The Performance Relation of Spatial Indexing on Hard Disk Drives and Solid State Drives

Anderson Chaves Carniel[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@gmail.com, cdac@icmc.usp.br

[2]Department of Computer Science – Federal University of São Carlos
15.565-905 – São Carlos – SP – Brazil

ricardo@dc.ufscar.br

Abstract. *Spatial indexing is a core aspect in spatial databases and Geographic Information Systems. Commonly, spatial indices like the R-tree and the R*-tree consider Hard Disk Drives (HDDs) as the main storage device in data management. On the other hand, flash memories in the form of Solid State Drives (SSDs) have widely been adopted in data servers. Due to their unique characteristics like the erase-before-update operation and the asymmetry between read and write costs, the impact of spatial indexing on SSDs needs to be studied. In this paper, we conduct an experimental evaluation in order to analyze the performance relation of spatial indexing on HDDs and SSDs. As a result, we show experimentally that spatial indices originally designed for HDDs should be redesigned for SSDs in order to take into account the unique characteristics of SSDs. We also propose guidelines to improve the performance of spatial indexing on SSDs by considering these characteristics.*

1. Introduction

Several advanced applications like agriculture systems, urban planning, and public transportation planning make use of geometric or spatial information in order to represent spatial phenomena. These applications commonly employ specialized systems to manage, analyze, and store a spatial phenomenon, such as *spatial database systems* and *Geographic Information Systems* (GIS). In order to aid in decision making, *spatial queries* return a set of spatial objects that satisfy some *topological predicate* (e.g., overlap, disjoint, inside) according to a given object [Gaede and Günther 1998]. For instance, a *spatial selection* that finds all rivers intersecting the Sao Paulo state. To speed up the processing of spatial queries, spatial indices are employed, such as the R-tree [Guttman 1984] and the R*-tree [Beckmann et al. 1990].

In general, these indices manage spatial objects stored in *Hard Disk Drives* (HDD), and thus take into account the slow mechanical access and the cost of search and rotational delay of magnetic disks. On the other hand, *flash memories* have widely been utilized in many applications since they are increasingly being used as the main storage device in mobile phones and laptops [Mittal and Vetter 2015, Suzuki and Swanson 2015]. *Solid State Drives* (SSD) are robust forms of flash memories that have also been popular in data centers and data servers. Flash memories have many positive characteristics compared to HDDs, such as (i) smaller size, (ii) lighter weight, (iii) lower power consumption, (iv) better shock resistance, and (v) faster reads and writes.

Although the positive characteristics of flash memories, these memories have other unique characteristics that may affect the performance of many applications [Chen et al. 2009, Mittal and Vetter 2015]. The main unique characteristic is the asymmetry between read and write costs, where a write requires more time and power consumption than a read. Another characteristic is that a random write is slower than a sequential write since the flash memory is block-oriented. Therefore, an evaluation of the performance of disk-based spatial indices (i.e., originally designed for HDDs) on flash memories is needed.

There are few approaches [Emrich et al. 2010, Fevgas and Bozanis 2015] that conduct an experimental evaluation of spatial indexing on flash memories. There are also approaches [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013] that make use of a buffer in the main memory, which stores the modifications of the spatial indices and thus, avoid random writes to the flash memory. When the buffer is full, a flushing operation is performed by applying sequential writes to the flash memory. However, these approaches face several problems, such as the lack of an analysis between the performance relation of spatial indexing on HDDs and flash memories, the conduction of a limited experimental evaluation that do not focus on the spatial query processing, and the lack of analysis of the parameterization impact of spatial indices on flash memories.

The goal of this paper is threefold. The first goal aims to check the relative performance results of an index on a SSD and a HDD, that is, if a spatial index that show the best results on the HDD also shows the best results on the SSD and vice-versa. For this purpose, we conducted an extensive experimental evaluation that varied several parameters of different spatial indices. The second goal aims to verify the impact of parameters on flushing operations. For instance, we analyzed if the size of bytes in a flushing operation (i.e., a flushing unit) has relation with the page size used in a spatial index. Finally, the third goal aims to analyze if the parameterization that could benefit the flushing operation also guarantee a good performance in the spatial query processing.

