UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Spatial indexing on flash-based solid state drives

Anderson Chaves Carniel

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Anderson Chaves Carniel

Spatial indexing on flash-based solid state drives

Thesis submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Cristina Dutra de Aguiar Ciferri

USP – São Carlos
February 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Carniel, Anderson Chaves
C289s Spatial indexing on flash-based solid state drives / Anderson Chaves Carniel; orientadora Cristina Dutra de Aguiar Ciferri. -- São Carlos, 2018.
174 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2018.

1. spatial database systems. 2. spatial indexing. 3. spatial access methods. 4. flash memory. 5. flash-aware spatial index. I. Ciferri, Cristina Dutra de Aguiar, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Anderson Chaves Carniel

Indexação espacial em dispositivos de estado sólido baseados em memória *flash*

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Cristina Dutra de Aguiar Ciferri

USP – São Carlos
Fevereiro de 2019

Este trabalho é dedicado aos meus pais,
Gerusa e Guilherme,
que sempre me apoiaram para chegar a esse momento.

ACKNOWLEDGEMENTS

My first thanks go to my parents, Gerusa and Guilherme, who always provided me the needed support for concluding each step of my life.

I am very thankful to the special support of Daniele, or for me, *Dany*. She always kindly understand when I needed to work on this thesis and helped to achieve my goals.

I would like also to thank the support of my supervisor, Prof. Dr. Cristina Ciferri. She always gives me insights and advices that certainly I will apply to not only my career, but also to the rest of my life. This special thanks also applies to the supervisor of my Master's work, Prof. Dr. Ricardo Ciferri.

I also need to thank the ideas, support, and discussions of other researchers that I met during the development of this PhD work. Among them, I appreciate the advices of Prof. Dr. Markus Schneider.

I would like to thank the supervision of Prof. Dr. Michael Vassilakopoulos during the research internship in Greece. He and his wife have kindly received me during July 2018. In addition, I also thank the receptiveness of George Roumelis, who was also involved in the research internship. Finally, I thank the initial collaboration started with Prof. Dr. Antonio Corral.

With respect to financial support of this PhD, these are my acknowledgments.

This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* - Brazil (CAPES) - Finance Code 001. This work has also been supported by the Brazilian federal research agency CNPq.

I have been supported by the grant #2015/26687-8, São Paulo Research Foundation (FAPESP). This grant is related to the research project of this PhD work and conducted at University of São Paulo, Brazil, entitled "Spatial Indexing in Non-Volatile Memories: Proposal of an Efficient and Robust Spatial Index with Durability".

I have also been supported by the grant #2018/10687-7, São Paulo Research Foundation (FAPESP), from July 1st to July 31st in 2018. This grant is related to the research project entitled "Porting Disk-based Spatial Indices to Flash-based Solid State Drives", conducted at University of Thessaly, Greece.

This work was also supported by the Microsoft Azure Sponsorship from the Microsoft Research. This support allowed us to use the Microsoft Azure in our experiments.

This work was also supported by the *Programa Unificado de Bolsas* of the University of São Paulo. This program allowed to me supervise undergraduate students.

“You, me, or nobody is gonna hit as hard as life. But it ain’t about how hard you hit. It’s about how hard you can get hit and keep movin’ forward. How much you can take and keep movin’ forward. That’s how winnin’ is done!”

(Rocky Balboa - the movie)

RESUMO

CARNIEL, A. C. **Indexação espacial em dispositivos de estado sólido baseados em memória *flash***. 2019. 174 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Sistemas de banco de dados espaciais empregam *estruturas de indexação espaciais* para acelerar o processamento de consultas espaciais. Muitos dos índices espaciais propostos na literatura, como a *R-tree*, assumem que os dispositivos de armazenamentos são os discos magnéticos (i.e., HDDs) e são denominados *índices espaciais baseados em disco*. Por outro lado, várias aplicações de banco de dados espaciais estão cada vez mais usando *Solid State Drives* (SSDs) baseados em memória *flash* e, assim, projetar índices espaciais para esses dispositivos tem ganhado cada vez mais atenção. Isso se deve ao fato de que, em comparação com os HDDs, os SSDs oferecem menor tamanho, menor peso, menor consumo de energia, melhor resistência a choques além de leituras e escritas mais rápidas. Assim, *índices espaciais para memória flash* têm sido propostos na literatura para lidar com as características intrínsecas dos SSDs, como os custos assimétricos de leituras e escritas. No entanto, a pesquisa até o momento não conseguiu estabelecer um índice espacial que realmente explora todos os benefícios dos SSDs. Esta tese de doutorado avança na literatura da seguinte forma. Primeiramente, é definida uma metodologia para criar conjuntos de dados espaciais para avaliações experimentais. Também é proposto *FESTIval*, um arcabouço versátil que fornece um ambiente comum e único para executar avaliações experimentais. Tais contribuições serviram como base para conduzir análises de desempenho ao longo deste trabalho de doutorado. Usando essa base, o comportamento de desempenho de índices espaciais em diferentes dispositivos de armazenamento, como HDDs e SSDs, é analisado. Além disso, discute-se a aplicabilidade de simuladores *flash* na avaliação experimental de índices espaciais. Os resultados desses experimentos contribuíram para a proposta de eFIND, uma estrutura genérica e eficiente para indexação espacial em memórias *flash*. eFIND é genérico porque pode portar uma ampla gama de índices espaciais baseados em disco para SSDs. eFIND também é eficiente porque é baseado em um conjunto de objetivos de projeto que exploram o desempenho do SSD. Os testes de desempenho mostraram que, em comparação com o estado da arte, eFIND melhorou a construção de índices espaciais portados e a execução de consultas espaciais. Para portar a *R-tree* (ou seja, a *eFIND R-tree*), eFIND mostrou melhorias de desempenho de 43% a 77% para construir índices espaciais e de 4% a 23% para executar consultas espaciais. Para portar a xBR$^+$-tree (ou seja, a *eFIND xBR$^+$-tree*), eFIND mostrou melhorias de 28% a 83% para construir índices espaciais e de até 35% no processamento de consultas espaciais.

Palavras-chave: sistemas de banco de dados espaciais, indexação espacial, métodos de acesso espaciais, memória flash, índice espacial para memória flash.

ABSTRACT

CARNIEL, A. C. **Spatial indexing on flash-based solid state drives**. 2019. 174 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Spatial database systems widely employ *spatial indexing structures* to speed up the processing of spatial queries. Many of the proposed spatial indices in the literature, such as the R-tree, assume magnetic disks (i.e., HDDs) as the underlying storage device. They are termed as *disk-based spatial indices*. On the other hand, several spatial database applications are increasingly using *flash-based Solid State Drives* (SSDs) and thus, designing spatial indices for these storage devices has gained increasing attention. This is due the fact that, compared to HDDs, SSDs offer smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes. Hence, specific indices for SSDs, termed *flash-aware spatial indices*, have been proposed in the literature to deal with 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. This PhD thesis advances on the literature as follows. We firstly define a methodology to create spatial datasets for experimental evaluations. We also propose FESTIval, a versatile framework that provides a common and unique environment to execute experimental evaluations. Such contributions served as a foundation to conduct performance analysis along this PhD work. By using this foundation, we analyze the performance behavior of spatial indices on different storage devices, such as HDDs and SSDs. Further, we discuss the applicability of employing flash simulators on the evaluation of spatial indices. The findings of these experiments contributed to the proposal of eFIND, a generic and efficient framework for flash-aware spatial indexing. eFIND is generic because it can port a wide range of disk-based spatial indices to SSDs. eFIND is also efficient because it is based on a set of design goals that exploits SSD performance. Performance tests showed that, compared to the state of the art, eFIND improved the construction of ported disk-based spatial indices and the execution of spatial queries. For porting the R-tree (i.e., the eFIND R-tree), eFIND showed performance reductions from 43% to 77% to build spatial indices, and from 4% to 23% to execute spatial queries. For porting the xBR$^+$-tree (i.e., the eFIND xBR$^+$-tree), eFIND showed reductions from 28% to 83% to build spatial indices and up to 35% in the spatial query processing.

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

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER

1

INTRODUCTION

This Chapter is organized as follows. Section 1.1 describes the context of this PhD work. Section 1.2 presents the motivation. Section 1.3 introduces the most relevant contributions. Finally, Section 1.4 shows the organization of this PhD thesis.

1.1 Context

Many database applications require the representation, storage, and management of spatial or geographic information to enrich data analysis. *Spatial database systems* and *Geographic Information Systems* (GIS) provide the foundation for these applications (RIGAUX; SCHOLL; VOISARD, 2001). For instance, mobile applications of social media (e.g., Twitter and Facebook) handle massive amounts of geographical data to enhance the user experience (SUI; LD, 2011). Another example is the use of geographic data as the main goal in applications dedicated to optimizing traffic routes (CASTRO *et al.*, 2013). Other research fields employing spatial or geographic data include agricultural applications that provide visualization and management of crops by overlaying them in maps (CANADA, 2018), urban planning applications that manage land resources (BATTY, 1992), and climatological applications that analyze meteorologic and precipitation data of specific global locations (STACKHOUSE *et al.*, 2015).

Spatial database systems and GIS often employ *spatial indexing structures* to speed up the processing of spatial queries (GAEDE; GÜNTHER, 1998; OOSTEROM, 2005). In general, a spatial query returns a set of spatial objects (e.g., point, line, and region objects) that satisfy a given *topological predicate* (e.g., intersects, overlaps, inside, meet) considering a search object. A typical query is the *intersection range query*, which returns all spatial objects that intersect a rectangular-shaped object called *query window*. The goal of using a spatial index is to reduce the processing time of spatial queries by avoiding the access of objects that certainly do not belong to the final answer of the query. Hence, it often reduces the search space of spatial queries and attempts to decrease the number of accesses to the storage device.

Intuitively, a spatial index groups nearest spatial objects in index pages. A common strategy is to organize these index pages in hierarchical (tree) structures; thus, an index page refers to a node in the tree. To this end, two main approaches are employed (GAEDE; GÜNTHER, 1998): (i) *data partitioning*, and (ii) *space partitioning*. Spatial indices based on the first approach organize the hierarchy oriented by the groups formed from the spatial objects; thus, they are termed as *data-driven access methods*. Examples are the well-known R-tree (GUTTMAN, 1984) and its variants, the R*-tree (BECKMANN *et al.*, 1990) and the Hilbert R-tree (KAMEL; FALOUTSOS, 1994). They are widely employed in spatial database systems. Spatial indices belonging to the second approach organize the hierarchy oriented by the division of the space in which the spatial objects are arranged; thus, they are termed as *space-driven access methods*. For instance, Quadtree-based indices (SAMET, 1984) such as the xBR$^+$-tree (ROUMELIS *et al.*, 2015). Such indices are widely employed to index points.

1.2 Motivation

In general, the majority of the spatial indexing structures proposed in the literature assume that the spatial objects are indexed in magnetic disks (i.e., *Hard Disk Drives* - HDDs). This is an implicit situation since HDDs have been very popular storage devices for a long time. Hence, such indices consider the slow mechanical access and the high cost of search and rotational delay of disks in their design. We term spatial indices designed for magnetic disks as *disk-based spatial indices*.

On the other hand, advanced database applications are interested in using modern storage devices like *flash-based Solid State Drives* (SSDs) (KOLTSIDAS; VIGLAS, 2011a; BRAYNER; FILHO, 2016; MITTAL; VETTER, 2016). This includes spatial database systems that employ spatial indices to efficiently retrieve spatial objects stored in SSDs (EMRICH *et al.*, 2010; KOLTSIDAS; VIGLAS, 2011b; CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a). The main reason for this interest is because SSDs, in contrast to HDDs, have smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

In a spatial database context, the replacement of an HDD to an SSD does not mean that spatial applications would inherit all the positive characteristics of SSDs. This is mainly due to the intrinsic characteristics of SSDs (AGRAWAL *et al.*, 2008; BOUGANIM; JÓNSSON; BONNET, 2009; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016), which have introduced a new paradigm in data management. A well-known characteristic is the asymmetric cost of reads and writes, where a write requires more time and power consumption than a read. Further, internally, SSDs are able to write data to empty pages only, which means that updating data in previously written pages requires an erase-before-update operation. Other factors that impact on SSD performance are the processing of

interleaved reads and writes, and the execution of reads on frequent locations. These factors are related to the internal controls of SSDs, such as its internal buffers and the read disturbance management (JUNG; KANDEMIR, 2013).

To deal with the intrinsic characteristics of SSDs, spatial indices specifically designed for SSDs, termed here as *flash-aware spatial indices*, have been proposed in the literature. They attempt to exploit all the benefits of the SSDs by specifying data structures and algorithms that take into account the intrinsic characteristics of SSDs. Unfortunately, current flash-aware spatial indices face several problems. First, the impact of SSDs on the spatial indexing context is understudied. This disallow us to design and develop efficient flash-aware spatial indices. Second, existing flash-aware spatial indices assume that the random read is the fastest operation of SSDs and thus, they execute more reads than writes in order to mitigate the poor performance of random writes. But this behavior degenerates SSD performance (JUNG; KANDEMIR, 2013). Third, there is no special treatment to minimize the effects of interleaved reads and writes, which also negatively impact SSD performance (BOUGANIM; JÓNSSON; BONNET, 2009; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). Finally, existing flash-aware spatial indices handle inefficient in-memory data structures, requiring high elapsed times to retrieve index pages in index operations like insertions and queries.

1.3 Contributions

The interest of the spatial database and GIS communities on using SSDs and the problems faced by related work provide the foundation for this PhD work. We advance on the state of the art by introducing the following contributions.

The definition of a methodology to create spatial datasets for conducting experimental evaluations. In order to understand the impact of SSDs on the spatial indexing context by means of extensive empirical evaluations, we propose two types of spatial datasets to be employed in experiments. The first type refers to the spatial datasets actually manipulated by index operations, such as insertions and queries. To create them, a systematic methodology is defined and employed in order to extract real data from the OpenSteetMaps[1] and then store the extracted spatial objects in relational tables of the PostgreSQL[2]. The second type of spatial datasets refers to the definition of the search objects employed in the spatial queries. To this end, we implement algorithms that create search objects considering the total extent of the spatial dataset. For instance, a search object can be randomly created or correlated to at least one spatial object of the spatial dataset. The main benefit of these spatial datasets is that the experiments conducted in the PhD can be reproduced and even extended by other researchers. The complete definition and specification of these spatial datasets are given in Carniel, Ciferri and Ciferri (2017c) (Chapter 2).

[1] <http://www.openstreetmap.org/>
[2] <https://www.postgresql.org/>

The proposal of FESTIval, a versatile framework to evaluate spatial indices. In order to conduct experiments in a unique environment, we propose the *Framework to Evaluate SpaTial Indices* (*FESTIval*), a versatile framework for conducting experimental evaluations of spatial indices on different storage devices. FESTIval is a PostgreSQL extension that implements a set of disk-based spatial indices and flash-aware spatial indices. The main advantage of FESTIval is that it provides a common design to execute a sequence of index operations, termed *workloads*. Its common design allows us to create workloads as user-defined functions in SQL Procedural Language of the PostgreSQL (PL/pgSQL) and execute them in SQL SELECT statements. The performance results of executed workloads are stored in a specific data schema managed by FESTIval. By using this schema, users are able to execute SQL SELECT statements in order to retrieve and analyze the performance results of experiments. The first version of FESTIval is detailed in Carniel, Ciferri and Ciferri (2016a) and then extended in Chapter 3.

An analysis of the performance behavior of spatial indices on different storage devices. By using the specified spatial datasets and FESTIval, we conduct an extensive experimental evaluation on HDDs and SSDs. The goal is 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. Such an analysis is useful to understand the impact of SSDs on the spatial indexing context. The results suggested that the direct use of existing disk-based spatial indices on SSDs does not guarantee efficiency. That is, there are several cases where a spatial index provides better performance on HDD. The main reason for this behavior is due to the expensive random writes and interleaved operations during the index construction. The first discussions in this direction are given in Carniel, Ciferri and Ciferri (2016c). Then, the experiments are extended in Carniel, Ciferri and Ciferri (2017a) (Chapter 4).

A study on the applicability of employing flash simulators on the evaluation of spatial indices. During the experiments conducted on the PhD, we concluded that conducting experiments on SSDs is a complicated and expensive task. We can cite three reasons for this. First, spatial data is complex data to manage, requiring high elapsed times to execute their specific algorithms. Second, a spatial index may have tunning parameter values, requiring the evaluation of many different configurations. Finally, flash memory has limited endurance. That is, after a number of internal writes and erases, a block of a flash memory is not able to store data. Flash simulators are promising tools to mitigate these issues since they simulate the behavior of flash memory in the main memory. Hence, we study the applicability of employing a flash simulator on the evaluation of spatial indices. Two main benefits are evidenced. First, experiments can be accelerated since they can be executed in the main memory. Second, a flash simulator can be used as a first step of an experimental evaluation on SSDs. That is, the number of configurations to be evaluated in the SSD can be reduced based on the results obtained in the flash simulator. The complete study on this topic is given in Carniel *et al.* (2017) (Chapter 5) and a first version of an extended study is given in Carniel, Silva and Ciferri (2018).

The proposal of eFIND, a generic and efficient framework for flash-aware spatial indexing. The findings discovered in the experiments allowed us to comprehend what should be modified in existing spatial indices in order to achieve good performance on SSDs. This analysis resulted in the proposal of a set of *design goals for designing flash-aware spatial indices*. These design goals correlate the intrinsic characteristics of SSDs and the observations from our empirical studies. They served as a foundation for proposing the *efficient Framework for spatial INDexing on SSDs* (eFIND), a generic framework that transforms a disk-based spatial index into a flash-aware spatial index. The main advantage of eFIND is that it does not require modifications in the structure and algorithms of the index being ported. Instead, eFIND defines efficient in-memory data structures and algorithms that only changes the way in which reads and writes are performed on the SSD. As a result, eFIND can be incorporated into existing spatial database systems with low implementation costs. Experiments have showed that eFIND is very efficient to port the R-tree to SSDs compared to related work, showing reductions to build spatial indices from 43% to 77%, and to execute spatial queries from 4% to 23%. Further, the effect of each design goal has been studied in order to understand how the intrinsic characteristic of SSD impact on the spatial indexing performance. The first version of eFIND is detailed in Carniel, Ciferri and Ciferri (2017b). Then, eFIND is extended in Carniel, Ciferri and Ciferri (2018) (Chapter 6).

The eFIND xBR$^+$-tree, an efficient flash-aware spatial index for points. eFIND has been employed to port other spatial indexing structures to SSDs, such as the xBR$^+$-tree. This effort is the result of an international collaboration started by a research project conducted with the funding of the São Paulo Research Foundation (FAPESP) under the program Research Internships Abroad (BEPE). This project was conducted at the University of Thessaly, Greece. The main result of this research internship is the proposal of the *eFIND xBR$^+$-tree*, a novel efficient flash-aware spatial index for points. It was created by porting the xBR$^+$-tree to SSDs using eFIND. During this porting, eFIND was slightly adapted to consider the specific characteristics of the xBR$^+$-tree, such as the sorting properties in the internal and leaf nodes. The advantage of using eFIND instead of other approach is showed by performance results. The eFIND xBR$^+$-tree overcame its closest competitor by reducing the elapsed time to construct the index from 28.4% to 83.5% and to execute spatial queries up to 34.6%, considering real and synthetic spatial datasets. Such reductions were possible because the data structures of eFIND fit well the properties and structure of the xBR$^+$-tree. These results are detailed in Carniel *et al.* (2018) (Chapter 8).

In addition to the aforementioned contributions, throughout the development of this PhD work, efforts to advance on other topics related to spatial database systems were also conducted. They certainly contributed to enrich the studies reported on this PhD thesis. They are summarized as follows.

Advances on the representation and manipulation of the spatial fuzziness on spatial database systems. This work is originated from my Master's dissertation (CARNIEL, 2014) and has as goal to define, develop, and specify conceptual and executable spatial algebras for vague (fuzzy)

spatial objects on spatial database systems and GIS. More details about the results of this research topic are given in Carniel, Ciferri and Ciferri (2015b), Carniel, Ciferri and Ciferri (2015a), Carniel *et al.* (2014), Carniel, Schneider and Ciferri (2015), Carniel, Ciferri and Ciferri (2016b), Carniel, Ciferri and Ciferri (2016d), Carniel and Schneider (2016), Carniel and Schneider (2017a), Carniel and Schneider (2017b), and Carniel and Schneider (2018).

Survey of parallel and distributed spatial data management systems. This work is the result of another collaboration. The main goal is to apply a user-centric view to characterize existing parallel and distributed spatial data management systems based on the Hadoop and Spark. More details about the results of this research topic is given in Castro, Carniel and Ciferri (2018).

1.4 Organization

This PhD thesis consists of a collection of research papers[3] written along the trajectory of this PhD. Each paper corresponds to a chapter and is specified as follows:

- **Chapter 2** attaches the paper that describes the spatial datasets employed in the experiments conducted during the PhD (CARNIEL; CIFERRI; CIFERRI, 2017c). The complete reference of this paper is provided as follows:
  - Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Spatial Datasets for Conducting Experimental Evaluations of Spatial Indices. In Proceedings of the 32nd Brazilian Symposium on Databases (SBBD 2017) - Workshop Dataset Showcase, p. 286-295, 2017.
- **Chapter 3** attaches an extended paper that describes FESTIval, a versatile framework for conducting experimental evaluations of spatial indices. This paper extends the first version of FESTIval (CARNIEL; CIFERRI; CIFERRI, 2016a) and has the following complete reference:
  - Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Experimental Evaluation of Spatial Indices with FESTIval. In Proceedings of the Satellite Events of the 31st Brazilian Symposium on Databases (SBBD 2016), p. 123–128, 2016.
- **Chapter 4** attaches the paper that conducts the extensive experimental evaluation to understand the impact of SSDs on the spatial indexing (CARNIEL; CIFERRI; CIFERRI, 2017a). This paper extends a previous experimental analysis (CARNIEL; CIFERRI; CIFERRI, 2016c) and has the following complete reference:

[3] The citations for references and the layout of the research papers have been slightly adapted to fit well with the format of this PhD thesis.

  - Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Analyzing the Performance of Spatial Indices on Hard Disk Drives and Flash-based Solid State Drives. Journal of Information and Data Management 8 (1), p. 34-49, 2017.

- **Chapter 5** attaches the paper that studies the applicability of flash simulators on the experimental evaluation of spatial indices (CARNIEL *et al.*, 2017). The complete reference of this paper is provided as follows:

  - Carniel, A. C.; Silva, T. B.; Bonicenha, K. L. S.; Ciferri, R. R.; Ciferri, C. D. A. Analyzing the Performance of Spatial Indices on Flash Memories using a Flash Simulator. In Proceedings of the 32nd Brazilian Symposium on Databases (SBBD 2017), p. 40-51, 2017.

- **Chapter 6** attaches the paper that describes eFIND (CARNIEL; CIFERRI; CIFERRI, 2018), a generic and efficient framework for porting disk-based spatial indices to SSDs. This paper extends a previous version of eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b) and has the following complete reference:

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

- **Chapter 7** attaches the paper that provides a summary of the results of this PhD work (CARNIEL, 2018). This paper was presented in a workshop dedicated for PhD students. It extends a previous paper that was also discussed in another forum for PhD students (CARNIEL; CIFERRI, 2016). The complete reference of the attached paper is given as follows:

  - Carniel, A. C. Spatial Indexing on Flash-based Solid State Drives. In Proceedings of the VLDB 2018 PhD Workshop (VLDB 2018), p. 1-4, 2018.

- **Chapter 8** attaches the paper that introduces the eFIND xBR$^+$-tree (CARNIEL *et al.*, 2018). The complete reference of this paper is provided as follows:

  - Carniel, A. C.; Roumelis, G.; Ciferri, R. R.; Vassilakopoulos, M.; Corral, A.; Ciferri, C. D. A. An Efficient Flash-aware Spatial Index for Points. In Proceedings of the XIX Brazilian Symposium on GeoInformatics (GEOINFO 2018), p. 68-79, 2018.

Finally, this PhD thesis is finished in Chapter 9 by presenting the conclusions and future research topics opened by this PhD work, such as the extension of the work in Carniel, Silva and Ciferri (2018).

CHAPTER

2

SPATIAL DATASETS FOR CONDUCTING EXPERIMENTAL EVALUATIONS OF SPATIAL INDICES

This Chapter attaches the following paper (CARNIEL; CIFERRI; CIFERRI, 2017c):

- Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Spatial Datasets for Conducting Experimental Evaluations of Spatial Indices. In Proceedings of the 32nd Brazilian Symposium on Databases (SBBD 2017) - Workshop Dataset Showcase, p. 286-295, 2017.

Abstract

Spatial database systems widely employ spatial indices to accelerate the processing time of spatial queries. To measure the performance of spatial indices, researchers conduct extensive experimental evaluations by varying, e.g., characteristics of the spatial datasets to be indexed and types of spatial queries to be issued. Thus, the public sharing of spatial datasets potentially benefits the research community that aims to reproduce experiments or to conduct new experiments. In this paper, we provide spatial datasets that we have been used in our experiments. We provide two types of datasets: (i) spatial datasets that represent real-world phenomena to be indexed by the spatial indices, and (ii) spatial datasets that aid in the construction of spatial queries to be issued on an indexed dataset. As a result, our spatial datasets can be used to create and perform experiments that aim to evaluate the performance of spatial indices.

2.1 Introduction

Spatial database systems and Geographic Information Systems (GIS) provide the needed foundation for applications that require the management of geometric and geographic phenom-

ena (RIGAUX; SCHOLL; VOISARD, 2001). To this end, applications characterize geographic phenomena by using spatial data types like points, lines, and regions (GÜTING, 1994). For instance, points representing hydrants, lines representing streets, and regions representing engineering buildings. In order to efficiently retrieve spatial objects, spatial database systems and GIS widely employ spatial indices (GAEDE; GÜNTHER, 1998; OOSTEROM, 2005). The use of spatial indices accelerates the processing time of several types of spatial queries, such as range queries and point queries.

Several spatial indices have been proposed in the literature (e.g., Gaede and Günther (1998) survey more than 40 spatial indices). Examples of spatial indices are hierarchical structures like the R-tree (GUTTMAN, 1984) and the R*-tree (BECKMANN *et al.*, 1990). They have different characteristics and can employ different parameter values. For instance, the R-tree can employ different split algorithms, while the R*-tree includes reinsertion policies. With the evolution of the storage devices, there are also specific spatial indices for newer storage devices. For instance, flash-aware spatial indices to exploit the advantages of flash-based solid state drives (SSDs). Examples are the RFTL (WU; CHANG; KUO, 2003) and the FOR-tree (JIN *et al.*, 2015), as well as generic frameworks for creating flash-aware spatial indices like FAST (SARWAT *et al.*, 2013) and eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b).

