Analyzing the Performance of Spatial Indices on Flash Memories using a Flash Simulator

Anderson Chaves Carniel[1], Tamires Brito da Silva[1], Kairo Luiz dos Santos Bonicenha[1], Ricardo Rodrigues Ciferri[2], Cristina Dutra de Aguiar Ciferri[1]

[1]Department of Computer Science – University of São Paulo
13.560-970 – São Carlos – SP – Brazil

{accarniel,tamiresbs,kairo.bonicenha}@usp.br, cdac@icmc.usp.br

[2]Department of Computer Science – Federal University of São Carlos
13.565-905 – São Carlos – SP – Brazil

ricardo@dc.ufscar.br

Abstract. *Spatial databases improve the spatial query processing by employing spatial indices. Due to the advantages of flash memories over magnetic disks like faster reads and writes, there is a special interest in managing spatial indices in these memories. However, many flash memories employ a Flash Translation Layer that does not provide open access to many important statistics, restricting the performance analysis of spatial indices. Flash simulators are promising tools to improve the performance analysis of spatial indices. In this paper, we analyze the performance of several distinct configurations of spatial indices by using a flash simulator and a real flash-based solid state drive. As a result, we provide correlations between these results to check the accuracy of a flash simulator in the spatial indexing context. In addition, we discuss the possibility of using a flash simulator as a first step for benchmarking spatial indices. That is, we check if the results provided by a flash simulator can be used to decrease the number of configurations to be evaluated in real flash memories, reducing the required time of an empirical analysis.*

1. Introduction

Advanced applications employ spatial database systems and Geographic Information Systems (GIS) for managing spatial information. For this purpose, spatial data types like *regions* are used [Güting 1994]. For instance, the area of a building is represented by a region. Commonly, applications issue *spatial queries* that return spatial objects. Typical queries involve *topological predicates*, e.g., "return all the buildings that *overlap* a given search area". To speed up the spatial query processing, spatial database systems and GIS make use of *spatial indices* [Gaede and Günther 1998], which discard spatial objects that certainly do not belong to the final result of a query. Examples of these indices are hierarchical structures like the R-tree and the R*-tree (see [Gaede and Günther 1998] for a survey). Since *Hard Disk Drivers* (HDDs) were the main storage devices used in previous decades, these indices assume that the spatial objects are stored in magnetic disks. We term them as *disk-based spatial indices*.

However, modern applications are increasingly requiring the use of newer storage devices like *flash memories* [Mittal and Vetter 2016, Brayner and Monteiro Filho 2016]. The reason is that, unlike HDDs, flash memories do not have mechanical parts and provide lower weight, size, power consumption, and read/write latency. An example of a storage device that employs flash memories is the flash-based Solid State Drive (SSD). These memories have intrinsic characteristics that introduce several system implications [Jung and Kandemir 2013]. For instance, the asymmetric cost between reads and writes, where a write requires more time and consumes more power than a read. Further, a write is only performed on empty pages of the flash memory; thus, an update is performed as an *erase-before-update* operation.

To improve the use of flash memories in current applications, the *Flash Translation Layer* (FTL) is employed [Chung et al. 2009]. It maps physical addresses of a flash memory into logical addresses and transforms reads and writes from application layers into a set of internal read, write, and erase operations. As a result, erase-before-updates can be avoided by applying an *out-of-place update* algorithm. Intuitively, this algorithm marks a page to be updated as invalid and writes its new content in another empty page. When a number of invalid pages are reached, a *garbage collector* erases the blocks with invalid pages, leading to erase-before-updates. Hence, a unique write performed on the application layer does not necessarily mean that only one write is performed on the flash memory. This also applies to reads since the data can be sliced into several physical pages and thus, a read from the application layer can lead to multiple reads on the flash memory.

To deal with the intrinsic characteristics of flash memories, *flash-aware spatial indices* have been proposed in the literature [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013, Jin et al. 2015]. In general, these indices avoid random writes by using an *in-memory buffer* that stores modifications of the index. When this buffer is full, a *flushing operation* is performed by writing a set of modified nodes, called *flushing unit*. For instance, FAST [Sarwat et al. 2013] is a framework that transforms a disk-based spatial index (e.g., the R-tree) into a flash-aware spatial index (e.g., the FAST R-tree).