This paper is organized as follows. Section 2 surveys related work. Section 3 summarizes underlying concepts from spatial indexing. Section 4 briefly describes the unique characteristics of flash memories. Section 5 details the conducted experimental evaluation. Section 6 concludes the paper and presents future work.

2. Related Work

There are few approaches [Emrich et al. 2010, Fevgas and Bozanis 2015] that conduct an experimental evaluation of spatial indexing on flash memories. In addition, there also approaches [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013] that propose new spatial indices for flash memories based on the R-tree. We classify these approaches according to the following characteristics: (i) the analysis of the performance of spatial indexing on HDDs and SSDs, (ii) the variety of spatial indices used in experiments, (iii) the parameters considered in the indices, and (iv) the focus of the experiments.

Regarding the first characteristic, almost all the approaches do not evaluate the performance of spatial indexing on HDDs and SSDs. This kind of evaluation is important to study and check if the performance of well-known indices (e.g., the R-tree and the R*-tree) are the same on HDDs and SSDs due to the different characteristic of these storage systems. Only [Emrich et al. 2010] conducts a performance evaluation considering both storage systems, HDDs and SSDs.

Regarding the second characteristic, the majority of the approaches [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013] employs the R-tree as baseline, while the remaining approaches [Emrich et al. 2010, Fevgas and Bozanis 2015] employ the R*-tree as baseline. However, it is important to consider both the R-tree and the R*-tree in a same experiment due to their good performance reported in the literature. In addition, many approaches [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013, Fevgas and Bozanis 2015] also propose new spatial indices for flash memories. These approaches conduct experimental evaluations to check the performance behavior of their indices. But, the lack of studies about the performance of well-known spatial indices on flash memories can ignore other characteristics that could improve their performance.

Regarding the third characteristic, the main parameter analyzed in the approaches is the buffer size in the main memory. Another parameter analyzed in [Sarwat et al. 2013] is the type of the flushing policy. However, parameterization plays an important role in spatial indexing and the variation of specific parameters of a spatial index could impact its performance (see Section 3). For instance, there is a lack of analysis of the relation between the page size considered in a spatial index on the HDD and SSD.

Regarding the fourth characteristic, the main focus of the experiments conducted in the approaches is insertion and deletion operations. The reason is that the flash memories introduce challenges in the maintenance of a spatial index since a random write is an expensive operation. But, it is also important to examine the spatial query processing since this is a very common operation in spatial databases. As a result, there is a lack of studies verifying the impact of query selectivity. For instance, since the flash memories provide fast reads, an open question is how to adapt the spatial organization to take into account this characteristic.

In this paper we conduct a performance evaluation considering different spatial indices with different parameters on both storage systems, HDDs and SSDs. For this purpose, we consider disk-based spatial indices (e.g., the R-tree) and flash-aware spatial indices (e.g., the FAST), which are summarized in the next section. Further, we analyze the performance results for creating and querying spatial indices.

3. Spatial Indexing

In general, hierarchical structures are employed to index spatial objects by using their Minimum Boundary Rectangles (MBR). The most known spatial index is the R-tree [Guttman 1984], which is composed of internal and leaf nodes where indexed spatial objects are stored in leaf nodes. The R*-tree [Beckmann et al. 1990] is a well-known variant of the R-tree that employs other aspects to organize the spatial objects in the nodes, such as the overlapping area among the entries, redistribution to maximize the storage utilization, and the margin of the nodes. Hence, the R*-tree modifies the insert algorithm of the R-tree in order to consider these aspects. Further, it applies a policy of reinsertion

Table 1. Comparison of the HDD[1] and the SSD[2] used in the experiments.

	4KB Random Transfers[3]		Power Consumption[4]		Endurance[5]
	Read	Write	Read	Write	
HDD	0.185MB/s	0.441MB/s	4.1W	4.1W	$> 10^{15}$
SSD	285.156MB/s	109.375MB/s	1.423W	2.052W	$10^4 - 10^5$

in order to decrease the number of split operations. Since these indices consider MBRs, a spatial query is composed of the filter and refinement steps [Gaede and Günther 1998].

