Spatial Indexing on Flash-Based Solid State Drives

Anderson C. Carniel
Supervised by Cristina D. A. Ciferri
University of São Paulo, Brazil
accarniel@gmail.com

ABSTRACT

The use of a *spatial index* is fundamental to process spatial queries on spatial database systems. With the growing use of *flash-based Solid State Drives* (SSDs), designing spatial indices for these storage devices has gained increasing attention. Hence, while there are spatial indices dedicated to magnetic disks (i.e., *disk-based spatial indices*), the literature has focused on to propose *flash-aware spatial indices* that consider the intrinsic characteristics of SSDs, such as the asymmetric costs of reads and writes. However, the research to date has not been able to establish a flash-aware spatial index that actually exploits all the benefits of SSDs. The principal goal of this PhD work is to propose an efficient flash-aware spatial indexing method by taking into account the system implications introduced by SSDs. The main preliminary result of this PhD is eFIND, a generic framework that transforms a disk-based spatial index into an efficient flash-aware spatial index. Performance tests showed that, compared to the state of the art, eFIND improved the construction of spatial indices from 60% to 77%, and the spatial query processing from 22% to 23%.

1. INTRODUCTION

Spatial database systems largely employ spatial indices to process spatial queries [6], such as intersection range queries (IRQs). A wide range of spatial indices like the R-tree and its variants assume that the indexed spatial objects are stored in magnetic disks (i.e., *Hard Disk Drives* - HDDs). Hence, they consider the slow mechanical access and the cost of search and rotational delay of disks in their design; we term them as *disk-based spatial indices*.

On the other hand, 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) [5, 2]. In fact, SSDs have been widely used as secondary storage in database servers. In contrast to HDDs, SSDs have a smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

SSDs have also intrinsic characteristics that introduce several system implications [8]. A well-known characteristic is that a write requires more time and power consumption than a read. In addition, SSDs require an erase-before-update operation to rewrite a page since they only are capable of writing to empty pages. To deal with these characteristics, some *flash-aware spatial indices* have been proposed in the literature [11, 9, 10, 7].

However, current flash-aware spatial indices do not exploit all the benefits of SSDs. First, the impact of SSDs on the spatial indexing context is understudied, particularly for designing of efficient flash-aware spatial indices. Second, existing indices assume that the random read is the fastest operation of SSDs and thus, they execute an excessive number of reads to minimize the number of random writes. But, this behavior degenerates SSD performance [8]. Third, there is no special treatment to minimize the effects of interleaved reads and writes, which also negatively impact SSD performance [8]. Finally, existing flash-aware spatial indices handle inefficient in-memory data structures.

In this PhD work, we aim to solve the aforementioned problems by pursuing three objectives. The first objective is to understand the impact of SSDs on the spatial indexing context by means of empirical evaluations. To this end, we analyzed the performance behavior of spatial indices on HDDs and SSDs [2]. The goal was to check whether a spatial index that shows the best results on the HDD also shows the best results on the SSD and vice-versa. Hence, we analyzed what should be modified in existing spatial indices in order to achieve good performance on SSDs.

The second objective is to propose a set of design goals for designing flash-aware spatial indices. To this end, we correlated the intrinsic characteristics of SSDs and observations from our empirical studies. The proposed design goals were validated through the implementation of the *efficient Framework for spatial INDexing on SSDs* (eFIND) [3]. eFIND 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. Our experiments showed that eFIND is very efficient since it provides efficient data structures specifically developed to achieve each design goal.

Proceedings of the VLDB 2018 PhD Workshop, August 27, 2018. Rio de Janeiro, Brazil.

The third, and last, objective is to propose a novel efficient and robust flash-aware spatial index. Currently, we are developing this index by taking the design goals of eFIND as a basis and by modifying the internal structure of the index to exploit the benefits of SSDs. An initial idea is to consider structures based on the R-tree, where levels nearest to leaf nodes might have an update-intensive workload. Thus, minimizing the number of split operations on these levels will lead to a decreasing number of writes to SSDs. Further, high levels of the tree can be buffered to minimize the number of reads from SSDs since the nodes in such levels are not frequently modified and are read-intensive.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 summarizes our studies on the impact of SSDs on the spatial indexing. Section 4 introduces our design goals and eFIND. Finally, Section 5 concludes the paper and presents future work.

2. RELATED WORK

We classify the existing approaches that study spatial indexing on SSDs in the following groups: (i) works that conduct experimental evaluations, and (ii) proposals of flash-aware spatial indices. With respect to the first group, there are some performance studies that analyze the affect of SSDs on the spatial indexing. Unfortunately, a common limitation of these studies is that they do not vary parameters that impact the performance of spatial indices, such as the page size (i.e., node size). An example of performance study is [5], which empirically analyzes the R*-tree in the computation of k-nearest neighbor queries on HDDs and SSDs.

With respect to the second group, the existing flash-aware spatial indices are inspired by unidimensional indices for flash memory (e.g., the *LA-tree* [1]) and often attempt to port the R-tree to be flash-aware. We detail the main characteristics of flash-aware spatial indices as follows.