However, analyzing the impact of flash memories on spatial indexing, such as to measure the performance of a given spatial index, is a problematic task. The reason is that the FTL is physically integrated into the flash memory with license restrictions of the manufacturer. Hence, these experiments often measure the processing time of index operations since the FTL restricts the collection of the number of writes, reads, and erases actually performed on the storage device. Flash simulators are promising tools for evaluating the performance behavior of spatial indices on flash memories. They emulate the structure of flash memories in the main memory, implement FTL algorithms, and enable the collection of several statistics, such as the number of writes, reads, and erases. Some flash simulators have been proposed in the literature [Su et al. 2009, Kim et al. 2009, Dong et al. 2012]. Among them, Flash-DBSim [Su et al. 2009] has been used in the (spatial) indexing context [Jin et al. 2015]. In addition, it is an open source, reusable, flexible and extensible simulator.

Despite the relevance of flash simulators, there are four important open questions that motivate this paper. They are:

1. Is a flash simulator capable of determining if a flash-aware spatial index provides better performance than a disk-based spatial index?

2. Does a spatial index that performs the best in a flash simulator also perform the best in a real flash memory, for example an SSD?
3. How can a flash simulator be used to evaluate the performance of spatial indices?
4. Is it possible to use a flash simulator as a filter of configurations of spatial indices to be evaluated in an SSD?

To answer these questions, in this paper we conduct an extensive performance evaluation of disk-based (the R-tree and the R*-tree) and flash-aware (the FAST R-tree and the FAST R*-tree) spatial indices by using two environments: Flash-DBSim and a real SSD. For all these compared indices, we varied several parameter values, resulting in different configurations of spatial indices.

We answer question 1 by analyzing the obtained results in Flash-DBSim. In this analysis, we aim to find out the performance gains of the flash-aware spatial indices over the disk-based spatial indices. Then, we correlate these performance gains with the obtained results in the real SSD. To answer question 2, we check if there is a configuration that shows the best results in both the environments. This helps us to measure the accuracy of the flash simulator. Based on the previous analyses, we answer question 3 by exploiting the applicability of flash simulators to evaluate the performance of spatial indices. By answering the questions 1 to 3, we lead to the answer of the most important question of this paper. To answer the last question, we analyze if only some configurations from a large set of configurations of spatial indices can be examined in a real SSD based on preliminary results obtained from experiments using a flash simulator.

The rest of this paper is organized as follows. Section 2 summarizes intrinsic characteristics of flash memories. Section 3 surveys related work. Section 4 details our performance evaluation. Section 5 answers our questions. Section 6 concludes the paper.

2. Flash Memories

Flash memories organize data in flash pages and flash blocks [Jung and Kandemir 2013]. A block consists of a fixed number of pages. Flash memories handle three types of operations: program (write), erase, and read. Read and write operations are performed at page level with asymmetric costs as a read requires much less time and consumes less power than a write; on the other hand, erase is performed at block level and is very expensive. In addition, flash memories do not provide an operation for updates. Thus, a three-step algorithm is performed to update the content of a page. First, the unchanged pages of its block are internally buffered. Second, the block is erased. Finally, the updated page and the buffered pages of the block are written back to the erased block. This operation is named *erase-before-update*. Moreover, flash memories have lower endurance capacity than HDDs [Mittal and Vetter 2016]. The endurance refers to the amount of write and erase operations that a block can receive before it is unavailable to support new operations.

The FTL component (see [Chung et al. 2009] for a survey) is employed to permit the use of flash memories in current computational environments. It allows a flash memory to be recognized by the operating system as a regular disk and provides only two operations for application layers: write and read. These operations are performed on logical page addresses mapped from physical page addresses of the flash memory [Mittal and Vetter 2016]. Each logical address of a page is marked either free, valid, or invalid. When a write is issued to the FTL, it writes the data into a free page and marks this page as valid. If a write is performed on a valid page (i.e., an update), the FTL marks this page as invalid and writes the new data to a free page. This operation is known as an *out-of-place update* and avoids erase-before-updates. When a number of blocks with invalid pages are reached and storage space is required, the *garbage collector* is dispatched. It selects a set of blocks to be erased, stores the content of valid pages of these blocks in other pages, and erases the selected blocks. To improve the endurance of flash memories, a *wear-leveling* algorithm is responsible for controlling the number of erases that a block can receive.

3. Related Work

There are several approaches that conduct experimental evaluations of spatial indices on flash memories. We classify them according to the following characteristics: (i) the performance evaluation of spatial indices under different storage devices [Emrich et al. 2010, Carniel et al. 2016b], and (ii) the performance evaluation of new flash-aware spatial indices on flash memories [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013, Jin et al. 2015].