With the increasing use of flash memories in applications, new spatial indices for flash memories were proposed [Wu et al. 2003, Lv et al. 2011, Sarwat et al. 2013, Fevgas and Bozanis 2015]. Despite the particular characteristic of each index, in general they make use of a buffer in the main memory that stores the most recent modifications of nodes of a spatial index instead of applying directly them to the flash memory. The goal is to avoid random writes, such as those that occur in splits. A *flushing operation* composed of sequential writes is performed when the buffer is full. Almost all the indices consider all modifications in the flushing operation, which can introduce overhead in write operations. On the other hand, [Sarwat et al. 2013] uses a refined *flushing policy* that chooses a set of nodes with modifications to be flushed. This set form a *flushing unit*, which has a fixed number of nodes to be written.

Parameterization plays an important role in spatial indexing [Gaede and Günther 1998]. Page (node) size as well as minimum and maximum number of entries of leaf and internal nodes are examples of typical parameters used by hierarchical structures. In addition, each index may include specific parameters according to its design. For instance, we are able to vary the reinsertion percentage of the R*-tree. Another example is to vary the buffer and flushing unit sizes of flash-aware spatial indices, which will impact directly on the performance of flushing operations. Therefore, there is a significant performance impact if *different* parameters are used in a spatial index in *different* datasets under *different* storage systems. We mainly conduct an empirical study regarding to it by analyzing the unique characteristics of flash memories, which are summaried in Section 4.

4. Flash Memories

Flash memories in the form of SSDs have been very popular in many applications [Suzuki and Swanson 2015, Mittal and Vetter 2015]. Table 1 shows a comparison of the SSD and HDD used in our experimental evaluation (Section 5). Flash memories have unique characteristics that need to be taken into account in the development of applications for them. Flash memories are block-oriented. This means that a fixed number of flash pages composes a flash block. Commonly, the flash page and block sizes of a SSD are 4KB and 256KB, respectively [Chen et al. 2009, Mittal and Vetter 2015].

[1]`https://support.wdc.com/product.aspx?ID=608&lang=en`
[2]`https://www.kingston.com/us/ssd/consumer/sv300s3`
[3]Measured by Iometer (`http://www.iometer.org/`).
[4]According to the manufactures[1 2].
[5]According to [Mittal and Vetter 2015].

Flash memories support the following operations: erase, read, and write (program) [Chen et al. 2009]. An erase is a block level operation that changes the bits from 0 to 1 of all pages contained in the block, and thus, this is the most expensive operation. The read and write are page level operations with asymmetric costs. A read requires much less time and power consumption than a write (see Table 1). A write is only able to change the bits from 1 to 0 in an erased block. Hence, a write operation in a previously written block requires an *erase-before-update operation* that leads to a very time-consuming operation. On the other hand, a write operation in a previously erased block is a lower latency operation and is denominated as a *sequential write*. Another important characteristic of flash memories is their lower endurance than HDDs (see Table 1). Endurance refers to the maximum number of writes and erases in a block before its unreliableness.

In order to avoid erase-before-update operations and improve the endurance of flash memories, the Flash Translation Layer (FTL) [Wu et al. 2009, Chung et al. 2009] is employed. For this purpose, FTL provides an interface that allows operation systems to use flash memories as a virtual disk and only enables reads and writes for application layers. Further, FTL maps physical page addresses of a flash memory into logical page addresses, which are effectively used by application layers. A logical page is marked as either free, valid, or invalid. A free logical page is able to store data. A valid logical page contains data previously written. A logical page is marked as invalid if a write is performed on a valid logical page; and its new content is stored in another free logical page. This operation is termed as an *out-of-place update* and avoids an erase-before-update operation. A *garbage collection* is needed when space is required and there is no sufficient free logical pages. This operation selects a set of blocks to apply erase operations causing erase-before-update operations. Hence, this is the most expensive operation performed by the FTL. The algorithms of the garbage collection and out-of-place update also consider a *wear leveling* in order to improve the endurance of flash blocks. For a survey of FTLs, see [Chung et al. 2009].

5. Performance Evaluation

We conduct an experimental evaluation in order to analyze the performance relation between the spatial indexing on HDDs and SSDs. Section 5.1 details the experimental setup used in the experiments, while Section 5.2 discusses the obtained results.

