figure 3a). Finally, it is appended to the log file (third line of Figure 4b).

**Cost Analysis.** Before discussing the cost of Algorithm 1, we first analyze $\mathcal{C}_{alg2}$ as the cost of retrieving the index page $P$ (Algorithm 2). Three possible cases compose $\mathcal{C}_{alg2}$. First, if the hash entry in the *Write Buffer Table* (i.e., *WBEntry*) has the *status* NEW or DEL, the cost $\mathcal{C}_{alg2}$ is minimized by accessing only one hash entry. The second case is if $P$ is cached in the *Read Buffer Table* (i.e., *RBEntry* is not null). It requires one access in each hash table, and the execution of the read buffer replacement policy (totaling $2 * \mathcal{H} + \mathcal{B}$). This case further requires the cost $\mathcal{M}$ of executing a merge operation, which is given by $\mathcal{M} = 0$ if *WBEntry* is null and $\mathcal{M} = \mathcal{O}(n + m)$ otherwise; where $n$ is the number of entries in *WBEntry* and $m$ is the number of entries in *RBEntry*. The last case occurs if $P$ has to be read from the SSD, which adds the costs of the read operation and the management of *RQ* (i.e., $\mathcal{R} + \mathcal{Q}$) to the cost of the second case. We formally define $\mathcal{C}_{alg2}$ as follows:

$$\mathcal{C}_{alg2} = \begin{cases} \mathcal{H} & \text{if } \textit{status} \text{ of } \textit{WBEntry} \text{ is NEW or DEL} \\ 2 * \mathcal{H} + \mathcal{B} + \mathcal{M} & \text{if } \textit{RBEntry} \text{ is not null} \\ 2 * \mathcal{H} + \mathcal{B} + \mathcal{M} + \mathcal{R} + \mathcal{Q} & \text{otherwise} \end{cases} \tag{1}$$

To calculate the cost of processing the modified index pages, we need the auxiliary cost function $\mathcal{C}_{amc}$. Its input is an index page $P$ that has modifications to be stored in the *Write Buffer Table*. $\mathcal{C}_{amc}$ returns the constant cost $\mathcal{F}$ of freeing the *mod_tree* of $P$, if it is a deleted index page. Otherwise, it returns the cost of updating all $m$ modifications of $P$. Both cases also include the cost of a write operation because of the log management. We formally define $\mathcal{C}_{amc}$ as follows:

$$\mathcal{C}_{amc}(P) = \begin{cases} \mathcal{F} + \mathcal{H} + \mathcal{W} & \text{if } P \text{ is a deleted index page} \\ \mathcal{H} + \sum_{i=1}^{m} (\mathcal{T}(P) + \mathcal{W}) & \text{otherwise} \end{cases} \tag{2}$$

Once defined $\mathcal{C}_{alg2}$ and $\mathcal{C}_{amc}$ (Equations 1 and 2), the cost of Algorithm 1, $\mathcal{C}_{alg1}$, is given by the sum of the cost of retrieving $r$ index pages needed for computing the in-memory maintenance operation and the cost of processing the modifications of $n$ modified index pages. $\mathcal{C}_{alg1}$ is formally defined as follows:

$$\mathcal{C}_{alg1} = \sum_{i=1}^{r} (\mathcal{C}_{alg2}) + \sum_{i=1}^{n} (\mathcal{C}_{amc}(P_i)) \tag{3}$$

**Algorithm 3:** Execution of the eFIND's flushing operation

```
1  let HEntries be a list containing $p$ of the oldest hash entries of Write Buffer
     Table;
2  filter HEntries informing their index page identifiers to the temporal
     control of writes (Algorithm 4);
3  sort in ascending order the list HEntries by the index page identifiers;
4  let FU be a list of flushing units created from the list HEntries;
5  foreach $FU_i$ in FU do
6      compute the value $d$ as $FP(FU_i)$;
7  let chosenFU be the flushing unit of FU with the greatest value $d$;
8  let pagesToFlush be an empty list of index pages;
9  foreach $CFU_i$ in chosenFU do
10     retrieve the index page $P$ with the index page identifier of $CFU_i$
         (Algorithm 2);
11     append $P$ to pagesToFlush;
12     invoke the temporal control of reads with $P$ as input (Algorithm 5);
13 write in batch the index pages contained in pagesToFlush to the SSD;
14 call Log Manager to append the identifiers contained in pagesToFlush to
     the log file;
15 add the identifiers contained in pagesToFlush to the WQ;
16 free the corresponding hash entries of pagesToFlush from the Write Buffer
     Table;
```

## *4.4. Flushing Operations*

**Algorithm Descriptions.** Algorithm 3 details the flushing operation (executed at line 23 in Algorithm 1). Instead of writing all the stored modifications to the SSD, eFIND smartly selects only some of them to be written to the SSD. To this end, the algorithm first creates the list *HEntries*, which contains the oldest hash entries of the *Write Buffer Table* by considering the *timestamp* attribute (line 1). The percentage value $p$ determines the size of *HEntries*. Our experiments showed best performance results if $p$ is equal to 60% (see Section 6.1).

Afterwards, the temporal control of writes discards hash entries of *HEntries* by using Algorithm 4 (line 2). Then, flushing units are created. Assuming a *flushing unit size* equal to $s$, each group of $s$ index pages of *HEntries*, previously sorted by the index page identifiers (line 3), define a flushing unit (line 4). Hence, flushing units are formed by sequential index pages.

Next, Algorithm 3 picks a flushing unit to be written according to a flushing policy. This is performed by using a function $FP$ that calculates, for each flushing unit, a degree $d$ (lines 5 and 6) that is then used to select the flushing unit with the greatest degree (line 7). In case of ties, any flushing unit might be picked. Different implementations of $FP$ are conceivable and can be based on distinct criteria. Here, we present the following set of flushing policies $\mathcal{P} = \{FP_m, FP_{mh},\ FP_{mha}, FP_{mhao}\}$. The input of each $FP \in \mathcal{P}$ is an array $FU$ of $n$ hash entries of the *Write Buffer Table*. The flushing policy $FP_m$ (Equation 4) is based on the number of modifications, that is, it calculates the total number of modifications of all index pages of $FU$. The remaining policies should be used by hierarchical indices, such as the eFIND R-tree. The flushing policy $FP_{mh}$ (Equation 5) is based on the number of modifications and height of nodes, that is, it uses the height of the modified node as weight on its number of modifications. The flushing policies $FP_{mha}$ and $FP_{mhao}$ (Equations 6 and 7) are extensions of $FP_{mh}$ and are based on the modified area and the overlapping modified area, respectively. They apply $\alpha$ as an additional weight that corresponds to the ratio of the modified area by the flushing unit and the total modified area stored in the write buffer. $FP_{mhao}$ also uses the $\beta$ weight, which is the ratio of the overlapping modified area the flushing unit and the total overlapping modified area stored in the write buffer. The idea behind the weights $\alpha$ and $\beta$ is the creation of flushing policies by using criteria based on geometric characteristics of the modifications. Each $FP \in \mathcal{P}$ is formally defined as follows:

$$FP_m(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count \tag{4}$$

$$FP_{mh}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) \tag{5}$$

$$FP_{mha}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) * \alpha \tag{6}$$

$$FP_{mhao}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) * \alpha * \beta \tag{7}$$

Our experiments showed best performance results for $FP_{mh}$ (see Section 6.1). The main reason is that this flushing policy gives higher degrees for index pages located in the highest levels of the index, which are not so frequently modified. In the sequence, Algorithm 3 retrieves the index pages that compose the chosen flushing unit (lines 9 to 12). Here, Algorithm 5 is called to execute the temporal control of reads (line 12). Then, Algorithm 3 writes in batch the index pages of the chosen flushing unit (line 13). Finally, the algorithm keeps log of the flushed index pages, adds the made writes to the queue *WQ* for handling the temporal control of writes, and frees the index pages from the *Write Buffer Table* (lines 14 to 16).

The temporal control of writes and reads plays an important role in the execution of Algorithm 3 (lines 2 and 12). Algorithm 4 shows the execution of the temporal control of writes. Its input is a list of index page identifiers. Its goal is to return index page identifiers that either (i) provide a sequential or almost sequential write pattern, (ii) or provide a write pattern composed of distant pages. That is, Algorithm 4 filters index pages to execute fast writes by considering previous writes. This algorithm divides its input into *Seq* and *Str* (lines 2 to 6). Sequential or almost sequential index pages to the pages stored in the $QW$ are appended to *Seq* (line 4), while pages very distant from the pages stored in the $QW$ are appended to *Str* (line 6). The concept of distance refers to the locality of index pages in the SSD. Our experiments showed better results when considering that two index pages are (almost) sequential if the distance between them is lesser than or equal to 10 and that two index pages are very distant if their distance is greater than 100. Algorithm 4 returns either *Seq* or *Str* if one of them is enough to create at least one flushing unit (lines 7 to 10). The priority is to return *Seq* since it potentially leads to a sequential write in the flushing operation. If *Seq* and *Str* do not contain enough pages to create one flushing unit, the algorithm returns either the union between them if it creates at least one flushing unit, or a copy of its input (line 12).

Algorithm 5 details the execution of the temporal control of reads. It has as input an index page and deals with two alternately cases. The first one relates to updating the content of the index page, if it is already cached in the *Read Buffer Table* (line 2). This is needed because the cached version of the index page was modified during the flushing operation (line 10 in Algorithm 3). In the second case, the index page is stored in the *Read Buffer Table*, if its identifier is contained in the $RQ$ (line 5). The advantage of this pre-caching is that it avoids a possible read after a write since the index page

**Algorithm 4:** Execution of the temporal control of writes

```
Input: IPages as a list of index page identifiers
Output: A list of index page identifiers
1  let Seq and Str be two empty lists of index page identifiers;
2  foreach $IP_i$ in IPages do
3      if $IP_i$ is a neighbor or almost neighbor of index page identifiers in WQ
        then
4          add $IP_i$ to Seq;
5      else if $IP_i$ is very distant from index page identifiers in WQ then
6          add $IP_i$ to Str;
7  if the size of Seq is greater than or equal to the flushing unit size then
8      return Seq;
9  else if the size of Str is greater than or equal to the flushing unit size then
10     return Str;
11 else
12     return either the combination between Seq and Str, or IPages;
```

**Algorithm 5:** Execution of the temporal control of reads

```
Input: P as an index page
1  if P is cached in the Read Buffer Table then
2      update the content of P;
3      apply the read buffer replacement policy;
4  else if the RQ contains the identifier of P then
5      insert P in the Read Buffer Table;
6      apply the read buffer replacement policy;
```

is frequently requested by retrieving operations. The read buffer replacement policy is applied as needed in both cases (lines 3 and 6).