The first group includes approaches that analyze the performance behavior of the spatial indexing on HDDs and SSDs. These approaches mainly compare the performance gains that SSDs provide over HDDs. For instance, [Carniel et al. 2016b] showed that an SSD provided reductions in the spatial query processing up to 96% over an HDD. However, flash simulators are not used in the experiments. This results in a limited analysis of an experiment since its focus is only on the elapsed time of a spatial index operation.

The second group consists of approaches that propose spatial indices specifically designed for SSDs. While straightforward adaptations of the R-tree are proposed [Wu et al. 2003, Lv et al. 2011], FAST [Sarwat et al. 2013] distinguishes itself by defining a generic framework to transform a disk-based spatial index into a flash-aware spatial index. In addition, the FOR-tree [Jin et al. 2015] is proposed to eliminate the split operations of the R-tree by allowing overflowed nodes. It uses a counter to decide when new primary nodes are created from an overflowed node, growing up the tree as needed. However, the counter for the root node is only incremented in search operations. Thus, instead of organizing nodes in a hierarchical structure, the construction of a FOR-tree index leads to an overflowed root node. Thus, it results in unacceptable processing times.

These approaches often conduct experiments measuring the performance of the proposed indices against disk-based spatial indices. However, the parameter values of these indices are not exploited. For instance, there is a lack of studies that measure the impact of the index page size (i.e., node size). In addition, the selectivity in spatial queries does not vary in workloads. Since the FTL is a black-box component of SSDs (see Section 2), Flash-DBSim was used in the experiments involving the FOR-tree. However, it is unclear if this simulator can be used as a filter for experiments beause there is a lack of performance comparisons between this simulator and a real SSD.

In this paper, we provide an extensive experimental evaluation for analyzing the applicability of flash memories to evaluate spatial indices by answering the four questions introduced in Section 1. Our experiments vary several parameter values and focus on the spatial query processing. As a result, we analyze the flash simulator as a tool for reducing the configurations to be evaluated in an empirical study.

4. Performance Evaluation

4.1. Configuration Setup

We used a real dataset containing 1,486,557 regions extracted from the OpenStreetMap[1]. This dataset represents the buildings of Brazil, such as hospitals, schools, universities, houses, stadiums, and so on.

We compared the configurations showed in Table 1. The disk-based spatial indices considered are: (i) the R-tree, with linear and quadratic splits, and (ii) the R*-tree, with the reinsertion policy (RP) of 30%. For these indices, we employed an in-memory buffer to cache the nodes in the highest levels of the index by using the least recently replacement (LRU) policy. The flash-aware spatial indices considered are the FAST-versions of the disk-based spatial indices: (i) the FAST R-tree, and (ii) the FAST R*-tree. For them, we applied the FAST* flushing policy (FP) because it reported the best results in [Sarwat et al. 2013]. The buffer size applied in all the configurations was equal to 512KB. In addition, we varied the size in bytes of an index node from 2KB to 16KB. Furthermore, we used the flushing unit size equal to 1 for the FAST-based spatial indices. We did not compare the FOR-tree because of the problems discussed in Section 3.

We executed two workloads: (i) index construction, and (ii) processing of intersection range queries (IRQ) [Gaede and Günther 1998]. The second workload includes the execution of three sets of 100 IRQs. These sets applied query windows corresponding respectively to 0.001%, 0.01%, and 0.1% of the area of the total extent of Brazil. Considering that the selectivity of a query is the ratio between the number of returned objects and the total objects, these sets of query windows form spatial queries with low, medium, and high selectivity, respectively.

These two workloads were executed as a sequence for each used configuration, i.e., we executed the construction of a spatial index and then the processing of the IRQs. To this end, we employed an extended version of FESTIval[2] [Carniel et al. 2016a], an open source PostgreSQL extension for benchmarking spatial indices. It was extended to integrate Flash-DBSim and to measure the performance of flash-aware spatial indices.

We used Flash-DBSim to simulate a flash memory of 512MB as depicted in Table 2. We employed this simulator because of its advantages (Section 1) and previous usage in the spatial indexing context. We collected the number of writes, reads, and erases required by each configuration to execute each workload (Section 4.2). We correlated these results by executing the same workloads on an SSD (Section 5). For this purpose, we used a local server equipped with an Intel® Core^{TM} i7-4770 with a clock frequency of 3.40GHz, 32GB of main memory, and an SSD Kingston V300 with a capacity of 480GB[3]. In this environment, we performed the tests locally to avoid network latency and flushed the system cache after the execution of each configuration. Further, we executed each workload 10 times and calculated the elapsed time as follows. For the first workload, we collected the average elapsed time. For the second workload, we collected the average elapsed time to execute the IRQs of each kind of selectivity. We employed Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, and PostGIS 2.2.0 in all the environments.