Frequently, extensive experimental evaluations measure the performance of spatial indices proposed in the literature. In these experiments, the parameter values of the spatial indices are varied in order to evaluate different configurations of these indices. In addition, the running environment of the experiments is also varied by indexing spatial datasets with different characteristics. For instance, the number of spatial objects, and the spatial data types and geometric configurations (e.g., number of points) of the spatial objects. In addition, the experiments may vary the types of spatial queries that will be issued to an indexed spatial dataset. For instance, the execution of points queries, and the execution of range queries with different sizes of search objects.

The public access and availability of spatial datasets employed in the experiments provide a valuable data sharing for the research community that wants to reproduce experiments or to conduct new experiments. Unfortunately, in the most of the cases, the spatial datasets of the experiments are not publicly available since they are synthetically generated by tools that do not have open access. Examples of these situations are found in Greene (1989), Wu, Chang and Kuo (2003), Emrich *et al.* (2010), and Sarwat *et al.* (2013). Although other experiments (e.g., in Lv *et al.* (2011b), Luo, Wong and Leong (2012), and Jin *et al.* (2015)) often use datasets based on the spatial dataset of the R-tree portal[1], the employed dataset limits the variation of other characteristics of the experiment. In addition, the experiments (e.g., in Sarwat *et al.* (2013) and Jin *et al.* (2015)) commonly do not provide details regarding the search objects used in the execution of spatial queries.

[1] <http://www.chorochronos.org/?q=node/59>

In this paper, we provide different sets of spatial datasets. We distinguish them in two types: (i) spatial datasets that can be indexed by the spatial indices, and (ii) spatial datasets to form spatial queries. Together, these types of datasets can be used in experiments to evaluate the performance of spatial indices. For the first type of spatial datasets, we provide real spatial datasets extracted from the OpenStreetMap[2] that vary (i) volume of data, (ii) spatial data types, and (iii) geometric configurations. For the second type of spatial datasets, we provide other spatial datasets that vary (i) the size of the search objects, (ii) the type of spatial query, and (ii) the correlation of the search objects with the indexed spatial dataset. We have been used these two types of spatial datasets in our experiments to measure the performance of spatial indices in SSDs (CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL *et al.*, 2017). Despite this fact, a complete description of these spatial datasets is only discussed in this paper. We also provide the open and public access to these spatial datasets at <http://gbd.dc.ufscar.br/festival/datasets.html>.

This paper is organized as follows. Section 2.2 details our spatial datasets, including the methodology of their creation and useful descriptions of them. Section 2.3 discusses how to use our spatial datasets in experiments. Finally, Section 2.4 finishes the paper.

2.2 Generated Spatial Datasets

In this section, we provide the complete description regarding the collection, generation, and description of our spatial datasets, following their types introduced in Section 2.1. Section 2.2.1 details the methodology that we employed to creating our spatial datasets. Our spatial datasets are stored in relational tables of the PostgreSQL with the spatial extension PostGIS. Section 2.2.2 provides the description of these relational tables. Finally, Section 2.2.3 presents the statistical descriptions of our spatial datasets.

2.2.1 Employed Methodology for Collecting and Generating the Spatial Datasets

We employed two different methodologies to collect and generate our spatial datasets, one methodology for each type of spatial dataset. They are detailed as follows.

Spatial datasets that can be indexed. We extract them from the OpenStreetMap (OSM). OSM is a project that provides an environment to anyone map any geographic event in the world. Thus, it crowdsources spatial data from different people. It also offers mechanisms to make available and public access of these collected spatial data[3].

[2] <http://wiki.openstreetmap.org/wiki/Main_Page>
[3] <http://wiki.openstreetmap.org/wiki/Downloading_data>

Our spatial datasets consist only of geographic events of Brazil and were extracted and treated as follows. We firstly downloaded OSM data by using GeoFabrik[4]. It is a web site that permits to download OSM data, as .osm files, of specific regions and countries of the world. Since OSM data is constantly updated by people of the world, GeoFabrik provides data that are almost daily updated.

Since the format .osm uses OSM data structures to represent geographic phenomena, tools for extracting spatial objects from these files are needed. We have used the tool osm2pgsql[5], which transforms the OSM data into spatial objects stored in relational tables of a database in the PostgreSQL with the spatial extension PostGIS. osm2pgsql creates three important relational tables: (i) *planet_osm_point*, which stores only points, (ii) *planet_osm_line*, which stores only lines, and (iii) *planet_osm_polygon*, which stores only regions. Each table has two important columns: (i) *osm_id*, which stores unique identifiers of OSM; and (ii) *way*, which stores spatial objects. The other columns of these tables represent the types of events that OSM is able to map[6]. Two important notes of osm2pgsql are: (i) different versions of this tool can import a different number of spatial objects from a same .osm file, and (ii) a specific parameter (-G) should be provided for storing spatial objects with multiple components.

After importing spatial objects into relational tables, we then created specific spatial datasets by issuing SQL queries on the relational tables created by osm2pgsql. That is, we created relational tables that store spatial datasets representing specific contexts. In addition, these spatial datasets have different volumes, spatial data types, and geometric characteristics that are important to be varied in experiments evaluating spatial indices. These details are given in Sections 2.2.2 and 2.2.3.

Spatial datasets to form spatial queries. We created spatial datasets to provide search objects for point queries and range queries (GAEDE; GÜNTHER, 1998). We focus on these kind of queries because of their common usage in spatial applications. Formally, they are queries that return all the objects o from a set D where the predicate P returns *true* for a search object s, i.e., $SpatialQuery(D,s,P) = \{o | o \in D \wedge P(s,o) = true\}$. A point query specifies that s is a point and that P is the predicate *intersects*, i.e., $PointQuery(D, s \in point) = SpatialQuery(D,s,intersects)$. A range query specifies that s is a rectangular-shaped object and that P can assume any predicate, i.e., $RangeQuery(D, s \text{ is a rectangle}, P) = SpatialQuery(D,s,P)$.

Based on that, our spatial datasets to form spatial queries consist of a dataset containing only points and another dataset containing rectangles. To create them, we follow two different approaches. The first approach generates random objects (i.e., points and rectangles) that intersect the region (i.e., polygon) representing Brazil. Thus, there is no guarantee that a spatial query to be issued to an indexed spatial dataset will return spatial objects. On the other hand, the second

[4] <http://download.geofabrik.de/>
[5] <http://wiki.openstreetmap.org/wiki/Osm2pgsql>
[6] <https://wiki.openstreetmap.org/wiki/Map_Features>

approach generates search objects that are correlated with at least one object of a spatial dataset that can be indexed. For generating rectangles, there are two types of correlations: containment and intersection. This means that spatial queries that employ these search objects potentially will return spatial objects from an indexed spatial dataset. To guarantee the correlations, we generated one spatial dataset to form spatial queries for each spatial dataset that can be indexed (previously extracted). We also guarantee that the rectangles have proportional sizes in relation to the total extent of Brazil. As a result, we can construct spatial queries that return a different number of spatial objects, as detailed in Section 2.2.3. Section 2.3 discusses how spatial queries are constructed by using the search objects from these spatial datasets.

We provide the public access to the algorithms, implemented as PL/pgSQL functions, responsible for generating points and rectangles at <http://gbd.dc.ufscar.br/festival/>. Thus, researchers can use these functions to create points and rectangles that are not stored in our original spatial datasets.

2.2.2 Data Schema for Storing the Spatial Datasets

Here, we describe the relational tables that store our two types of spatial datasets.

Spatial datasets that can be indexed. We have extracted the OSM data related to Brazil in two different dates, 3 May 2016 and 16 January 2017. We call these files as *brazil2016.osm* and *brazil2017.osm*, respectively. Then, by following the methodology in Section 2.2.1, we created the relational tables detailed in Table 1. Thus, these relational tables store our spatial datasets. The columns of Table 1 are detailed as follows. The first column provides the name of the relational tables. The second column gives the .osm file of origin. The third column specifies the version of the employed osm2pgsql. Finally, the last column characterizes the context of the relational table, that is, the real-world phenomena that the spatial objects represent. Note that *brazil2017* is a relational table derived from the union of tables generated by osm2pgsql, forming, therefore, a bigger relational table. Other relational tables can be also combined. All these relational tables have the same columns. They are:

- *id*: has sequential values and is the primary key of a table.
- *osm_id*: provides the unique identifier used by OSM. By using this value, we can get the full description of a spatial object by using tools like Nominatim[7].
- *way*: stores spatial objects.

Spatial datasets to form spatial queries. According to the methodology in Section 2.2.1, we created two types of spatial datasets. We store them in two respective relational tables, *generated_point* and *generated_rectangles*. Their structures are detailed in Tables 2 and 3, respectively.

[7] <http://wiki.openstreetmap.org/wiki/Nominatim>

These relational tables store randomly created or correlated spatial objects. In addition, the generated rectangles have proportional sizes in relation to a given spatial dataset that can be indexed. The number of generated rectangles for each proportional size is given in the third column in Table 3.

Table 1 – Description of the relational tables that store our spatial datasets that can be indexed.

Name	**.osm File of Origin**	**osm2pgsql Version**	**Spatial Data Type of *way***	**Context of the Spatial Objects**
brazil_buildings2016	*brazil2016.osm*	0.91.0-dev (64 bit id space)	region	buildings[8] of Brazil, e.g., universities, hotels, schools, hospitals, warehouses, stadiums, houses, churches, etc.
brazil_buildings2017	*brazil2017.osm*			
brazil_highways2017	*brazil2017.osm*	0.93.0-dev (64 bit id space)	line	highways[9] of Brazil, e.g., roads, footpaths, streets, cycleways, raceways, etc.
brazil_points2017	*brazil2017.osm*		point	locations mapped as points, e.g., toilets, telephones, banks, hydrants, etc.
brazil2017	*brazil2017.osm*		region, line, point	union among all the regions, lines, and points of the *brazil2017.osm*

Table 2 – Description of the relational table (*generated_point*) that stores points to form point queries.

Column	**Data Type**	**Description**
id	integer	sequential numerical values representing the primary key
is_correlated	Boolean	Boolean values are used to indicate if the point is correlated to a spatial dataset or not
dataset	text	possible textual values are the names of the spatial datasets in Table 1. If *is_correlated* is *true*, it indicates the dataset in which *geom* is correlated; otherwise, it may indicate the target dataset to process the spatial query containing a search object from *geom*
geom	point	there are 100 randomly created points and 100 points correlated for each spatial dataset in Table 1

2.2.3 Statistical Description

Table 4 describes the statistical description of our relational tables that store the spatial datasets that can be indexed. The first feature is with respect to the data volume, that is, the number of spatial objects of each dataset (Table 4a). In addition, we show the minimum, maximum, and

[8] <http://wiki.openstreetmap.org/wiki/Map_Features#Building>
[9] <http://wiki.openstreetmap.org/wiki/Map_Features#Highway>

Table 3 – Description of the relational table (*generated_rectangle*) that stores rectangles to form range queries.

Column	Data Type	Description
id	integer	sequential numerical values representing the primary key
is_correlated	Boolean	Boolean values are used to indicate if the point is correlated to a spatial dataset or not
type	text	stores NULL, if *is_correlated* is *false*. Otherwise, it stores the type of correlation that was considered to create the rectangle: *intersects* or *contains*. This means that a rectangle *intersects* (or *contains*) at least one spatial object from the *dataset*
dataset	text	possible textual values are the names of the spatial datasets in Table 1. If *is_correlated* is *true*, it indicates the dataset in which *geom* is correlated; otherwise, it may indicate the target dataset to process the spatial query containing a search object from *geom*
percentage	double	percentage area of the total extent of Brazil. Examples are: 0.001%, 0.01%, 0.1%, and 1%
geom	region	there are 100 randomly created rectangles and 100 rectangles correlated for each spatial dataset in Table 1 considering values of *type*

average value of the following features: (i) number of points (Table 4a); (iv) length (Table 4b), only for datasets storing lines, and (ii) area and perimeter (Table 4c), only for datasets storing regions;. We can conclude from Table 4 that our spatial datasets have a great diversity of geometry characteristics, which can aid in the evaluation of spatial indices.

Table 4 – Statistical description of the relational tables that store our spatial datasets that can be indexed.

Relational Table	# of Spatial Objects	# of Points		
		Min	Max	Avg
brazil_buildings2016	534,926	4	506	7
brazil_buildings2017	1,486,557	4	744	8.29
brazil_highways2017	2,644,432	2	1,550	9.30
brazil_points2017	770,842	1	1	1
brazil2017	5,577,373	1	94,047	14.86

(a)

Relational Table	Length		
	Min	Max	Avg
brazil_highways2017	0.02	100,000	674.38
brazil2017	0	100,000	1,054.92

(b)

Relational Table	Area			Perimeter		
	Min	Max	Avg	Min	Max	Avg
brazil_buildings2016	0.0005	4,756,911	672.81	0.23	22,730.19	19.09
brazil_buildings2017	∼3.66e-05	4,756,911	368.56	0.03	22,730.19	63.4
brazil2017	0	∼9.30e12	13,421,569.76	0	24,572,026.11	670.32

(c)

The statistical description of *generated_point* and *generated_rectangle* are showed in Tables 5 and 6, respectively. These tables show the minimum, maximum, and average number of spatial objects that are returned when we execute point queries and range queries based on these spatial datasets. The number of spatial queries followed the number of objects reported in Tables 2 and 3. More specifically, the number of point queries was of 200 for each indexed spatial dataset: 100 with random points, and 100 with correlated points. The number of range queries was of 300 for each indexed spatial dataset and for each percentage value: 100 with random rectangles generated for the indexed spatial dataset, 100 correlated rectangles with the

type of intersection, and 100 correlated rectangles with the type of containment. This means that we consider only the correlated objects that were generated taking as a basis the indexed spatial dataset (Table 1). For instance, the first row of Table 5 shows that the maximum number of objects from *brazil_buildings2016* returned by the 100 point queries using correlated points based on *brazil_buildings2016* is equal to 3. Note that in Table 5, in most of cases, the point queries using random points did not return objects from the indexed datasets since the search objects were randomly generated considering the total extent of Brazil (Section 2.2.1).

In Table 6, we use (I) to indicate correlated rectangles based on the intersection and (C) to indicate rectangles based on the containment. Further, these symbols also indicate the predicate used in our range queries, where (I) means *intersects*, and (C) means *contains*. Note that the percentage area of the rectangles is shown in the second column of Table 6.

Table 5 – Statistical description of *generated_point*.

Indexed Dataset	**Random Points**			**Correlated Points**		
	# of Spatial Objects			**# of Spatial Objects**		
	Min	**Max**	**Avg**	**Min**	**Max**	**Avg**
brazil_buildings2016	0	0	0	1	3	1.03
brazil_buildings2017	0	0	0	1	2	1.01
brazil_highways2017	0	0	0	1	5	1.7
brazil_points2017	0	0	0	1	1	1
brazil2017	1	10	7.5	4	15	9.04

2.3 Utilization of the Spatial Datasets in Performance Evaluations

Our spatial datasets can be used according to the purpose in which they are created. The spatial datasets described in Table 1 have the purpose to be handled by spatial indices, such as:

- constructing a spatial index on a spatial dataset. For instance, create an R-tree on *brazil_buildings2016*. Future operations like updates, queries, and deletions are possible after the construction.
- constructing a spatial index on a part of a spatial dataset. For instance, create an R-tree on a percentage of the elements contained in *brazil_buildings2016*. Thus, the remaining elements can be used in future operations like insertions after the processing of spatial queries.

The spatial datasets described in Tables 2 and 3 have the purpose to form spatial queries. That is, these spatial datasets are dedicated to creating point queries and range queries. This creation can be made as follows:

Table 6 – Statistical description of *generated_rectangle*.

Indexed Dataset	Perc.	Random Rectangles # of Spatial Objects			Correlated Rectangles (I) # of Spatial Objects			Correlated Rectangles (C) # of Spatial Objects		
		Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
brazil_buildings2016	0.001%	0	39	0.55	1	74,206	21,679.51	1	74,200	22,279.39
	0.01%	0	11,719	204.35	3	95,023	24,144.46	17	95,053	30,326.52
	0.1%	0	5,261	216.31	20	99,479	37,079.97	199	99,501	38,193
	1%	0	85,304	7,822.47	211	113,261	55,214.83	728	102,912	53,780.87
brazil_buildings2017	0.001%	0	82	1.2	6	278,021	88,266.88	9	277,630	101,127.05
	0.01%	0	11,771	134.96	6	515,477	206,265.15	17	515,475	223,487.47
	0.1%	0	515,117	13,443.87	190	520,496	281,023.15	760	519,977	256,348.36
	1%	0	144,873	14,166.93	641	529,540	242,520.75	780	657,213	267,309.71
brazil_highways2017	0.001%	0	1,027	47.12	8	16,582	2,633.49	8	15,859	2,405.85
	0.01%	0	4,994	284.75	44	108,791	12,349.97	139	83,037	12,778.12
	0.1%	0	215,533	10,843.76	843	217,628	36,309.8	826	232,674	37,941.99
	1%	37	344,322	39,497.31	3,392	470,558	159,296.06	2,037	473,662	153,861.3
brazil_points2017	0.001%	0	132	6.34	1	31,102	3,889.45	1	31,778	2,727.45
	0.01%	0	3,139	116.19	23	47,212	8,818.45	4	47,527	9,169.48
	0.1%	9	73,036	3,741.52	72	72,344	18,426.5	38	72,417	14,098.62
	1%	37	90,343	11,723.36	389	141,498	51,208.44	429	140,891	48,481.03
brazil2017	0.001%	2	1,350	78.05	22	292,907	52,215.29	3	294,792	39,986.56
	0.01%	12	26,358	684.35	69	573,095	70,012.62	117	572,782	82,713.27
	0.1%	80	569,545	30,808.87	795	624,866	196,012.61	409	617,289	182,953.84
	1%	506	593,943	76,815.42	1,052	979,319	394,746	4,016	952,634	404,238.33

- for point queries: we select a list of points P from *generated_point*. For each point p in P, we can form point queries $PointQuery(D, p)$, where D is a given indexed spatial dataset. For instance, we can form 100 point queries to be processed by an R-tree created on *brazil_buildings2016*, where the points are correlated to the indexed spatial dataset.

- for range queries: we select a list of rectangles R from *generated_rectangle*. For each rectangle r in R and given a predicate P, we can form range queries $RangeQuery(D, r, P)$. For instance, we can form 100 range queries with the predicate *contains* to be processed by an R-tree created on *brazil_buildings2016*, where the rectangles have a correlation of containment with the indexed spatial dataset and percentage size of 0.01%.

The main benefits to using our spatial datasets to be handled by spatial indices are that (i) they are based on real geographic data and thus do not have fixed distributions of objects, (ii) they provide different geometric complexities and data volume, and (iii) they can be combined to create other specific datasets. These factors contribute to a complete empirical evaluation of spatial indices that can be done by creating different workloads based on our spatial datasets.

With respect to the benefits of our spatial datasets to form spatial queries, we provide several types of search objects that directly stress a spatial index. The main reason is that we include two types of search objects: randomly created search objects, which aid to evaluating if the spatial index shows a good performance in discarding of data that do not belong to the

answer, and correlated search objects, which may force the traversal of many pages of the spatial index. In addition, the rectangles of range queries permit the variation of the number of objects to be returned since they have different sizes.

We have been used the spatial datasets of this paper in experiments to evaluate the performance of spatial indices in different scenarios. We first analyzed and correlated the performance of spatial indices managed in hard disks and SSDs by using the spatial dataset *brazil_buildings2016* and by executing range queries by using rectangles randomly generated and stored in *generated_rectangle* (CARNIEL; CIFERRI; CIFERRI, 2016c). The use of these datasets in this experiment led us to identify important design goals that flash-aware spatial indices should address to deliver good performance on SSDs. As a result, we propose eFIND as a solution for spatial indexing on SSDs. Its performance was measured by indexing the spatial dataset *brazil_buildings2017* and by processing range queries based on randomly created rectangles stored in *generated_rectangle* (CARNIEL; CIFERRI; CIFERRI, 2017b). Another experiment using the same spatial datasets were conducted in order to measure the performance of spatial indices in flash simulators (CARNIEL *et al.*, 2017). Future work includes the extension of these experiments by indexing the spatial dataset called *brazil_buildings2017_v2*, which was created by using the osm2pgsql version 0.93. This spatial dataset contains 1,485,866 regions and has similar characteristics to the *brazil_buildings2017*. In addition, we will index the other spatial datasets storing lines and points. In addition, we plan to execute range and point queries with correlated search objects. Finally, all the experiments employ FESTIval (CARNIEL; CIFERRI; CIFERRI, 2016a), an open-source PostgreSQL extension to benchmark spatial indices on different storage devices. FESTIval has several default spatial datasets that can be indexed, which include the ones provided in this paper. A complete documentation of FESTIval is available at <http://gbd.dc.ufscar.br/festival/>.

2.4 Conclusions and Future Work

This paper provides several spatial datasets that can be employed in experiments to evaluate the performance of spatial indices. Two types of spatial datasets are provided: spatial datasets to be handled by spatial indices, and spatial datasets to form spatial queries. They have particular features that aid in the evaluation of spatial indices, including empirical experiments on different storage devices (CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL *et al.*, 2017). Our spatial datasets can be downloaded at <http://gbd.dc.ufscar.br/festival/>, which also provides a framework to conduct experimental evaluations of spatial indices. As a result, researchers can reproduce old experiments and create new experiments to better understand the performance of spatial indices under different scenarios.

We plan to extract other spatial datasets from the OpenStreetMap. In addition, future work includes the generation of more rectangles and points to act as search objects of spatial

queries. In fact, this task can be made by using our generator algorithm.

Acknowledgements

This work has been supported by the following Brazilian research agencies: CAPES, CNPq, and 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.

CHAPTER

3

FESTIVAL: A VERSATILE FRAMEWORK FOR CONDUCTING EXPERIMENTAL EVALUATIONS OF SPATIAL INDICES

This Chapter attaches an extended paper that describes FESTIval (CARNIEL; CIFERRI; CIFERRI, 2016a), originally published as follows:

- Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Experimental Evaluation of Spatial Indices with FESTIval. In Proceedings of the Satellite Events of the 31st Brazilian Symposium on Databases (SBBD 2016), p. 123–128, 2016.

Abstract

The use of a spatial index is a common strategy to improve the performance of spatial queries in spatial database systems and Geographic Information Systems. Choosing the right spatial index to be employed in a given context requires a quantitative method to analyze the performance of spatial indices. This is done through extensive experimental evaluations. However, conducting these evaluations is an expensive, error-prone, and challenging task because (i) spatial objects are complex data to manage, (ii) spatial indices can apply different parameter values and thus assume distinct configurations, and (iii) there are indices specifically developed for different storage systems, such as disks and flash memories. In this article, we propose FESTIval, a versatile framework for conducting experimental evaluations of spatial indices. FESTIval has the following main advantages:

- the support for different types of disk-based and flash-aware spatial indices;
- the specification and execution of user-defined workloads;

- the use of a data schema that stores index configurations and statistical data of executed workloads.

Because of its characteristics, FESTIval allows users to reproduce executed experiments. Further, FESTIval provides an extensible environment, where any spatial dataset can be handled by spatial indices. FESTIval has been used to validate new proposals of flash-aware spatial indices, such as eFIND-based indices.

3.1 Method Details

This article introduces the *Framework to Evaluate SpaTial Indices* (*FESTIval*), a versatile method for conducting experimental evaluations of spatial indices. Before describing the details of FESTIval, we shortly discuss the context and motivation behind its development.

3.1.1 Context and Motivation

Spatial database systems and *Geographic Information Systems* (GIS) widely make use of spatial indices to accelerate the processing of spatial queries, such as *spatial selections*, *range queries*, and *point queries* (GAEDE; GÜNTHER, 1998; OOSTEROM, 2005). A huge set of spatial indices has been proposed in the literature. Examples of traditional disk-based spatial indices are the R-tree (GUTTMAN, 1984) and its variants, the R*-tree (BECKMANN *et al.*, 1990) and the Hilbert R-tree (KAMEL; FALOUTSOS, 1994).

The development of spatial indices for newer storage devices like *flash-based Solid State Drives* (SSDs) has attracted the attention of the research community (EMRICH *et al.*, 2010; KOLTSIDAS; VIGLAS, 2011b; LIU *et al.*, 2012; CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a). The main reason is that SSDs have several improved characteristics than HDDs, such as smaller size, lighter weight, lower power consumption, and faster reads and writes. However, SSDs have intrinsic characteristics that introduce several system implications (AGRAWAL *et al.*, 2008; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016), such as the asymmetric costs between reads and writes, the performance interference of interleaved reads and writes, and the read disturbance management. To take into account the intrinsic characteristics of SSDs, *flash-aware spatial indices* have been proposed in the literature, such as FAST-based indices (SARWAT *et al.*, 2013), the FOR-tree (JIN *et al.*, 2015), and eFIND-based indices (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018).

With the increasing number of spatial indices, choosing the best spatial index to be employed in a given context requires the execution of extensive performance evaluations (GAEDE; GÜNTHER, 1998). However, conducting these evaluations is an expensive, error-prone, and challenging task because (i) spatial objects are complex data to manage, (ii) spatial indices can

apply different parameter values and thus assume distinct configurations, and (iii) there are indices specifically developed for different storage systems, such as disks and flash memories. A performance evaluation usually requires the execution of *user-defined workloads* on a given spatial dataset. A workload consists of a set of index operations, such as insertions, deletions, or updates of spatial objects, and the processing of spatial queries.

To the best of our knowledge, there are no methods that provide needed functionalities for creating and executing workloads to benchmark disk-based and flash-aware spatial indices on different storage devices. The reason is that existing approaches (KORNACKER; SHAH; HELLERSTEIN, 1998; GURRET *et al.*, 1999; MYLLYMAKI; KAUFMAN, 2002) face several problems (see next section). In general, they are not extensible since users are not able to define their own workloads. More importantly, they do not provide support for flash-aware spatial indices.

In this article, we propose FESTIval, a versatile framework for conducting experimental evaluations of spatial indices that:

- provides support for different types of disk-based and flash-aware spatial indices;
- allows the specification and execution of user-defined workloads under a unique environment;
- allows the reproduction of executed experiments;
- employs a data schema that stores index configurations and statistical data of executed workloads.

3.1.2 FESTIval

FESTIval is an open-source PostgreSQL extension implemented in C by using the extensibility provided by the PostgreSQL internal library[1]. FESTIval is also based on the PostGIS[2], a widely used PostgreSQL extension to manage spatial objects. The complete documentation of FESTIval is available at <https://accarniel.github.io/FESTIval/>.