5.1. Experimental Setup

We used a real dataset extracted from the OpenStreetMap[6], which consisted of 534.926 complex regions with holes. This dataset represents the buildings of Brazil, such as hospitals, schools, universities, houses, stadiums, and so on. We used the PostgreSQL database management system with the PostGIS extension to store this dataset in a relational table.

In order to conduct our experiments, we employed FESTIval [Carniel et al. 2016]. FESTIval is a PostgreSQL extension that enables the performance comparison of different spatial indices with different parameters under different storage systems by using a unique environment. We used it to compare the performance of the following spatial indices designed for HDDs: the R-tree and the R*-tree. Further, we implemented the

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

Table 2. Configurations of the spatial indices and their corresponding specific parameters used in the experiments.

Configuration Name	Spatial Index	Specific Parameters
Linear R-tree	R-tree	Split: Linear
Quadratic R-tree	R-tree	Split: Quadratic
R-tree 20%*	R*-tree	RP: 20%
R-tree 30%*	R*-tree	RP: 30%
R-tree 40%*	R*-tree	RP: 40%
FAST Linear R-tree	FAST R-tree	Split: Linear; FP: FAST*
FAST Quadratic R-tree	FAST R-tree	Split: Quadratic; FP: FAST*
FAST R-tree 20%*	FAST R*-tree	RP: 20%; FP: FAST*
FAST R-tree 30%*	FAST R*-tree	RP: 30%; FP: FAST*
FAST R-tree 40%*	FAST R*-tree	RP: 40%; FP: FAST*

Table 3. Generic parameters used in the experiments.

Page Size	Minimum Occupancy	Maximum Occupancy
2KB	28	56
4KB	57	113
8KB	114	227
16KB	228	455
32KB	455	910
64KB	910	1820

FAST [Sarwat et al. 2013], which is an approach that adapts a spatial index to be efficiently used in flash memories. Note that FAST does not change the index structure but only changes the way in which the nodes are written in the flash memory. We applied the FAST to be used together with the R-tree and R*-tree, and thus, we formed the FAST R-tree and the FAST R*-tree, respectively.

FESTIval also enables the configuration of a spatial index by using specific and generic parameters. Specific parameters only determine the configuration of a specific spatial index. For the R-tree, we varied its split algorithm. For the R*-tree, we varied the reinsertion percentage (RP) since it impacts on the number of writes in the structure. Further, based on the recommendations for the R*-tree [Beckmann et al. 1990], we considered the close reinsert and fixed the number of elements to be examined in the insertion algorithm as 32. For the FAST-based indices, we considered the FAST* flushing policy (FP) due to its advantages over other flushing policies [Sarwat et al. 2013]. In addition, we studied the effect of the variation of the size of the buffer and the flushing unit (Section 5.2.1). Based on that, we employed the configurations depicted in Table 2.

We also varied generic parameters, which can be applied for any spatial index. For instance, the page size (i.e., the size of a node) and the minimum and maximum number of entries of a node. Further, we considered the DIRECT I/O to avoid operational system caching in read and write operations. Table 3 shows the generic parameters employed for all the configurations in Table 2.

We executed two workloads defined as follows. The first workload focused on the *index construction*, and thus, we collected the time processing in seconds for creating a spatial index (Section 5.2.1). The creation of a spatial index was performed by inserting element by element according to the original insert algorithm of the respective index.

The second workload focused on the *spatial query processing* (Section 5.2.2). Due to its utilization in many applications, we executed *intersection range queries* (IRQ) [Gaede and Günther 1998]. This kind of query returns from a dataset D, a set of spatial objects R that intersects a given query window QW, i.e., $R = \{o | o \in D \land intersects(o, QW) = true\}$. Here, we employed rectangular-shaped objects as query windows. We synthetically generated three sets of query windows. Each set was composed of 100 query windows, which had a $x\%$ of area of the bounding box of Brazil. The first set had 0.1%, the second set had 0.5%, and the third set had 1%. They correspond to query windows with low, medium, and high selectivities, respectively.