[1]http://www.openstreetmap.org/
[2]http://gbd.dc.ufscar.br/festival/
[3]https://www.kingston.com/us/ssd/consumer/sv300s3

Table 1. Configurations of the spatial indices used in the experiments.

Name	Spatial Index	Buffer Type	Parameters
Quadratic R-tree	R-tree	LRU	Split: Quadratic
Linear R-tree	R-tree	LRU	Split: Linear
R-tree*	R*-tree	LRU	RP: 30%
FAST Quadratic R-tree	FAST R-tree	FAST Buffer	Split: Quadratic; FP: FAST*
FAST Linear R-tree	FAST R-tree	FAST Buffer	Split: Linear; FP: FAST*
FAST R-tree*	FAST R*-tree	FAST Buffer	RP: 30%; FP: FAST*

Table 2. The emulated flash memory[4].

Latency			Size	
Read	Write	Erase	Block	Page
30µs	300µs	2,500µs	128KB (= 64 pages)	2KB

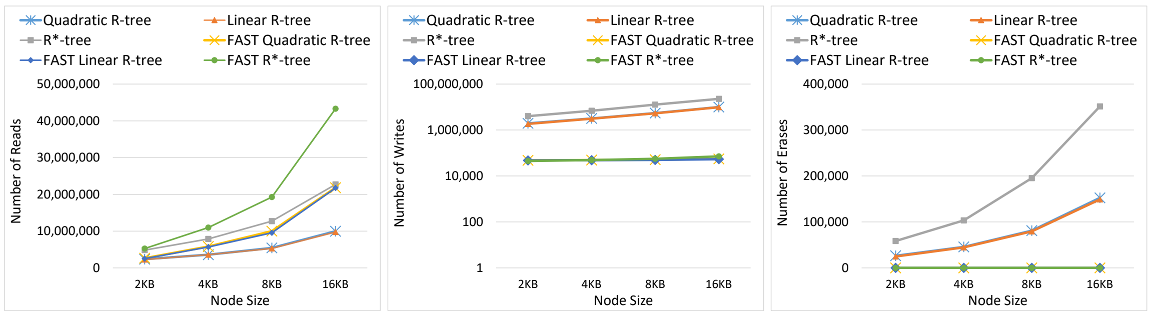
4.2. Execution on the Flash Simulator

4.2.1. Index Construction

Figure 1 depicts the results obtained for building the spatial indices in the flash simulator. Clearly, the FAST-based spatial indices required far fewer writes than disk-based spatial indices for all the page sizes. The reduction of the number of writes varied from 98% to 99%. This expressive result led to the non-use of erase operations. This is due to the buffer management of the FAST-based spatial indices, which reduced the number of writes compared to the traditional LRU buffer. On the other hand, the number of reads required by the FAST-based spatial indices was higher than the reads required by disk-based spatial indices. The reason is that the FAST buffer is only dedicated to storing the modifications of the index and is not used for reading purposes.

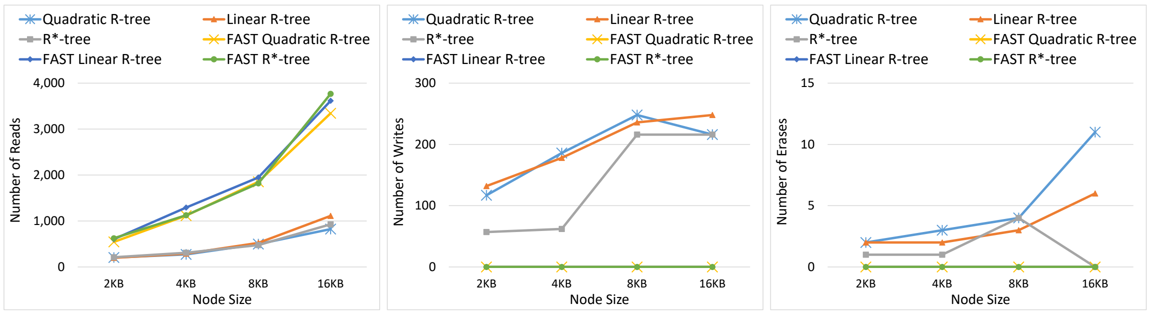
In general, the construction of the indices by using the page sizes equal to 2KB and 4KB required less operations than using other page sizes. The page size equal to 16KB provided the worst results since the index nodes have to be split into several pages of the flash simulator. In this case, each node of the index is stored on 8 pages of the flash simulator. In addition, because of the R*-tree reinsertion policy, the R*-tree generated the highest number of reads, writes, and erases.