Currently, FESTIval provides support for the following disk-based spatial indices: the R-tree, the R*-tree, and the Hilbert R-tree. FESTIval also provides support for the following flash-aware spatial indices: the FAST R-tree, the FAST R*-tree, the FAST Hilbert R-tree, the FOR-tree, the eFIND R-tree, the eFIND R*-tree, and the eFIND Hilbert R-tree. Further, FESTIval allows the execution of user-defined workloads on real storage devices (e.g., HDDs and SSDs) and on the Flash-DBSim (SU *et al.*, 2009), which is a flash simulator that emulates the behavior of flash memory in the main memory. The use of a flash simulator is useful because the Flash Translation

[1] For more information about writing PostgreSQL extensions, please access related PostgreSQL documentation at <https://www.postgresql.org/docs/10/static/extend-extensions.html>

[2] <https://postgis.net/>

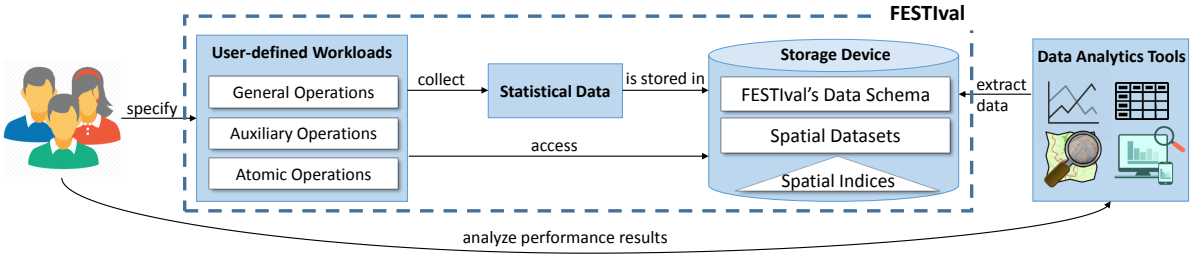


Figure 1 – The overview of FESTIval.

Layer (FTL) (KWON *et al.*, 2011) of a real flash memory usually does not provide access to the number of internal operations actually performed on the flash memory.

Figure 1 depicts a general view of FESTIval, which is detailed as follows.

Workloads. FESTIval allows the creation and execution of user-defined workloads by using the Structured Query Language (SQL). By using the FESTIval's SQL functions, a user (e.g., database administrator, researcher, and software developer) is able to define the sequence of index operations that should be executed and then analyzed. Index operations include insertions, updates, and deletions of spatial objects, and the execution of spatial queries (i.e., *general and atomic operations*). Further, users can also determine the exact moment that statistical data should be collected and stored in the FESTIval's data schema (i.e., *auxiliary operations*).

Storage Device. It stores three main elements: (i) the spatial indices, (ii) the spatial datasets, and (iii) the FESTIval's data schema. The spatial indices are handled by FESTIval during the execution of workloads, whereas the spatial datasets provide spatial objects to these indices. The FESTIval's data schema stores information of spatial datasets, parameters used by spatial indices, and statistical data of executed workloads. Collected statistical data can be used in mathematical models to measure the performance of spatial indices, considering employed parameter values, characteristics of the spatial dataset, and the employed storage device.

3.1.2.1 The FESTIval's Data Schema

Figure 2 depicts the FESTIval's logical data schema, called *fds*. In this figure, we only show the primary and foreign keys of the relational tables to illustrate their relationships. Table 7 enriches this figure by listing the attributes of each relational table. Here, we only provide a general view of this schema, detailing the most important tables only. The complete description can be found at the FESTIval's documentation.

There are two categories of data managed by FESTIval: (i) configuration of a spatial index, and (ii) storage of statistical data.

Configuration of a spatial index. It consists of four components. The first component is the *spatial dataset*, which is the source of spatial objects to be used by a spatial index. A spatial

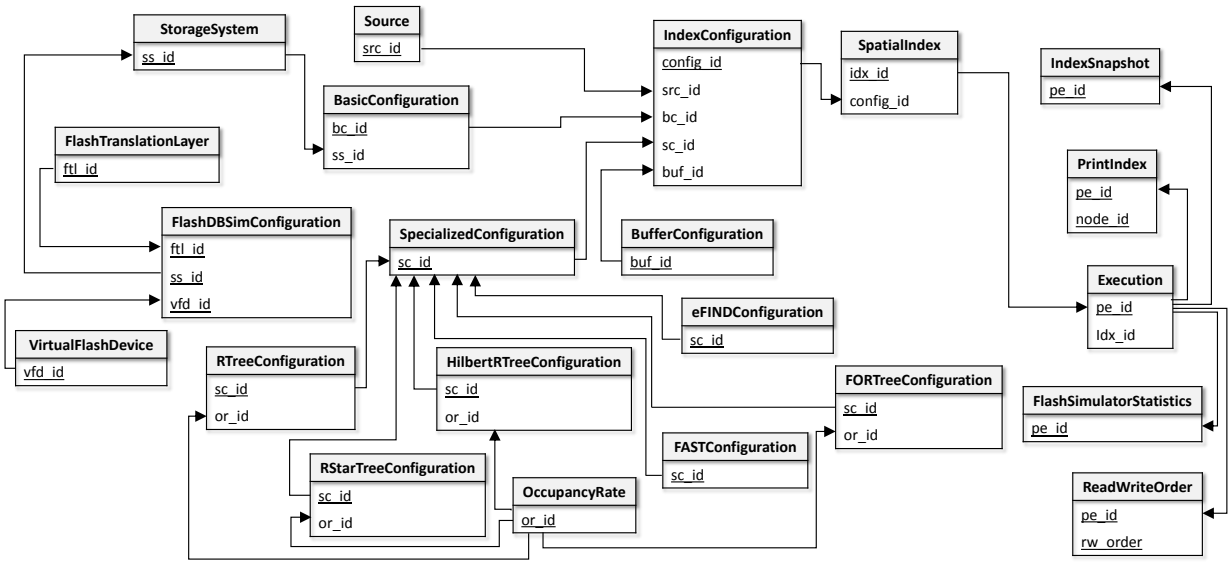


Figure 2 – The FESTIval's logical data schema. This figure shows only the primary and foreign keys of the relational tables to illustrate the relationships among the tables. Primary keys are highlighted. The relationship between a primary key and a foreign key is represented by a directed arrow from the primary key to the foreign key. Table 7 details the attributes of each relational table of this figure.

dataset is a PostgreSQL relational table that contains a column storing spatial objects. The needed information of spatial datasets is stored in the table Source. To insert a new spatial dataset in the FESTIval's data schema, we should provide its PostgreSQL schema name (*schema_name*), table name (*table_name*), column name that contains spatial objects (*column_name*), and the primary key of this table (*pk_name*). Hence, each tuple in Source represents a dataset that can be indexed. By using this strategy, users can use any spatial dataset in experiments. This fact contributes to providing a versatile platform to conduct empirical analyses.

The second component of a configuration refers to *basic parameters* that are employed by any spatial index. Basic parameters are stored in the table BasicConfiguration, which contains the storage system that stores the index (*ss_id*), the page size in bytes that an index page should have (*page_size*), the method of access to the storage device (*io_access*), and the library used in the refinement step in the spatial query processing (*refinement_type*). Currently, the attribute *io_access* is either the classical access to storage devices (i.e., using the *libio.h* in C) or the DIRECT I/O (i.e., using the *fcntl.h* in C) that allows us to bypass the caching system of reads and writes of the operating system; while the attribute *refinement_type* is either the use of the GEOS library or the use of PostGIS algorithms. As for the storage system, related data is stored in the table StorageSystem. This table contains the type of the storage device (*storage_system*) and its description (*description*). It can be HDDs, SSDs, or simulated flash memories using Flash-DBSim. For simulated flash memories, the primary key of the StorageSystem is linked to the table FlashDBSimConfiguration, which is an aggregated table of two needed information of the Flash-DBSim: (i) the flash device (table VirtualFlashDevice), and (ii) the flash translation layer

Table 7 – Attributes of each relational table of the FESTIval's data schema.

Relational Table	**Attributes**
Source	src_id, schema_name, table_name, column_name, pk_name
BasicConfiguration	bc_id, ss_id, page_size, io_access, refinement_type
StorageSystem	ss_id, storage_system, description
FlashDBSimConfiguration	ss_id, ftl_id, vfd_id
VirtualFlashDevice	vfd_id, nand_device_type, block_count, page_count_per_block, page_size1, page_size2, erase_limitation, read_random_time, read_serial_time, program_time, erase_time
FlashTranslationLayer	ftl_id, ftl_type, map_list_size, wear_leveling_threshold
BufferConfiguration	buf_id, buf_type, buf_size
SpecializedConfiguration	sc_id, description
RTreeConfiguration	sc_id, or_id, split_type
RStarTreeConfiguration	sc_id, or_id, reinsertion_perc_internal_nodes, reinsertion_perc_leaf_nodes, reinsertion_type, max_neighbors_exam
HilbertRTreeConfiguration	sc_id, or_id, order_splitting_policy
FORTreeConfiguration	sc_id, or_id, buffer_size, flushing_unit_size, ration_flushing, x, y
FASTConfiguration	sc_id, index_type, db_sc_id, buffer_size, flushing_unit_size, flushing_policy, log_size
eFINDConfiguration	sc_id, index_type, db_sc_id, buffer_size, read_buffer_perc, temporal_control_policy, read_temporal_control_perc, write_temporal_control_size, write_temporal_control_mindist, write_temporal_control_stride, timestamp_percentage, flushing_unit_size, flushing_policy, read_buffer_policy, log_size
OccupancyRate	or_id, min_fill_int_nodes, min_fill_leaf_nodes, max_fill_int_nodes, max_fill_leaf_nodes
IndexConfiguration	config_id, src_id, bc_id, sc_id, buf_id
SpatialIndex	idx_id, config_id, idx_name, idx_path, idx_creation, idx_last_mod
Execution*	pe_id, idx_id, execution_name, total_time, index_time, read_time, write_time, split_time, reads_num, writes_num, total_cpu_time, index_cpu_time, read_cpu_time, write_cpu_time, split_cpu_time, ...
ReadWriteOrder	pe_id, rw_order, op_type, op_timestamp, page_id
FlashSimulatorStatistics	pe_id, read_count, write_count, erase_count, read_latency, write_latency, erase_latency
IndexSnapshot*	pe_id, height, num_int_nodes, num_leaf_nodes, num_entries_int_nodes, num_entries_leaf_nodes, avg_num_entries_pnode, avg_coverage_area_pnode, ...
PrintIndex	pe_id, node_id, geom, elem_position, node_height

* attributes were suppressed.

All attributes are fully described in the FESTIval's documentation at <https://accarniel.github.io/FESTIval/>.

(table FlashTranslationLayer). The attributes of these tables correspond to the same parameters required by Flash-DBSim, represented by the table FlashDBSimConfiguration, to simulate a flash memory as detailed in Su *et al.* (2009) and in the FESTIval's documentation.

The third component of a configuration refers to the *generic buffer management* of the spatial index. A generic buffer manager is a general-purpose method employed to reduce the number of accesses to the storage device; thus, any spatial index can employ a buffer manager. Parameters of the generic buffer manager are stored in the table BufferConfiguration. The attributes of this table consist of the size of the buffer in bytes (*buf_size*), and the type of the page replacement algorithm (*buf_type*). Currently, FESTIval provides support for the following buffer managers: LRU (DENNING, 1980), LRU storing preferentially the highest nodes of the tree, called HLRU (as used in Carniel, Ciferri and Ciferri (2017b)), and the two versions of 2Q (JOHNSON; SHASHA, 1994). The size of the buffer of a spatial index equal to 0 means that the spatial index has not a general buffer manager. This is the case if the spatial index has its own specialized buffer manager. For instance, flash-aware spatial indices (e.g., FAST-based and eFIND-based indices) have specialized buffer managers with specific parameter values (see below). Although it is possible to also employ generic buffer managers in flash-aware spatial indices, performance evaluations usually do not employ general buffer managers when analyzing the performance of flash-aware spatial indices (SARWAT *et al.*, 2013; JIN *et al.*, 2015; CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018; CARNIEL *et al.*, 2017).

Finally, the fourth component of a configuration refers to *specific parameters* that are used by an index. That is, each spatial index has its own set of parameters, and the table SpecializedConfiguration generalizes this concept by providing a unique identifier for this specific set of parameters. For instance, the R-tree permits to specify its split algorithm (*split_type*), which can be exponential, quadratic, and linear (GUTTMAN, 1984). Other split algorithms are also conceivable, such as the Greene-split (GREENE, 1989) and the AngTan-split (ANG; TAN, 1997). This specific information is stored in the specialized table RTreeConfiguration and for each entry in this table, there is also an entry in the table SpecializedConfiguration that includes a short description (*description*). This strategy is similarly employed to store the specific parameters of other supported indices, that is, the R*-tree (table RStarTreeConfiguration), the Hilbert R-tree (table HilbertRTreeConfiguration), the FOR-tree (table FORTreeConfiguration), FAST-based indices (table FASTConfiguration), and eFIND-based indices (table eFINDConfiguration). The attributes of these tables are based on the corresponding parameters of their indices as specified in their original research papers. For FAST- and eFIND-based spatial indices, the use of the attribute *db_sc_id* that refers to the identifier of an entry of the SpecializedConfiguration allows to combine the specific parameters of the underlying index pointed by this attribute and the specific parameters of FAST and eFIND.

In addition, we can vary the *occupancy rate* (table OccupancyRate) of index pages. This

occupancy rate is informed by the attribute *or_id* that is present in the tables storing specific parameters of indices. We can specify the maximum capacity of leaf and internal nodes by using percentage values *max_fill_leaf_nodes* and *max_fill_int_nodes*, respectively. These percentage values specify how much space from the page size should be allocated to accommodate node entries. We can also specify the minimum capacity of leaf and internal nodes using respectively the attributes *min_fill_leaf_nodes* and *min_fill_int_nodes*, indicating the minimum occupancy rate considering the total available space (calculated from the maximum capacity of the nodes).

FESTIval provides an SQL script, named *festival-inserts.sql*, that contains SQL INSERT INTO statements for inserting default parameter values into all the aforementioned relational tables related to the configuration of spatial indices. As for the default spatial datasets, they can be downloaded at <https://github.com/accarniel/FESTIval/wiki/> and a detailed specification of them is also given in Carniel, Ciferri and Ciferri (2017c). Users are also able to insert new values to the aforementioned tables by executing SQL INSERT INTO statements. The informed values are checked by using triggers and SQL CHECK constraints in order to ensure the consistency of the parameters. For instance, it is impossible to insert a new R*-tree configuration that has a reinsertion type not defined in the original R*-tree paper.

The combination of the values of the attributes *src_id*, *bc_id*, *sc_id*, and *buf_id* of the tables Source, BasicConfiguration, SpecializedConfiguration, and BufferConfiguration, respectively, creates one spatial index configuration (table IndexConfiguration). Then, a spatial index (table SpatialIndex) consists of a configuration from the table IndexConfiguration (*config_id*) and other specific data, such as the name of the index (*idx_name*), the directory storing the index file (*idx_path*), time of its creation (*idx_creation*), and the time of its last modification (*idx_last_mod*). Note that multiple spatial indices might have the same configurations; however, their states (i.e., content) might be different because of the executed operations. Only FESTIval insert entries into the tables IndexConfiguration and SpatialIndex, that is, users should not manually insert entries into these relational tables.

Storage of statistical data. FESTIval collects and stores two types of statistical data. The first type refers to *statistical data collected from the execution of index operations*. This data is maintained in the table Execution, which is a non-normalized table in order to avoid excessive joins when retrieving performance results. Each entry in this table means that at least one index operation like insertion, deletion, and query was performed. To identify the type of execution, the user can set a name (*execution_name*). This aspect is further discussed in the next sections of this article. Here, we only provide a general view of the main attributes of the table Execution: the total processing time of the index (*index_time*), the time spent to perform reads and writes (*read_time* and *write_time*, respectively), the processing time to execute splitting operations (*split_time*), the number of reads and writes (*reads_num* and *writes_num*, respectively), the CPU time of processing specific operations (e.g., *index_cpu_time*), and other attributes. Note that the total processing time of the index considers other times, such as the processing time of

splits, reads, and writes. By collecting and storing detailed statistical data, FESTIval allows us to better analyze the composition of the total processing time (*total_time*). Further, FESTIval stores the order of reads and writes performed on the storage device (table ReadWriteOrder). To this end, a sequential identifier of the operation (*rw_order*), the type of operation (*op_type*, which can be either read or write), the moment that the operation was performed (*op_timestamp*), and the identifier of the index page (*page_id*) are stored. The order of reads and writes is optionally collected and is useful to discover access patterns. Finally, if the executed workload employed a simulated flash memory, statistical data collected by Flash-DBSim are stored in the table FlashSimulatorStatistics. This data includes the number of read, writes, and erases actually performed on the simulated flash memory (*read_count*, *write_count*, and *erase_count*, respectively).

The second type of collected *statistical data refers to the structure of the spatial index*. This data is stored in the table IndexSnapshot. Each entry in this table allows us to analyze the structure of an index after executing operations that modify its structure, such as insertions and deletions. Here, we only provide a general view of the main attributes of the table IndexSnapshot: the height of the index (*height*), the number of internal and leaf nodes (*num_int_nodes* and *num_leaf_nodes*, respectively), the number of entries in internal and leaf nodes (*num_entries_int_nodes* and *num_entries_leaf_nodes*, respectively), summary data per node (e.g., the average number of entries - *avg_num_entries_pnode*), and other related attributes. Further, FESTIval provides the table PrintIndex that allows us to graphically visualize a spatial index by using a GIS, such as QGIS[3] and ArcGIS[4]. To this end, FESTIval stores data related to each node entry of the spatial index, such as position in the node (*elem_position*), its minimum bounding rectangle (MBR) (*geom*), height of the node (*node_height*), and the identifier of the node (*node_id*). This allows us to understand the structure of a spatial index for different purposes, such as educational.

3.1.2.2 The FESTIval's Operations

FESTIval provides a set of SQL functions that allows users to create and execute workloads by using a common design. Each SQL function has the prefix *FT_* and calls one C function internally implemented in the FESTIval's internal library that is responsible for performing the desired processing. Hence, an index can be seen as an *abstract data type* (STONEBRAKER; RUBENSTEIN; GUTTMAN, 1983) that has common parameters (i.e., its configuration) and a set of operations (i.e., SQL functions). The main advantage of this strategy is that complex implementations are hidden from users, who now can manage and test different indices under the same environment (i.e., using the same SQL functions).

There are three types of operations: (i) general operations, (ii) auxiliary operations, and (iii) atomic operations. They are described as follows.

[3] <https://qgis.org/>
[4] <https://www.arcgis.com/index.html>

General operations. They are responsible for handling index structures. Since FESTIval provides a common design, the same SQL function can be used for any type of index. We detail each general operation of FESTIval as follows:

1. `boolean FT_CreateEmptySpatialIndex(integer index_type, text apath, integer src_id, integer bc_id, integer sc_id, integer buf_id);`
2. `boolean FT_Insert(text apath, integer pointer, geometry geom);`
3. `boolean FT_Delete(text apath, integer pointer, geometry geom);`
4. `boolean FT_Update(text apath, integer old_pointer, geometry old_geom, integer new_pointer, geometry new_geom);`
5. `setof query_result FT_QuerySpatialIndex(text apath, integer query_type, geometry search_obj, integer predicate, integer proc_option=1);`
6. `boolean FT_ApplyAllModificationsForFAI(text apath);`
7. `boolean FT_ApplyAllModificationsFromBuffer(text apath);`

All these functions have a common parameter, *apath*, that indicates the absolute path of the index file. The first SQL function creates an empty spatial index (i.e., without any spatial objects) according to a set of parameters. It returns *true* if the index is successfully created, and *false* otherwise. The parameter *index_type* is an identifier that specifies the type of index to be created. Currently, FESTIval employs integer values from 1 to 10 to respectively represent the R-tree, the R*-tree, the Hilbert R-tree, the FAST R-tree, the FAST R*-tree, the FAST Hilbert R-tree, the FOR-tree, the eFIND R-tree, the eFIND R*-tree, and the eFIND Hilbert R-tree. The parameter *src_id* is a primary key value of the table Source that binds the spatial objects to the index. The parameters *bc_id*, *sc_id*, and *buf_id* specify the basic, specific, and buffer parameters of the spatial index by using the primary key values originated from the tables BasicConfiguration, SpecializedConfiguration, and BufferConfiguration, respectively. Since the specific parameters refer to only one type of index, FESTIval checks if the index to be constructed (i.e., the parameter *index_type*) is compatible with the values of *sc_id*. In summary, *FT_CreateEmptySpatialIndex* prepares all internal structures needed to handle a spatial index.

The three SQL functions *FT_Insert*, *FT_Delete*, and *FT_Update* execute operations that modify the index structure by respectively inserting, deleting, and updating spatial objects. They return *true* if the modification is successfully executed, and *false* otherwise. To insert and delete spatial objects, two additional parameters are needed: *pointer* and *geom*. The parameter *pointer* is the primary key value of the spatial object being inserted (or deleted), while the parameter

geom is the geometry representing the spatial object. Spatial objects handled by FESTIval are PostGIS objects (i.e., *geometry* objects), guaranteeing a full integration of FESTIval with spatial applications that use PostGIS. To update a spatial object, *FT_Update* requires information about the spatial object being updated (parameters *old_pointer* and *old_geom*) to a new value (parameters *new_pointer* and *new_geom*). In general, an update is an atomic operation that sequentially deletes the old spatial object (and its pointer) and then inserts the new spatial object (and its pointer).

To apply the functions *FT_Insert*, *FT_Delete*, and *FT_Update*, it is needed to first create the index file (i.e., *apath*) by using the function *FT_CreateEmptySpatialIndex*. Further, since a spatial index is related to a specific dataset, it is important to apply the corresponding modification firstly in its dataset. For instance, add a new spatial object with its primary key value as a new tuple in the indexed dataset by using an SQL INSERT INTO statement before calling *FT_Insert*. FESTIval does not perform any changes in the dataset; thus, the user should perform modifications in the dataset as needed. The main advantage of this treatment is that we can isolate the time processing of a modification performed on the spatial index from the modification performed on the dataset.

The fifth SQL function executes spatial queries. It is a set-returning function of the PostgreSQL. It returns a set of *query_result* rows, formed by a primary key value (*id*) and a spatial object (*geom*) of the indexed dataset. The parameter *query_type* specifies the type of spatial query to be processed. There are many types of spatial queries proposed in the literature (GAEDE; GÜNTHER, 1998; OOSTEROM, 2005). FESTIval provides support for *spatial selections* (*query_type*=1), *range queries* (*query_type*=2), and *point queries* (*query_type*=3). Spatial selection is a general type of query that returns a set of spatial objects that satisfy some topological predicate (e.g., overlap, inside) for a given spatial object, called *search object*. Range query is similar to spatial selection but considering the search object as a rectangular-shaped object. Point query specializes spatial selection by allowing only the use of intersects as the topological predicate and points as search objects. The parameter *search_obj* is the search object, which is a PostGIS object. Some restrictions with respect to the geometric format of *search_obj* may be applicable. If *query_type* is equal to 2, the MBR of *search_obj* is considered. If *query_type* is equal to 3, *search_obj* must be a simple point object. The parameter *predicate* specifies the topological predicate (SCHNEIDER; BEHR, 2006) to be used in the spatial query. It can assume the following topological predicates: intersects, overlap, disjoint, meet, inside, coveredBy, contains, covers, and equals (they are integer values from 1 to 8, respectively). Finally, the last parameter *proc_option* refers to the type of the result of the spatial query, which is often executed by using two steps, filter and refinement (GAEDE; GÜNTHER, 1998). If *proc_option* is equal to 1, its default value, *FT_QuerySpatialIndex* yields the final result of the query, that is, it executes the filter and refinement steps. In this case, it is important to maintain the spatial index compatible with the indexed dataset, as previously discussed. If *proc_option* is equal to 2, *FT_QuerySpatialIndex* returns the candidates of the query returned by the filter step only.

The last two SQL functions are responsible for executing a flushing operation that writes to the storage device all buffered index modifications. They return *true* if the flushing operation is successfully executed, and *false* otherwise. *FT_ApplyAllModificationsForFAI* flushes all buffered modifications stored in the specialized buffers of flash-aware spatial indices (i.e., eFIND-based indices), while *FT_ApplyAllModificationsFromBuffer* flushes all buffered modifications stored in the generic buffers (e.g., LRU, 2Q). These SQL functions write all modifications contained in the main memory, cleaning the corresponding buffer.

Auxiliary operations. They are designed for helping the process of creating workloads. They are mainly related to collecting and storing statistical data. We detail each auxiliary operation of FESTIval by providing its synopsis together with its short description as follows:

1. `boolean FT_StartCollectStatistics(boolean rw=false);`

2. `boolean FT_CollectOrderOfReadWrite();`

3. ```
   integer FT_StoreStatisticalData(text apath,
   integer statistic_option=1, integer loc_stat_data=1,
   text file=NULL);
   ```

4. ```
   boolean FT_StoreIndexSnapshot(text apath, integer execution_id,
   boolean print_index=false, integer loc_stat_data=1,
   text file=NULL);
   ```

5. ```
   boolean FT_SetExecutionName(text execution_name,
   integer loc_stat_data=1);
   ```

The functions returning Boolean values yield *true* if the processing is successfully performed, and *false* otherwise. The first auxiliary operation is the SQL function *FT_StartCollectStatistics*. After invoking this function, FESTIval starts to collect statistical data in the main memory. If the parameter *rw* is *false*, its default value, the order of reads and writes made on the storage device are not be collected. Otherwise, this order is collected, requiring extra computation. When an user performs the SQL function *FT_StartCollectStatistics*(*false*), and afterwards wants to collect the order of reads and writes, the user should call the SQL function *FT_CollectOrderOfReadWrite*(). This allows users to collect the order of reads and writes only for specific index operations since this collection is expensive; but it is important to understand access patterns.

Collected in-memory statistical data are only stored in the FESTIval's data schema after calling the SQL function *FT_StoreStatisticalData*, which returns an integer value that consists of the primary key value of the row inserted into the table Execution (i.e., the value of the column *pe_id*). The parameter *apath* is the absolute path of the index file and the parameter
   ```

statistic_option refers to the type of statistical data that is stored. Independently of the value of *statistic_option*, FESTIval stores typical statistical data about the executed index operations (i.e., the attributes of the table Execution). If *statistic_option* is equal to 1, its default value, FESTIval inserts a new tuple in the table Execution only, without any other additional information. Optionally, FESTIval stores a new tuple in the table IndexSnapshot (*statistic_option*=2 or *statistic_option*=4), requiring the traversal of all index pages in order to collect statistical data related to the index structure. Further, FESTIval stores new tuples in the table PrintIndex (*statistic_option*=3 or *statistic_option*=4), also requiring the traversal of all index pages to visualize the index structure. The cost of traversing the tree is not take into account when collecting typical statistical data. Storing data in the tables IndexSnapshot and PrintIndex is particularly useful after the execution of operations that modify the index structure (e.g., insertions, deletions, and updates). As for the parameter *loc_stat_data*, it defines where the statistical data should be stored. If its value is equal to 1, its default value, the statistical data is stored directly in the FESTIval's data schema. If its value is equal to 2, the statistical data is stored in an SQL file that can be latter loaded into the FESTIval's data schema. In this case, the absolute path of this SQL file should be informed by using the parameter *file*. Particularly, setting the value 2 for *loc_stat_data* is useful to avoid reads and writes performed on the FESTIval's data schema during the execution of a workload. This aspect is important for SSDs because of the interference between reads and writes (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). Hence, the statistical data can be first stored in a file located in other storage device (e.g., an HDD). Finally, at any moment, the user is also able to collect and store statistical data related to the index structure by using the SQL function *FT_StoreIndexSnapshot*. It has almost the same parameters as the SQL function *FT_StoreStatisticalData*, expect for the parameters *execution_id* and *print_index*. The parameter *execution_id* is the primary key value of the table Execution that links the collected statistical data with an execution, while the parameter *print_index* indicates whether the structure of index should be stored in the table PrintIndex or not.