**Example of Execution.** Consider that the *Write Buffer Table* of Figure 3a is full, requiring a flushing operation. Let us assume that *HEntries* contains 60% of the oldest hash entries of this hash table, that is, $L_2$, $L_3$, $N_1$, and $I_1$. Considering that all these index pages are almost sequential to the pages stored in the *WQ* (Figure 4a), the temporal control of writes (Algorithm 4) returns the same list of index pages. Afterwards, the pages are sorted according to its identifiers. Here, we sort them by considering their appearance in Figure 2b, from the highest level to the lowest level of the tree. The resulting list is then $I_1$, $L_2$, $N_1$, and $L_3$.

Assuming the flushing unit size equal to 2, the flushing units $FU_1 = \{I_1, L_2\}$ and $FU_2 = \{N_1, L_3\}$ are created. By applying the flushing policy $FP_{mh}$ (Equation 5), the flushing unit $FU_1$ is chosen because its degree (i.e., 6) is higher than the degree of $FU_2$ (i.e., 5). Next, the nodes $I_1$ and $L_2$ are retrieved by using Algorithm 2. Only one read is required to retrieve $L_2$ since $I_1$ is cached in the *Read Buffer Table* (Figure 3b). In the sequence, the temporal control of reads (Algorithm 5) updates the content of $I_1$ and applies the read buffer replacement policy. $L_2$ is cached in the *Read Buffer Table* because it is contained in the $RQ$ (Figure 4a). Finally, the most recent versions of $I_1$ and $L_2$ are written to the SSD, and this flushing operation is appended as a new log entry 12 in the log file of Figure 4b.

**Cost Analysis.** The cost of Algorithm 3 requires the cost analysis of Algorithms 4 and 5. The cost of Algorithm 4, $\mathcal{C}_{alg4}$, includes the traversal of a list containing $n$ index page identifiers. For each identifier, the algorithm checks whether it provides a (almost) sequential or stride write, considering the $m$ index page identifiers of $WQ$. We formally define $\mathcal{C}_{alg4}$ as follows:

$$\mathcal{C}_{alg4} = \mathcal{O}(n * m) \tag{8}$$

As for the cost of Algorithm 5, $\mathcal{C}_{alg5}$, two alternately cases are possible. The first case relates to the cost $\mathcal{U}$ of updating the content of $P$ in the *Read Buffer Table*. For the second case, $\mathcal{C}_{alg5}$ requires the cost $\mathcal{H}$ of inserting a new entry in the *Read Buffer Table*. Both cases add the cost $\mathcal{B}$ of executing the read buffer replacement policy. We formally define $\mathcal{C}_{alg5}$ as follows:

$$\mathcal{C}_{alg5} = \begin{cases} \mathcal{U} + \mathcal{B} & \text{if } P \text{ is cached in the } \textit{Read Buffer Table} \\ \mathcal{H} + \mathcal{B} & \text{if } RQ \text{ contains the identifier of } P \end{cases} \tag{9}$$

Now we can estimate the average cost of Algorithm 3, $\mathcal{C}_{alg3}$. First, it requires the linear cost $\mathcal{A}$ needed to collect the $p$ oldest hash entries in the *Write Buffer Table*. These entries are filtered by the temporal control of writes with the cost $\mathcal{C}_{alg4}$ (Equation 8). Then, a sorting operation of cost $\mathcal{S}$ is made, which typically corresponds to $\mathcal{O}(n \log n)$ for $n$ as the number of index page identifiers being sorted. Next, a flushing unit is chosen by requiring the linear cost $\mathcal{T}$. To write the flushing unit, its $f$ index pages are retrieved and processed by the temporal control of reads, requiring the cost $\mathcal{C}_{alg2} + \mathcal{C}_{alg5}$ (Equations 1 and 9). Writing the chosen flushing unit leads the cost $\mathcal{W} + \mathcal{W}_{batch}$, which corresponds to the maintenance of the log file plus the batch operation. Finally, let $\mathcal{Q}+\mathcal{D}$ be the cost of adding a flushed index page in $QW$ and of deleting its hash entry from the *Write Buffer Table.* $\mathcal{C}_{alg3}$ is the sum of all previous costs as follows:

$$\mathcal{C}_{alg3} = \mathcal{A} + \mathcal{C}_{alg4} + \mathcal{S} + \mathcal{T} + \mathcal{W} + \mathcal{W}_{batch} + \sum_{i=1}^{f} (\mathcal{C}_{alg2} + \mathcal{C}_{alg5} + \mathcal{Q} + \mathcal{D}) \tag{10}$$

## *4.5. Search Operations*

**Algorithm Description.** eFIND does not change the search algorithm of the underlying index. But, the search algorithm should use Algorithm 2 to retrieve index pages. To serve as an illustration, Algorithm 6 shows how an eFIND R-tree should execute a search operation. Its inputs are a search object *SO*, a topological predicate *TP* (e.g., *intersects*, *inside*), and a node (index page) identifier. Starting from the root node of an eFIND R-tree, the algorithm first retrieves its index page by using Algorithm 2 (lines 1 and 2). If it is an internal node, Algorithm 6 is called recursively for all child nodes that satisfy the topological predicate *TP* by considering the search object *SO* (lines 3 to 6). When the node is a leaf, the search operation returns the most recent version of the entries that answer the spatial query (line 8).

**Example of Execution.** To exemplify a search operation, let us consider the execution of a point query [1]. Let further consider that the search object *SO* is a point contained in *O17*, that is, *SO* $\in$ *O17*. The query starts by traversing the root node $R$. Hence, the most recent version of $R$ is retrieved by Algorithm 2. Among the entries of $R$, $I_2$ intersects the search object. Then, Algorithm 6 is called recursively for such node. $I_2$ is retrieved from the *Read Buffer Table* (first line of Figure 3b) and is merged to its modification contained in the *Write Buffer Table* (third line of Figure 3a). The last call of Algorithm 6 is for the leaf node $L_3$, which is read from the SSD and has its modification appropriately applied (fifth line of Figure 3a). In this calling, the algorithm returns *O17*. Note that the use of the *Read Buffer Table* avoids reads from the SSD. Without the use of the read buffer, 3 reads should be made to answer the point query; instead, eFIND did perform 1 read only.

**Cost Analysis.** A number of executions of Algorithm 2 are made until a given spatial query is completely answered. This number depends on the

**Algorithm 6:** Execution of the search algorithm for the eFIND R-tree, starting from its root node

```
Input: $SO$ as a search object, $TP$ as a topological predicate, and $P_{id}$ as a
       node identifier of the eFIND R-tree
Output: A set of entries containing the candidates that answer the query
1 let $LE$ be a list of entries;
2 let $P$ be the index page yielded by the retrieving algorithm with $P_{id}$ as
    input (Algorithm 2);
3 if $P$ is an internal node then
4     foreach entry $En_i$ in $P$ do
5         if $En_i$ satisfies the topological predicate $TP$, considering the search
            object $SO$ then
6             invoke the search algorithm with $SO$, $TP$, and the node
                identifier pointed by $En_i$ as inputs;
7 else
8     add all entries whose satisfy the topological predicate $TP$ for the search
        object $SO$ to $LE$;
9 return $LE$;
```

spatial organization of the underlying index. Since eFIND does not change this spatial organization, we assume $n$ as the number of accessed index pages needed to answer a spatial query. We formally define the cost of Algorithm 2 as follows:

$$\mathcal{C}_{alg6} = \sum_{i=1}^{n} \mathcal{C}_{alg2} \tag{11}$$

*4.6. Restart Operations*

**Algorithm Description.** Algorithm 7 shows how eFIND recovers all modifications that were not effectively applied to the index, after a system crash, fatal error, or failure power. It has the log file of eFIND as input and yields a new possibly compacted log file. The first step of the restart operation is to read the log file in reverse order (lines 4 to 9), that is, from the most recent modifications to the oldest ones. During this step, all log entries of flushing operations are added to a list (line 6), while log entries not contained in this

**Algorithm 7:** Execution of a restart operation, rebuilding the *Write Buffer Table* and compacting the log

```
   Input: LFile as the log file of eFIND
   Output: A new possibly compacted log file
1  let LEntries be an empty list of log entries;
2  let Stack be an empty stack of log entries;
3  let LE be the last log entry of LFile;
4  while LE is not NULL do
5      if the type_mod of LE is FLUSH then
6          add LE to LEntries;
7      else if LE is not contained in LEntries then
8          push LE into Stack;
9      let LE become the next log entry of LFile or NULL if there is no more
         entries, respecting the reverse order;
10 create a new log file LFile';
11 while Stack is not empty do
12     pop the log entry SE;
13     insert SE into the Write Buffer Table while keeping log in LFile';
14 erase LFile;
15 return LFile';
```

list are pushed in a stack of log entries (line 8) because they were not applied yet to the SSD. By using this stack, eFIND creates a new log file and appends all modifications contained in the stack to the *Write Buffer Table* (lines 11 to 13). Finally, the new log file should be used in future maintenance operations (lines 14 and 15).

Another benefit of Algorithm 7 is that it may compact the log file. Compacting the log is a needed operation to improve the space utilization of eFIND. Instead of inserting log entries of the stack into the *Write Buffer Table* (line 13), the compaction algorithm should only append these log entries to the new log file.