[4]Values extracted from `https://www.digchip.com/datasheets/parts/datasheet/710/TC58NYG2S3ETA00.php`



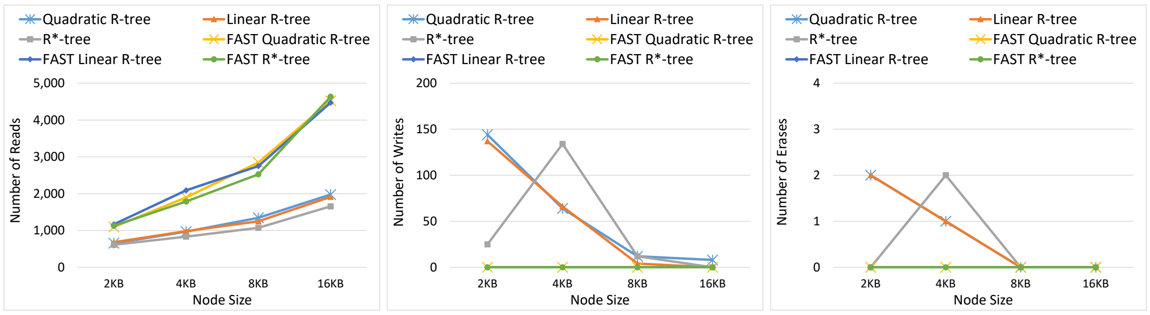
(a) reads (b) writes (c) erases

Figure 1. Number of reads (a), writes (b), and erases (c) for building spatial indices in the flash simulator.



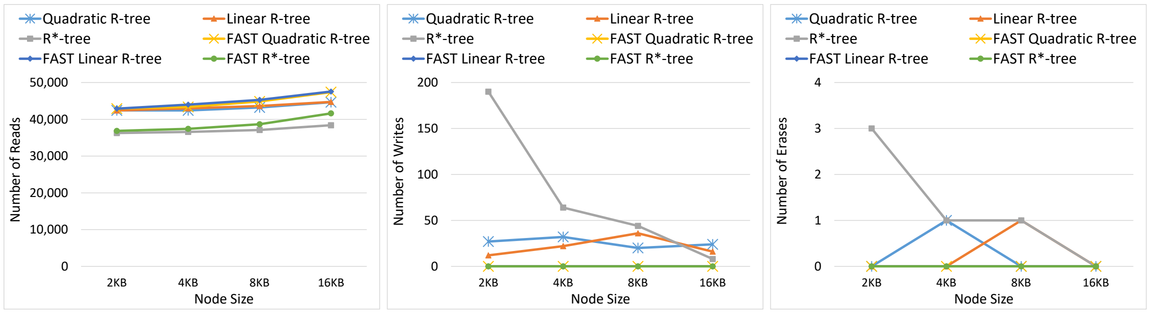
reads writes erases

(a) query windows with 0.001%



reads writes erases

(b) query windows with 0.01%



reads writes erases

(c) query windows with 0.1%

Figure 2. Results obtained for executing the IRQs in the flash simulator.

4.2.2. Spatial Query Processing

Figure 2 depicts the results obtained for processing of IRQs in the flash simulator. In most of the cases, the LRU buffer employed by the disk-based spatial indices required writes because of the LRU buffer that contained some modifications (from the index construction) to be applied. Hence, it also occasionally caused erases. The number of writes and erases decreased as the IRQs were executed because of the sequence of execution of workloads. On the other hand, the FAST-based spatial indices did not perform writes or erases because the employed buffer is not used for caching nodes without modifications. Thus, the elements stored in the buffer were not replaced.