The last auxiliary operation is the SQL function *FT_SetExecutionName*. Its main applicability is to set a name for the execution of a workload through the parameter *execution_name*. By creating workloads with this function, users are able to easily retrieve statistical data of executed workloads by issuing SQL queries on the table Execution. As previously described, the parameter *loc_stat_data* defines where the statistical data should be stored.

Atomic operations. They are combinations of some aforementioned functions and help the construction of workloads. An atomic operation is an SQL function that is executed as a unique and indivisible operation, following the principles of the atomicity of database systems (HARDER; REUTER, 1993). That is, if any function inside an atomic operation fails, the entire atomic operation fails. In general, an atomic operation is formed by the following sequence of operations: (i) *FT_StartCollectStatistics*, (ii) the requested operation, and (iii) *FT_StoreStatisticalData*. The atomic operations of FESTIval start with *FT_A* and are specified as follows:

Algorithm 1: The source code of the SQL function *FT_AInsert*, an atomic operation of FESTIval.

```
1 CREATE OR REPLACE FUNCTION FT_AInsert(apath text, pointer
    int4, geom geometry, statistic_option int4 default 1,
    location_stat_data int4 default 1, file text default NULL)
2 RETURNS int4 AS
3 $BODY$
4   BEGIN
5     PERFORM FT_StartCollectStatistics(); -the default value
       of its parameter is false
6     PERFORM FT_Insert(apath, pointer, geom);
7     RETURN FT_StoreStatisticalData(apath, statistic_option,
       location_stat_data, file);
8 LANGUAGE plpgsql VOLATILE
9 COST 100;
```

1. `integer FT_AInsert(text apath, integer pointer, geometry geom, integer statistic_option=1, integer loc_stat_data=1, text file=NULL);`

2. `integer FT_ADelete(text apath, integer pointer, geometry geom, integer statistic_option=1, integer loc_stat_data=1, text file=NULL);`

3. `integer FT_AUpdate(text apath, integer old_pointer, geometry old_geom, integer new_pointer, geometry new_geom, integer statistic_option=1, integer loc_stat_data=1, text file=NULL);`

4. `setof query_result FT_AQuerySpatialIndex(text apath, integer query_type, geometry search_obj, integer predicate, integer proc_option=1, integer statistic_option=1, integer loc_stat_data=1, text file=NULL);`

Algorithm 1 depicts the source code of the SQL function *FT_AInsert*, which illustrates a first usage of FESTIval's operations. The remaining atomic operations have very similar codifications. In fact, *FT_AInsert*, *FT_ADelete*, *FT_AUpdate*, and *FT_AQuerySpatialIndex* are atomic versions of the functions that respectively insert, delete, update, and query spatial objects from a spatial index. They combine the parameters of the SQL function responsible for executing the index operation and the parameters of *FT_StoreStatisticalData*. They assume that the order of reads and writes is not collected. Hence, collecting this kind of information requires the invocation of the SQL function *FT_CollectOrderOfReadWrite* before executing an atomic operation.

3.1.2.3 Creating and Executing Workloads

FESTIval provides a common design to create workloads. A workload consists of a sequence of index operations and can be created by using the SQL Procedural Language of the PostgreSQL (PL/pgSQL). Hence, users create workloads as user-defined functions in PL/pgSQL and execute them in SQL SELECT statements. Here, we illustrate two examples of workloads.

The first workload, depicted in Algorithm 2, executes a sequential insertion of spatial objects stored in a given dataset. That is, it constructs a spatial index by inserting spatial objects one-by-one. Due to the importance of this workload, FESTIval includes this function in its source code. The inputs of *FT_CreateSpatialIndex* are similar to those of the aforementioned SQL functions, except for the Boolean parameters *apply_fai* and *apply_stdbuffer* (line 1) that are employed to decide whether flushing operations should be performed in the end of the index creation. First, *FT_CreateSpatialIndex* extracts needed data about the dataset that is being indexed (line 10). This includes the names of its schema, table, column storing spatial objects, and primary key column. Then, the total number of rows of this table is retrieved (line 11) to identify how many spatial objects should be inserted into the spatial index. Afterwards, statistical data should be collected when executing the next index operations (line 12). The first index operation is the creation of an empty spatial index (line 13). Next, a sequence of insertions is made (lines 15 to 25). To better manage the main memory, the workload retrieves 100,000 spatial objects by time from the dataset (lines 16 and 17). The loop stops when all spatial objects are inserted into the spatial index (lines 19 to 21). In the sequence, the workload checks whether a flushing operation should be made in order to write all the remaining in-memory modifications after the insertions (lines 26 to 31). This includes the specialized in-memory buffer managers of flash-aware spatial indices (lines 26 to 28), and general buffer managers (lines 29 to 31). Finally, the workload stores statistical data related to the creation of the spatial index (line 32).

Before introducing the second workload, we show how the workload depicted in Algorithm 2 can be employed and how users can collect performance results. For this, we show a set of SQL statements that can be incrementally executed in order to reproduce our example.

In order to provide a name for the execution of *FT_CreateSpatialIndex*, we need to first execute an SQL SELECT statement that invokes *FT_SetExecutionName*, as shown below:

```
SELECT FT_SetExecutionName('Creating R-tree on brazil_points2017');
```

Then, an index can be created by executing *FT_CreateSpatialIndex*. The next SQL SELECT statement creates an R-tree, called *linear_rtree* and stored in */opt/*, that indexes the spatial objects stored in the dataset identified by *src_id*=7 and with the parameter values *bc_id*=6, *sc_id*=18, and *buf_id*=4. These values are included in *festival-inserts.sql*, which can be loaded into the FESTIval's data schema after installing FESTIval. More precisely, this command builds an R-tree with page (node) size of 4KB, employing the linear splitting algorithm, and a general LRU buffer of 512KB. The indexed spatial objects are from the dataset named *brazil_-*

Algorithm 2: The workload written in PL/pgSQL for creating spatial indices.

```
1  CREATE OR REPLACE FUNCTION FT_CreateSpatialIndex(index_type
     int4, apath text, src_id int4, bc_id int4, sc_id int4,
     buf_id int4, apply_fai bool default false, apply_stdbuffer
     bool default false, statistic_option int4 default 1,
     location_stat_data int4 default 1, file text default NULL)
2  RETURNS bool AS
3  $BODY$
4      DECLARE
5          src REFCURSOR; rec RECORD;
6          i INTEGER; total INTEGER := 0;
7          sch VARCHAR; tab VARCHAR; colu VARCHAR; pk VARCHAR;
8      BEGIN
9          EXECUTE 'SELECT schema_name, table_name, column_name,
            pk_name FROM fds.source WHERE src_id = $1' INTO sch,
            tab, colu, pk USING src_id;
10         EXECUTE 'SELECT count(*) from ' || sch || '.' || tab
            INTO total;
11         PERFORM FT_StartCollectStatistics();
12         PERFORM FT_CreateEmptySpatialIndex(index_type, apath,
            src_id, bc_id, sc_id, buf_id);
13         i := 0;
14         WHILE (i <= total) LOOP
15             OPEN src FOR
16             EXECUTE 'SELECT ' || pk || ' as pk, ' || colu || '
                as geom FROM ' || sch || '.' || tab || ' ORDER BY '
                || pk || ' LIMIT 100000 OFFSET $1' USING i;
17             LOOP
18                 FETCH src INTO rec;
19                 EXIT WHEN NOT FOUND;
20                 PERFORM FT_Insert(apath, rec.pk, rec.geom);
21             CLOSE src;
22             i := i + 100000;
23         IF (apply_fai AND (index_id = 4 OR index_id = 5 OR
            index_id = 6 OR index_id = 7 OR index_id = 8 OR
            index_id = 9 OR index_id = 10)) THEN
24             PERFORM FT_ApplyAllModificationsForFAI(apath);
25         IF (apply_stdbuffer) THEN
26             PERFORM FT_ApplyAllModificationsFromBuffer(apath);
27         PERFORM FT_StoreStatisticalData(apath,
            statistic_option, location_stat_data, file);
28         RETURN true;
29 LANGUAGE plpgsql VOLATILE
30 COST 500;
```

points2017 (CARNIEL; CIFERRI; CIFERRI, 2017c). Since this command does not change default values for arguments of *FT_CreateSpatialIndex*, it does not flush any modification

remaining in the buffer after the insertions, and it stores statistical data only for the table Execution that is directly inserted into the FESTIval's data schema.

```
SELECT FT_CreateSpatialIndex(1, '/opt/linear_rtree', 7, 6, 18, 4);
```

By issuing SQL SELECT statements, we are also able to retrieve and analyze performance results. For instance, the following command returns the required index time to build the previous R-tree (i.e., *linear_rtree*):

```
SELECT index_time
FROM fds.execution
WHERE execution_name = 'Creating R-tree on brazil_points2017';
```

Note the importance of setting a name for the execution, which can be used to retrieve performance results of executed workloads. The previous SQL SELECT statement could also include other columns containing statistical values.

If *FT_CreateSpatialIndex* is executed multiple times, we are able to capture average statistical results from these executions. In this case, the index is built with different names but with the same configurations. Assuming that the previous R-tree is created multiple times (with different names, such as *linear_rtree2*, *linear_rtree3*, and so on), the following SQL SELECT statement returns the average index time and its standard deviation of these executions:

```
SELECT avg(index_time), stddev(index_time)
FROM fds.execution
WHERE execution_name = 'Creating R-tree on brazil_points2017';
```

Another example of execution of *FT_CreateSpatialIndex* is to vary its parameter values in order to collect statistical data related to the index structure, as shown in the next two SQL SELECT statements. The first one denominates the corresponding execution name. The second one creates another R-tree, called *linear_rtree_comp* and stored in */opt/*, with the same parameter values of the previous R-tree (i.e., *linear_rtree*); but collecting statistical data related to its structure, which is useful to analyze the spatial organization of the index. In addition, FESTIval also stores the nodes of the built index, which is useful to visualize the index by using specialized programs like QGIS.

```
SELECT FT_SetExecutionName('Creating R-tree on brazil_points2017');
SELECT FT_CreateSpatialIndex(1, '/opt/linear_rtree_comp', 7, 6, 18,
  4, false, false, 4);
```

The next SQL SELECT statement shows an example of a query that returns the height, the number of internal and leaf nodes, and the average number of entries per node of the previously built R-tree (considering that only the aforementioned SQL statements were executed):

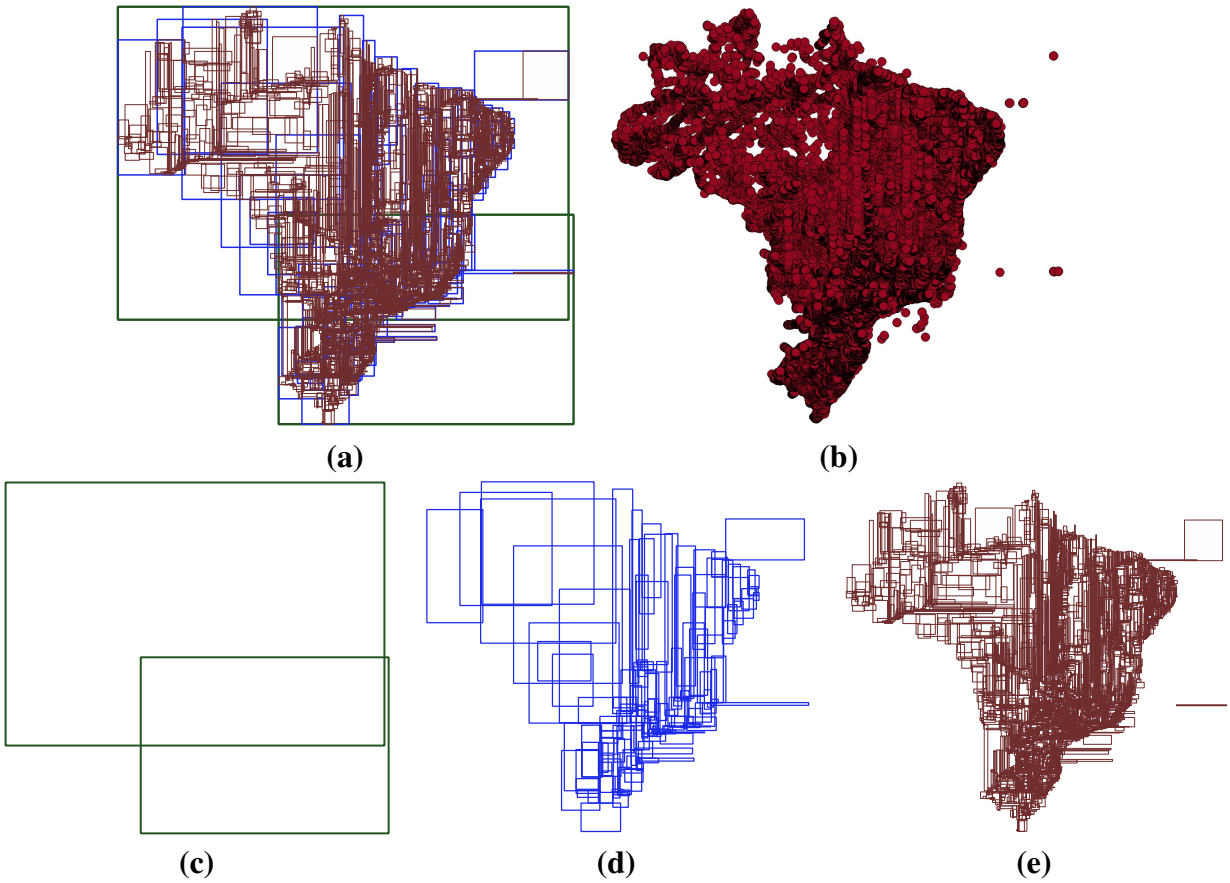


Figure 3 – Visualization of the MBRs of node entries of an R-tree. This R-tree (a) is built over the *brazil_points2017* (b) and has height equal to 3. The entries of each level, from the highest to the lowest level, are shown in (c), (d), and (e), respectively.

```
SELECT height, num_internal_nodes, num_leaf_nodes,
       avg_num_entries_pnode
FROM fds.execution e, fds.indexsnapshot is
WHERE e.pe_id = is.pe_id AND
  execution_name = 'Creating R-tree on brazil_points2017';
```

Further, the user is also able to visualize this index by retrieving rows from the PrintIndex. Every row in this table represents an entry of an index page, which has at least a pointer, height, and a geometry object representing its MBR. Figure 3 depicts the structure of the built R-tree (i.e., *linear_rtree_comp*) by using the QGIS. Different layers of geometries are employed to see the MBRs of each level of the tree (Figures 3c, d, and e). This visualization is particularly useful to graphically represent indices, such as for educational purposes.

Algorithm 3 depicts another workload, named *FT_QueryWorkload*. This workload has been used to understand the impact of SSDs on the spatial indexing context (CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a; CARNIEL *et al.*, 2017; CARNIEL; SILVA; CIFERRI, 2018; CARNIEL, 2018) and to measure the performance gains

Algorithm 3: The workload written in PL/pgSQL for executing spatial queries.

```
1  CREATE OR REPLACE FUNCTION FT_QueryWorkload(index_type int4,
     apath text, src_id int4, bc_id int4, sc_id int4, buf_id
     int4, file text default NULL)
2  RETURNS bool AS
3  $BODY$
4      DECLARE
5          query_windows RECORD;
6          tab VARCHAR;
7      BEGIN
8          EXECUTE 'SELECT table_name FROM fds.source WHERE src_id
            = $1' INTO tab USING src_id;
9          PERFORM FT_SetExecutionName('Query Worload - Index
            Creation');
10         PERFORM FT_CreateSpatialIndex(index_type, apath,
            src_id, bc_id, sc_id, buf_id, false, false, 1, 2,
            file);
11         PERFORM FT_SetExecutionName('Query Worload - Execution
            of IRQs with Query Windows of 0.001%');
12         FOR query_windows IN EXECUTE 'SELECT geom FROM
            generated_rectangle WHERE dataset = ''' || tab || '''
            AND type = ''intersection'' AND percentage = 0.001 AND
            is_correlated is TRUE'
13         LOOP
14             PERFORM FT_AQuerySpatialIndex(apath, 2,
                query_windows.geom, 1, 2, 1, 2, file);
15         PERFORM FT_SetExecutionName('Query Worload - Execution
            of IRQs with Query Windows of 0.01%');
16         FOR query_windows IN EXECUTE 'SELECT geom FROM
            generated_rectangle WHERE dataset = ''' || tab || '''
            AND type = ''intersection'' AND percentage = 0.01 AND
            is_correlated is TRUE'
17         LOOP
18             PERFORM FT_AQuerySpatialIndex(apath, 2,
                query_windows.geom, 1, 2, 1, 2, file);
19         PERFORM FT_SetExecutionName('Query Worload - Execution
            of IRQs with Query Windows of 0.1%');
20         FOR query_windows IN EXECUTE 'SELECT geom FROM
            generated_rectangle WHERE dataset = ''' || tab || '''
            AND type = ''intersection'' AND percentage = 0.1 AND
            is_correlated is TRUE'
21         LOOP
22             PERFORM FT_AQuerySpatialIndex(apath, 2,
                query_windows.geom, 1, 2, 1, 2, file);
23         RETURN true;
24 LANGUAGE plpgsql VOLATILE
25 COST 500;
```

of eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018). Its main goal is to build an index and to execute intersection range queries (IRQs). The employed query windows (stored in the relational table called *generated_rectangle*) are correlated to the indexed dataset (CARNIEL; CIFERRI; CIFERRI, 2017c) and are available at <https://github.com/accarniel/FESTIval/wiki/>. All inputs of this workload have the same meaning of the common inputs of the workload depicted in Algorithm 2. *FT_QueryWorkload* first gets the name of the dataset to be indexed in this workload (line 9). Then, the workload sets the name of the execution (line 10) to identify that the next operation is the index construction (line 11). After building the index, three different sets of IRQs are processed (lines 12 to 16, lines 17 to 21, and lines 22 to 26). To this end, three different sets of query windows are employed. Each set has 100 query windows with specific sizes of the area of the total extent of Brazil. These sizes are 0.001%, 0.01%, and 0.1%, respectively. Considering that the selectivity of a spatial query is the ratio of the number of returned objects and the total objects, these sets of query windows form spatial queries with low, medium, and high selectivity, respectively. Each execution of a spatial query is performed by the atomic operation *FT_AQuerySpatialIndex*. Finally, all statistical data is stored in a file located in the HDD (parameter *file* in lines 11, 15, 20, and 25) since this workload handles the spatial index file stored in an SSD. The use of this workload to validate a flash-aware spatial index is further discussed in the next section.

3.2 Method Validation: Employing FESTIval to measure the efficiency of eFIND

In this section, we show how *FT_QueryWorkload* (Algorithm 3) is employed to validate eFIND and how the statistical results can be extracted to measure its performance gains. eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018) is a generic approach that transforms a disk-based spatial index (e.g., the R-tree) into an efficient flash-aware spatial index (e.g., eFIND R-tree). The examples described in this section details a part of the experimental evaluation conducted by us in Carniel, Ciferri and Ciferri (2018). These experiments compare eFIND against FAST, which is the closest competitor to eFIND among existing approaches to implementing flash-aware spatial indices. eFIND and FAST are employed to port the traditional R-tree to SSDs, forming the following configurations: the *eFIND R-tree* and the *FAST R-tree*, respectively. The used spatial dataset (i.e., *src_id*) is the *brazil_buildings2017* (CARNIEL; CIFERRI; CIFERRI, 2017c), containing 1,485,866 regions that represent the buildings of Brazil. Both configurations employed an in-memory buffer of 512KB, log size of 10MB, and the flushing unit size equal to 5. The best parameter values were applied for the remaining specific parameters. This means that we did not vary specific parameter values for each configuration (i.e., *sc_id=50031* for the eFIND R-tree and *sc_id=1323* for the FAST R-tree). On the other hand, basic parameter values (i.e., *bc_id*) are varied to evaluate the eFIND R-tree and the FAST R-tree under page sizes from 2KB to 32KB. A generic buffer was

not employed in the experiments (i.e., *buf_id=1*). We conducted the experiments on a Kingston SSD V300 of 480GB.

The next SQL SELECT statements execute the workload depicted in Algorithm 3 to evaluate the performance of the eFIND R-tree and the FAST R-tree, respectively, using the page size equal to 4KB (i.e., *bc_id=53*). The files *efind_results.sql* and *fast_results.sql*, maintained in an HDD, are employed to store statistical data for the eFIND R-tree and the FAST R-tree, respectively:

```
SELECT FT_QueryWorkload(8, '/opt/efind_rtree1', 5, 53, 50031, 1,
  '/HDD/efind_results.sql');
SELECT FT_QueryWorkload(4, '/opt/fast_rtree1', 5, 53, 1323, 1,
  '/HDD/fast_results.sql');
```

Similar SQL SELECT statements are issued to evaluate the eFIND R-tree and the FAST R-tree for different page sizes. Each SQL SELECT statement is executed 5 times, varying the name of each index file, in order to calculate the average index time of the index construction and the execution of the IRQs. The cache of the PostgreSQL and the operating system is cleaned between the executions.

As previously discussed, collecting statistical data requires the execution of SQL SELECT statements on the FESTIval's data schema. For instance, the next query returns the average index time and its standard deviation to construct an eFIND R-tree for each employed page size:

```
SELECT b.page_size, avg(e.index_time), stddev(e.index_time)
FROM fds.basicconfiguration b, fds.specializedconfiguration sc,
     fds.indexconfiguration ic, fds.spatialindex si,
     fds.execution e
WHERE b.bc_id = ic.bc_id AND sc.sc_id = ic.sc_id AND
      ic.config_id = si.config_id AND si.idx_id = e.idx_id AND
      ic.sc_id = 50031 AND
      execution_name = 'Query Worload - Index Creation'
GROUP BY b.page_size
ORDER BY b.page_size;
```

The same structure of query can be also used for extracting performance results of the FAST R-tree.

In our analyzes, we collected the average of the total elapsed time required to execute each set of 100 IRQs. The following SQL SELECT statement is performed when collecting results for the query windows with 0.001% of the area of Brazil:

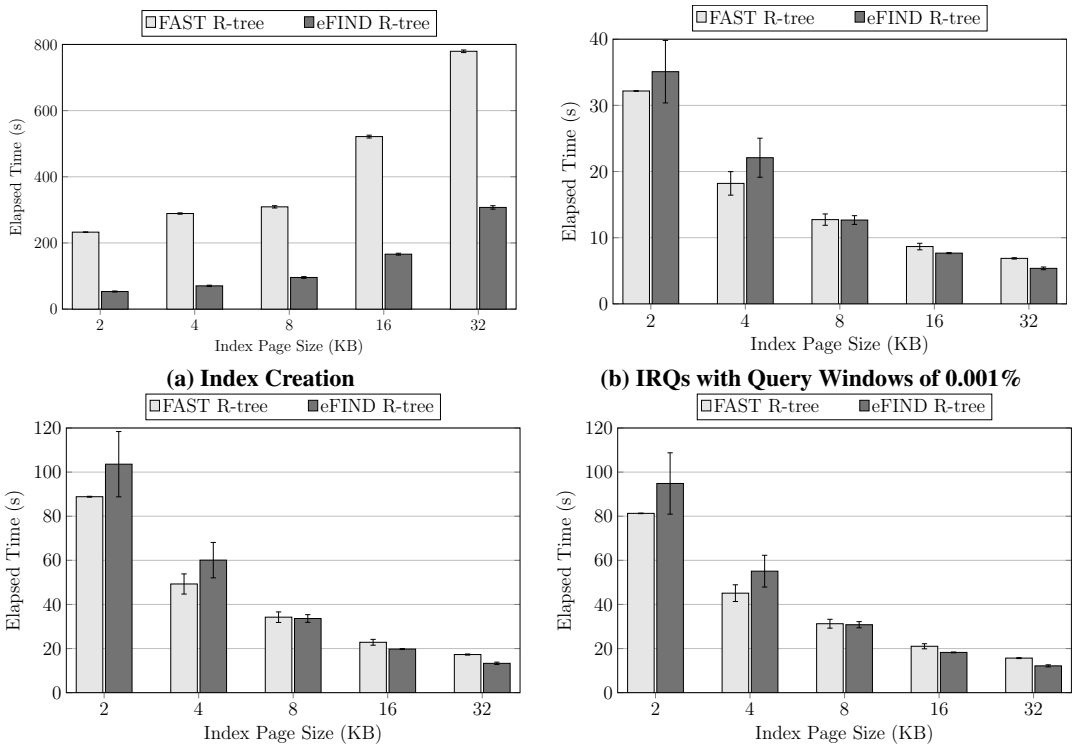
```
SELECT t.page_size, avg(t.s), stddev(t.s)
FROM (
        SELECT si.idx_name, b.page_size, sum(e.index_time) as s
        FROM fds.basicconfiguration b,
          fds.specializedconfiguration sc,
          fds.indexconfiguration ic,
          fds.spatialindex si, fds.execution e
        WHERE b.bc_id = ic.bc_id AND sc.sc_id = ic.sc_id AND
          ic.config_id = si.config_id AND
          si.idx_id = e.idx_id AND ic.sc_id = 50031 AND
          execution_name = 'Query Worload -
            Execution of IRQs with Query Windows of 0.001%'
        GROUP BY si.idx_name, b.page_size
) as t
GROUP BY t.page_size
ORDER BY t.page_size;
```

The subquery returns the sum of the index time required to process the 100 IRQs with query windows of 0.001% for each built index and page size. Note that we have 5 spatial indices with the same configurations created by the multiple executions of *FT_QueryWorkload*; each spatial index has a different name stored in *idx_name*. Then, the outer query returns the average and the standard deviation of the total elapsed time for each page size. A similar approach is used to extract the performance results of the FAST R-tree.

By using the returned results of the SQL SELECT statements, we can employ data analytics tools (Figure 1) to analyze the performance results. For instance, we can visualize the results by using bar graphs, where the x-axis is the first returned column (i.e., *page_size*) and the y-axis is the average time with the errors bar for the standard deviation, as shown in Figure 4. In our experiments, the eFIND R-tree overcame the FAST R-tree when building indices in all employed page sizes. Its performance gains were very expressive, ranging from 60% to 77% for the Kingston SSD (Figure 4a). As for the query processing (Figures 4b, c, and d), the eFIND R-tree provided the best performance only for larger pages, showing gains of 22% and 23% for the index pages of 16KB and 32KB, respectively. The efficiency of eFIND comes from the use of a set of design goals specified to fully exploit SSD performance (CARNIEL; CIFERRI; CIFERRI, 2017b).