We collected the total elapsed time in seconds taken to execute the 100 IRQs of each set of query windows. The total elapsed time was calculated as follows. For a specific set of query windows, we executed each IRQ 10 times, collected the average elapsed time of the execution, and then calculated the sum of the average elapsed times of the 100 IRQs. We performed the tests locally to avoid network latency and flushed the system cache after the execution of each IRQ.

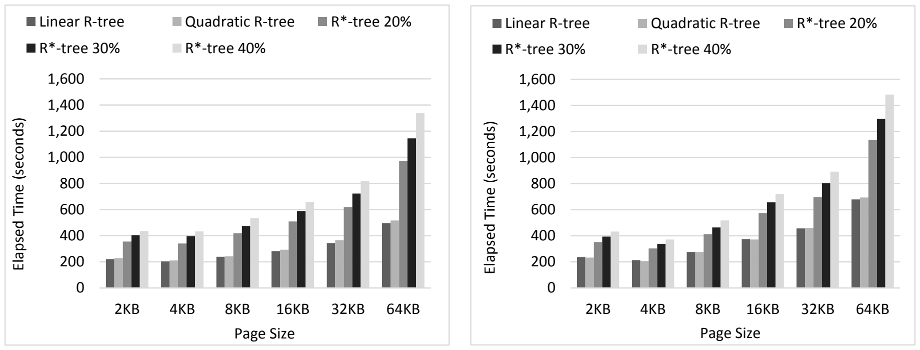
The experiments were conducted on a computer with an Intel® Core™ i7-4770 with frequency of 3.40GHz and 32GB of main memory. For the experiments with HDD we used a 2TB Western Digital with 7200RPM, while for the experiments with SSD we used a 480GB Kingston V300. We employed the Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, PostGIS 2.2.0, and GEOS 3.5.2. GEOS is used by PostGIS for the computation of topological predicates.

5.2. Performance Results

The obtained results related to the spatial index construction and spatial query processing are discussed in Sections 5.2.1 and 5.2.2, respectively.

5.2.1. Spatial Index Construction

Figure 1 depicts the elapsed time for creating disk-based spatial indices according to Tables 2 and 3. We obtained the best performance results for all spatial indices on both storage systems by using the page size equal to 4KB, indicating that this page size should be used for creating spatial indices. Considering the page size equal to 4KB, the performance gain of the indices on the SSD overcame the HDD in at most 14.45% since 4KB is the page size of the SSD. A performance gain is the percentage that shows how much one configuration is more efficient than another configuration. On the other hand, for other page sizes (i.e., 2KB, 8KB, 16KB, 32KB, and 64KB) we obtained best performance results on the HDD with performance gains between 1.88% and 27.05%, which increased as the page size also increased. This case happens since an index construction mix many random writes and reads, and thus, it can degenerate the performance on flash memories (also discussed in [Lee and Moon 2007] and [Chen et al. 2009]).



(performed on the HDD) (performed on the SSD)

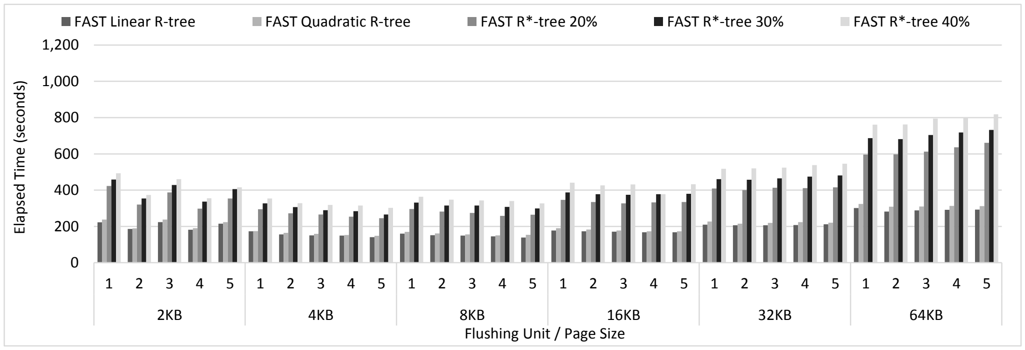
Figure 1. Performance results for creating disk-based spatial indices.