With respect to query windows with 0.001% (Figure 2a), all the configurations showed the best results by using the page size equal to 2KB and increased the number of reads as the used node size also increased. The FAST-based spatial indices required much more reads than the corresponding disk-based spatial indices. This is a consequence of

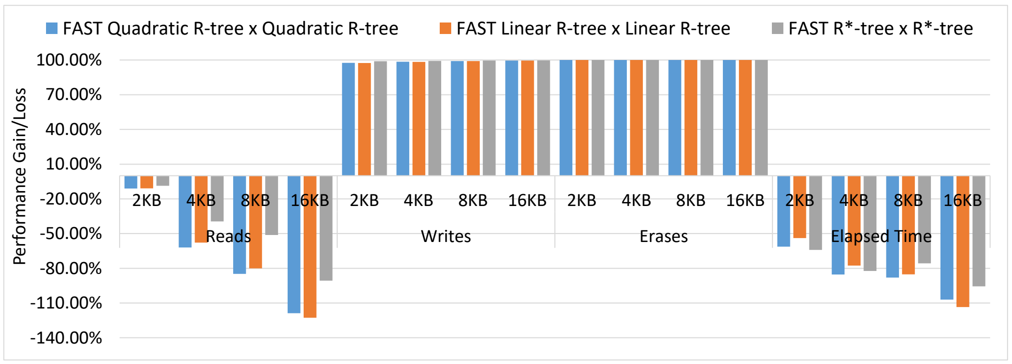


Figure 3. Correlating the performance gains and losses of the FAST-based spatial indices over the corresponding disk-based spatial indices for building indices.

the buffer strategy employed by FAST, which only avoids a read from the storage device when the node to be retrieved is entirely stored in the buffer. The best performance results were obtained by using the Linear R-tree with the page size equal to 2KB.

With regard to query windows with 0.01% (Figure 2b), the number of writes and erases performed by the disk-based spatial indices decreased, compared to the query windows with 0.001%. Again, the FAST-based spatial indices required much more reads than the other configurations. In these IRQs, the best performance results were obtained by using the R*-tree with the page size equal to 2KB.

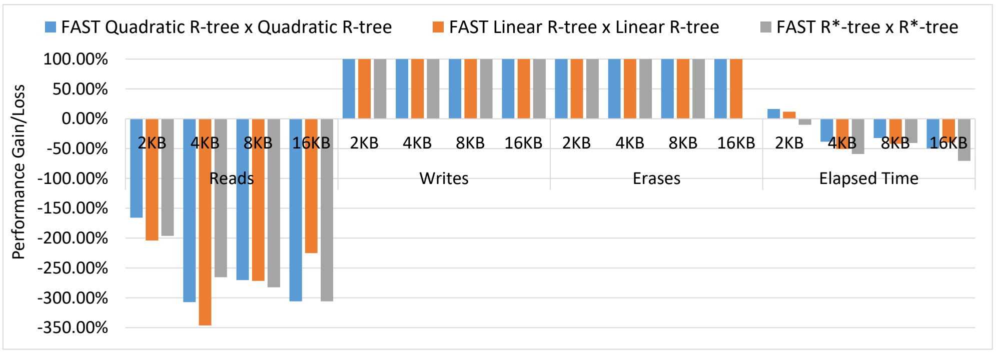
With respect to query windows with 0.1% (Figure 2c), the FAST-based spatial indices decreased the overhead of the number of reads compared to the execution of the previous query windows. In addition, the impact related to the size of nodes employed in the index was minimized, if compared to the execution of the query windows with 0.001%. Due to the high selectivity of the query windows with 0.1%, more objects have to be loaded in the main memory to process the topological predicates. Consequently, if a node has a greater capacity, the index will require fewer reads from the storage device. The best performance results were obtained by using the R*-tree using the page size equal to 2KB, which generated 4,754 nodes of which 1.07% of them were accessed.

5. The Performance Relation between the Flash Simulator and the Real SSD

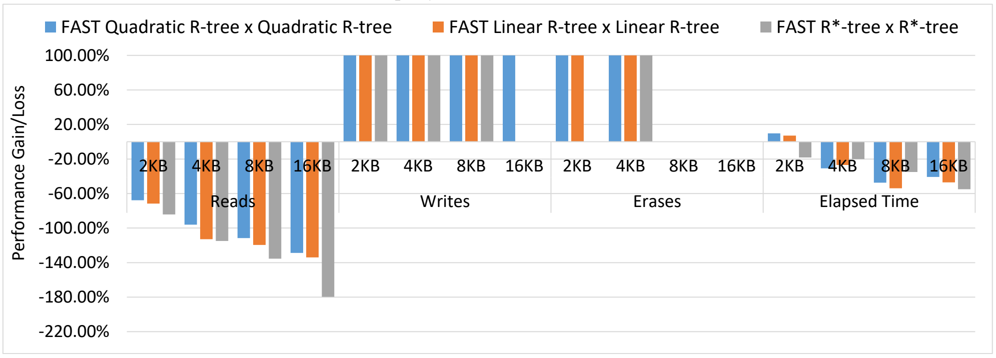
To answer the questions in Section 1, we compare the results obtained in the flash simulator (Section 4.2) and in the SSD. Figures 3 and 4 depict the performance gains and losses of the FAST-based spatial indices over the corresponding disk-spatial indices (e.g., the FAST R*-tree and the R*-tree) for constructing indices and for processing spatial queries, respectively. A performance gain shows as a percentage value how much a configuration was better compared to another configuration. Conversely, a performance loss shows how much a configuration increased a compared value in relation to another configuration. Note that the percentage values of the number of reads, writes, and erases were calculated from the results of the flash simulator (Section 4.2), while the elapsed time percentage values were obtained from the results of the SSD.