The positive characteristics of FESTIval allow its use in distinct experimental evaluations, such as experiments for analyzing the impact of SSDs in the spatial indexing context (CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a), and experiments for validating new proposals of spatial indexing on SSDs (e.g., eFIND-based indices (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018)). Moreover, external data

analytics tools can access the FESTIval's data schema to generate different types of graphics, to plot maps, and to process statistical data in mathematical models. These aspects are very important when benchmarking spatial indexing structures in spatial database systems and GIS.



(a) Index Creation

(b) IRQs with Query Windows of 0.001%

(c) IRQs with Query Windows of 0.01%

(b) IRQs with Query Windows of 0.1%

Figure 4 – FESTIval was very useful to measure the performance gains of the eFIND R-tree, which reduced the time spent when building spatial indices (a) and when processing IRQs (b, c, and d).

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work has also been supported by the Brazilian federal research agency CNPq, as well as by the São Paulo Research Foundation (FAPESP). Anderson C. Carniel has been supported by the grant #2015/26687-8, FAPESP. Ricardo R. Ciferri has been supported by the grant #311868/2015-0, CNPq. Cristina D. A. has been supported by the grant #2018/22277-8, FAPESP.

3.3 Supplementary material and/or Additional information

3.3.1 Installation

FESTIval has the following dependencies: PostgreSQL 9.5 (or later), and PostGIS 2.2.1 (or later) with its dependencies. Further, a main dependency of PostGIS and, consequently of FESTIval, is the GEOS library. We strongly recommend the most recent version of GEOS for both systems. After configuring the Makefile of FESTIval, the user can install it by issuing the following commands in the terminal:

```
sudo make all
sudo make install
```

Once installed, FESTIval should be enabled in a PostgreSQL database for its usage. To this end, the following SQL CREATE EXTENSION statement should be issued:

```
CREATE EXTENSION festival;
```

The FESTIval's data schema is automatically created by FESTIval when it is enabled in a PostgreSQL database. The default values of the FESTIval's data schema can be loaded by executing the following command:

```
psql -U user -d database -vfestivaldir=/f_path/
     -f /f_path/festival-inserts.sql
```

where *user* is the user name of the PostgreSQL, *database* is the database name in which FESTIval is enabled, and */f_path/* is the full path of the FESTIval's source code.

3.3.2 Previous version of FESTIval

The first version of FESTIval was described in Carniel, Ciferri and Ciferri (2016a). The version introduced in this article greatly extends the previous version of FESTIval as follows. First, we extend FESTIval to provide support for other indexing structures, such as the Hilbert R-tree and flash-aware spatial indices. Second, FESTIval now also provides support for generic buffers, such as the LRU and the 2Q. Third, a common design was introduced to provide the creation of user-defined workloads. Fourth, spatial indices can be also evaluated on emulated flash memories. Finally, the current version of FESTIval also allows to manage refined statistical data, such as the order of read and writes.

We are constantly improving the source code of FESTIval. These extensions allow us to continuously use FESTIval in research papers, such as in Carniel, Ciferri and Ciferri (2016c), Carniel, Ciferri and Ciferri (2017a), Carniel, Ciferri and Ciferri (2017b), Carniel, Ciferri and Ciferri (2018), Carniel *et al.* (2017), Carniel, Silva and Ciferri (2018), and Carniel (2018).

CHAPTER

4

ANALYZING THE PERFORMANCE OF SPATIAL INDICES ON HARD DISK DRIVES AND FLASH-BASED SOLID STATE DRIVES

This Chapter attaches the following paper (CARNIEL; CIFERRI; CIFERRI, 2017a):

- Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. Analyzing the Performance of Spatial Indices on Hard Disk Drives and Flash-based Solid State Drives. Journal of Information and Data Management 8 (1), p. 34-49, 2017.

This paper is an extended version of the following work (CARNIEL; CIFERRI; CIFERRI, 2016c):

- Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. The Performance Relation of Spatial Indexing on Hard Disk Drives and Solid State Drives. In Proceedings of the XVII Brazilian Symposium on GeoInformatics (GEOINFO 2016), p. 263-174, 2016.

Abstract

Spatial database systems and Geographic Information Systems frequently employ *disk-based spatial indices* like the R-tree and the R*-tree to speed up the processing of spatial queries, such as *spatial range queries*. Commonly, these indices are originally designed for *Hard Disk Drives* (HDDs) and thus, they take into account the slow mechanical access and the cost of search and rotational delay of magnetic disks. On the other hand, *flash-based Solid State Drives* (SSDs) have widely been adopted in local data centers and cloud data centers like the Microsoft Azure environment. Because of intrinsic characteristics of SSDs like the erase-before-update property and the asymmetric costs between reads and writes, the impact of spatial indexing on SSDs

needs to be studied. In this article, we conduct an experimental evaluation in order to analyze the performance relation of spatial indexing on HDDs and SSDs. For this purpose, we execute our experiments on a local server equipped with an HDD and an SSD, as well as on virtual machines equipped with HDDs and SSDs and allocated in the Microsoft Azure environment. 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 intrinsic characteristics of SSDs. This means that a spatial index that showed a good performance on an HDD often did not show the same good performance on an SSD.

4.1 Introduction

Several advanced applications like agricultural systems, urban planning applications, and public transportation systems make use of geometric or spatial information in order to represent spatial phenomena. Commonly, these applications require specialized systems to manage, store, and retrieve spatial phenomena. *Spatial database systems* and *Geographic Information Systems* (GIS) fulfill these requirements allowing the processing of *spatial queries* on spatial objects (GÜTING, 1994). For instance, a spatial selection that finds all the roads intersecting the Sao Paulo state. Another example is a spatial range query that returns all the buildings overlapping a given rectangular object, called query window. Due to the expensive processing of topological predicates (e.g., intersects, overlap), spatial database systems and GIS widely employ *spatial indices* to speed up the processing of spatial queries (GAEDE; GÜNTHER, 1998). Frequently, hierarchical indices like the R-tree (GUTTMAN, 1984) and the R*-tree (BECKMANN *et al.*, 1990) are employed.

In general, these indices are designed to be handled in *Hard Disk Drives* (HDD) and thus, they take into account the slow mechanical access and the cost of search and rotational delay of magnetic disks. We term them as *disk-based spatial indices*. On the other hand, *flash-based Solid State Drives* (SSDs) have widely been used in many applications (KRYDER; KIM, 2009; MITTAL; VETTER, 2016) because of 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. As a result, SSDs are being used as the main storage device in local data centers and in cloud data centers, such as the Azure Premium Storage of the Microsoft Azure environment[1].

However, SSDs have intrinsic characteristics (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; MITTAL; VETTER, 2016) that may affect the performance of many spatial database applications. The main intrinsic characteristic is the asymmetric cost between reads and writes where a write requires more time and power consumption than a read. Another characteristic is the erase-before-update property where an erase operation is performed in order

[1] <https://azure.microsoft.com>

to update the content of a flash page, leading to an expensive operation.

Because of the intrinsic characteristics of SSDs, spatial database applications increasingly require the performance evaluation of disk-based spatial indices on SSDs to analyze if there is a relation with the well-known performance obtained on HDDs. As a result, we can analyze how similar are the performance behavior of spatial indices on such devices. There are few approaches (EMRICH *et al.*, 2010; FEVGAS; BOZANIS, 2015) that conduct experimental evaluations of spatial indices on SSDs. There are also approaches (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012; SARWAT *et al.*, 2013; JIN *et al.*, 2015) that propose *flash-aware spatial indices* by adapting the R-tree in order to deal with the intrinsic characteristics of SSDs. The performance of these indices is measured through experimental evaluations. However, the experiments conducted on these approaches face several problems, such as the lack of an analysis of the performance relation of spatial indexing on HDDs and SSDs, the lack of focus on the spatial query processing, and the lack of an analysis with respect to the use of different parameter values of spatial indices on SSDs and HDDs.

To fill these gaps, this article has the following objectives. First, we aim to correlate the performance results of spatial indices managed on SSDs and HDDs. The goal is 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. For this purpose, we conduct an extensive experimental evaluation on SSDs and HDDs using a local server and using virtual machines maintained in the Microsoft Azure environment. Second, we aim to verify the performance impact of different parameter values in the evaluated spatial indices. For this purpose, we analyze the performance relation between the used parameter values and the obtained results. Finally, we aim to analyze if the same set of parameter values that shows a good performance to build spatial indices also results in good performance to process spatial queries.

This article is organized as follows. Section 4.2 surveys related work. Section 4.3 summarizes underlying concepts from spatial indexing. Section 4.4 briefly describes intrinsic characteristics of SSDs. Section 4.5 details our experiments and discusses the obtained results. Section 4.6 concludes the paper and presents future work.

4.2 Related Work

There are few approaches that conduct experimental evaluations of disk-based spatial indices on SSDs (EMRICH *et al.*, 2010; FEVGAS; BOZANIS, 2015). Further, there also approaches that propose flash-aware spatial indices based on the R-tree (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012; SARWAT *et al.*, 2013; JIN *et al.*, 2015). We classify these approaches according to the following characteristics: (i) the performance evaluation of spatial indices on HDDs and SSDs, (ii) the variety of spatial indices used in experiments, (iii) the parameter values considered in the indices, and (iv) the types of workloads

employed in the experiments.

Regarding the first characteristic, almost all the approaches do not evaluate the performance of spatial indices 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) is the same on HDDs and SSDs. Only Emrich *et al.* (2010) conduct experiments considering both storage systems, HDDs and SSDs, to evaluate the performance of the R*-tree in the computation of k-nearest neighbor queries. In contrast to Emrich *et al.* (2010), we conduct experiments by taking into account other spatial indices, and other kinds of workloads.

Regarding the second characteristic, the majority of the approaches (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012; SARWAT *et al.*, 2013; JIN *et al.*, 2015) employ the R-tree as a baseline in the experiments, while the remaining approaches (EMRICH *et al.*, 2010; FEVGAS; BOZANIS, 2015) employ the R*-tree as a baseline. However, it is important to consider both the R-tree and the R*-tree in the same experiment because of their good performance reported in the literature (GAEDE; GÜNTHER, 1998). In addition, many approaches (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012; SARWAT *et al.*, 2013; FEVGAS; BOZANIS, 2015; JIN *et al.*, 2015) also propose flash-aware spatial indices and thus, they conduct experimental evaluations to measure the performance of the proposed indices. However, the lack of studies about the performance of well-known spatial indices on SSDs can hide findings that could improve the performance of the proposed flash-aware spatial indices.

Regarding the third characteristic, the main parameter analyzed in the approaches is the buffer size employed in the main memory. However, parameterization plays an important role in spatial indexing and the variation of specific parameter values of a spatial index can impact on its performance (see Section 4.3). For instance, a well-known parameter used on hierarchical structures is the page (node) size, which is ignored in the existing experiments. Thus, there is a lack of analysis of the impact of the page size used in a spatial index handled on HDDs and SSDs.

Regarding the fourth characteristic, the existing experiments only focus on insertion and deletion operations since random writes are expensive operations on SSDs. However, it is also important to verify the performance of spatial queries since this is a very common operation in spatial database systems and GIS. Unfortunately, there is a lack of studies for verifying the impact of the spatial selectivity on the spatial query processing using indices stored on SSDs.

As a conclusion, we can note that there is a lack of experimental evaluations that understand and correlate the performance of spatial indices managed on HDDs and SSDs. In this article, we conduct experiments considering different spatial indices with different parameter values on both storage systems. For this purpose, we consider disk-based spatial indices (i.e., the R-tree and the R*-tree) and flash-aware spatial indices, which are summarized in the next section.

We extend our previous work in Carniel, Ciferri and Ciferri (2016c) by performing experiments in virtual machines equipped with HDDs and SSDs allocated in the Microsoft Azure environment, which is a cloud data center. Further, we present details of our experimental setup, as well as discussions regarding newly obtained results. We also compare the performance of the spatial indices managed on the different environments in order to provide guidelines for spatial database applications.

4.3 Spatial Indexing

In general, hierarchical structures are employed to index spatial objects. Due to the complexity to store and handle complex shapes, spatial objects are approximated to Minimum Bounding Rectangles (MBR). The MBR of a spatial object is a box that minimally encloses all the points of the spatial object. Since MBRs are employed to represent spatial objects, two steps are required in order to process spatial queries (GAEDE; GÜNTHER, 1998): *filter step*, and *refinement step*. The filter step employs a spatial index to discard portions of data where the answer of a spatial query cannot be found and returns the possible candidates that answer the spatial query. Then, the refinement step checks whether each object really belongs to the final answer or not. Thus, this step is a time-expensive operation since spatial objects are accessed from the storage device and complex algorithms from the geometry computational are employed to compute topological predicates (GAEDE; GÜNTHER, 1998).

The R-tree (GUTTMAN, 1984) is a common hierarchical spatial index used in spatial database systems and GIS. It is composed of internal and leaf nodes. A leaf node comprises entries in the format (mbr_o, p_o), where mbr_o refers to the MBR of the spatial object o and p_o is a pointer that guarantees the direct access to o. An internal node comprises entries in the format (mbr_c, p_c), where mbr_c is the MBR that encompasses all the entries of the child node c and p_c is a pointer to the child node c. The minimum and maximum capacity of internal and leaf nodes are parameters determined respectively by m and M such that $m \leq M/2$. If the maximum capacity of a node is achieved by an insertion, a split operation is performed. There are different split algorithms with different complexities and results, such as the linear split and the quadratic split.

The R*-tree (BECKMANN *et al.*, 1990) is a well-known variant of the R-tree that applies other aspects to index spatial objects. For instance, the utilization of overlapping areas among the entries in internal nodes, the redistribution of entries of overflowed nodes to maximize the storage utilization, and the utilization of margin of nodes. Hence, the R*-tree modifies the insertion algorithm of the R-tree in order to take into account these aspects. Further, it employs another split algorithm and applies a reinsertion policy to decrease the number of split operations.

With the increasing use of SSDs in applications, flash-aware spatial indices have been proposed in the literature. The RFTL (WU; CHANG; KUO, 2003), the ARFTL (PAWLIK; MACYNA, 2012), the LCR-tree (LV *et al.*, 2011b), and the FOR-tree (JIN *et al.*, 2015) are

straightforward adaptations of the R-tree. A generic framework, called FAST (SARWAT *et al.*, 2013), is proposed to convert a disk-based spatial index into a flash-aware spatial index. FAST does not change the index structure but only changes the way in which the nodes are written to the SSD. Thus, we are able to define *FAST-based spatial indices*. For instance, the FAST R-tree and the FAST R*-tree are based on the R-tree and R*-tree, respectively.

Despite the particular characteristic of each flash-aware spatial index, they often make use of a buffer in the main memory that stores the most recent modifications of nodes instead of applying directly the modifications to the SSD. 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. While there are indices (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012) that flush all the modifications leading to an expensive flushing operation, FAST and the FOR-tree employ a *flushing policy* that chooses a set of nodes with modifications to be flushed. This set is called *flushing unit*.

Parameterization plays an important role in spatial indexing (GAEDE; GÜNTHER, 1998). Examples of typical parameters are the page (node) size and the minimum and the maximum number of entries of leaf and internal nodes. In addition, each index may include specific parameters according to its design. For instance, the reinsertion percentage of the R*-tree. Moreover, flash-aware spatial indices introduced several new parameters, such as the buffer and flushing unit sizes, which impact directly on the performance of flushing operations. Hence, there is a significant performance impact if *different* parameter values are used in a spatial index under *different* storage systems.

4.4 Flash-based Solid State Drives

Flash-based Solid State Drives (SSDs) have been very popular in many applications (KRYDER; KIM, 2009; MITTAL; VETTER, 2016). Table 8 shows a comparison of the HDD and SSD used in our experiments in the local server (Section 4.5). SSDs have intrinsic characteristics that introduce several implications (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013) for the management of data in such devices.

SSDs store data in a block-oriented model, which means that a fixed number of flash pages composes a flash block. Commonly, the flash page size is 4KB and the flash block size is 256KB (CHEN; KOUFATY; ZHANG, 2009; MITTAL; VETTER, 2016). SSDs provide the following operations: erase, read, and write (program) (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). 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. Read and write are page level operations with asymmetric costs. A read requires much less time and power consumption than a write (see Table 8). A write is only able to change the bits from 1 to 0 in an erased block. Hence, to write in a previously written block (i.e., update), an *erase-before-update*

Table 8 – Comparison of the HDD[2] and the SSD[3] used in the experiments conducted in the local server.

	4KB Random Transfers[4]		Power Consumption[5]		Endurance[6]
	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	$10^4 - 10^5$

operation is performed leading to a 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 SSDs is their lower endurance than HDDs (see Table 8). Endurance refers to the maximum number of writes and erases in a block before its inability to save new data.

A Flash Translation Layer (FTL) (CHUNG *et al.*, 2009) is employed to avoid erase-before-update operations, to facilitate the integration of SSDs with current systems, and to improve the endurance of SSDs. For this purpose, FTL provides an interface that allows operating systems to use SSDs as a usual storage device and only enables reads and writes for application layers. Further, FTL maps physical page addresses of the SSD 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 a page ready 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; 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 performed when space is required and there is a great number of invalid 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 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).

4.5 Performance Evaluation

We conduct an experimental evaluation to analyze the performance of spatial indices on HDDs and SSDs. Section 4.5.1 details the experimental setup. The obtained results related to the spatial index construction and spatial query processing are discussed in Sections 4.5.2 and 4.5.3, respectively.

2 <https://support.wdc.com/product.aspx?ID=608&lang=en>
3 <https://www.kingston.com/us/ssd/consumer/sv300s3>
4 Measured by Iometer (<http://www.iometer.org/>).
5 According to the manufactures[2,3].
6 According to Mittal and Vetter (2016).

Table 9 – Configurations employed 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 10 – 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	1,820

4.5.1 Experimental Setup

We used a real dataset extracted from the OpenStreetMap[7], which consisted of 534,926 complex regions possibly 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 have extended the first version of FESTIval (CARNIEL; CIFERRI; CIFERRI, 2016a) by adding new functionalities like the creation of user-defined workloads. FESTIval[8] is an open-source PostgreSQL extension that enables the benchmarking of disk-based and flash-aware spatial indices with different parameter values under different storage systems by issuing workloads in a unique environment. We used FESTIval to compare the following disk-based spatial indices: the R-tree and the R*-tree. Further, we evaluate the following flash-aware spatial indices: the FAST R-tree and the FAST R*-tree, which are FAST-based spatial indices (Section 4.3). Since we have updated the version of the FESTIval used in our previous work (CARNIEL; CIFERRI; CIFERRI, 2016c), in this article we have re-executed all the experiments.

FESTIval also enables the configuration of a spatial index by using specific and generic parameter values. Specific parameters determine the configuration of a specific spatial index

[7] <http://www.openstreetmap.org/>
[8] <http://gbd.dc.ufscar.br/festival/>

only. For the R-tree, we varied its split algorithm by considering the linear and quadratic splits. For the R*-tree, we varied the reinsertion percentage (RP) since it impacts on the number of writes. Further, based on the recommendations for the R*-tree (BECKMANN *et al.*, 1990), we considered the close reinsert policy. For the FAST-based spatial indices, we considered the FAST* flushing policy (FP) (SARWAT *et al.*, 2013). As a result, we employed the configurations depicted in Table 9. We also studied the effect of the variation of the buffer and the flushing unit size. For all FAST-based indices, we executed our experiments considering the following buffer sizes: 128KB, 256KB, and 512KB. We have reported results only for the buffer equal to 512KB since it showed similar performance behavior compared to the other buffer sizes. In addition, we varied the flushing unit size from 1 to 5, which contributed to analyzing the impact of this parameter in the flash-aware spatial indexing.

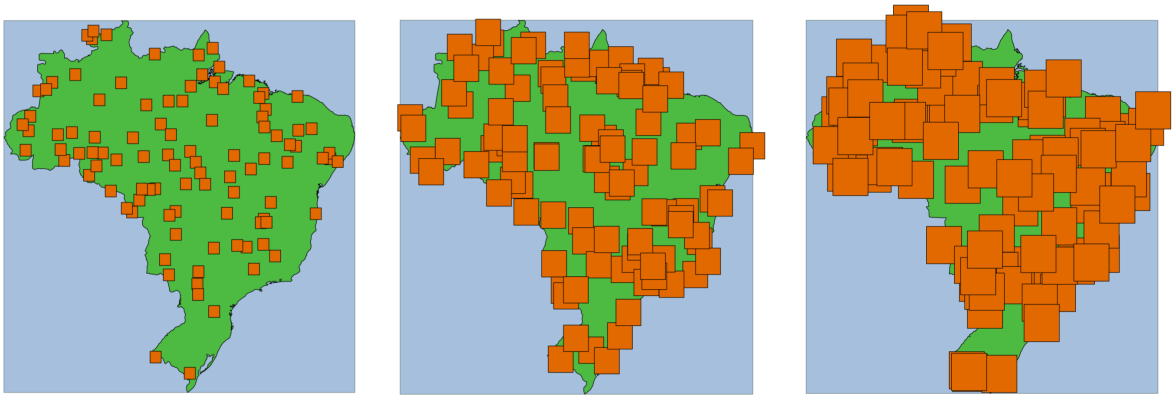
We also varied generic parameters, which can be applied for any spatial index of FESTIval. For instance, the size in bytes of a node (i.e., page size), and the minimum and the maximum number of entries of a node. Further, we considered the DIRECT I/O to avoid operational system caching in reads and writes. Table 10 shows the page sizes, with its minimum and maximum occupancy, employed for all the configurations in Table 9.

We executed two workloads defined as follows. The first workload focused on the *index construction* and thus, we collected the elapsed time in seconds for creating a spatial index (see Section 4.5.2). 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* (see Section 4.5.3). We executed *intersection range queries* (IRQ) (GAEDE; GÜNTHER, 1998) because of its wide utilization in many applications. This kind of query returns from a dataset D, a set of spatial objects R that intersects a given rectangular-shaped query window QW, i.e., $R = \{o | o \in D \wedge intersects(o, QW) = true\}$. We synthetically generated three sets of query windows, which are depicted in Figure 5. Each set was composed of 100 query windows, which had a $x\%$ of the area of the total extent of Brazil (i.e., the MBR of Brazil). The first set had 0.1% (Figure 5a), the second set had 0.5% (Figure 5b), and the third set had 1% (Figure 5c). Considering selectivity as the proportion between the number of returned elements in the query and the number of total elements in the dataset, these query windows composed IRQs with low, medium, and high selectivity, 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 10 times each IRQ, collected the average elapsed time of its execution, and then calculated the sum of the average elapsed times of the 100 IRQs. We flushed the system cache after each execution. Further, we consider the elapsed time with respect to the index time.

The experiments were conducted on two different hardware environments. The first environment was a local server equipped with an Intel® Core™ i7-4770 with a frequency of



(a) (b) (c)

Figure 5 – Query windows employed in the processing of IRQs: query windows with 0.1% (a), query windows with 0.5% (b), and query windows with 1% (c).

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. Hence, this environment gives us experiment results under controlled conditions since we executed the workloads in a local server to avoid network latency.

The second environment was composed of two virtual machines maintained in the Microsoft Azure in order to analyze the efficiency of spatial indices in cloud servers, which are used by many applications. The first virtual machine had the Standard A6 size (4 cores, 28GB of main memory) with an HDD of 500GB. The second virtual machine had the Standard DS12 v2 size (4 cores, 28GB of main memory) with an SSD of 500GB. These two virtual machines and their storage devices were allocated in the South Central US. This environment is not as much controlled as the first environment because of the characteristics of the Microsoft Azure, such as the lack of guarantee that the storage device is in the same physical place than the central processing unit. However, this environment gives us the chance to analyze if the spatial indexing on HDDs and SSDs employed in such virtual machines have a performance behavior similar to the found in a controlled environment. For instance, we can find out if a spatial index that shows a good performance on the SSD in the first environment also shows a good performance on a virtual machine on a cloud data center with SSD. This would help applications hosted in the cloud to make use of the better spatial index.

The software employed in all the environments are Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, and PostGIS 2.2.0.

4.5.2 Spatial Index Construction

Here, we detail the obtained results for creating spatial indices. Sections 4.5.2.1 and 4.5.2.2 discuss the results for the local server and for the virtual machines of the Microsoft Azure, respectively. Correlations are made in Section 4.5.2.3.

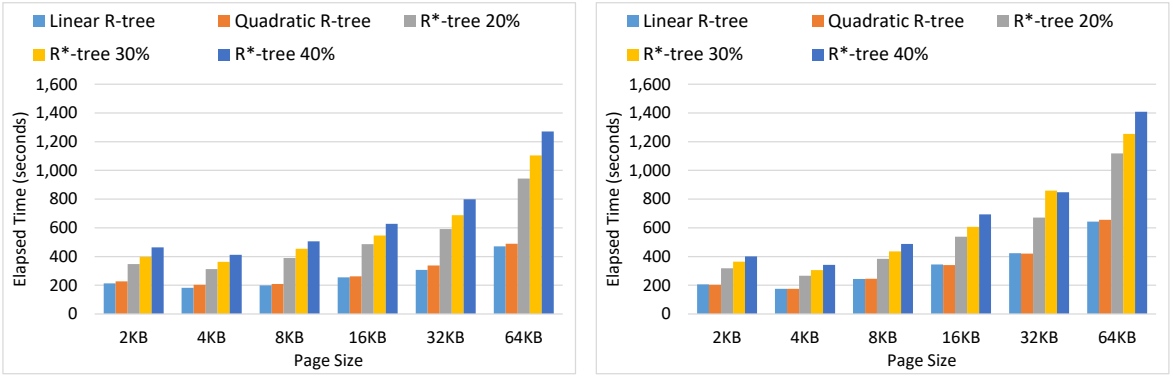
4.5.2.1 Execution on the Local Server

Figure 6 depicts the elapsed time for creating disk-based spatial indices on the HDD and SSD. The page size equal to 4KB gathered the best performance results for all spatial indices on both storage systems, indicating that this page size should be used for creating spatial indices. Considering this page size, the SSD showed better performance results than the HDD. The SSD showed performance gains between 4% and 16% over the HDD. A performance gain is a percentage that shows how much one configuration is more efficient than another configuration. On the other hand, for the page sizes equal to 16KB, 32KB, and 64KB, we obtained best performance results by using the HDD. For these page sizes, the SSD introduced a performance loss ranging from 6% to 38% over the HDD. The main reason for this performance degradation is due to the interleaved writes and reads that the index construction performs on the storage device. Thus, workloads that mix many writes and reads tend to degenerate the performance of SSDs, such as discussed by Lee and Moon (2007) and Chen, Koufaty and Zhang (2009). 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 demonstrated that it is important to take into account the intrinsic characteristics of SSDs to achieve good performance in these memories. Thus, the redesign of spatial indices originally designed for HDDs is needed, such as the framework provided by FAST that transforms a disk-based spatial index into a flash-aware spatial index.