**Example of Execution.** Consider that a flushing operation was performed (Section 4.4) before a power failure. As a result, the modifications of the *Write Buffer Table* (Figure 3a) are lost and the current log file, shown in Figure 5a, contains now 13 log entries. The last log entry has *type_mod* equal to FLUSH, *result* equal to $I_1, L_2$, and null for the remaining attributes.

| log# | page_id | h | type_mod | result |
|---|---|---|---|---|
| 1 | $L_2$ | 0 | MOD | O15 |
| 2 | $L_2$ | 0 | MOD | O16 |
| 3 | $I_1$ | 1 | MOD | ($L_2$, *MBR*($L_2$)) |
| 4 | $L_3$ | 0 | MOD | O17 |
| 5 | $N_1$ | 0 | NEW | - |
| 6 | $N_1$ | 0 | MOD | O18 |
| 7 | $N_1$ | 0 | MOD | O19 |
| 8 | $N_1$ | 0 | MOD | O20 |
| 9 | $I_1$ | 1 | MOD | ($N_1$, *MBR*($N_1$)) |
| 10 | R | 2 | MOD | ($I_1$, *MBR*($I_1$)) |
| 11 | $L_6$ | 0 | DEL | - |
| 12 | $I_2$ | 1 | MOD | ($L_6$, ∅) |
| 13 | - | - | FLUSH | $I_1$, $L_2$ |

**(a)** Log file after the flushing operation

| log# | page_id | h | type_mod | result |
|---|---|---|---|---|
| 1 | $L_3$ | 0 | MOD | O17 |
| 2 | $N_1$ | 0 | NEW | - |
| 3 | $N_1$ | 0 | MOD | O18 |
| 4 | $N_1$ | 0 | MOD | O19 |
| 5 | $N_1$ | 0 | MOD | O20 |
| 6 | R | 2 | MOD | ($I_1$, *MBR*($I_1$)) |
| 7 | $L_6$ | 0 | DEL | - |
| 8 | $I_2$ | 1 | MOD | ($L_6$, ∅) |

**(b)** Compacted log file

Figure 5: Illustrating how the restart operation may also compact the log file.

Since the log file is read in reverse order, the first log entry to be read refers to the flushed nodes $I_1$ and $L_2$. These nodes are appended to *LEntries*. Then, the log entries 12, 11, and 10 are pushed into the stack. The log entry 9 contains one modification for nodes of *LEntries* and is ignored. The log entries from 8 to 4 are pushed into the stack, while the log entries 3, 2, and 1 are ignored. Finally, the log entries in the stack are written to a new log file and their modifications are inserted into the *Write Buffer Table*. As a result, the algorithm rebuilds the same version of the *Write Buffer Table* before the system crash. Further, the log file is compacted by containing 8 log entries instead of 13 log entries, as shown in Figure 5b.

**Cost Analysis.** The cost of Algorithm 7, $\mathcal{C}_{alg7}$, involves the cost of two loops. The first loop refers to the traversal of the log file in reverse order. For each log entry, this requires the sum of the cost of reading the corresponding log entry (i.e., $\mathcal{R}$), and the cost of inserting the log entry either in a list or in a stack (i.e., $\mathcal{K}$). The second loop refers to the traversal of a stack to rebuilt the *Write Buffer Table*. The cost of this loop is defined by the cost function $\mathcal{C}_{aml}$ (Equation 12). Its input is a log entry $L$. $\mathcal{C}_{aml}$ has the fixed cost $\mathcal{W}$ of writing a log entry in the new log file. If the *type_mod* of $L$ is DEL, the cost $\mathcal{F}$ of freeing modifications of $L$ in the *Write Buffer Table* is added to $\mathcal{C}_{aml}$. Otherwise, $\mathcal{C}_{aml}$ adds the cost $\mathcal{T}(L)$ of inserting the modification of $L$ in the *Write Buffer Table*. In addition to the cost of these two loops, $\mathcal{C}_{alg7}$ also considers the cost $\mathcal{E}$ of erasing the old version of the log file. Let $n$ be the number of log entries stored in the log file, and $m$ be the number of stacked log entries. We formally define $\mathcal{C}_{aml}$ and $\mathcal{C}_{alg7}$ respectively as follows:

$$\mathcal{C}_{aml}(L) = \begin{cases} \mathcal{F} + \mathcal{W} & \text{if } L \text{ has the } \textit{type\_mod} \text{ DEL} \\ \mathcal{T}(L) + \mathcal{W} & \text{otherwise} \end{cases} \tag{12}$$

$$\mathcal{C}_{alg7} = \sum_{i=1}^{n} (\mathcal{K} + \mathcal{R}) + \sum_{j=1}^{m} (\mathcal{C}_{aml}(L_j)) + \mathcal{E} \tag{13}$$

## 5. Experimental Evaluation

In this section, we measure the performance gains of eFIND against the state of the art. Section 5.1 describes the experimental setup employed in the experiments. Section 5.2 discusses results related to the construction of spatial indices, while Section 5.3 reports results for executing spatial queries.

### *5.1. Experimental Setup*

**Dataset.** We used a real spatial dataset from the OpenStreetMap[1], which consisted of 1,485,866 complex regions possibly with holes representing the buildings of Brazil like hospitals, universities, schools, houses, and stadiums. Geometric and statistical descriptions of this dataset can be found in [30] through the name *brazil_buildings2017_v2*.

**Configurations.** We compared two configurations: (i) the *FAST R-tree*, and (ii) the *eFIND R-tree*. These configurations applied the quadratic split algorithm of the R-tree, as well as had a buffer of 512KB and log capacity of 10MB. We used a fixed buffer size because our goal is to analyze the performance behavior of these configurations, which is usually similar for different sizes of the buffer [6, 7, 14, 15]. For the FAST R-tree, we used the FAST* flushing policy, which provided the best results according to [14]. For the eFIND R-tree, we employed the best parameter values based on the analysis conducted in Section 6: $FP_{mh}$ as the flushing policy, the value of 60% for $p$, the allocation of 20% of the buffer for the read buffer, and the

[1]`http://www.openstreetmap.org/`

simplified 2Q as the read buffer replacement policy. We considered FAST as the state of the art because it provides the best characteristics among the existing flash-aware spatial indices, as detailed in Section 7. We did not compare the native (plain) R-tree [19] because it requires a prohibitive number of writes, reads, and erases, not being suitable for SSDs, as shown in [15, 31].

**Varied Parameters.** We employed index page sizes (i.e., node sizes) from 2KB to 32KB, and flushing unit sizes from 1 to 5. To simplify the visualization of the graphics, we only report results for the flushing unit size equal to 5 because it provided the best results for both configurations.

**Workloads.** We executed two workloads: (i) index construction, and (ii) execution of 300 intersection range queries (IRQ) [1]. Three different sets of query windows were used. These sets were respectively composed of 100 query windows with 0.001%, 0.01%, and 0.1% of the area of the total extent of Brazil. Considering that the selectivity of a spatial query is the ratio of the number of returned objects and the total objects, these sets of query windows form spatial queries with low, medium, and high selectivity, respectively. For each configuration, we executed the workloads as a sequence, that is, the index construction followed by the processing of IRQs. Each sequence was executed 5 times. We flushed the system cache after the execution of each sequence and calculated the elapsed time as follows. 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 FESTIval [32], an open-source PostgreSQL extension that benchmarks spatial indices. The source code and the documentation of FESTIval (and eFIND) are publicly available at `https://github.com/accarniel/festival`. We performed the tests locally to avoid network latency. The experiments were conducted on a local server equipped with an Intel Core® i7-4770 with a frequency of 3.40GHz, 32GB of main memory, and two SSDs: (i) one Kingston V300 of 480GB, and (ii) one Intel Series 535 of 240GB. The Intel SSD is a high-end SSD that provides faster reads and writes than the low-end Kingston SSD. This allowed us to measure the performance of eFIND by considering different architectures of SSDs. The software we used was Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, and PostGIS 2.2.

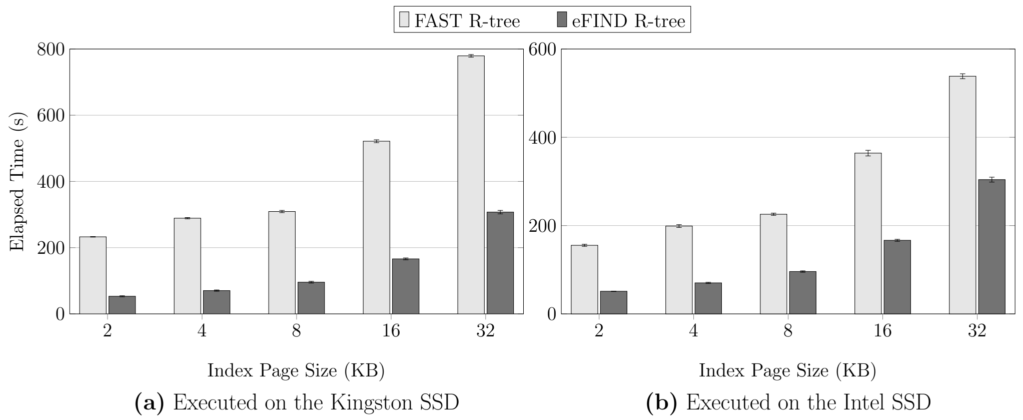


Figure 6: The eFIND R-tree showed expressive performance gains when building spatial indices on both SSDs.

## *5.2. Index Construction*

As shown in Figure 6, the eFIND R-tree overcame the FAST R-tree on both SSDs and for all employed page sizes. Its performance gains were very expressive, ranging from 60% to 77% for the Kingston SSD (Figure 6a), and ranging from 43% to 67% for the Intel SSD (Figure 6b).

The eFIND R-tree exploited the benefits of the SSDs because it is based on the design goals defined in Section 3. The contribution of the read buffer to obtain these results was significantly relevant even using a relatively small percentage of the buffer size, as discussed in Section 6.2. Another important contribution was the use of the temporal control, which guaranteed that frequently accessed index pages were stored beforehand in the read buffer. Further, eFIND improved the space utilization of the write buffer. Instead of using a list that allows repeated elements, eFIND leverages efficient data structures to manage index modifications. This led to the faster retrieval of index pages, reflecting in the elapsed time when building spatial indices in the SSDs.

Both configurations presented their best performance for the index page size of 2KB. This is due to the high cost of writing flushing units with larger index pages (e.g., 32KB) since a write made on the application layer can be split into several internal writes in the SSD. Hence, the construction of the indices required more time as the page size also increased.