In the following, we answer each question of Section 1 by firstly providing an analysis to serve as a foundation.

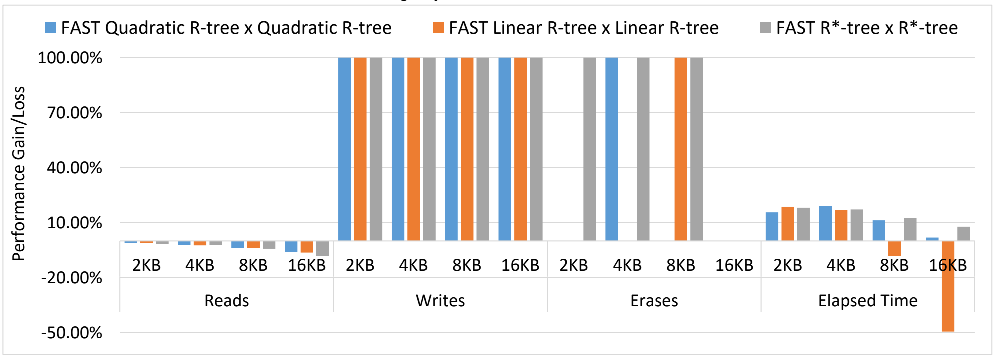
Analysis to Answer Question 1. For the index construction (Figure 3), in spite of the



(a) query windows with 0.001%



(b) query windows with 0.01%



(c) query windows with 0.1%

Figure 4. Correlating the performance gains and losses of the FAST-based spatial indices over the disk-based spatial indices for processing the IRQs.

expressively positive results in the flash simulator, we did not obtain performance gains by using the FAST-based spatial indices in the SSD. In this case, the collected elapsed times only reported performance losses. The main reason was that these indices reduced the number of writes and erases, but at the same time increased the number of reads. In fact, the number of reads doubled for the node size equal to 16KB.

For the spatial query processing (Figure 4), the performance of the disk-based spatial indices was impacted by the writes performed in the SSD. Thus, although the FAST-based spatial indices have required more reads, it showed reductions in the processing time of IRQs in some cases. These reductions were more frequent to process the query windows with 0.1%, which showed the lower losses in the number of reads. In this case, the flash simulator was capable of determining that the FAST-based spatial indices would have a better performance compared to their corresponding disk-based indices.

Table 3. Best configurations obtained in the environments. In parentheses is showed the node size used in the configuration. The cells with '-' means that more than five configurations showed the best results.

		Flash Simulator		SSD
Workload	**# of Reads**	**# of Writes**	**# of Erases**	**Elapsed Time**
Index Construction	Linear R-tree (2KB)	FAST R*-tree (2KB)	FAST-based indices	Linear R-tree (4KB)
Query Windows with 0.001%	Linear R-tree (2KB)	FAST-based indices	FAST-based indices	R*-tree (4KB)
Query Windows with 0.01%	R*-tree (2KB)	-	-	Linear R-tree (8KB)
Query Windows with 0.1%	R*-tree (2KB)	FAST-based indices	-	Linear R-tree (16KB)

Answer of Question 1. For the index construction, the answer is yes. Our experiments showed that when the number of reads in the flash simulator increased by more than 8%, a flash-aware spatial index tends to be inefficient in the SSD. For the spatial query processing, the answer is yes for the most of the cases. Our experiments showed that often when there is a low performance loss in the number of reads, the simulator is capable of determining that the spatial index will provide a good performance on an SSD.