The results demonstrated that spatial indices originally designed for HDD should be redesigned for flash memories in order to take into account the unique characteristics of these memories. FAST is an approach that adapts disk-based spatial indices to be used efficiently on flash memories. In order to evaluate the performance of FAST-based spatial indices, we varied the flushing unit size from 1 to 5. For instance, for configurations with page size equal to 16KB, the flushing unit size equal to 3 performs writes of 48KB in each flushing operation. The flushing unit sizes were combined with each page size, which allowed to verify the impact of this parameter on the index construction. We further considered three different buffer sizes: 128KB, 256KB, and 512KB. In Figure 2, we only report the performance results for creating FAST-based spatial indices by using the buffer size equal to 512KB since it had the same behavior than the other buffer sizes.

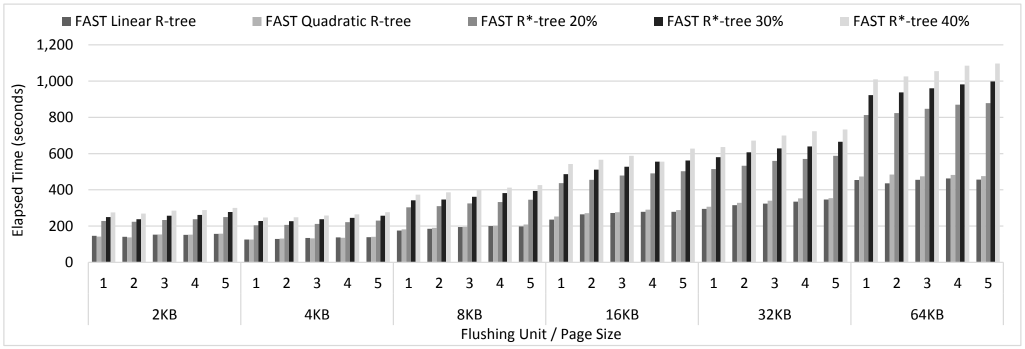
The performance results showed that FAST-based indices improved the performance of their counterparts on both storage systems. For instance, the *FAST R*-tree 30%* (Figure 2) configuration with 4KB performed on the HDD imposed a performance gain from 17.23% to 32.84% compared to the *R*-tree 30%* with 4KB (Figure 1) on the HDD. This also occurred for the SSD, where the performance gains were yet more expressive, varying from 23.85% to 32.86%. Further, we clearly note that we did not obtain the same performance behavior on the storage systems for constructing FAST-based indices. We emphasize two main differences. The first difference was that the spatial indices showed best performance results on the HDD by utilizing the page size equal to 4KB and the flushing unit size equal to 5. On the other hand, the best performance results on the SSD utilized the page size equal to 4KB and the flushing unit equal to 1. The second difference was that with the increase of page and flushing unit sizes in the index construction on the SSD, the time processing also increased. This was even much slower than the construction performed on the HDD. For instance, for the page size equal to 64KB, the results demonstrated that the performance gains on the HDD were higher than the SSD because big writes turned out problematic on the SSD.

5.2.2. Spatial Query Processing

Figures 3, 4, and 5 depict the obtained results for processing spatial queries by employing IRQs with 0.1%, 0.5%, and 1%, respectively. Note that we use a different scale to report



(Performed on the HDD considering buffer equal to 512KB)



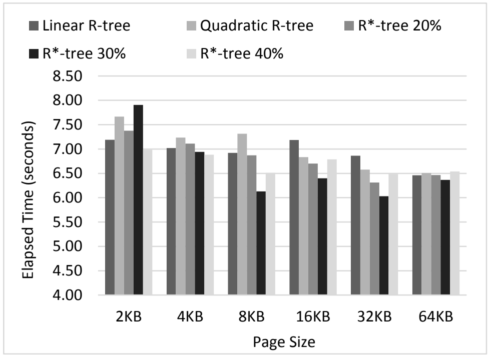
(Performed on the SSD considering buffer equal to 512KB)

Figure 2. Performance results for creating FAST-based spatial indices.