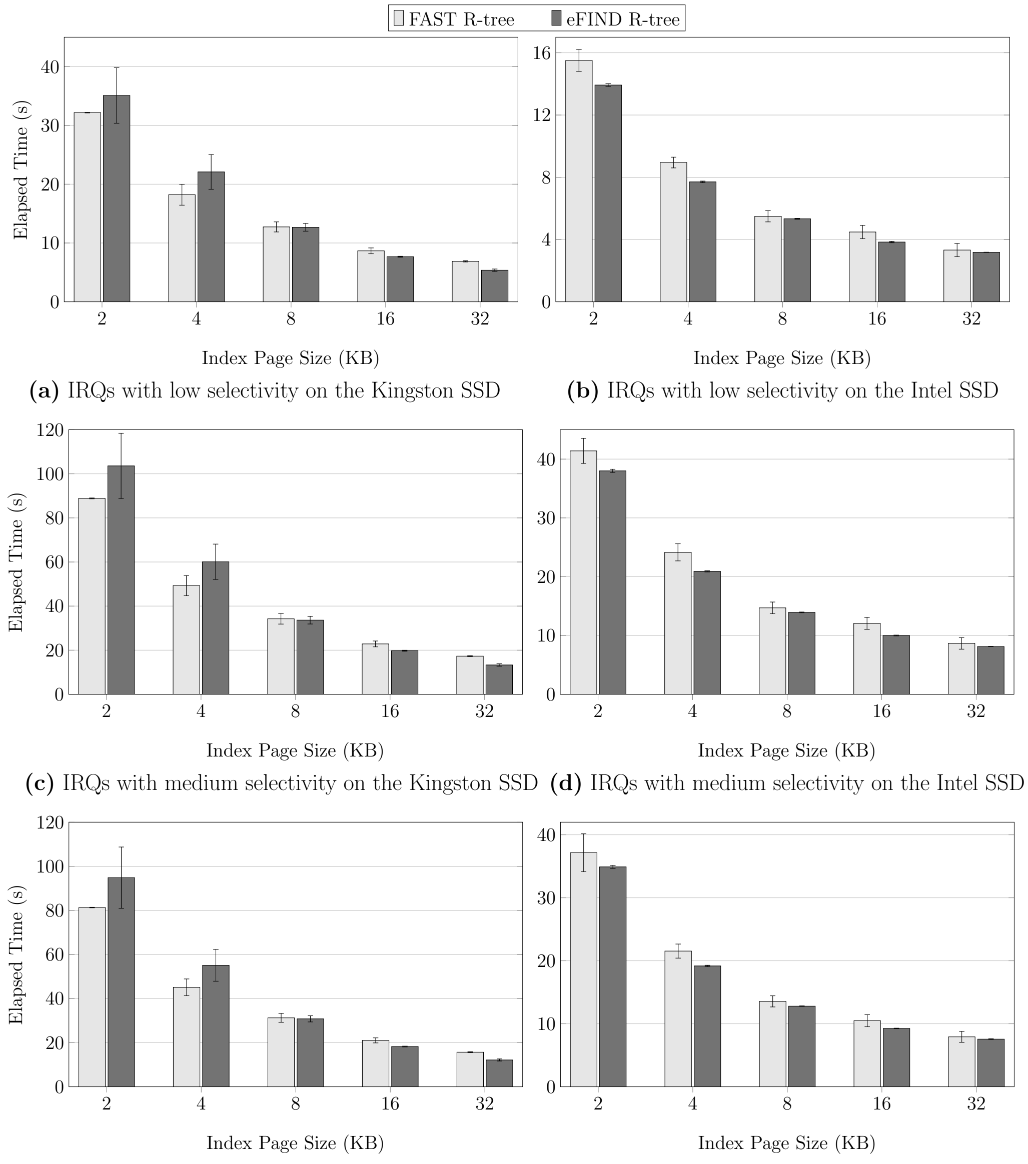
### *5.3. Spatial Query Processing*

For the high-end Intel SSD, the eFIND R-tree always provided the best performance, but only slightly overcame its competitor (Figures 7b, d, and f). Its performance gains ranged from 4% to 6%. The internal characteristics of the Intel SSD like its fast read performance contributed to these results. As for the low-end Kingston SSD (Figures 7a, c, and e), the eFIND R-tree provided the best performance only for larger pages but delivered better performance gains. Its gains were of 22% and 23% for the index pages of 16KB and 32KB respectively. In both SSDs, we believe that the performance gains were not more expressive because of the high cost of loading index pages read from the SSD to the main memory. This is further analyzed in Section 6.3.

The time spent by the configurations to process the IRQs decreased as the index page size increased because a large page size allows that more entries be loaded into the main memory, requiring fewer reads. Hence, the index page size of 32KB provided the best results for both configurations and SSDs. For this page size, the eFIND R-tree was better than the FAST R-tree for all selectivity levels. The IRQs using query windows with 0.001% (Figures 7a and b) required less time to be executed than the other query windows because of their low selectivity. The IRQs with medium and high selectivity (Figures 7c and d, and Figures 7e and f) required more elapsed time because of the high number of random reads performed on the SSD together with the traversal of several entries in the main memory.

## 6. Experimental Analysis of the Design Goals

In this section, we evaluate the effect of each design goal in the performance behavior of eFIND. For these experiments, we used the Intel SSD to execute the same workloads from Section 5.1. Specifically here, in addition to index construction, we report the results only for IRQs with low selectivity because we gathered similar performance behavior for the other selectivity levels. Throughout the discussions conducted in this section, we show the results in two different ways of visualization. The first one is an overview that shows the performance behavior of the eFIND R-tree for all tested configurations. The second one applies a zoom in the overview by considering the index page sizes that demonstrated the best performance results: 2KB and 4KB for building indices, and 16KB and 32KB for processing IRQs. This



**(a)** IRQs with low selectivity on the Kingston SSD **(b)** IRQs with low selectivity on the Intel SSD

**(c)** IRQs with medium selectivity on the Kingston SSD **(d)** IRQs with medium selectivity on the Intel SSD

**(e)** IRQs with high selectivity on the Kingston SSD **(f)** IRQs with high selectivity on the Intel SSD

Figure 7: The configurations showed the best performance to process the IRQs when using large index page sizes. For these cases, the eFIND R-tree outperformed the FAST R-tree for all selectivity levels and on both SSDs.

zooming strategy allows us to better visualize and compare the differences in the results.

## *6.1. Effect of the Flushing Operation on the Write Buffer*

In this section, we analyze the effect of parameters related to the write buffer (Goal 1), which also includes the flushing operation (Goal 2). We varied two tuning parameters of Algorithm 3: (i) the function employed by the flushing policy to calculate the degree of a flushing unit, and (ii) the percentage value $p$. Since $p$ determines the amount of index page identifiers that the flushing policy must handle, we first varied the flushing policies and fixed $p$ to 60%. Then, we fixed a flushing policy and varied $p$. For all experiments of this section, we used the size of log equal to 10MB and turned off the support for the read buffer and the temporal control.

**Varying the flushing policy.** We evaluated the set $\mathcal{P} = \{FP_m, FP_{mh}, FP_{mha}, FP_{mhao}\}$ of flushing policies (Section 4.4). Figure 8 depicts the performance results of each flushing policy in $\mathcal{P}$. The flushing policies $FP_m$ and $FP_{mh}$ showed the best elapsed times when building indices (Figures 8a and b). Considering the number of reads and writes made by them, $FP_{mh}$ required up to 3% less operations than $FP_m$ because it prioritizes the nodes in the highest levels of the index, which are not so frequently updated. The use of other characteristics like the modified and overlapping areas of the modifications did not improve the performance of the index construction because of their computational costs. The flushing policies $FP_{mha}$ and $FP_{mhao}$ showed the worst results with losses up to 34% compared to the $FP_{mh}$.

With respect to the execution of IRQs, the flushing policies showed similar elapsed times to process them (Figures 8c and d). That means the order in which the index pages were written to the SSD did not impair the performance of the IRQs. Because of these results, we assume the flushing policy $FP_{mh}$ in the remainder of the experiments.

**Varying $p$.** We now analyze the impact of the tuning parameter $p$, which determines the number of index pages to be considered by the flushing policy based on the balance between the recency of modifications and number of modifications. We evaluated each value in $\{20\%, 40\%, 60\%, 80\%\}$ as shown in Figure 9. Our analysis focus on the index construction since the experiments for processing the IRQs (Figures 9c and d) showed similar elapsed times than those depicted in (Figures 8c and d), respectively.

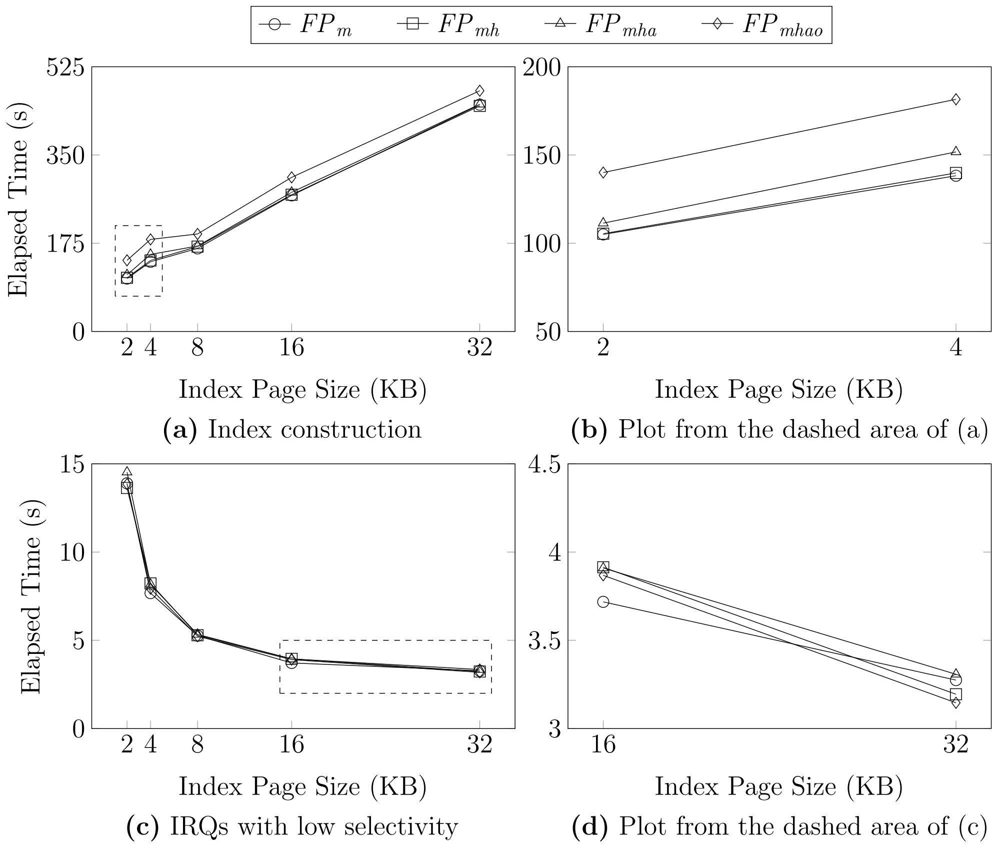


Figure 8: Effect of different flushing policies. The flushing policy $FP_{mh}$ showed the best results when building eFIND R-trees (a) and (b). These flushing policies did not significantly affect the query performance (c) and (d).