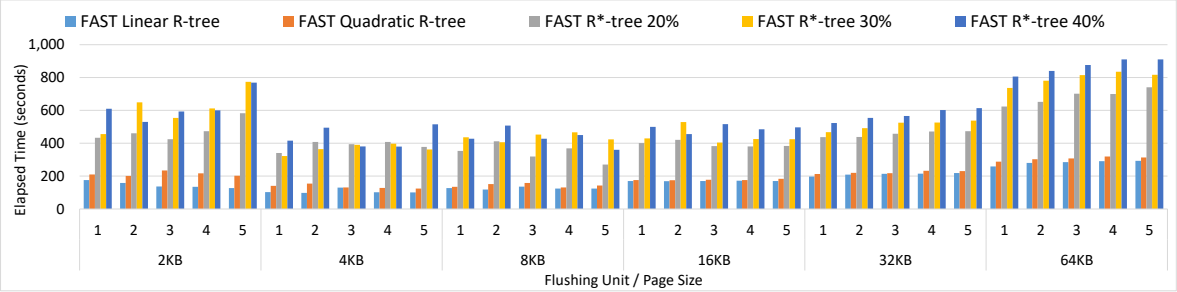
Figure 7 shows the performance results for constructing spatial indices by using FAST. In general, FAST-based spatial indices improved the performance of their counterparts on both storage systems. For instance, the *FAST Linear R-tree* (Figure 7) showed lower elapsed times than the *Linear R-tree* (Figure 6) on both storage systems. The *FAST Linear R-tree* imposed performance gains ranging from 22% to 40% on the SSD. Although FAST is designed for SSDs, it also showed performance gains varying from 34% to 46% on the HDD.

An interesting fact was that the performance behavior for the construction of the FAST-

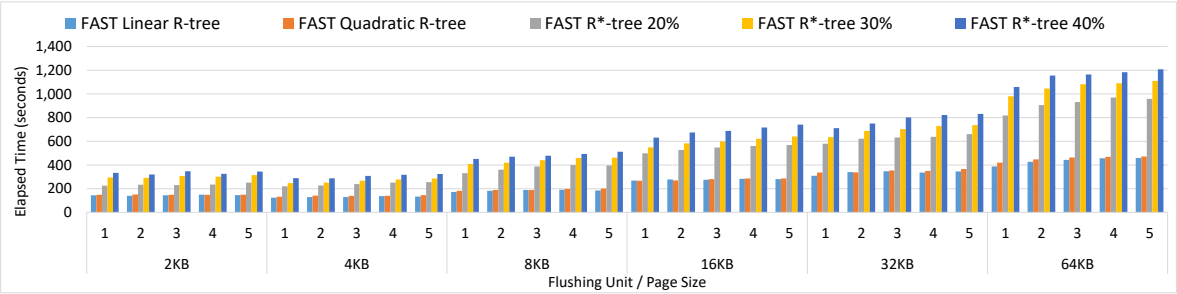


(performed on the HDD) (performed on the SSD)

Figure 6 – Performance results for creating disk-based spatial indices on the local server.



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



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

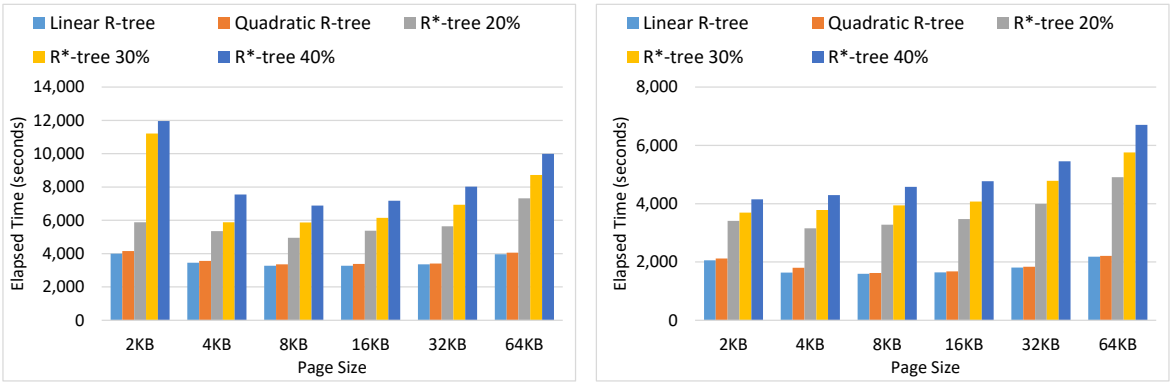
Figure 7 – Performance results for creating FAST-based spatial indices on the local server.

based indices on the SSD was different from the performance behavior obtained in the HDD. We emphasize two main differences. The first difference was that the majority of the spatial indices showed best performance results on the HDD by using the page size equal to 4KB and the flushing unit size equal to 5. On the other hand, all the spatial indices showed the best performance results on the SSD by employing 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 HDD was faster than the SSD since big writes turned out problematic on the SSD.

4.5.2.2 *Execution on the Microsoft Azure*

Figure 8 depicts the elapsed time for creating disk-based spatial indices on the virtual machines of the Microsoft Azure. Note that we use different scales in the graphics to better compare the performance results. Clearly, the construction of spatial indices on the virtual machine with SSD required much less processing time than the construction of the same spatial indices on the virtual machine with HDD. The main reason is that the Microsoft Azure offers a special attention for the use of SSDs, denominated as *Azure Premium Storage*[9]. It has a maximum throughput of 200MB/s and maximum of 5,000 input/output operations per second (IOPS), while a common storage device (i.e., HDD) has a maximum throughput of 60MB/s and

[9] <https://docs.microsoft.com/en-us/azure/storage/storage-about-disks-and-vhds-linux>



(performed on the virtual machine with HDD) (performed on the virtual machine with SSD)

Figure 8 – Performance results for creating disk-based spatial indices on the virtual machines of the Microsoft Azure.

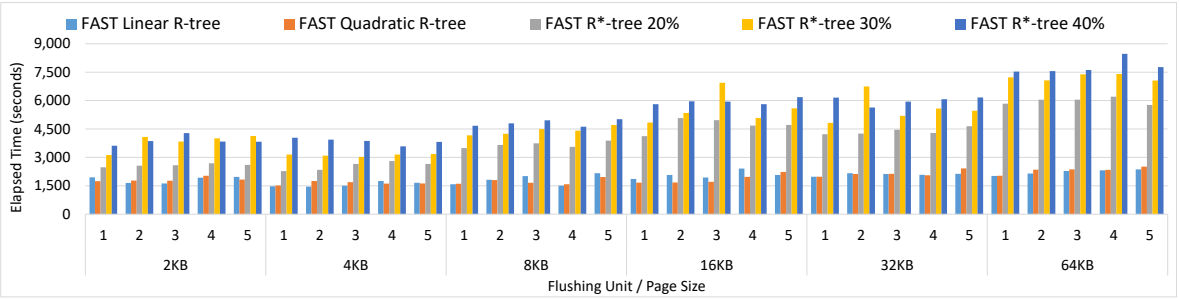
a maximum of 500IOPS. It contributed to obtaining performance gains varying from 29% to 67% for constructing spatial indices on the virtual machine with SSD. In general, the page size equal to 8KB showed the best elapsed times for both the virtual machines.

Although the expressive performance gains of the SSD over the HDD, these gains were yet improved when constructing spatial indices by using FAST. Figure 9 depicts the performance results obtained to construct FAST-based spatial indices on the virtual machines of the Microsoft Azure. The performance gains imposed by FAST ranged from 19% to 72% for constructing indices on the HDD. These gains were increased for constructing indices on the SSD, varying from 82% to 93%. It shows that FAST improved the performance of this kind of workload on the virtual machine using SSD because it takes into account intrinsic characteristics of these storage devices. Thus, we note that these intrinsic characteristics are still present in the cloud data center, such as the Microsoft Azure environment.

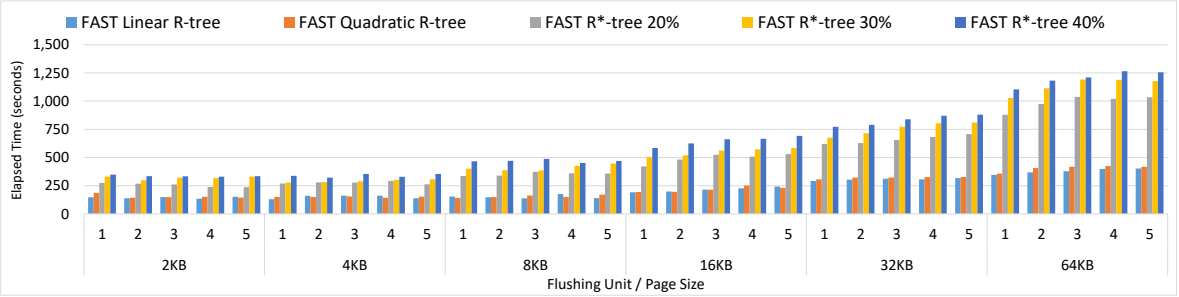
4.5.2.3 Comparing the Obtained Results

The main difference among the results was that the disk-based spatial indices executed on the local server with HDD showed, in general, better performance results than the execution on the local server with SSD. This performance behavior did not happen in the virtual machines of the Microsoft Azure. The main reason is that this cloud environment performs a special treatment for the SSDs, where only specific virtual machines are able to use this kind of storage system.

On the other hand, we can enumerate the following similarities among the results. First, since FAST takes into account the intrinsic characteristics of SSDs, it showed the best performance results for the environments using SSDs. It also improved the performance results for the environments using HDDs due to the use of a buffer in the main memory. Second, in general, the page size equal to 4KB gathered the best performance results for the FAST-based spatial indices on the SSD and HDD. Third, the use of page sizes greater than 4KB required an



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



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

Figure 9 – Performance results for creating FAST-based spatial indices on the virtual machines of the Microsoft Azure.

increasing elapsed time to construct a FAST-based index in the SSD, as the page size and flushing unit size also increases. Finally, in both the environments, the *FAST Linear R-tree* showed the fastest elapsed times, while the *Linear R-tree* was the best configuration among the disk-based spatial indices.

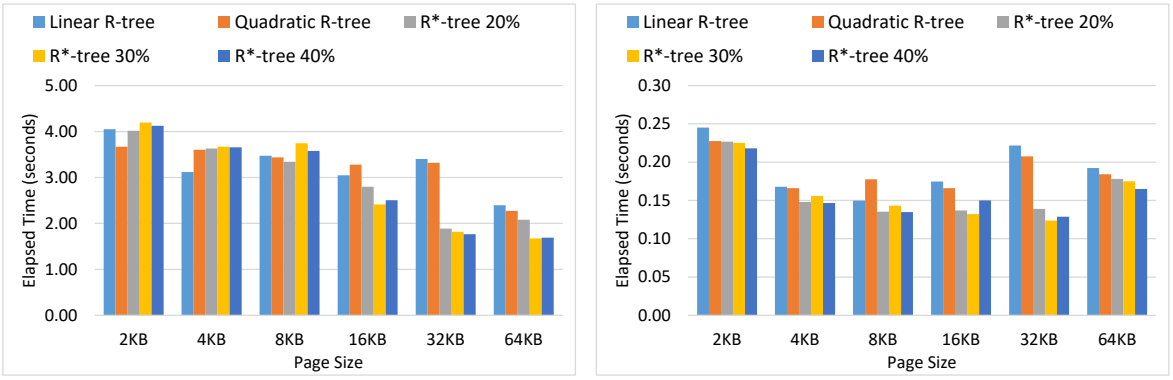
4.5.3 Spatial Query Processing

This section details the obtained results for spatial query processing on the local server (Section 4.5.3.1) and on the virtual machines of the Microsoft Azure (Section 4.5.3.2). Finally, these results are correlated in Section 4.5.3.3.

4.5.3.1 Execution on the Local Server

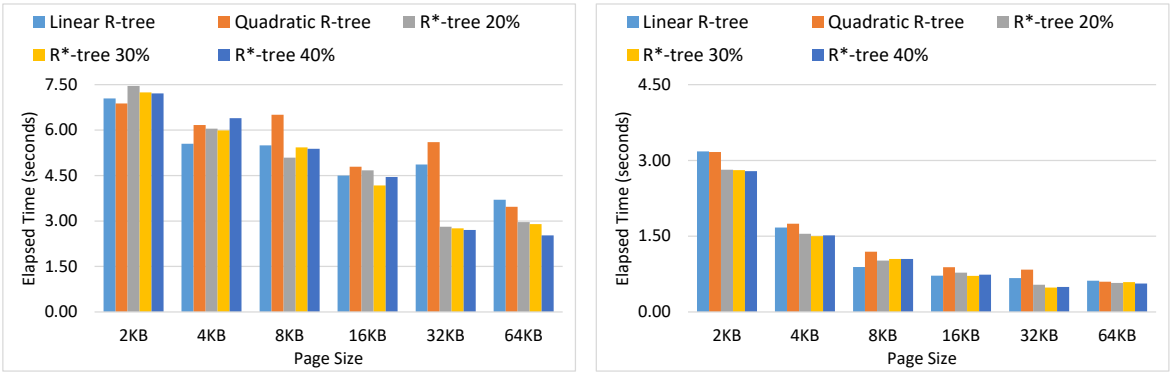
Figures 10, 11, and 12 depict the obtained results for processing spatial queries by employing IRQs with 0.1%, 0.5%, and 1%, respectively. In these figures, we use a different scale to report the results for the SSD. Further, we considered only configurations based on the R-tree and the R*-tree (Table 9) since FAST does not change the structure of a spatial index (e.g., the *Linear R-tree* using the page size equal to 4KB is the same as the *FAST Linear R-tree* using the page size equal to 4KB). Thus, we are able to compare the performance behavior of the underlying index that can organize spatially the data in different forms.

Clearly, the performance of spatial query processing on the SSD overcame the HDD in all



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

Figure 10 – Performance results to process spatial queries on the local server, considering IRQs with 0.1%.

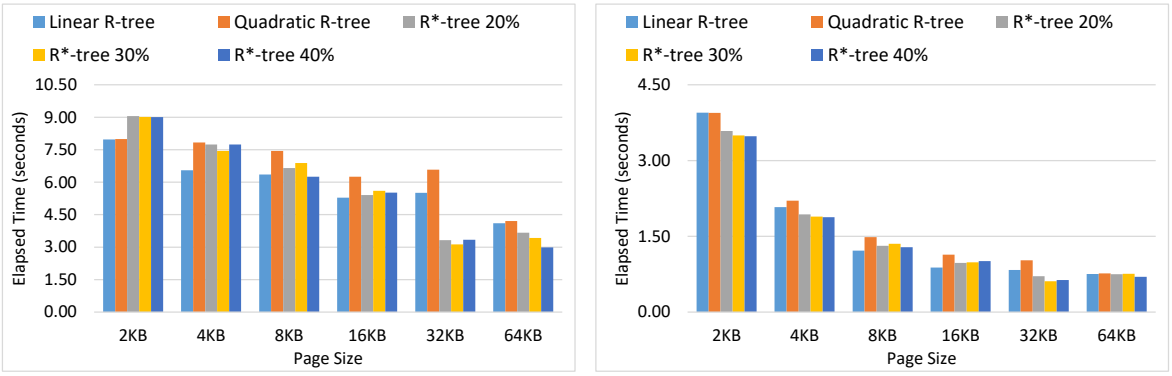


(performed on the HDD) (performed on the SSD)

Figure 11 – Performance results to process spatial queries on the local server, considering IRQs with 0.5%.

experiments because of its faster random reads. Considering all the page sizes, the performance gains of the SSD varied from 89% to 96% for the query windows with 0.1%. Gains ranging from 54% to 86% and from 51% to 85% were obtained for the query windows with 0.5% and 1%, respectively. The performance gains were decreasing as the selectivity increased because of the number of elements returned by the spatial queries. That is, the performance of the SSD benefited more spatial queries with low selectivity than the other spatial queries because of the number reads made on the storage device.

With respect to the execution on the HDD, almost all the configurations improved their performance by increasing page sizes. The reason is that if a spatial index employs large page sizes, more elements are processed in the main memory with only a few reads from the storage device. For the query windows with 0.1% (Figure 10), the *R*-tree 30%* using the page size equal to 64KB provided the best results. For the IRQs with medium and high selectivity (Figures 11 and 12), the best configuration was the *R*-tree 40%* using the page size equal to 64KB. The reason is that these queries returned more elements than the spatial queries using the other query



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

Figure 12 – Performance results to process spatial queries on the local server, considering IRQs with 1%.

windows. Since the *R*-tree 40%* improved the storage utilization and reduced the number of nodes, fewer accesses to the disk were required, improving the elapsed time.

With respect to the execution on the SSD, the performance behavior was slightly different than the HDD. For all IRQs, we obtained the best results by using the *R*-tree 30%* with the page size equal to 32KB. For the majority of the configurations, the use of the page size equal to 64KB deteriorated the performance, if compared to the use of the 32KB. We observed this behavior here because the SSD has a poor performance to retrieve large node sizes, thus degenerating the performance of the reads. The results of this experiment demonstrated that the spatial organization for an efficient spatial query processing on SSDs tends to be different.

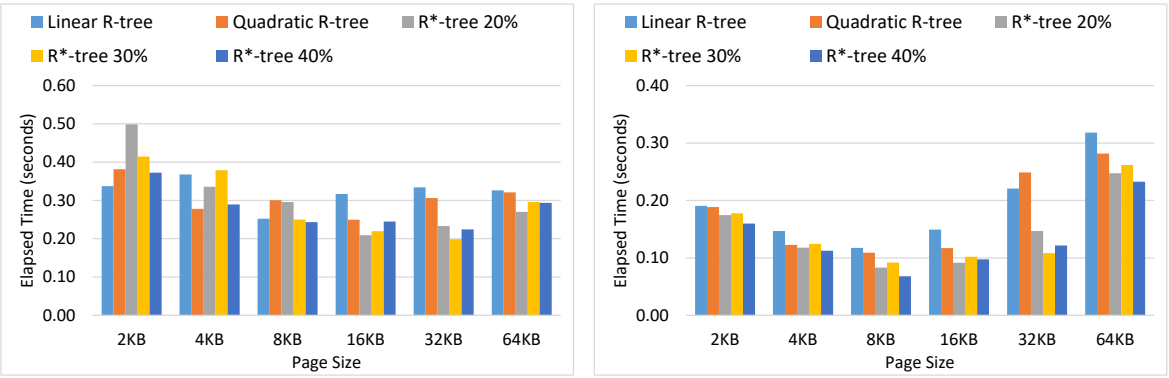
4.5.3.2 Execution on the Microsoft Azure

Figures 13, 14, and 15 show the obtained results for processing spatial queries on the virtual machines of the Microsoft Azure. As in other figures, we used a different scale to provide a better comparison of the results. Further, we considered only configurations based on the R-tree and the R*-tree (Table 9).

For the majority of the cases, the virtual machine equipped with the SSD showed better performance results to process spatial queries. Considering the query windows with 0.1% (Figure 13), the performance gains ranged from 3% to 72%. For both virtual machines, the *R*-tree 30%* using the page size equal to 32KB showed the fastest elapsed time to process the spatial queries.

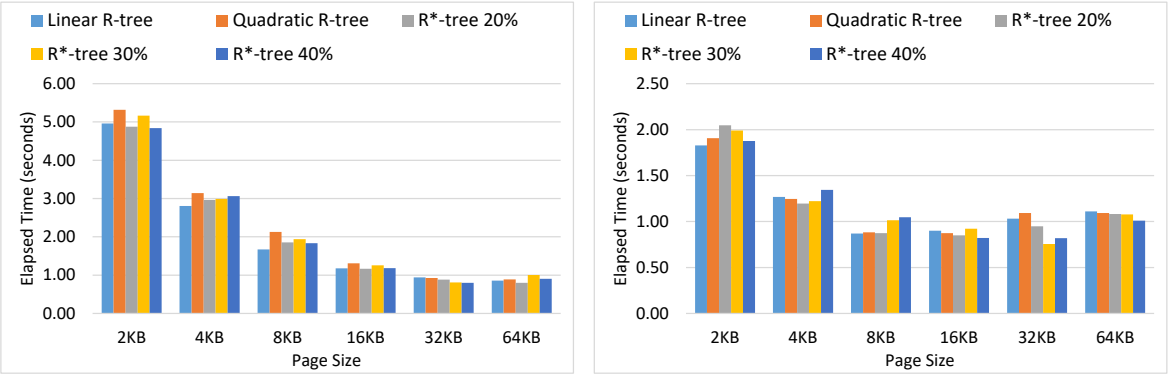
On the other hand, for the page sizes equal to 32KB and 64KB, the virtual machine with the HDD overcame the virtual machine with the SSD to process the query windows with 0.5% and 1% (Figures 14 and 15). The performance loss of the SSD is due to the great size of the nodes to be processed by the virtual machine, which may have its storage device in a different location from the processing unit. Smaller page sizes improved the performance.

For the majority of the indices, the page size equal to 16KB showed the best performance



(performed on the virtual machine with HDD) (performed on the virtual machine with SSD)

Figure 13 – Performance results to process spatial queries on the virtual machines of the Microsoft Azure, considering IRQs with 0.1%.



(performed on the virtual machine with HDD) (performed on the virtual machine with SSD)

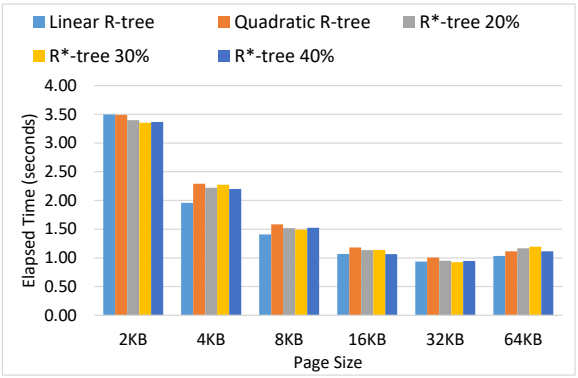
Figure 14 – Performance results to process spatial queries on the virtual machines of the Microsoft Azure, considering IRQs with 0.5%.

results in the virtual machine with SSD. Considering this page size, gains ranging from 23% to 33% were obtained for the query windows with 0.5% and gains varying from 4% to 20% were obtained for the query windows with 1%. For the query windows with 0.5%, the *R*-tree 30%* was the best configuration in both virtual machines, while the *R*-tree 40%* was the best configuration for the query windows with 1%.

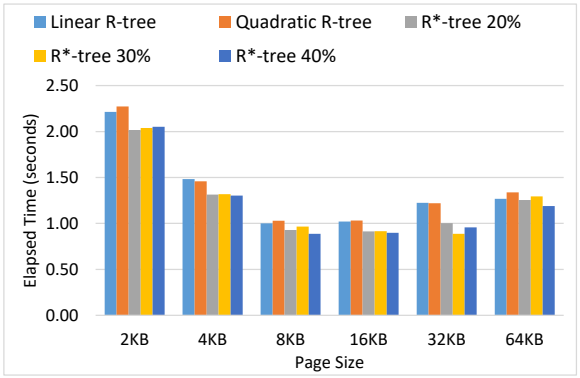
4.5.3.3 Comparing the Obtained Results

For the local server, the experiments showed that the SSD always guaranteed the best performance results. On the other hand, we obtained a different behavior in the virtual machines of the Microsoft Azure. The main difference was in the use of large page sizes (i.e., 32KB and 64KB), where the HDD guaranteed the best performance results to process query windows with medium and high selectivity because of the peculiarity of the cloud environment. However, this behavior disappeared for smaller page sizes.

On the other hand, a common behavior was the benefit of using large pages in the HDD



(performed on the virtual machine with HDD)



(performed on the virtual machine with SSD)

Figure 15 – Performance results to process spatial queries on the virtual machines of the Microsoft Azure, considering IRQs with 1%.

to process the spatial queries. In addition, the *R*-tree 30%* showed the best performance results for the HDD and SSDs for the majority of the spatial queries.

4.6 Conclusions and Future Work

In this article, we conducted an extensive experimental evaluation to study the performance of spatial indices on HDDs and SSDs under a local server and a cloud server. We considered the disk-based spatial indices R-tree and R*-tree because of their positive characteristics well-known in the literature. We also employed FAST in order to transform the disk-based spatial indices into flash-aware spatial indices. By varying several parameters, we were able to analyze at least 210 distinct configurations of spatial indices in our experiments.

We empirically evaluated spatial indices under two different workloads: spatial index construction and spatial query processing. In the first workload, our experiments showed that the direct use of disk-based spatial indices on SSDs did not guarantee good performance results. In the local server, the use of the HDD showed better elapsed times to build the majority of the disk-based spatial indices, if compared to the SSD. This means that to efficiently build spatial indices on the SSD, the design of the spatial indices should consider the intrinsic characteristics of SSDs, such as the poor performance of random writes. Based on that, the experiments showed that FAST-based spatial indices improved the elapsed time for building spatial indices on SSDs and even on HDDs since FAST uses a buffer in the main memory.

On both the environments considered in this article, our experiments showed that the FAST 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. In special, the page size equal to 16KB guaranteed better performance on spatial queries with high selectivity. Regarding the cloud server, we highlight the following two situations. First, the Microsoft Azure offers a special treatment for virtual machines equipped with SSDs, resulting in a better

maximum throughput and IOPS compared to virtual machines equipped with conventional disks. Thus, our experiments showed the best performance results on the virtual machine with SSD, in the most of the cases. Second, we can improve the performance of spatial indices stored on virtual machines with SSDs by taking into account the intrinsic characteristics of these storage devices. The reason is that the FAST-based spatial indices showed expressive performance gains in this environment. Therefore, our experiments showed that flash-aware spatial indices often improve the performance of spatial indexing on SSDs, independently of the running environment, compared to the direct use of disk-based spatial indices without any additional treatment.

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; FALOUTSOS, 1994). Finally, we aim to propose a new flash-aware spatial index that takes into account the findings of this article.

Acknowledgments.

This work has been supported by the Brazilian federal research agencies CAPES and CNPq, as well as by the São Paulo Research Foundation (FAPESP). A. C. Carniel has been supported by the grant #2015/26687-8, FAPESP. R. R. Ciferri has been supported by the grant #311868/2015-0, CNPq. We also thank the Microsoft Azure Sponsorship from the Microsoft Research.

CHAPTER

5

ANALYZING THE PERFORMANCE OF SPATIAL INDICES ON FLASH MEMORIES USING A FLASH SIMULATOR

This Chapter attaches the following paper (CARNIEL *et al.*, 2017):

- Carniel, A. C.; Silva, T. B.; Bonicenha, K. L. S.; Ciferri, R. R.; Ciferri, C. D. A. Analyzing the Performance of Spatial Indices on Flash Memories using a Flash Simulator. In Proceedings of the 32nd Brazilian Symposium on Databases (SBBD 2017), p. 40-51, 2017.

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.

5.1 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; 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; VETTER, 2016; BRAYNER; 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; 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; CHANG; KUO, 2003; LV *et al.*, 2011b; 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 5.2 summarizes intrinsic characteristics of flash memories. Section 5.3 surveys related work. Section 5.4 details our performance evaluation. Section 5.5 answers our questions. Section 5.6 concludes the paper.