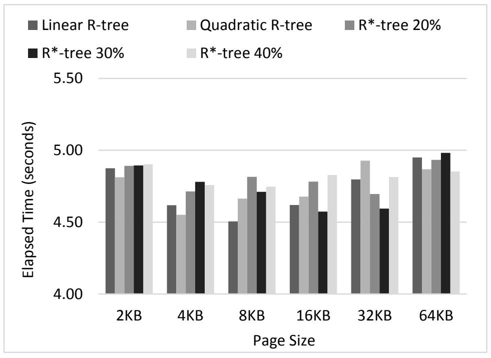
the results for the SSD in order to better compare the performance results. We considered only configurations based on the R-tree and the R*-tree (Table 2) since the FAST does not change the structure of a spatial index (e.g., the *Linear R-tree* using 4KB as page size has the same structure than the *FAST Linear R-tree* using 4KB as page size). Clearly, the performance of spatial query processing on the SSD overcame the HDD in all experiments because of its faster random reads.

With respect to the execution on the HDD, almost all the configurations improved their performance by increasing page sizes. The reason is that if we use large page sizes, we are able to retrieve more elements from the disk with few reads. The best results were obtained by using the *R*-tree 30%* configuration. For the query windows with 0.1% (Figure 3), the page size equal to 32KB provided the best results since the number of performed reads in the tree required for answering the queries was lower than the other IRQs due to the low selectivity. Hence, for the IRQs with medium and high selectivity (Figures 4 and 5), the use of the page size equal to 64KB improved the elapsed time.

With respect to the execution on the SSD, the performance behavior was slightly different than the HDD. For the IRQs with low selectivity (Figure 3), we obtained the best result by using the *Linear R-tree* configuration with the page size equal to 8KB. For this kind of selectivity, the number of traversed paths on the index to answer the query impacted on the elapsed time. It led to perform sequential reads, which is a very low latency operation [Chen et al. 2009]. Further, for the majority of the cases, the performance

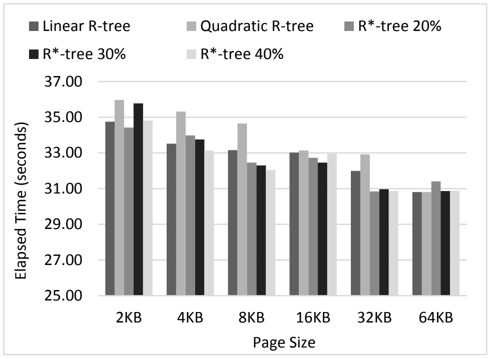


(performed on the HDD)

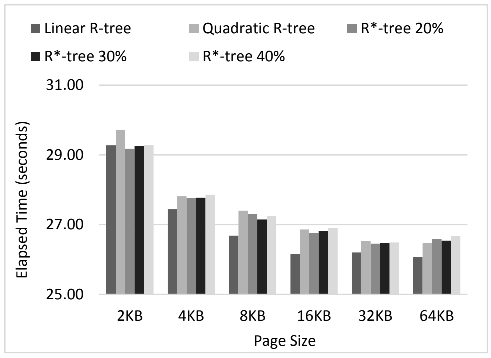


(performed on the SSD)

Figure 3. Performance results of spatial queries considering IRQs with 0.1%.



(performed on the HDD)



(performed on the SSD)

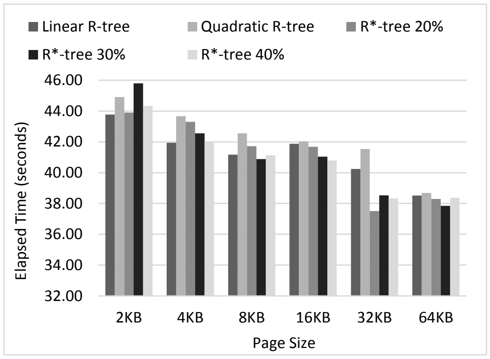
Figure 4. Performance results of spatial queries considering IRQs with 0.5%.

deteriorated as the page size increased. On the other hand, by increasing the page size for medium and high selectivities (Figures 4 and 5), we guaranteed more efficient time processing. This is due to the fact that small page sizes introduce much more random reads.