The *RFTL* [11] is a first 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 main problem of RFTL is the flushing operation because it writes all modifications stored in the write buffer, requiring high elapsed times. Another problem is related to the data durability. This means that the modifications stored in the write buffer are lost after a system crash or power failure.

The *LCR-tree* [9] emerged to improve the flushing operation of 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.

FAST [10] generalized the write buffer for allowing the transformation of any disk-based hierarchical index into a flash-aware index, such as the creation of the FAST R-tree from the R-tree. The FAST's buffer improves the search performance by storing the results of index modifications. In addition, FAST provides a specialized flushing algorithm to create space for new modifications whenever the write buffer is full. Another characteristic of FAST is its support for data durability. However, 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. These problems impact on FAST performance, as detailed in [3].

The *FOR-tree* [7] 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 stored in the main memory. When a specific number of accesses to an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting them as entries 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 create spatial indices over voluminous datasets.

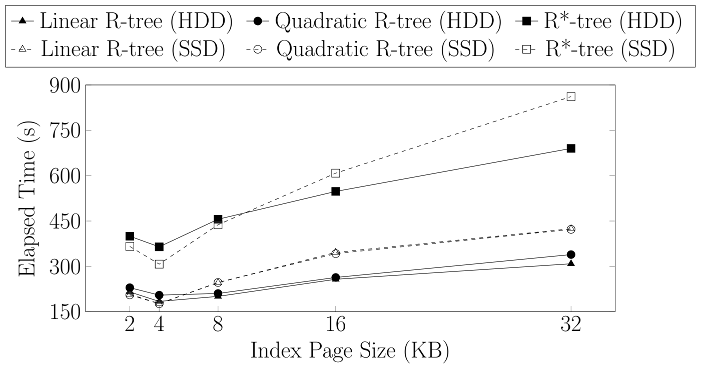

Figure 1: Performance results when creating R-trees and R*-trees on an HDD (denoted by filled marks) and on an SSD (denoted by empty marks).

3. THE IMPACT OF FLASH MEMORY ON THE SPATIAL INDEXING CONTEXT

In this section, we provide a summary of an empirical analysis of spatial indices on HDDs and SSDs that was conducted in our work [2]. Considering a dataset extracted from the OpenStreetMap containing 534,926 regions (specified in [4]), here we report the elapsed time when creating R-trees (varying the split algorithm) and R*-trees on an HDD and SSD (Figure 1). In general, the index page size of 4KB gathered the best performance results for all spatial indices. For this page size, the SSD showed better performance results with performance gains between 4% and 16% over the HDD.

On the other hand, for the index page sizes greater than 8KB we obtained best performance results by using the HDD. For these page sizes, the construction of spatial indices on the SSD showed a performance loss ranging from 6% to 38% over the HDD. This result could be considered as counter-intuitive since SSDs often have faster reads and writes than HDDs. However, the main reason for this performance degradation is due to the interleaved writes and reads during the index construction. This kind of performance behavior is well studied in the literature, such as discussed in [8]. In addition, the lack of a special treatment for random writes was also determinant since writes are the most expensive operations of SSDs.

The results suggested that the direct use of existing disk-based spatial indices does not guarantee efficiency. Therefore, we should design flash-aware spatial indices that consider the intrinsic characteristics of SSDs. This topic is addressed in Section 4.

4. DESIGNING EFFICIENT FLASH-AWARE SPATIAL INDICES

Although the intrinsic characteristics of SSDs have been well studied in the literature, it remains unclear how to deal with them to deliver good spatial indexing performance. We solve this problem by specifying a set of *design goals for flash-aware spatial indices* (Section 4.1), and by proposing eFIND (Section 4.2).

4.1 Design Goals

Our five design goals are inspired by analysis of experimental evaluations on SSDs (Section 3) and techniques used by existing indices for flash memory (Section 2). These design goals should be employed 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 internal SSD algorithms [8]. To achieve Goal 1, a flash-aware spatial index should employ an efficient in-memory buffer, called *write buffer*, to store the most recent modifications of the index. Whenever the write buffer is full, a *flushing algorithm* should be executed, which 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 flushes all modifications contained in the write buffer degenerates performance and might writes index pages frequently modified [10]. 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 (e.g., number of modifications, and the moment of its last modification). The flushing unit creator should create a flushing unit as sequential index pages, following the flushing policy, and determine the size of data that is written to the SSD in each flushing operation.

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 [8]. To achieve Goal 3, a flash-aware spatial index should use an in-memory buffer dedicated to 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. Further, the management of the read buffer should include a *read buffer replacement policy*, such as the LRU.

Goal 4 - Avoid interleaved reads and writes. Mixing reads and writes negatively affect SSD performance because of the interference between these operations [8]. 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 to aid in the management of these buffers.

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 non-flushed modifications in a log file.

To the best of our knowledge, there is no flash-aware spatial index that fulfills all these design goals. 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 (Section 2). Therefore, we consider FAST as the state of the art in spatial indexing for SSDs by comparing it in our experiments.