5.2 Flash Memories

Flash memories organize data in flash pages and flash blocks (JUNG; 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; 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; 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.

5.3 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; CIFERRI; CIFERRI, 2016c), and (ii) the performance evaluation of new flash-aware spatial indices on flash memories (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; 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, Ciferri and Ciferri (2016c) 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; CHANG; KUO, 2003; LV *et al.*, 2011b), 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 5.2), 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 5.1. 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.

5.4 Performance Evaluation

5.4.1 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 11. 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 5.3.

We executed two workloads: (i) index construction, and (ii) processing of intersection range queries (IRQ) (GAEDE; 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; CIFERRI; CIFERRI, 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 12. We employed this simulator because of its advantages (Section 5.1) 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 5.4.2). We correlated these results by executing the same workloads on an SSD (Section 5.5). For this purpose, we used a local server equipped with an Intel® Core™ 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

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

Table 11 – 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 12 – The emulated flash memory[4].

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

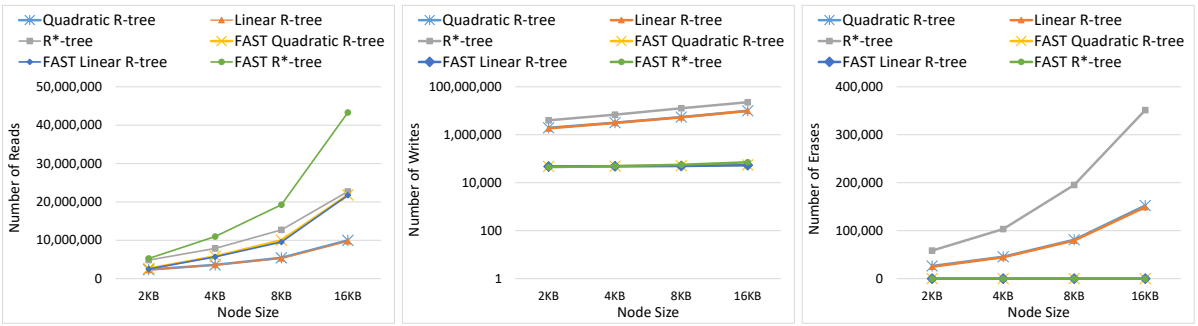
execution of each configuration. Further, we executed each workload 10 times and calculated the elapsed time as follows. For the first workload, we collected the average elapsed time. For the second workload, we collected the average elapsed time to execute the IRQs of each kind of selectivity. We employed Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, and PostGIS 2.2.0 in all the environments.

5.4.2 Execution on the Flash Simulator

5.4.2.1 Index Construction

Figure 16 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

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



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

Figure 16 – Number of reads (a), writes (b), and erases (c) for building spatial indices in the flash simulator.

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.

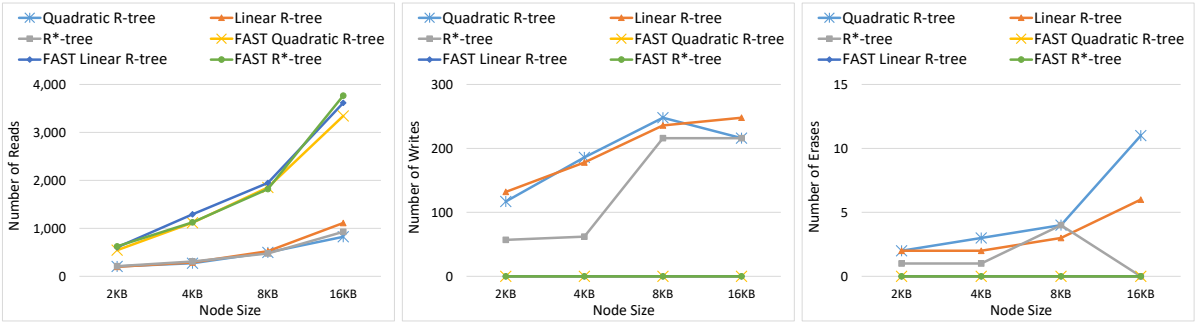
5.4.2.2 Spatial Query Processing

Figure 17 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 17a), 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 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.

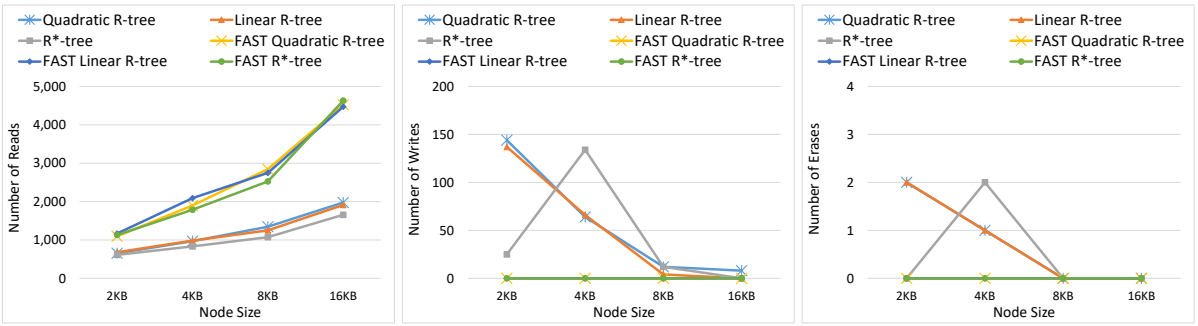
With regard to query windows with 0.01% (Figure 17b), the number of writes and erases performed by the disk-based spatial indices decreased, compared to the query windows with 0.001%. Again, the FAST-based spatial indices required much more reads than the other configurations. In these IRQs, the best 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 17c), 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



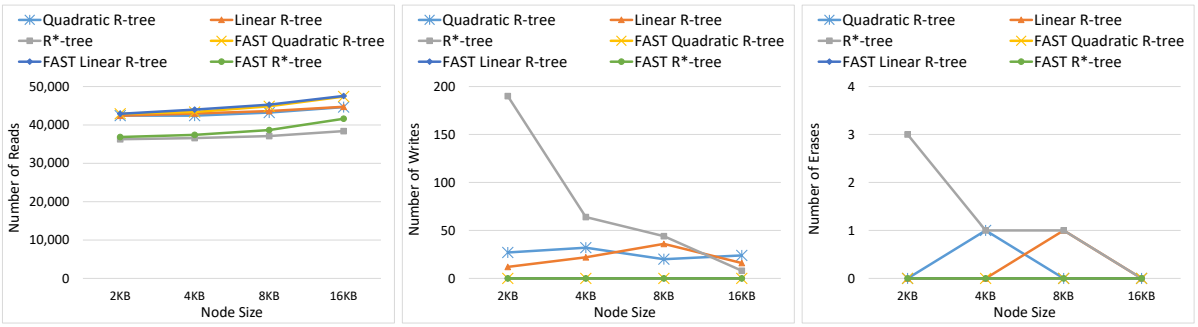
reads writes erases

(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 17 – Results obtained for executing the IRQs in the flash simulator.

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.

Figure 18 – Correlating the performance gains and losses of the FAST-based spatial indices over the corresponding disk-based spatial indices for building indices.

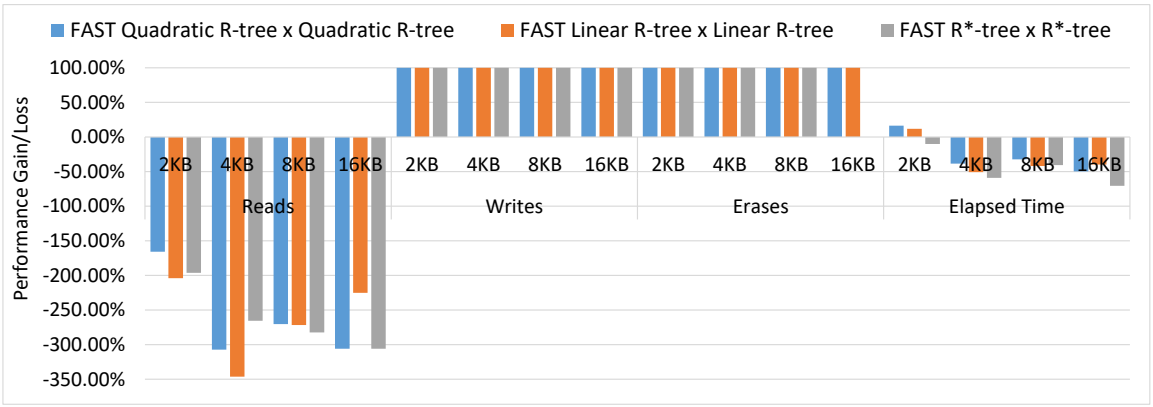
5.5 The Performance Relation between the Flash Simulator and the Real SSD

To answer the questions in Section 5.1, we compare the results obtained in the flash simulator (Section 5.4.2) and in the SSD. Figures 18 and 19 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 5.4.2), while the elapsed time percentage values were obtained from the results of the SSD.

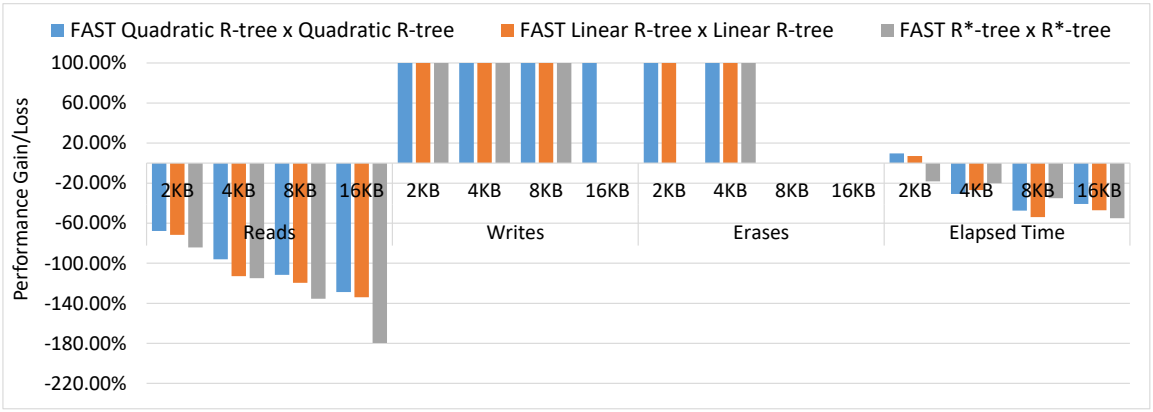
In the following, we answer each question of Section 5.1 by firstly providing an analysis to serve as a foundation.

Analysis to Answer Question 1. For the index construction (Figure 18), in spite of the 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.

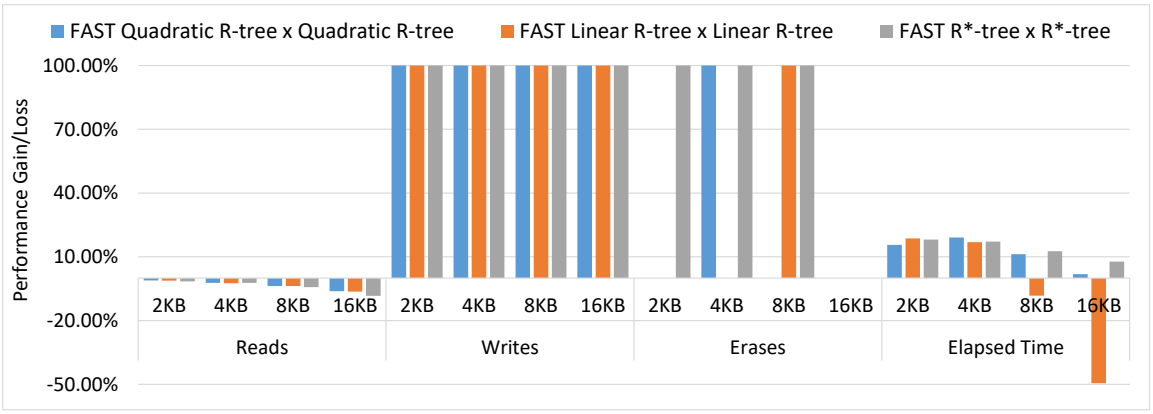
For the spatial query processing (Figure 19), the performance of the disk-based spatial indices was impacted by the writes performed in the SSD. Thus, although the FAST-based spatial indices have required more reads, it showed reductions in the processing time of IRQs in some cases. These reductions were more frequent to process the query windows with 0.1%, which



(a) query windows with 0.001%



(b) query windows with 0.01%



(c) query windows with 0.1%

Figure 19 – Correlating the performance gains and losses of the FAST-based spatial indices over the disk-based spatial indices for processing the IRQs.

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.

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

Table 13 – 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.

Workload	**Flash Simulator**			**SSD**
	# 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)

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 13 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 17).

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.

5.6 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; GÜNTHER, 1998) and eFIND-based spatial indices (CARNIEL; CIFERRI; CIFERRI, 2017b).

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.

CHAPTER

6

A GENERIC AND EFFICIENT FRAMEWORK FOR FLASH-AWARE SPATIAL INDEXING

This Chapter attaches the following paper (CARNIEL; CIFERRI; CIFERRI, 2018):

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

This paper is an extended version of the following work (CARNIEL; CIFERRI; CIFERRI, 2017b):

- Carniel, A. C.; Ciferri, R. R.; Ciferri, C. D. A. A Generic and Efficient Framework for Spatial Indexing on Flash-based Solid State Drives. In Proceedings of the 21st European Conference on Advances in Databases and Information Systems (ADBIS 2017), p. 229-243, 2017.

Abstract

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

Performance tests showed the efficiency of eFIND. Compared to the state of the art, eFIND improved the construction of spatial indices from 43% to 77%, and the spatial query processing from 4% to 23%.

6.1 Introduction

Spatial indices are largely employed to improve spatial query processing since they reduce the search space by discarding portions of the dataset where the answer cannot be found (GAEDE; GÜNTHER, 1998; OOSTEROM, 2005). Nowadays, there is an increasing number of spatial database applications requiring the use of spatial indices to retrieve efficiently spatial objects stored in *flash-based Solid State Drives* (SSDs) (EMRICH *et al.*, 2010; KOLTSIDAS; VIGLAS, 2011b; LIU *et al.*, 2012; CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a). In fact, SSDs have been widely used as secondary storage in notebooks, desktops, and database servers because of their improved characteristics compared to Hard Disk Drives (HDDs) (MITTAL; VETTER, 2016). These characteristics include smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

On the other hand, SSDs have intrinsic characteristics that introduce several system implications (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). A well-known characteristic is that a write requires more time and power consumption than a read. In addition, random writes can lead to high costly erase-before-update operations and thus, sequential writes are preferable. To deal with these characteristics, some *flash-aware spatial indices* have been proposed in the literature (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; PAWLIK; MACYNA, 2012; SARWAT *et al.*, 2013; JIN *et al.*, 2015; LIN *et al.*, 2006; LI *et al.*, 2013; FEVGAS; BOZANIS, 2015). Among them, FAST-based indices (SARWAT *et al.*, 2013) distinguish themselves by providing efficiency and data durability.

Commonly, existing flash-aware spatial indices extend spatial indices originally designed for HDDs like the R-tree (GUTTMAN, 1984) (termed as *disk-based spatial indices*). Instead of directly performing random writes to the SSD, flash-aware spatial indices store index modifications in an in-memory buffer. When this buffer is full, a flushing policy picks a set of modified index pages to be sequentially written to the SSD.

However, current flash-aware spatial indices do not exploit all the benefits of SSDs. First, the management of the modifications contained in the buffer is based on inefficient data structures, such as lists with repeated elements. Second, these indices execute an excessive number of random reads, which can degenerate SSD performance (JUNG; KANDEMIR, 2013). Third, they perform interleaved reads and writes, also negatively impacting the SSD performance (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). Finally, they employ flushing algorithms that may lead to unnecessary writes to SSDs.

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

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

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

The rest of this article is organized as follows. Section 6.2 describes the intrinsic characteristics of SSDs and their impact on applications. Section 6.3 introduces our design goals that serve as a basis for eFIND. Section 6.4 proposes eFIND, which deploys specific data structures and algorithms to fulfill the design goals. Section 6.5 experimentally compares eFIND with the state of the art. Section 6.6 conducts a performance evaluation to study the effect of our design goals. Section 6.7 surveys related work and describes how this article extends our previous work (CARNIEL; CIFERRI; CIFERRI, 2017b). Finally, Section 6.8 concludes the article and presents future work.

6.2 An Overview of Flash-based Solid State Drives

An SSD consists of several components, such as multiple cores, internal DRAM/SRAM buffer, and storage media (AGRAWAL *et al.*, 2008; JUNG; KANDEMIR, 2013). Our focus is on the storage media, which is an array of *flash memory packages* that delivers high storage capacity and internal parallelism of reads and writes. A flash memory package is made up of a set of dies (chips). A die is divided into multiple planes, where each plane consists of several

blocks and a few registers. Each block is composed of a number of pages; thus, flash memory is *block-oriented*. A page stores data and metadata, such as the Error Correcting Code (ECC).

Flash memory supports erase, read and write (program) operations (AGRAWAL *et al.*, 2008; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013). Erase is performed in block granularity, leading to the most expensive operation of the flash memory. Read and write are page level operations with asymmetric costs, where normally a read requires much less time and power consumption than a write. The write operation is only performed on cleaned pages since update operations are not supported. Hence, a *sequential write*, where pages in one block are written sequentially throughout the block, is preferable. Otherwise, an expensive *erase-before-update* operation is performed. This operation first saves the data of the block containing the page being updated, then erases the block, and finally writes the content of the modified page together with the non-modified content of the block. Further, flash memory has a limited endurance. After a number of writes and erases on a block, it can no longer store data.

There are a number of other factors that impact on SSD performance (BOUGANIM; JÓNSSON; BONNET, 2009; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016). The use of ECC, which introduces some overhead on reads and writes, can correct some bit errors on pages. However, bit error rate increases exponentially as writes and erases are performed on a block. Blocks with a high error rate, reaching to an error recovery coverage limit, are marked as bad blocks and need a *bad block management* (JUNG; KANDEMIR, 2013) that degrades system performance.

The *read disturbance management* (JUNG; KANDEMIR, 2013) is another management that requires extra computational time of SSDs. This management is needed to avoid read disturbances, which occur if multiple reads are issued on the same page without any erase. Because of a disturbance, reads can require long latencies similar to the write latency.

Another factor is that *reads and writes interfere with each other* (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016). A critical interference is when a write is performed on a page and this page is subsequently read. In this case, the read operation must wait for the write request because the write is internally buffered. Similarly to read disturbances, the interference makes reads require long latencies. Hence, mixing reads and writes negatively impact SSD performance.

The last factor is related to the *Flash Translation Layer* (FTL) (CHEN; KOUFATY; ZHANG, 2009; CHUNG *et al.*, 2009; KWON *et al.*, 2011; XIE; CHEN; ROTH, 2017). FTL is physically located inside of the SSD and deploys sophisticated algorithms to optimize SSD performance. It emulates the interface of HDDs to provide an easy integration of SSDs with existing computational systems. This interface maps the physical addresses of the array of flash memory packages into logical addresses that are used by applications. Further, FTL employs an out-of-place update algorithm to avoid erase-before-update operations, a garbage collector to reclaim pages and blocks, and a wear leveling to improve flash memory endurance. These

complex algorithms can introduce additional computational costs if applications do not take into account the aforementioned intrinsic characteristics of SSDs.

6.3 Design Goals for Flash-Aware Spatial Indices

Despite the fact that intrinsic characteristics of SSDs have been well studied in the literature, it remains unclear how to deal with them to achieve good spatial indexing performance. In this section, we provide a conceptual discussion of the underlying ideas to solve this problem by introducing a set of *design goals for flash-aware spatial indices*. They are inspired by existing flash-aware techniques (Section 6.7) and represent reasonable solutions that take into account the intrinsic characteristics of SSDs (Section 6.2). Hence, our design goals can even serve as a foundation to implement other types of flash-aware algorithms, such as flash-aware unidimensional index structures (e.g., B-trees). In this article, we focus on spatial indexing on SSDs and our discussion is directed toward using these design goals as a basis to create efficient and robust flash-aware spatial indices. We detail each design goal as follows.

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

Goal 2 - Dynamically pick modifications to be sequentially flushed. A flushing operation that writes all modifications contained in the write buffer leads to a big write to the SSD, degenerating its performance. Further, it writes index pages that would be potentially modified soon (SARWAT *et al.*, 2013). To achieve Goal 2, a flash-aware spatial index should include a *specialized flushing algorithm* consisting of a *flushing policy* and a *flushing unit creator*. The flushing policy should pick the modified index pages to be written, according to distinct criteria. For instance, a modified index page can be picked based on its number of modifications, the moment of its last modification, and/or internal characteristics (e.g., its height). The flushing unit creator should organize index pages in flushing units, following the flushing policy, and determine the size of data that is written to the SSD in each flushing operation. Ideally, a flushing unit should contain only sequential index pages. But, when it is not possible, a flushing unit should be composed of (i) almost sequential index pages, or (ii) distant index pages (e.g., at least 100 index pages of distance). The former leads to a performance as similar as a sequential write,

while the latter leads to a better performance compared to random writes (CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016). Finally, the flushing operation should also take into account Goal 4.

Goal 3 - Avoid excessive random reads in frequent locations. The common assumption that the random read is the fastest operation of SSDs is not always valid because of the read disturbance management. As a result, this operation can take as long as a write or erase. To achieve Goal 3, a flash-aware spatial index should use an in-memory buffer dedicated to the management of reads, called *read buffer*. Thus, instead of performing a random read directly from the SSD to obtain a frequently accessed index page, the index page can be obtained from the read buffer. For instance, in hierarchical indices like the R-tree, nodes (i.e., index pages) located near to the root are frequently accessed in search operations and are reasonable candidates to be cached in the read buffer. Further, the management of the read buffer should include a *read buffer replacement policy*. Examples are the well-known Least Recently Used (LRU) replacement algorithm (DENNING, 1980), the two versions of the 2Q replacement algorithm (JOHNSON; SHASHA, 1994), and the Adaptive Replacement Cache (ARC) (MEGIDDO; MODHA, 2003).

Goal 4 - Avoid interleaved reads and writes. Mixing reads and writes negatively affects SSD performance because of the interference between these operations. More importantly, this interference severely degrades read performance. To achieve Goal 4, a flash-aware spatial index should use read and write buffers together with a *temporal control*, which temporally stores the identifiers of the last read and written index pages. The temporal control of reads should determine whether a flushed index page must be cached to avoid a read after a write. The temporal control of writes should help the flushing operation by discarding index pages that do not lead to the desired organization, such as a sequential write.

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

6.4 The Efficient Framework for Spatial Indexing on SSDs

In this section, we detail how to achieve each design goal conceptually discussed in Section 6.3. This is done by proposing eFIND, a generic and efficient framework that transforms a disk-based spatial index into a flash-aware spatial index. Figure 20 depicts the eFIND's architecture, which consists of three sophisticated managers to meet the requirements of the design goals.

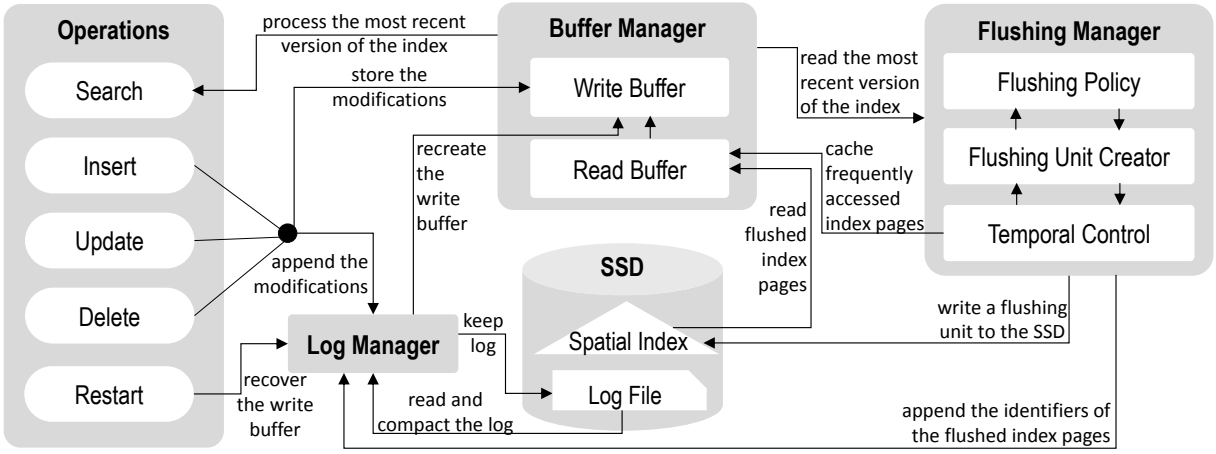


Figure 20 – The architecture of eFIND.

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

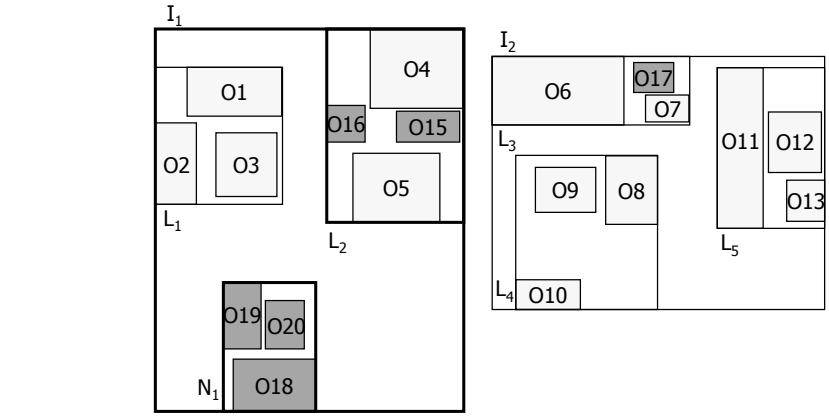
Flushing Manager. It contains three interacting components to perform a flushing operation. The first component is the *flushing unit creator*, which builds flushing units by grouping sequential index pages. The second component is the *flushing policy*, which ranks flushing units according to different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

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

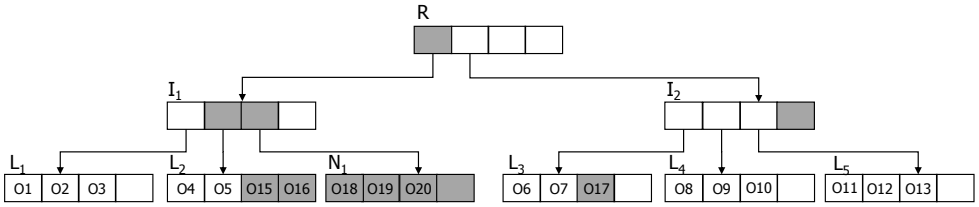
Before describing how eFIND works in Sections 6.4.2 to 6.4.6, we introduce in Section 6.4.1 a running example used throughout this article.