For building indices, the best balance was obtained for $p$ equal to 60% (Figures 9a and b). If $p$ is small (i.e., 20%), the flushing algorithm picks index pages containing only a few modifications, reducing the available space of the write buffer after the flushing operation. In this case, the number of flushing operations needed to create space for storing new modifications is increased. $p = 20\%$ required 7% more writes than $p = 60\%$. If $p$ is very large (i.e., 80%), index pages frequently modified are flushed multiple times, decreasing the performance of the index construction. $p = 60\%$ outperformed $p = 80\%$ by showing reductions up to 5%.

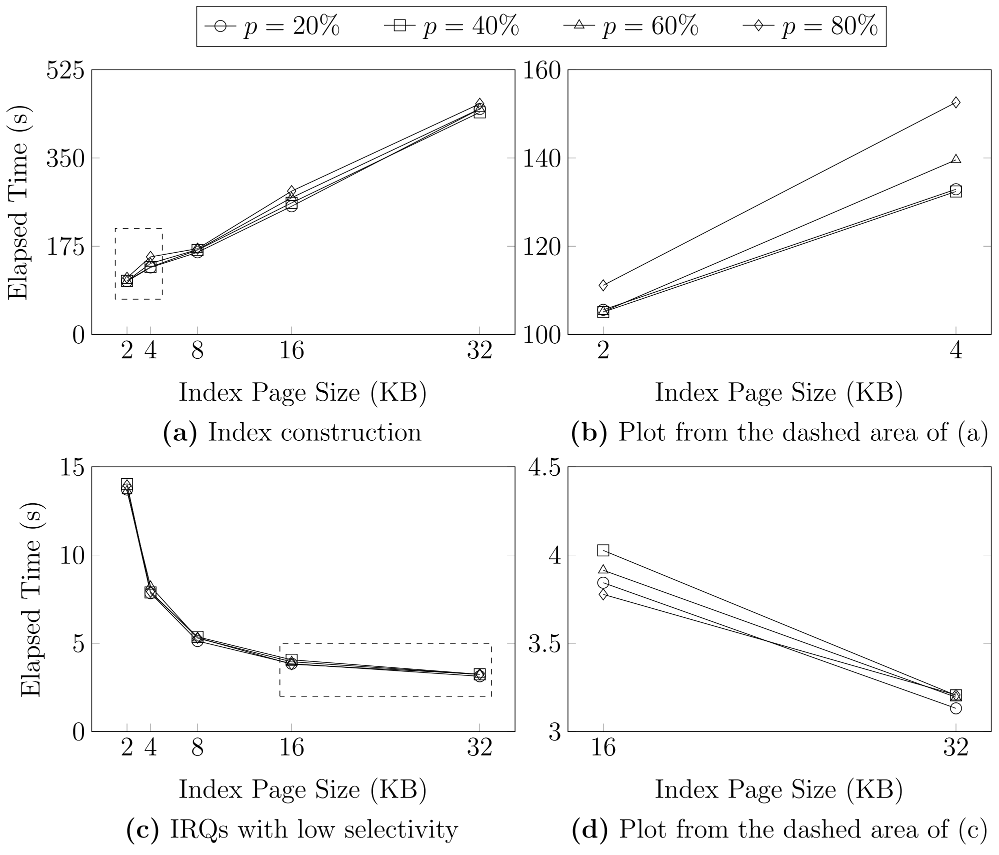


Figure 9: Effect of the percentage value $p$ used in the first line of Algorithm 3. Although $p = 20\%$ showed better elapsed times in the index construction (a) and (b), it required more writes than $p = 60\%$, which in turn showed a better performance than $p = 80\%$. Varying $p$ did not significantly affect the query performance (c) and (d).

## *6.2. Effect of the Read Buffer*

In this section, we analyze the effect of allocating a percentage of the buffer for the read buffer (Goal 3). We varied this percentage in 0%, 20%, 40%, 60%, and 80%, forming the configurations *Without RB*, *RB 20%*, *RB 40%*, *RB 60%*, and *RB 80%*, respectively. As a result, each configuration divided the buffer of the eFIND R-tree (i.e., 512KB) into two parts, the read buffer, and the write buffer. Since our focus is to analyze the balance between the sizes of the read buffer and the write buffer, we did not vary the read buffer replacement policy by considering only the LRU. For all experiments

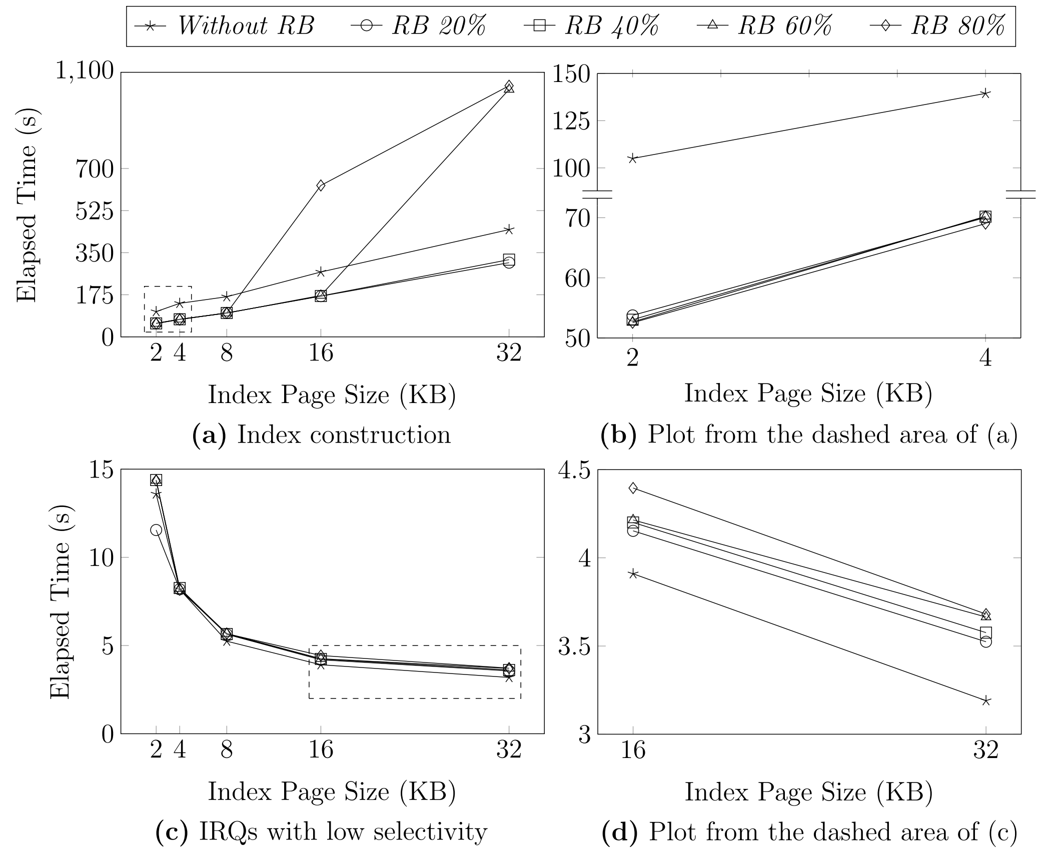


Figure 10: Effect of allocating a part of the buffer for the read buffer. The read buffer showed to be very promising to improve the performance of the index construction (a) and (b). But, this was not the case for the execution of IRQs (c) and (d) because of the cost of loading pages from the SSD.

of this section, we used the flushing policy $FP_{mh}$, $p = 60\%$, and log size of 10MB. The support for the temporal control was turned off.

In most cases, the use of the read buffer, independently of its dedicated percentage, greatly improved the elapsed time for building indices, if compared to *Without RB* (Figures 10a and b). The best configuration was *RB 20%*. Compared to *Without RB*, it showed reductions between 31% to 49%. Compared to *RB 40%*, *RB 60%*, and *RB 80%*, *RB 20%* showed reductions in the number of writes varying from 2% to 99%. If larger allocations for the read buffer are used, fewer modifications can be stored in the write buffer leading to an increasing number of flushing operations, especially for large sizes of the index page. For instance, *RB 80%* showed the worst performance results for the index page sizes equal to 16KB and 32KB. In general, the effectiveness of the read buffer can be also improved by prefetching frequently accessed nodes, as discussed in Section 6.3.

As for the processing of IRQs (Figures 10c and d), allocating space for the read buffer did not improve this processing. This occurred even when a large allocation for the read buffer was used (i.e., 80%). Because of the cost of loading the index pages read from the SSD to the main memory, the read buffer impaired the performance of the IRQs. Another fact that contributed to these results was the high cost of processing spatial data in the main memory because of the traversal of multiple paths of the tree to answer an IRQ.