4.2 eFIND as a Solution

This section details eFIND, a generic and efficient framework that transforms a disk-based spatial index into a flash-aware spatial index. The eFIND's architecture consists of three sophisticated managers to meet the requirements of the design goals introduced in Section 4.1. More details about eFIND are given in [3].

Buffer Manager. It leverages two in-memory buffers to deal with random writes and reads. The first one is the *write buffer*, which stores the most recent index modifications 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 guarantees data durability (Goal 5) by keeping a log of all modifications stored in the write buffer and of flushing operations. 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.

Now, we discuss the performance gains of an extension of eFIND against the state of the art, FAST. The experiments and their results are detailed as follows.

Setup. We used a dataset from the OpenStreetMap containing 1,485,866 regions that represent the buildings of Brazil (specified in [4]). We compared the *FAST R-tree*, and the *eFIND R-tree*. They employed an in-memory buffer of 512KB. We used the best parameter values for these configurations and executed two workloads: (i) index construction, and (ii) execution of IRQs [6]. Here we show the results of the execution of 100 IRQs with query windows of 0.001% of the area of Brazil. The running environment employed was FESTIval (available at `https://github.com/accarniel/festival`), a PostgreSQL extension implemented during this PhD that allows us to conduct performance tests of spatial indices. Finally, we conducted the experiments on a Kingston SSD V300 of 480GB.

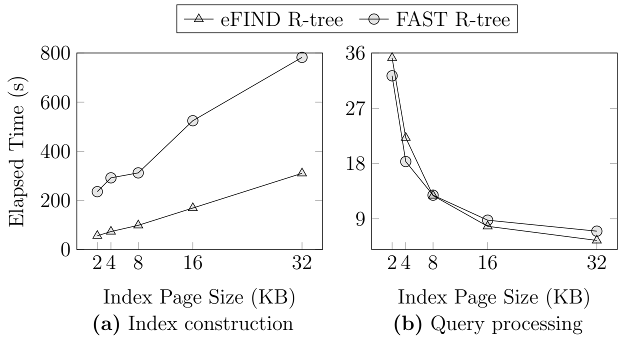

Figure 2: The eFIND R-tree showed expressive performance gains when building spatial indices (a). It also showed the best performance to process the IRQs when using large index page sizes (b).

Index Construction. As shown in Figure 2a, the eFIND R-tree overcame the FAST R-tree for all employed page sizes. Its performance gains were very expressive, ranging from 60% to 77%. The eFIND R-tree exploited the benefits of the SSDs because it is based on the design goals defined in Section 4.1. The contribution of the read buffer to obtain these results was significantly relevant even using a relatively small portion of the buffer size (20%). 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 by leveraging 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 SSD.
Spatial Query Processing. Figure 2b depicts that, for both configurations, larger page sizes provided better elapsed times since more entries are loaded into the main memory, requiring fewer reads from the SSD. For these page sizes (16KB and 32KB), the eFIND R-tree showed gains of 22% and 23% respectively mainly because of the read buffer, which contributes to reducing the number of reads.

5. CONCLUSIONS AND OUTLOOK

This paper describes a PhD work on spatial indexing on SSDs. The PhD encompasses three main goals: (i) understand the impact of SSDs on the spatial indexing context, (ii) design methods for efficient flash-aware spatial indexing, and (iii) propose a new efficient flash-aware spatial index. The first goal was achieved through extensive experimental evaluations to analyze the performance behavior of spatial indices on HDDs and SSDs. As a result, we studied the deficiencies of disk-based spatial indices when applied to SSDs. With this study, we then achieved the second goal of this PhD by proposing eFIND, which is based on a set of design goals that exploit the benefits of SSDs. eFIND is a generic framework that can be applied in a wide range of spatial indices, such as the R-tree and its variants, without changing original algorithms of the underlying index. eFIND is also efficient, showing expressive performance gains that ranged from (i) 60% to 77% when building spatial indices and from (ii) 22% to 23% when processing spatial queries.

The next activities of this PhD include the proposal of a novel flash-aware spatial index. By using the design goals of eFIND and other findings of the conducted experiments, we plan to design a tree structure that exploits the benefits of SSDs. For instance, we learned from the experiments that a high number of writes is performed from splitting operations in the bottom levels of the tree. Thus, by allowing overflowed nodes in the bottom levels, we could improve the performance of the index in maintenance operations without impairing the spatial organization of the index. Finally, the novel flash-aware spatial index will be evaluated by means of extended performance tests, that is, with other types of spatial indices in addition to IRQs, and other spatial datasets.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

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

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

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

[4] A. C. Carniel, R. R. Ciferri, and C. D. A. Ciferri. Spatial datasets for conducting experimental evaluations of spatial indices. In *Satellite Events of the Brazilian Symp. on Databases*, pages 286–295, 2017.

[5] T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma. On the impact of flash SSDs on spatial indexing. In *Int. Work. on Data Management on New Hardware*, pages 3–8, 2010.

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

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

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

[9] Y. Lv, J. Li, B. Cui, and X. Chen. Log-Compact R-tree: An efficient spatial index for SSD. In *Int. Conf. on Database Systems for Advanced Applications*, pages 202–213, 2011.

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

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