Analysis to Answer Question 2. We use Table 3 to show the best configurations by comparing the number of reads, writes, erases obtained in the flash simulator and the elapsed time to process the workload in the SSD. For the index construction, our experiments showed that the reduction of writes and erases was not enough to provide the best performance results in the SSD. For the spatial query processing, we gathered different best configurations in the environments. But, we can note that the reduction of writes and erases associated to a small increase of required reads in the flash simulator was enough to guarantee the performance gains for the FAST-based indices in the SSD (Figure 2).

Answer of Question 2. In this case, the best configurations were not exactly the same in the environments. But, the flash simulator was able to indicate the following relation in the index construction. The configuration that generated the smallest number of reads showed also the best performance results in the SSD (e.g., the Linear R-tree, in spite of the difference in the used node size).

Analysis to Answer Questions 3 and 4. Our analysis is based on the spatial query processing since the spatial index should be constructed beforehand. For the IRQs with low selectivity (i.e., query windows with 0.001%), we did not need to evaluate the configurations using the node size equal to 8KB and 16KB. The main reason is that the use of smaller page sizes required the processing of fewer entries in the main memory, reducing the number of writes, reads, and erases. Consequently, it also reduced the processing time in the SSD. On the other hand, to efficiently process the IRQs with medium and high selectivity, the usage of small page sizes did not guarantee the better performance in the SSD. However, the flash simulator did not show this behavior.

Answers of Questions 3 and 4. The flash simulator showed to be applicable. Our experiments indicated that we could exclude several configurations to be evaluated in the SSD based on results obtained in the flash simulator.

In addition, we are able to avoid more configurations if an analysis has as goal to discover the best possible configurations to be employed on an SSD. Based on the number of reads obtained in the flash simulator, we can exclude configurations with the highest number of reads. For instance, we would avoid evaluating the FAST-based spatial indices since they failed to reduce the number of reads compared to the disk-based spatial indices. Moreover, if the index construction is a critical step in the goal of the analysis, the flash simulator showed even more useful to exclude configurations to be evaluated in the SSD.

6. Conclusions and Future Work

This paper provides an extensive experimental evaluation in order to analyze the use of a flash simulator as a first step in the performance evaluation of spatial indices in flash memories. We considered the disk-based spatial indices the R-tree and the R*-tree because of their positive characteristics reported in the literature. In addition, we also considered their flash-aware versions, that is, the FAST R-tree and the FAST R*-tree.

As main conclusions, we can cite the following performance behaviors that lead us to answer the most important question of this paper. That is, the question that studies the applicability of a flash simulator as a filter of configurations to be evaluated in an SSD. Firstly, for building spatial indices, our experiments showed that the flash simulator is capable of avoiding configurations to be executed in the SSD based on the number of reads, writes, and erases. The indices that showed the greater number of operations in the simulator also showed poor performance in the SSD. Secondly, for spatial query processing, the number of reads obtained in the flash simulator can be used as a basis to exclude configurations with the highest number of reads. The reason is that these configurations showed the worst performance results in the SSD. We can conclude that a flash simulator is an interesting filter to determine configurations to be evaluated in an SSD. Consequently, the time required to perform empirical analysis can be decreased.

Future work will deal with the execution of other workloads mixing insertions, updates, and queries. In addition, our plan is to conduct the same evaluation in other flash simulators to compare the accuracy among them, as well as take into account other spatial indices, such as the Hilbert R-tree [Gaede and Günther 1998] and eFIND-based spatial indices [Carniel et al. 2017].

Acknowledgments. This work has been supported by CAPES, CNPq, and FAPESP. The first author has been supported by the grant #2015/26687-8, FAPESP. The second and third authors have been supported by the PUB/USP. The fourth author has been supported by the grant #311868/2015-0, CNPq.

References

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

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. (2016a). Experimental evaluation of spatial indices with FESTIval. In *Brazilian Symp. on Databases - Demos*, pages 123–128.

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

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

Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. (2009). A survey of flash translation layer. *Journal of Systems Architecture: the EUROMICRO Journal*, 55(5-6):332–343.

Dong, X., Xu, C., Xie, Y., and Jouppi, N. (2012). NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007.

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

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

Güting, R. H. (1994). An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399.

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

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

Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B. (2009). FlashSim: A simulator for NAND flash-based solid-state drives. In *Int. Conf. on Advances in System Simulation*, pages 125–131.

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

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

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

Su, X., Jin, P., Xiang, X., Cui, K., and Yue, L. (2009). Flash-DBSim: A simulation tool for evaluating flash-based database algorithms. In *IEEE Int. Conf. on Computer Science and Information Technology*, pages 185–189.

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