## *6.3. Effect of the Temporal Control*

In this section, we analyze the use of the temporal control to avoid interleaved reads and writes (Goal 4). In addition, we evaluate how to improve the cost of loading index pages read from the SSD to the main memory. For this, we employed 2Q as the read buffer replacement policy because of its good performance on newer memories [33]. There are two versions of 2Q: (i) the simplified version, and (ii) the full version. The simplified version of 2Q, namely *S2Q*, caches the most recent accessed pages in an LRU queue, and stores identifiers of frequently accessed pages in a FIFO queue. Since this FIFO queue is equivalent to the read queue *RQ* of eFIND (Section 4.2), S2Q can employ our *RQ* for its management. The full version of 2Q, namely *F2Q*, caches frequently accessed pages in a FIFO queue, stores recently accessed pages in an LRU queue, and maintains the identifiers of pages with at least two accesses in a FIFO queue. The later FIFO queue is also compatible with *RQ* of eFIND.

We compared the non-support for the temporal control, termed here *Without TC*, to the following configurations with temporal control: (i) the *LRU TC*, which employed the LRU; (ii) the *S2Q TC*, which employed the S2Q; and (iii) the *F2Q TC*, which employed the F2Q. For all experiments of this section, we used the flushing policy $FP_{mh}$, $p = 60\%$, 20% of the buffer dedicated for the read buffer, and log size of 10MB.

Figure 11 shows that the combination of 2Q algorithms with the temporal control improved the performance not only when building indices (Figures 11a and b), but also when processing IRQs (Figures 11c and d). Re-

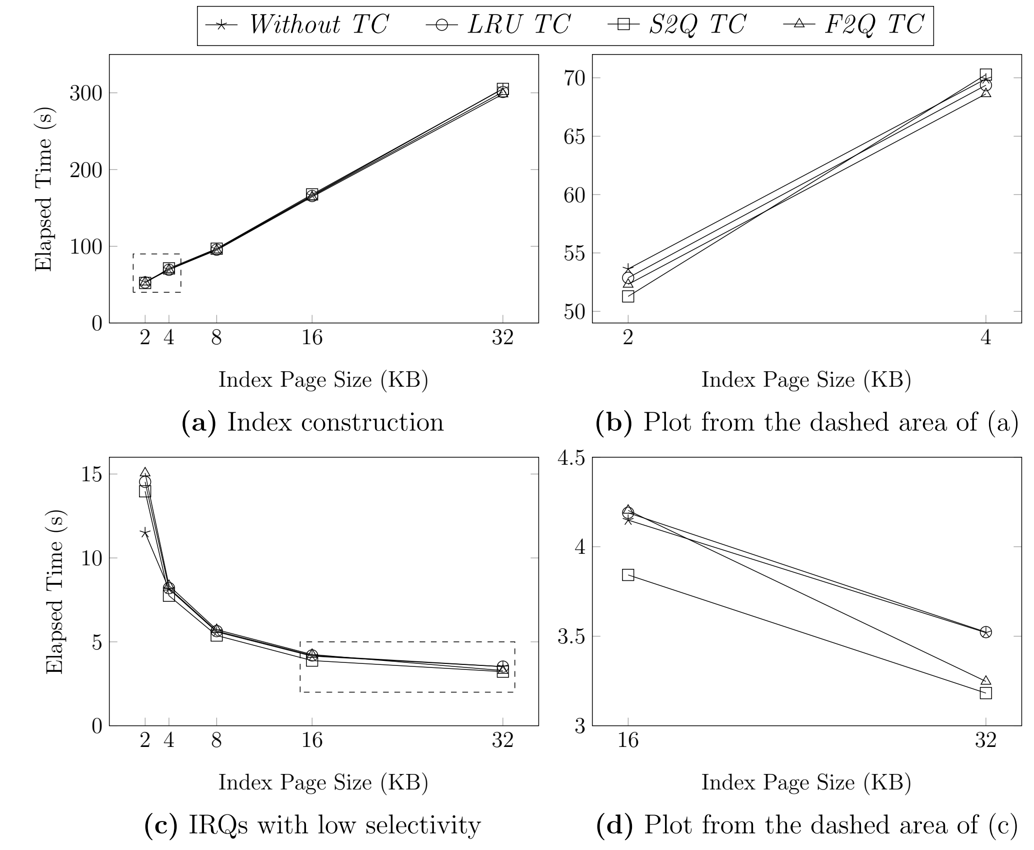


Figure 11: Effect of the temporal control when building eFIND R-trees (a) and when executing IRQs with low selectivity (c). The temporal control with the S2Q, *S2Q TC*, showed the best results as visualized in the zoomed areas (b) and (d).

garding index construction, *S2Q TC* and *F2Q TC* showed the best elapsed times. *Without TC* showed to be inefficient because of interleaved reads and writes. Although *LRU TC* improved the performance of *Without TC*, the use of LRU did not show the best results because it did not provide a full integration with the temporal control. Considering the index pages of 2KB and 4KB (Figure 11b), the performance gains of *S2Q TC* and *F2Q TC* over *LRU TC* were between 2% and 4%, respectively. This means that the temporal control using both versions of 2Q showed best performance because it prefetches frequently accessed nodes and provides specific write patterns to improve index performance on SSDs.

As for the execution of IRQs (Figures 11c and d), *S2Q TC* showed the best elapsed times when considering the index pages of 16KB and 32KB (Figures 11d). While LRU faced problems regarding the cost of loading objects from the main memory, F2Q required complex management of the cached objects. Hence, *S2Q TC* guaranteed reductions of 8% and 10% over *LRU TC*, and reductions of 2% and 9% over *F2Q TC*.

## *6.4. Effect of the Log Size*

In this section, we analyze the effect of guaranteeing data durability. We compared the following configurations that varied the log size: *0MB*, *5MB*, *10MB*, *50MB*, and *100MB*. Since the management of the log is only related to index modifications, we focus on the index construction only. For all experiments of this section, we used the flushing policy $FP_{mh}$, $p = 60\%$, and S2Q combined with the temporal control.

Compared to the non-support for data durability (i.e., *0MB*), compacting the log clearly required extra time (Figure 12a). But, this extra time decreased as the size of the index page increased. This is related to the efficiency of the compaction, which refers to the number of log entries that can be discarded. A log entry is only discarded if its index page was already flushed. For small index pages, a low number of log entries is discarded because of the high range of created pages. In this case, the compaction was inefficient and was executed multiple times. On the other hand, the range of created pages was low for large index pages, increasing the number of discarded log entries and improving the compaction of the log. Hence, for the page size of 32KB, compacting the log required the lowest overhead, varying from 6% to 8%.

The size of the log also impacts on the performance of its compaction. In general, building indices took more time for large log files (i.e., *50MB* and *100MB*) since the compaction log algorithm traverses all log entries to compact the log. But, the size of the log cannot be very small; otherwise, the compaction is inefficient. Our experiments showed best elapsed times for the log sizes of *5MB* and *10MB* (Figure 12b), showing reductions from 3% to 12% compared to *50MB* and *100MB*.

Another aspect that affects the efficiency of the compaction log is the size of the write buffer. If the write buffer is very large, the compaction of the log file is not so efficient because only a few number of flushing operations is made. Further, if the log size is also very small, probably the log file cannot be compacted because flushing operations were not executed. In this

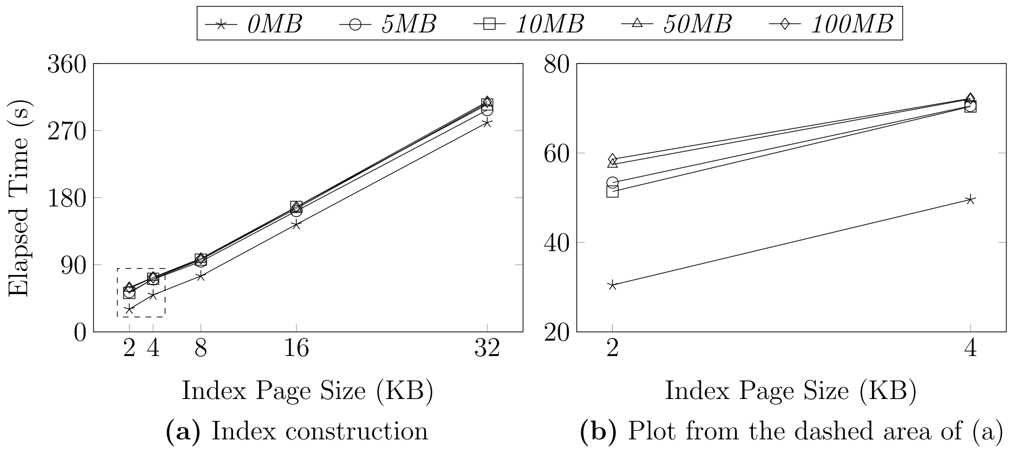


Figure 12: Effect of the log size when building eFIND R-trees (a). Although the additional cost, compacting the log is needed to better use the space allocated to provide data durability. The log size *10MB* showed the best results for small index pages (b).

case, the compaction algorithm should force the execution of some flushing operations to compact the log file, if space is critical for the application. These aspects are also discussed in [14].

## *6.5. Summary of the Effects of the Design Goals*

Throughout Section 6, we have evaluated the effects of each design goal by employing an incremental strategy. That is, we first discovered the best parameters related to the write buffer and the flushing operation without considering the read buffer and temporal control. Then, we added the support for the read buffer, and subsequently, considering the best configuration of the read buffer, we added the support for the temporal control. Finally, we varied the log size to analyze its effect. This incremental strategy allowed us to understand the effects of each design goal isolatedly. The summary of the main findings are detailed as follows.

**Write buffer and flushing operation (Goals 1 and 2).** We empirically analyzed two tuning parameters: (i) the flushing policy, and (ii) the balance between the recency of modifications and the number of modifications of modified nodes (i.e., $p$). For building indices, the flushing policy $FP_{mh}$ showed performance gains up to 34% and executed less number of writes, while the tuning parameter $p = 60\%$ showed the best results in terms of the number of writes and elapsed time. Varying flushing policies and $p$ did not significantly affect the query performance because they are parameters for flushing operations.

**Read buffer (Goal 3).** We empirically found that allocating a portion of the buffer for the read buffer is a very effective approach to improve the performance when building indices. Allocating 20% for the read buffer and 80% for the write buffer showed the best results. Compared to the non-use of the read buffer (i.e., 100% of the buffer for the write buffer), it showed reductions between 31% to 49%. Compared to other allocation sizes for the read buffer, it showed reductions in the number of writes varying from 2% to 99%. The read buffer did not improve significantly query processing because of the cost of loading index pages read from the SSD to the main memory.