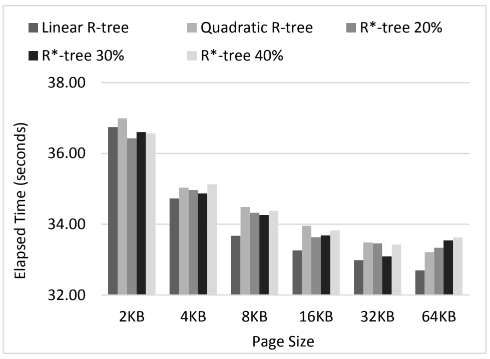
The results of this experiment demonstrated that the spatial organization for an efficient spatial query processing on SSDs tends to be different than on HDDs. A main finding is that we can exploit the good performance of random reads by using larger page sizes for higher selectivities. Conversely, we can use smaller page sizes for spatial query processing with lower selectivities.

6. Conclusions and Future Work

In this paper, we conducted an extensive experimental evaluation to check the performance relation of spatial indexing on HDDs and SSDs. We considered the disk-based spatial indices R-tree and R*-tree due to their positive characteristics known in the literature. We also considered flash-aware spatial indices based on the FAST due to its positive features, such as the support of flushing policies and flushing units. For all these indices, we varied several parameters and as a result, at least 180 distinct configurations of spatial indices were analyzed in our experiments.



(performed on the HDD)



(performed on the SSD)

Figure 5. Performance results of spatial queries considering IRQs with 1%.

As main conclusions we can cite the following performance behaviors, which can lead us to a set of guidelines to further improve the performance of spatial indexing on SSDs. Firstly, in all experiments we see that the direct employment of disk-based spatial indices that showed good performance results on HDDs did not guarantee the best performance results on SSDs. For instance, the R-trees provided better performance on the SSD in the spatial query processing than the R*-trees; but the R*-trees led to best performances on HDDs. This means that the spatial indexing should consider other aspects in order to explore the positive characteristics of flash memories. Based on that, the experiments showed that FAST-based indices improved the elapsed time for the creation of spatial indices on the SSD and even on the HDD since it uses a buffer in the main memory. In general, the performance evaluation showed that employing the FAST for the R-tree with page sizes varying from 4KB to 16KB did not require much time for creating the indices and provided a good performance in the spatial query processing. While the use of the page size equal to 4KB should be recommended for queries with low selectivity, the page size equal to 16KB should be recommended for queries with higher selectivities.

Future work will consider new workloads by including insertions, deletions, and updates of spatial objects in order to analyze the performance behavior for maintaining spatial indices. We will also extend the experiments by considering other spatial indices, like the Hilbert R-tree [Kamel and Faloutsos 1994].

Acknowledgments

This work has been supported by the Brazilian research agencies CAPES, CNPq, and São Paulo Research Foundation (FAPESP). The first author has been supported by the grant #2015/26687-8, FAPESP. The second author has been supported by the grant #311868/2015-0, CNPq.

References

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331.

Carniel, A. C., Ciferri, R. R., and Ciferri, C. D. A. (2016). Experimental evaluation of spatial indices with FESTIval. In *Proceedings of the Brazilian Symposium on Databases - Demonstration Track*, pages 123–128.

Chen, F., Koufaty, D. A., and Zhang, X. (2009). Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *SIGMETRICS Perform. Eval. Rev.*, 37(1):181–192.

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.

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.

Fevgas, A. and Bozanis, P. (2015). Grid-file: Towards to a flash efficient multidimensional index. In *Proceedings of the International Conference on Database and Expert Systems Applications*, pages 285–294. Springer International Publishing.

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

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. *SIGMOD Record*, 14(2):47–57.

Kamel, I. and Faloutsos, C. (1994). Hilbert R-tree: An improved R-tree using fractals. In *Int. Conf. on Very Large Data Bases*, pages 500–509.

Lee, S.-W. and Moon, B. (2007). Design of flash-based DBMS: An in-page logging approach. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 55–66.

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. (2015). A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. on Parallel and Distributed Systems*, PP(99):1–14.

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.

Suzuki, K. and Swanson, S. (2015). A survey of trends in non-volatile memory technologies: 2000-2014. In *IEEE Int. Memory Workshop*, pages 1–4.

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.

Wu, P.-L., Chang, Y.-H., and Kuo, T.-W. (2009). A file-system-aware FTL design for flash-memory storage systems. In *Conf. on Design, Automation and Test in Europe*, pages 393–398.