**Temporal control (Goal 4).** To improve the effectiveness of the read buffer and avoid interleaved reads and writes, we analyzed the use of the temporal control. The combination of the temporal control with the read buffer replacement policy S2Q showed the best performance results because it prefetches frequently accessed nodes and provides specific write patterns in flushing operations. For building indices, the performance gains of this configuration were between 2% and 4%, compared to the LRU. For query processing, the reductions were between 8% and 10%.

**Data durability (Goal 5).** We showed that guaranteeing data durability requires an extra cost since the log file should be compacted in order to improve the space utilization of SSDs. The extra cost decreased as the size of the index page increased because in this case, the compaction is more efficient. On the other hand, the extra cost increased as the size of the log file increased because of the number of log entries to be processed. For small page sizes, our experiments showed the best results for the log size of 10MB. Compared to other log sizes, it presented reductions between 3% to 12% when building indices.

## 7. Related Work

Since many characteristics of flash-aware spatial indices are inspired by buffer managers and unidimensional indices for flash memory, in Sections 7.1 and 7.2 we present overviews of *flash-aware buffer managers* and *flash-aware unidimensional indices* respectively. Next, in Section 7.3 we detail existing flash-aware spatial indices and compare them according to our design goals.

## *7.1. An Overview of Flash-Aware Buffer Managers*

Database buffer management is a traditional field that provides general solutions to speed up accesses to storage devices [34]. Commonly, a page replacement algorithm is employed by a buffer to decide which pages should be maintained in the main memory. For HDDs, we can cite the classical Least Recently Used (LRU) replacement algorithm [27], the two versions of the 2Q replacement algorithm [28], and the Adaptive Replacement Cache (ARC) [29] as examples. There are also general buffers proposed to deal with the intrinsic characteristics of flash memory. In this section, we provide the central idea of some flash-aware buffer managers.

Many flash-aware buffer managers are based on the LRU. The *Clean-First LRU* (*CFLRU*) [35] divides the LRU into two regions, the working region that stores recently used pages, and the clean-first region that stores candidates for eviction. CFLRU prioritizes clean pages, stored in the clean-first region, to be evicted. The *Clean-First Dirty-Clustered* (*CFDC*) [36] improves the page replacement of CFLRU by dividing the clean-first region into two other regions. CFDC also changes the priority of the candidates for eviction by considering their locality in the flash memory. The *Flash-based Operation-aware buffer Replacement*$^+$ (*FOR*$^+$) [37] is also based on the LRU and divides it into two regions, hot and cold, according to the access frequency. Each page cached in the FOR$^+$ has a weight that is based on region membership, which is used for eviction. Finally, the *Flash Based-ARC* (*FBARC*) [38] is based on the ARC. FBARC distinguishes itself by creating a write list that uses the locality of evicted pages as a temporal control to produce semi-sequential write patterns.

Commonly, these general-purpose buffer managers do not use special knowledge from the data structure in their page replacement algorithms. That is, they are not specially designed to deal with index structures. As we discuss in Sections 7.2 and 7.3, proposals of unidimensional and spatial indices usually adapt well-known buffer managers to include specific characteristics of index structures. For instance, instead of storing an entire modified index page in an in-memory buffer, flash-aware indices often store modified entries only, improving memory management. Another example is the development of specialized flushing algorithms that consider both intrinsic characteristics of SSDs and index structures.

## *7.2. An Overview of Flash-Aware Unidimensional Indices*

Unidimensional indices manipulate alphanumeric data in order to deliver fast search operations. For HDDs, we can cite the traditional B-tree and its variants, the B+-tree and the B*-tree, as examples [39]. Motivated by the positive characteristics of SSDs, researchers proposed adaptations for the B-tree and its variants [40, 41] or even new tree structures [42, 43, 44] for indexing alphanumeric data on SSDs efficiently. Our goal here is not to describe in detail all flash-aware unidimensional indices, but show their characteristics to index data on SSDs.

The main focus of the flash-aware unidimensional indices is to deal with the poor performance of random writes. The first strategy employed to this end is to possibly execute an excessive number of sequential writes and random reads to avoid random writes as much as possible. Examples of flash-aware unidimensional indices that employ this strategy are the *CHC-tree* [42] and the *Lazy-Adaptive tree* [43].

Another strategy is to store index modifications in an in-memory buffer and flush them in batch when space is needed. Existing flash-aware unidimensional indices mainly differ on how the write buffer is managed. The *B-tree over the FTL* [40] makes use of logical addresses of the FTL to organize the write buffer and packs the modifications into logical blocks of the FTL in a flushing operation. The *FD-tree* [44] focuses on large-capacity SSDs and organizes the write buffer in different levels of the tree, respecting ascending order. The *read/write optimized B+-tree* [41] allows overflowed nodes to reduce random writes and leverages Bloom filters to reduce extra reads to these overflowed nodes.

## *7.3. Flash-Aware Spatial Indices*

There are few flash-aware spatial indices proposed in the literature. The *R-tree over FTL* (*RFTL*) [11], is a straightforward extension of the R-tree. It does not change the structure of the R-tree and only employs a write buffer to deal with the well-known poor performance of random writes of SSDs. The RFTL faces two main problems. First, the write buffer does not store the results of the modifications of nodes, but only how they must be performed. This means that to retrieve a node, the RFTL has to remake the modifications on this node, degrading search performance. Second, the flushing operation is an expensive operation since it writes all modifications stored in the write buffer.

The *Log-Compact R-tree* (*LCR-tree*) [12] emerged to improve the flushing and search operations of the RFTL by using a log-structured format to store the modifications in its write buffer. However, the management of this write buffer requires an additional computational cost to keep the log-structured format. Another problem is the lack of a flushing policy for the flushing operation, which still requires long times to write all buffered modifications. In addition, its write buffer does not store the results of the modifications.

The *aggregated RFTL* (*aRFTL*) [13] was proposed to be used in decision making systems. It stores numeric values together with the MBR of the nodes to represent the different aggregations of a system. Since the aRFTL is only a simple extension of the RFTL, it does not advance on solving its aforementioned problems.

The *Framework for Search Trees* (*FAST*) [14] is an upper-level solution that transforms any disk-based hierarchical index into a flash-aware index. To this end, FAST generalizes the write buffer to be applied to any hierarchical index, which includes unidimensional and spatial indices, such as the creation of the FAST R-tree from the R-tree. This buffer also stores the results of the modifications, improving search performance. On such buffer, FAST applies a specialized flushing algorithm to create space for new modifications. Another characteristic of FAST is its support for data durability. Despite the aforementioned positive characteristics, FAST faces the following problems. First, it can write a flushing unit containing a node without modification, 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, the modifications of a node are stored in a list that allows repeated elements. That means this list can store the result of old modifications and a full scan is needed to retrieve the most recent version of a node.

In this article, we consider FAST as the state of the art and empirically compare it against eFIND in Section 5. FAST has the same goal of eFIND in terms of porting any tree structure for SSDs. However, eFIND distinguishes from FAST because it employs efficient in-memory data structures for the write buffer in order to store only the latest version of modified entries. Further, eFIND employs a flushing algorithm that does not lead to unnecessary writes to the SSD since it considers only the modifications stored in the write buffer. Finally, eFIND provides a read buffer and a temporal control to deal with other intrinsic characteristics of SSDs that FAST does not consider (see Table 2). These advantages contributed to the performance gains of eFIND, as detailed in Section 5.

The *Flash-Optimized R-tree* (*FOR-tree*) [15] improves the flushing algorithm of FAST by dynamically creating flushing units with only the modifications stored in the write buffer. It also abolishes splitting operations of full nodes by allowing overflowed nodes. When a specific number of accesses in an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting elements in its parent, 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, spatial objects are stored in the overflowed root node in a sequential form when building an index. This critical problem disallowed us to execute our experiments since the FOR-tree failed to construct the index over our dataset.

There are also flash-aware spatial indices for multidimensional points [16, 17, 18]. A common limitation of these indices is that they mainly focus on two-dimensional points only. *MicroHash* and *MicroGF* (*MH* & *MGF*) [16] are index structures to execute spatial queries on flash-based sensor devices with very limited main memory and low processing capabilities. Since these indices are designed to sensor devices, they employ write buffers only. The *K-D-B-tree over flash memory* (*F-KDB*) [17] adapts the K-D-B-tree [45] to be implemented over the FTL. It employs a write buffer that stores modified entries of the K-D-B-tree, called logging entries. When its write buffer is full, a flushing policy selects some logging entries to be written. The main problem of F-KDB is that retrieving a node is a complex operation since logging entries of a node might be scattered over different flash pages. Finally, the *Grid file for flash memory* (*GFFM*) [18] employs a buffer strategy based on the LRU to cache modifications of the grid file [46]. A flushing operation only writes to the SSD those index pages that are classified as cold pages. However, the number of modifications is not taken into account, leading to a possibly high number of flushing operations.

To the best of our knowledge, there is no flash-aware spatial index that fulfills all design goals of Section 3, as shown in Table 2. Existing flash-aware spatial indices do not improve the performance of reads and do not avoid interleaved reads and writes. Among them, FAST provides the best characteristics. Hence, we consider FAST as the state of the art in spatial indexing for SSDs by comparing it in our experiments.

On the other hand, eFIND consists of sophisticated managers that fulfill all design goals. In this article, we extend the first version of eFIND [20] as follows. First, we leverage efficient data structures to manipulate the write

Table 2: Comparison of flash-aware spatial indices according to our design goals (Section 3).

| | *Write buffer* | *Specialized flushing algorithm* | *Read buffer* | *Temporal control* | *Data durability* |
|---|---|---|---|---|---|
| RFTL [11] | ✓ | | | | |
| LCR-tree [12] | ✓ | | | | |
| aRFTL [13] | ✓ | | | | |
| FAST [14] | ✓ | ✓ | | | ✓ |
| FOR-tree [15] | ✓ | ✓ | | | |
| MH & MGF [16] | ✓ | | | | |
| F-KDB [17] | ✓ | ✓ | | | |
| GFFM [18] | ✓ | ✓ | | | |
| eFIND | ✓ | ✓ | ✓ | ✓ | ✓ |

buffer and improve the computational complexity of the flushing algorithm. Second, we generalize the read buffer to accept a read buffer replacement policy as a parameter. Third, we refine the temporal control algorithms of eFIND and link them to the read buffer. Finally, we specify an algorithm to compact the log used for guaranteeing data durability. Another important extension refers to the performance tests. Here, we measure the efficiency of eFIND by considering two SSDs with different characteristics and by analyzing the impact of each design goal of eFIND.

## 8. Conclusions and Future Work

This article proposes eFIND, a new generic and efficient framework that transforms disk-based spatial indices into flash-aware spatial indices. eFIND is generic because it can be applied in a wide range of spatial indices, such as the R-tree [19], the R*-tree [47], the Hilbert R-tree [48], and the xBR+-tree [49], without changing original algorithms of the underlying index. Instead, it only changes the way in which reads and writes are performed on the SSD. This allows us to integrate eFIND into spatial database systems with a low-cost implementation.

eFIND is efficient because it is based on distinctive design goals that exploit the benefits of SSDs. To achieve the first design goal, eFIND implements efficient data structures to manage the write buffer, which deals with the poor performance of random writes of SSDs. The second design goal is achieved by specifying a flushing algorithm that employs a flushing policy to pick a set of modified index pages to be sequentially written to the SSD. To improve the performance of random reads and thus achieve the third design goal, a read buffer is specified. A temporal control is employed to avoid interleaved reads and writes, achieving the fourth design goal. Finally, a log-structured approach is used to achieve the last design goal, the support for data durability.

This article also deeply studied the effect of these design goals by showing their performance behavior when building indices and when processing intersection range queries. With these results, we provided the best parameter values of the eFIND R-tree, which outperformed the state-of-the-art FAST R-tree. Regarding the index construction, eFIND showed expressive performance gains that ranged from 43% to 77%. As for the query processing, eFIND showed performance gains from 4% to 23%.

Future work will deal with a number of topics. Currently, eFIND does not provide support for parallel transactions and concurrency control. Hence, a future topic is to include this support by also considering the ACID properties [50]. Another future topic is to evaluate eFIND by using other underlying spatial indices, such as the R*-tree, the Hilbert R-tree, and the xBR+-tree. In addition, the internal structure of these indices might be changed, allowing us to analyze the performance of different spatial organizations on SSDs. To this end, we aim to execute workloads containing different types of spatial queries (e.g., point query and containment range query) on large spatial datasets with different types of spatial data (e.g., points, lines, and regions). We further plan to execute these extended workloads by employing *flash simulators* [51, 52, 53]. The objective is to evaluate the performance of eFIND-based indices when using different configurations of SSDs.

Finally, by considering the emerging of *non-volatile main memories* (NVMM) like ReRAM, STT-RAM, and PCM [8, 54], we also aim to port eFIND for these memories. Although many characteristics of NVMMs are similar to SSDs (e.g., asymmetric costs of reads and writes), eFIND would not exploit all the advantages of NVMMs. The main reason is that NVMMs are byte-addressable, allowing to access persistent data with CPU load and store instructions. Therefore, our last future topic is to study how eFIND should be extended to deal with intrinsic characteristics of NVMMs.

## Acknowledgments

This work has been supported by CAPES, CNPq, and FAPESP. A. C. Carniel has been supported by the grant #2015/26687-8, FAPESP. R. R. Ciferri has been supported by the grant #311868/2015-0, CNPq.

## References

[1] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.

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

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

[4] I. Koltsidas, S. D. Viglas, Spatial data management over flash memory, in: International Conference on Advances in Spatial and Temporal Databases, 2011, pp. 449–453.

[5] Q. Liu, H. Nie, K. Bu, H. Liu, Z. Sun, M. Li, Q. Xie, An efficient flash-based remote sense image storage approach for fast access geographic information system, in: International Conference on Digital Manufacturing Automation, 2012, pp. 175–178.

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

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

[8] S. Mittal, J. S. Vetter, 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) (2016) 1537–1550.

[9] F. Chen, D. A. Koufaty, X. Zhang, 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, 2009, pp. 181–192.

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

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

[12] Y. Lv, J. Li, B. Cui, X. Chen, Log-Compact R-tree: An efficient spatial index for SSD, in: International Conference on Database Systems for Advanced Applications, 2011, pp. 202–213.

[13] M. Pawlik, W. Macyna, Implementation of the aggregated R-tree over flash memory, in: International Conference on Database Systems for Advanced Applications, 2012, pp. 65–72.

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

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

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

[17] G. Li, P. Zhao, L. Yuan, S. Gao, Efficient implementation of a multidimensional index structure over flash memory storage systems, The Journal of Supercomputing 64 (3) (2013) 1055–1074.

[18] A. Fevgas, P. Bozanis, Grid-file: Towards to a flash efficient multidimensional index, in: International Conference on Database and Expert Systems Applications, 2015, pp. 285–294.

[19] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.

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

[21] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference, 2008, pp. 57–70.

[22] L. Bouganim, B. Jónsson, P. Bonnet, uFLIP: Understanding flash IO patterns, in: Fourth Biennial Conference on Innovative Data Systems Research, 2009.

[23] F. Chen, B. Hou, R. Lee, Internal parallelism of flash memory-based solid-state drives, ACM Transactions on Storage 12 (3) (2016) 13:1–13:39.

[24] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, H.-J. Song, A survey of flash translation layer, Journal of Systems Architecture 55 (5) (2009) 332–343.

[25] S. J. Kwon, A. Ranjitkar, Y.-B. Ko, T.-S. Chung, FTL algorithms for NAND-type flash memories, Design Automation for Embedded Systems 15 (3-4) (2011) 191–224.

[26] W. Xie, Y. Chen, P. C. Roth, ASA-FTL: An adaptive separation aware flash translation layer for solid state drives, Parallel Computing 61 (2017) 3–17.

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

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

[29] N. Megiddo, D. S. Modha, ARC: A self-tuning, low overhead replacement cache, in: USENIX Conference on File and Storage Technologies, 2003, pp. 115–130.

[30] A. C. Carniel, R. R. Ciferri, C. D. A. Ciferri, Spatial datasets for conducting experimental evaluations of spatial indices, in: Satellite Events of the Brazilian Symposium on Databases - Dataset Showcase Workshop, 2017, pp. 286–295.

[31] A. C. Carniel, T. B. Silva, K. L. S. Bonicenha, R. R. Ciferri, C. D. A. Ciferri, Analyzing the performance of spatial indices on flash memories using a flash simulator, in: Brazilian Symposium on Databases, 2017, pp. 40–51.

[32] A. C. Carniel, R. R. Ciferri, C. D. A. Ciferri, Experimental evaluation of spatial indices with FESTIval, in: Satellite Events of the Brazilian Symposium on Databases - Demonstration Track, 2016, pp. 123–128.

[33] L. Lersch, I. Oukid, I. Schreter, W. Lehner, Rethinking DRAM caching for LSMs in an NVRAM environment, in: European Conference on Advances in Databases and Information Systems, 2017, pp. 326–340.

[34] W. Effelsberg, T. Haerder, Principles of database buffer management, ACM Transactions on Database Systems 9 (4) (1984) 560–595.

[35] S. yeong Park, D. Jung, J. uk Kang, J. soo Kim, J. Lee, CFLRU: a replacement algorithm for flash memory, in: International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006, pp. 234–241.

[36] Y. Ou, T. Härder, P. Jin, CFDC: A flash-aware buffer management algorithm for database systems, in: European Conference on Advances in Databases and Information Systems, 2010, pp. 435–449.

[37] Y. Lv, B. Cui, B. He, X. Chen, Operation-aware buffer management in flash-based systems, in: ACM SIGMOD International Conference on Management of Data, 2011, pp. 13–24.

[38] P. Dubs, I. Petrov, R. Gottstein, A. Buchmann, FBARC: I/O asymmetry-aware buffer replacement strategy, in: Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, 2013, pp. 58–69.

[39] D. Cormer, Ubiquitous B-tree, ACM Computing Surveys 11 (2) (1979) 121–137.

[40] C.-H. Wu, T.-W. Kuo, L.-P. Chang, An efficient B-tree layer implementation for flash-memory storage systems, ACM Transactions on Embedded Computing Systems 6 (3).

[41] P. Jin, C. Yang, C. S. Jensen, P. Yang, L. Yue, Read/write-optimized tree indexing for solid-state drives, The VLDB Journal 25 (5) (2016) 695–717.

[42] S. Byun, M. Hur, An index management using CHC-cluster for flash memory databases, Journal of Systems and Software 82 (5) (2009) 825–835.

[43] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, S. Singh, Lazy-adaptive tree: An optimized index structure for flash devices, VLDB Endowment 2 (1) (2009) 361–372.

[44] Y. Li, B. He, R. J. Yang, Q. Luo, K. Yi, Tree indexing on solid state drives, VLDB Endowment 3 (1-2) (2010) 1195–1206.

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

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

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

[48] I. Kamel, C. Faloutsos, Hilbert R-tree: An improved R-tree using fractals, in: International Conference on Very Large Databases, 1994, pp. 500–509.

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

[50] T. Harder, A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Surveys 15 (4) (1993) 287–317.

[51] X. Su, P. Jin, X. Xiang, K. Cui, L. Yue, Flash-DBSim: A simulation tool for evaluating flash-based database algorithms, in: IEEE International Conference on Computer Science and Information Technology, 2009, pp. 185–189.

[52] Y. Kim, B. Tauras, A. Gupta, B. Urgaonkar, FlashSim: A simulator for NAND flash-based solid-state drives, in: International Conference on Advances in System Simulation, 2009, pp. 125–131.

[53] A. C. Carniel, T. B. Silva, C. D. A. Ciferri, Understanding the applicability of flash simulators on the experimental evaluation of spatial indices, in: 9th Annual Non-Volatile Memories Workshop, 2018, pp. 1–2.

[54] Y. Zhang, S. Swanson, A study of application performance with non-volatile main memory, in: Symposium on Mass Storage Systems and Technologies, 2015, pp. 1–10.