6.4.1 Running Example

In this running example, we apply eFIND to an R-tree resulting in the *eFIND R-tree* shown in Figure 21. The objects *O1* to *O13* depicted in light gray represent the spatial objects stored in the SSD. The rectangles L_1, L_3, L_4, L_5, and I_2 represent entries that are also stored in the SSD. The other objects shown in dark gray and rectangles with thick lines are not stored in the SSD. They are derived from the following modifications: (i) insertion of 6 new spatial objects (i.e., *O15* to *O20*), which leads to the creation of the new node N_1, and (ii) deletion of the leaf node L_6 that contained one spatial object and was stored in the last entry of I_2. These



(a) Graphical representation



(b) Hierarchical representation

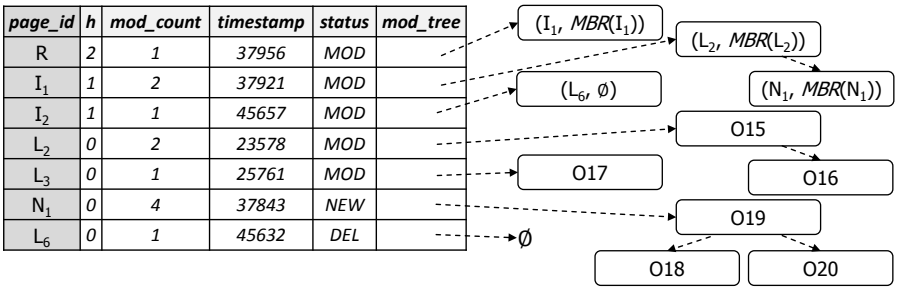
Figure 21 – The eFIND R-tree of our running example.

modifications should be handled by the eFIND's data structures.

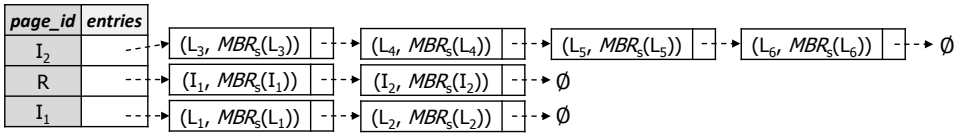
This eFIND R-tree has a height equal to 2 and indexes 19 spatial objects. Each node of the tree represents an *index page* and consists of a fixed number of entries (four, in the current example). Each entry has the format (p, r). If the entry is contained in a leaf node, then p is a unique identifier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

6.4.2 Data Structures

eFIND leverages specific data structures to deal with the design goals of Section 6.3. To implement the write buffer (Goal 1), a hash table named *Write Buffer Table* is employed. It stores the modifications of index pages that were not applied to the SSD yet. We use a hash table as the structure for the write buffer because we can access modified entries of index pages usually in constant time. The key of this hash table is the identifier of an index page (*page_id*) and its value stores modifications in the format (h, *mod_count*, *timestamp*, *status*, *mod_tree*). Here, h refers to a specific parameter for hierarchical indices and stores the height of the modified index page; *mod_count* is the quantity of in-memory modifications; *timestamp* informs when the last modification was made; and *status* is the type of modification made and can be NEW, MOD, or DEL for representing newly created index pages in the buffer, index pages stored in the SSD but with modified entries, and deleted index pages, respectively. This information is mainly



(a) *Write Buffer Table*



(b) *Read Buffer Table*

Figure 22 – Graphical representation of the buffers managed by eFIND for applying the modifications of Figure 21.

used in flushing operations (see Section 6.4.4). For *status* equal to NEW or MOD, *mod_tree* is a red-black tree containing the result of modified entries; otherwise, it is null. An element of a red-black tree has the format (*e*, *mod_result*), where *e* is the key and corresponds to the unique identifier of an entry of the index page and *mod_result* stores the most recent version of this entry, assuming null if *e* was removed. The main benefit of this strategy is that only the latest version of a modified entry is stored in the write buffer. This is achieved by the use of a red-black tree to store the modified entries of an index page instead of using a list with repeated elements. Consequently, the space of the write buffer is better managed with a low cost of retrieving the most recent version of an index page (see Section 6.4.3). Further, the cost of updating *mod_tree* is amortized by the use of a red-black tree.

Figure 22a shows the *Write Buffer Table* for our running example. In this figure, *MBR* is a function for computing the rectangle that encompasses all entries of a node by considering current modifications in the write buffer. Hence, the same format of an entry of the underlying index is used for each element in *mod_tree*. That is, (*e*, *mod_result*) is equivalent to (p,r) (Section 6.4.1). For instance, the first line of the hash table in Figure 22a shows that *R*, located in the *height* 2, has the *status* MOD to store the entry $(I_1, MBR(I_1))$. Note that this entry now corresponds to the most recent version of the first entry of *R*. This modification occurred in the *timestamp* 37956 and is derived from the adjustment of I_1 after the creation of N_1.

To implement the read buffer (Goal 3), another hash table named *Read Buffer Table* is employed. It caches the index pages that are already stored in the SSD, prioritizing pages frequently accessed. We employ a hash table because of its usually constant time to access cached index pages. Further, we do not use the same hash table of the write buffer because the read buffer has a different purpose and requires a read buffer replacement policy like LRU and 2Q to decide which index pages should be cached. The key of *Read Buffer Table* is the unique

RQ	WQ
R	L_3
L_2	L_4
L_3	I_2
L_4	L_2

(a) Temporal control

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

(b) Log file

Figure 23 – The queues of the temporal control and the log file after applying the modifications of Figure 21.

index page identifier (*page_id*) and its value (*entries*) stores a list of entries of the index page. For the R-tree, *entries* stores a set of (p, r) values. Figure 22b depicts that I_2, R, and I_1 are cached in the *Read Buffer Table* by considering only entries stored in the SSD. In this figure, MBR_S is a function for extracting the stored MBR. For instance, the entries of I_2 includes L_6 even after its deletion, which is indicated in the third line of the hash table in Figure 22a.

To deal with the temporal control (Goal 4), eFIND employs two queues named *RQ* and *WQ*. Each queue is a First-In-First-Out (FIFO) data structure. *RQ* stores identifiers of the index pages read from the SSD, while *WQ* keeps the identifiers of the last index pages written to the SSD. Figure 23a shows that the last read nodes are R, L_2, L_3, and L_4, and the last flushed nodes are L_3, L_4, I_2, and L_2.

To guarantee data durability (Goal 5), eFIND sequentially writes to a log file the modifications contained in the *Write Buffer Table*. The format (*page_id*, *h*, *type_mod*, *result*) is employed to store each log entry. It has a very similar format to that of the *Write Buffer Table* because we need to be able to recover the write buffer after a fatal problem. That is, *page_id* is the identifier of an index page, *h* is the height of the modified page, and *type_mod* corresponds to the *status*, and *result* is an element of the corresponding *mod_tree*. Hence, different modifications made on the same index page are stored in sequential log entries. Further, index pages written to the SSD are also stored in the log file to allow log compaction, as explained in Section 6.4.6. In this case, *type_mod* assumes the value FLUSH, *result* is the set of flushed index pages, and the value null is stored for the remaining attributes. Figure 23b shows the log file that follows the chronological order of the modifications stored in the *Write Buffer Table* (Figure 22a). For instance, the first log entry corresponds to the first needed modification to insert *O16*, that is, the accommodation of this object in L_2.

The main advantage of the eFIND's data structures is their low-cost integration into existing spatial database systems since they do not require the changing of the underlying spatial

Table 14 – Employed notations to measure the cost of the eFIND's algorithms.

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

Algorithm 4: Execution of a maintenance operation by using eFIND

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

index. Thus, to estimate the average time to process the operations of Figure 20, we consider the eFIND's algorithms only. For this, we make use of the notations in Table 14 in our cost analyses. Note that the employed data structures have also a low maintenance cost ($\mathscr{H}$, $\mathscr{T}(P)$, and $\mathscr{Q}$).

6.4.3 Maintenance Operations

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

To guarantee data durability, Algorithm 4 calls the *Log Manager* to keep the log of all modifications stored in the *Write Buffer Table* (line 21). This is a low latency operation since it involves only sequential writes. The final step of Algorithm 4 is the execution of a flushing operation (Section 6.4.4) whenever the write buffer is full (line 23).

The execution of an in-memory maintenance operation (line 1 in Algorithm 4) involves retrieving index pages. For instance, to choose an index page to accommodate or delete a spatial object. Because parts of the underlying spatial index can be stored in three different locations, the SSD, the *Read Buffer Table*, and the *Write Buffer Table*, we specify Algorithm 5 to retrieve index pages by considering all these locations.

Algorithm 5 has the identifier P_{id} of an index page as input. If this page is a newly created index page or a deleted index page, the algorithm returns the index page pointed by its corresponding entry in the *Write Buffer Table* (line 3). Otherwise, it gets the last stored version of the index page that is either buffered in the *Read Buffer Table* (lines 6 and 7) or stored in the SSD (lines 9 to 11). In the former case, we avoid a read operation. In the latter case, the index page is read from the SSD and stored in the *Read Buffer Table* (lines 9 and 10). Further, the read operation is added to the FIFO queue *RQ* (line 11) designated for handling the temporal control of reads. Afterwards, the read buffer replacement policy is applied (line 12). Finally, Algorithm 5 returns either the index page read from the SSD or the index page cached in the *Read Buffer Table*, if it does not have modifications in the *Write Buffer Table* (line 15). If P_{id} has modifications (line 13), a merge operation is executed to return the most recent version of the index page (line 14). This merge operation replaces the old version of modified entries by the entries stored in the *Write Buffer Table*, maintains the entries that were not modified, and adds newly created entries if any.

Example of Execution. To illustrate the execution of a maintenance operation, let us consider

Algorithm 5: Retrieving an index page by using eFIND

Input: P_{id} as the identifier of an index page to be returned
Output: The current version of the index page identified by P_{id}

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

the second modification on the eFIND R-tree (Figure 21), that is, the insertion of *O16*. First, a leaf node should be chosen to accommodate this object. Since the MBR of the internal node I_1 intersects *O16*, Algorithm 5 gets the cached version of I_1 from the *Read Buffer Table* (last line of Figure 22b) without requiring reads from the SSD. Then, the leaf node L_2 is chosen to accommodate *O16*. This node is read from the SSD since it is not cached in the *Read Buffer Table*. This read operation is registered in the *RQ* (second line of Figure 23a).

Accommodating *O16* in L_2 results in two modified pages to be traversed. The first modification is stored in the *Write Buffer Table* (fourth line of Figure 22a). It indicates that L_2, with height 0, has 2 modifications. The last of them is the insertion of *O16* at the time 23578. At the same time, this modification is also stored in the log file (second line of Figure 23b). The second modification corresponds to the adjustment of I_1. It is stored in its *mod_tree* as $(L_2, MBR(L_2))$ (second line of Figure 22a). Finally, it is appended to the log file (third line of Figure 23b).

Cost Analysis. Before discussing the cost of Algorithm 4, we first analyze $\mathscr{C}_{alg2}$ as the cost of retrieving the index page P (Algorithm 5). Three possible cases compose $\mathscr{C}_{alg2}$. First, if the hash entry in the *Write Buffer Table* (i.e., *WBEntry*) has the *status* NEW or DEL, the cost $\mathscr{C}_{alg2}$ is minimized by accessing only one hash entry. The second case is if P is cached in the *Read Buffer Table* (i.e., *RBEntry* is not null). It requires one access in each hash table, and the execution of the read buffer replacement policy (totaling $2 * \mathscr{H} + \mathscr{B}$). This case further requires the cost $\mathscr{M}$ of executing a merge operation, which is given by $\mathscr{M} = 0$ if *WBEntry* is null and $\mathscr{M} = \mathscr{O}(n+m)$ otherwise; where n is the number of entries in *WBEntry* and m is the number of

entries in *RBEntry*. The last case occurs if P has to be read from the SSD, which adds the costs of the read operation and the management of *RQ* (i.e., $\mathscr{R}+\mathscr{Q}$) to the cost of the second case. We formally define $\mathscr{C}_{alg2}$ as follows:

$$\mathscr{C}_{alg2} = \begin{cases} \mathscr{H} & \text{if } \textit{status} \text{ of } \textit{WBEntry} \text{ is NEW or DEL} \\ 2*\mathscr{H}+\mathscr{B}+\mathscr{M} & \text{if } \textit{RBEntry} \text{ is not null} \\ 2*\mathscr{H}+\mathscr{B}+\mathscr{M}+\mathscr{R}+\mathscr{Q} & \text{otherwise} \end{cases} \tag{6.1}$$

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

$$\mathscr{C}_{amc}(P) = \begin{cases} \mathscr{F}+\mathscr{H}+\mathscr{W} & \text{if } P \text{ is a deleted index page} \\ \mathscr{H}+\sum_{i=1}^{m}(\mathscr{T}(P)+\mathscr{W}) & \text{otherwise} \end{cases} \tag{6.2}$$

Once defined $\mathscr{C}_{alg2}$ and $\mathscr{C}_{amc}$ (Equations 6.1 and 6.2), the cost of Algorithm 4, $\mathscr{C}_{alg1}$, is given by the sum of the cost of retrieving r index pages needed for computing the in-memory maintenance operation and the cost of processing the modifications of n modified index pages. $\mathscr{C}_{alg1}$ is formally defined as follows:

$$\mathscr{C}_{alg1} = \sum_{i=1}^{r}(\mathscr{C}_{alg2}) + \sum_{i=1}^{n}(\mathscr{C}_{amc}(P_i)) \tag{6.3}$$

6.4.4 Flushing Operations

Algorithm Descriptions. Algorithm 6 details the flushing operation (executed at line 23 in Algorithm 4). Instead of writing all the stored modifications to the SSD, eFIND smartly selects only some of them to be written to the SSD. To this end, the algorithm first creates the list *HEntries*, which contains the oldest hash entries of the *Write Buffer Table* by considering the *timestamp* attribute (line 1). The percentage value p determines the size of *HEntries*. Our experiments showed best performance results if p is equal to 60% (see Section 6.6.1).

Afterwards, the temporal control of writes discards hash entries of *HEntries* by using Algorithm 7 (line 2). Then, flushing units are created. Assuming a *flushing unit size* equal to s, each group of s index pages of *HEntries*, previously sorted by the index page identifiers (line 3), define a flushing unit (line 4). Hence, flushing units are formed by sequential index pages.

Next, Algorithm 6 picks a flushing unit to be written according to a flushing policy. This is performed by using a function *FP* that calculates, for each flushing unit, a degree d (lines 5 and 6) that is then used to select the flushing unit with the greatest degree (line 7). In case of ties, any

Algorithm 6: Execution of the eFIND's flushing operation

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

flushing unit might be picked. Different implementations of FP are conceivable and can be based on distinct criteria. Here, we present the following set of flushing policies $\mathscr{P} = \{FP_m, FP_{mh}, FP_{mha}, FP_{mhao}\}$. The input of each $FP \in \mathscr{P}$ is an array FU of n hash entries of the *Write Buffer Table*. The flushing policy FP_m (Equation 6.4) is based on the number of modifications, that is, it calculates the total number of modifications of all index pages of FU. The remaining policies should be used by hierarchical indices, such as the eFIND R-tree. The flushing policy FP_{mh} (Equation 6.5) is based on the number of modifications and height of nodes, that is, it uses the height of the modified node as weight on its number of modifications. The flushing policies FP_{mha} and FP_{mhao} (Equations 6.6 and 6.7) are extensions of FP_{mh} and are based on the modified area and the overlapping modified area, respectively. They apply α as an additional weight that corresponds to the ratio of the modified area by the flushing unit and the total modified area stored in the write buffer. FP_{mhao} also uses the β weight, which is the ratio of the overlapping modified area the flushing unit and the total overlapping modified area stored in the write buffer. The idea behind the weights α and β is the creation of flushing policies by using criteria based on geometric characteristics of the modifications. Each $FP \in \mathscr{P}$ is formally defined as follows:

$$FP_m(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count \tag{6.4}$$

$$FP_{mh}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) \tag{6.5}$$

$$FP_{mha}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) * \alpha \tag{6.6}$$

$$FP_{mhao}(FU) = \sum_{i=1}^{FU.n} FU[i].mod\_count * (FU[i].h + 1) * \alpha * \beta \quad (6.7)$$

Our experiments showed best performance results for FP_{mh} (see Section 6.6.1). The main reason is that this flushing policy gives higher degrees for index pages located in the highest levels of the index, which are not so frequently modified. In the sequence, Algorithm 6 retrieves the index pages that compose the chosen flushing unit (lines 9 to 12). Here, Algorithm 8 is called to execute the temporal control of reads (line 12). Then, Algorithm 6 writes in batch the index pages of the chosen flushing unit (line 13). Finally, the algorithm keeps log of the flushed index pages, adds the made writes to the queue *WQ* for handling the temporal control of writes, and frees the index pages from the *Write Buffer Table* (lines 14 to 16).

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

Algorithm 8 details the execution of the temporal control of reads. It has as input an index page and deals with two alternately cases. The first one relates to updating the content of the index page, if it is already cached in the *Read Buffer Table* (line 2). This is needed because the cached version of the index page was modified during the flushing operation (line 10 in Algorithm 6). In the second case, the index page is stored in the *Read Buffer Table*, if its identifier is contained in the *RQ* (line 5). The advantage of this pre-caching is that it avoids a possible read after a write since the index page is frequently requested by retrieving operations. The read buffer replacement policy is applied as needed in both cases (lines 3 and 6).

Example of Execution. Consider that the *Write Buffer Table* of Figure 22a is full, requiring a flushing operation. Let us assume that *HEntries* contains 60% of the oldest hash entries of this hash table, that is, L_2, L_3, N_1, and I_1. Considering that all these index pages are almost sequential to the pages stored in the *WQ* (Figure 23a), the temporal control of writes (Algorithm 7) returns

Algorithm 7: Execution of the temporal control of writes

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

Algorithm 8: Execution of the temporal control of reads

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

the same list of index pages. Afterwards, the pages are sorted according to its identifiers. Here, we sort them by considering their appearance in Figure 21b, from the highest level to the lowest level of the tree. The resulting list is then I_1, L_2, N_1, and L_3.

Assuming the flushing unit size equal to 2, the flushing units $FU_1 = \{I_1, L_2\}$ and $FU_2 = \{N_1, L_3\}$ are created. By applying the flushing policy FP_{mh} (Equation 6.5), the flushing unit FU_1 is chosen because its degree (i.e., 6) is higher than the degree of FU_2 (i.e., 5). Next, the nodes I_1 and L_2 are retrieved by using Algorithm 5. Only one read is required to retrieve L_2 since I_1 is cached in the *Read Buffer Table* (Figure 22b). In the sequence, the temporal control of reads (Algorithm 8) updates the content of I_1 and applies the read buffer replacement policy. L_2 is cached in the *Read Buffer Table* because it is contained in the *RQ* (Figure 23a). Finally, the most recent versions of I_1 and L_2 are written to the SSD, and this flushing operation is appended as a new log entry 12 in the log file of Figure 23b.

Cost Analysis. The cost of Algorithm 6 requires the cost analysis of Algorithms 7 and 8. The cost of Algorithm 7, $\mathscr{C}_{alg4}$, includes the traversal of a list containing n index page identifiers. For each identifier, the algorithm checks whether it provides a (almost) sequential or stride write, considering the m index page identifiers of *WQ*. We formally define $\mathscr{C}_{alg4}$ as follows:

$$\mathscr{C}_{alg4} = \mathscr{O}(n * m) \tag{6.8}$$

As for the cost of Algorithm 8, $\mathscr{C}_{alg5}$, two alternately cases are possible. The first case relates to the cost $\mathscr{U}$ of updating the content of P in the *Read Buffer Table*. For the second case, $\mathscr{C}_{alg5}$ requires the cost $\mathscr{H}$ of inserting a new entry in the *Read Buffer Table*. Both cases add the cost $\mathscr{B}$ of executing the read buffer replacement policy. We formally define $\mathscr{C}_{alg5}$ as follows:

$$\mathscr{C}_{alg5} = \begin{cases} \mathscr{U} + \mathscr{B} & \text{if } P \text{ is cached in the } \textit{Read Buffer Table} \\ \mathscr{H} + \mathscr{B} & \text{if } RQ \text{ contains the identifier of } P \end{cases} \tag{6.9}$$

Now we can estimate the average cost of Algorithm 6, $\mathscr{C}_{alg3}$. First, it requires the linear cost $\mathscr{A}$ needed to collect the p oldest hash entries in the *Write Buffer Table*. These entries are filtered by the temporal control of writes with the cost $\mathscr{C}_{alg4}$ (Equation 6.8). Then, a sorting operation of cost $\mathscr{S}$ is made, which typically corresponds to $\mathscr{O}(n \log n)$ for n as the number of index page identifiers being sorted. Next, a flushing unit is chosen by requiring the linear cost $\mathscr{T}$. To write the flushing unit, its f index pages are retrieved and processed by the temporal control of reads, requiring the cost $\mathscr{C}_{alg2} + \mathscr{C}_{alg5}$ (Equations 6.1 and 6.9). Writing the chosen flushing unit leads the cost $\mathscr{W} + \mathscr{W}_{batch}$, which corresponds to the maintenance of the log file plus the batch operation. Finally, let $\mathscr{Q} + \mathscr{D}$ be the cost of adding a flushed index page in QW and of deleting its hash entry from the *Write Buffer Table*. $\mathscr{C}_{alg3}$ is the sum of all previous costs as follows:

$$\mathscr{C}_{alg3} = \mathscr{A} + \mathscr{C}_{alg4} + \mathscr{S} + \mathscr{T} + \mathscr{W} + \mathscr{W}_{batch} + \sum_{i=1}^{f} (\mathscr{C}_{alg2} + \mathscr{C}_{alg5} + \mathscr{Q} + \mathscr{D}) \tag{6.10}$$

6.4.5 Search Operations

Algorithm Description. eFIND does not change the search algorithm of the underlying index. But, the search algorithm should use Algorithm 5 to retrieve index pages. To serve as an illustration, Algorithm 9 shows how an eFIND R-tree should execute a search operation. Its inputs are a search object *SO*, a topological predicate *TP* (e.g., *intersects*, *inside*), and a node (index page) identifier. Starting from the root node of an eFIND R-tree, the algorithm first retrieves its index page by using Algorithm 5 (lines 1 and 2). If it is an internal node, Algorithm 9 is called recursively for all child nodes that satisfy the topological predicate *TP* by considering the search object *SO* (lines 3 to 6). When the node is a leaf, the search operation returns the most recent version of the entries that answer the spatial query (line 8).

Example of Execution. To exemplify a search operation, let us consider the execution of a point query (GAEDE; GÜNTHER, 1998). Let further consider that the search object *SO* is a point contained in *O17*, that is, $SO \in O17$. The query starts by traversing the root node R. Hence, the most recent version of R is retrieved by Algorithm 5. Among the entries of R, I_2 intersects the search object. Then, Algorithm 9 is called recursively for such node. I_2 is retrieved from the *Read*

Algorithm 9: Execution of the search algorithm for the eFIND R-tree, starting from its root node

Input: *SO* as a search object, *TP* as a topological predicate, and P_{id} as a node identifier of the eFIND R-tree
Output: A set of entries containing the candidates that answer the query

```
1 let LE be a list of entries;
2 let P be the index page yielded by the retrieving algorithm with P_id as input (Algorithm 5);
3 if P is an internal node then
4     foreach entry En_i in P do
5         if En_i satisfies the topological predicate TP, considering the search object SO then
6             invoke the search algorithm with SO, TP, and the node identifier pointed by En_i
                as inputs;
7 else
8     add all entries whose satisfy the topological predicate TP for the search object SO to LE;
9 return LE;
```

Buffer Table (first line of Figure 22b) and is merged to its modification contained in the *Write Buffer Table* (third line of Figure 22a). The last call of Algorithm 9 is for the leaf node L_3, which is read from the SSD and has its modification appropriately applied (fifth line of Figure 22a). In this calling, the algorithm returns *O17*. Note that the use of the *Read Buffer Table* avoids reads from the SSD. Without the use of the read buffer, 3 reads should be made to answer the point query; instead, eFIND did perform 1 read only.

Cost Analysis. A number of executions of Algorithm 5 are made until a given spatial query is completely answered. This number depends on the spatial organization of the underlying index. Since eFIND does not change this spatial organization, we assume n as the number of accessed index pages needed to answer a spatial query. We formally define the cost of Algorithm 5 as follows:

$$\mathscr{C}_{alg6} = \sum_{i=1}^{n} \mathscr{C}_{alg2} \tag{6.11}$$

6.4.6 Restart Operations

Algorithm Description. Algorithm 10 shows how eFIND recovers all modifications that were not effectively applied to the index, after a system crash, fatal error, or failure power. It has the log file of eFIND as input and yields a new possibly compacted log file. The first step of the restart operation is to read the log file in reverse order (lines 4 to 9), that is, from the most recent modifications to the oldest ones. During this step, all log entries of flushing operations are added to a list (line 6), while log entries not contained in this list are pushed in a stack of log entries (line 8) because they were not applied yet to the SSD. By using this stack, eFIND creates a new log file and appends all modifications contained in the stack to the *Write Buffer Table* (lines 11 to 13). Finally, the new log file should be used in future maintenance operations (lines 14 and 15).

Algorithm 10: Execution of a restart operation, rebuilding the *Write Buffer Table* and compacting the log

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

Another benefit of Algorithm 10 is that it may compact the log file. Compacting the log is a needed operation to improve the space utilization of eFIND. Instead of inserting log entries of the stack into the *Write Buffer Table* (line 13), the compaction algorithm should only append these log entries to the new log file.

Example of Execution. Consider that a flushing operation was performed (Section 6.4.4) before a power failure. As a result, the modifications of the *Write Buffer Table* (Figure 22a) are lost and the current log file, shown in Figure 24a, contains now 13 log entries. The last log entry has *type_mod* equal to FLUSH, *result* equal to I_1, L_2, and null for the remaining attributes. Since the log file is read in reverse order, the first log entry to be read refers to the flushed nodes I_1 and L_2. These nodes are appended to *LEntries*. Then, the log entries 12, 11, and 10 are pushed into the stack. The log entry 9 contains one modification for nodes of *LEntries* and is ignored. The log entries from 8 to 4 are pushed into the stack, while the log entries 3, 2, and 1 are ignored. Finally, the log entries in the stack are written to a new log file and their modifications are inserted into the *Write Buffer Table*. As a result, the algorithm rebuilds the same version of the *Write Buffer Table* before the system crash. Further, the log file is compacted by containing 8 log entries instead of 13 log entries, as shown in Figure 24b.

Cost Analysis. The cost of Algorithm 10, $\mathscr{C}_{alg7}$, involves the cost of two loops. The first loop refers to the traversal of the log file in reverse order. For each log entry, this requires the sum of the cost of reading the corresponding log entry (i.e., $\mathscr{R}$), and the cost of inserting the log entry either in a list or in a stack (i.e., $\mathscr{K}$). The second loop refers to the traversal of a stack to rebuilt

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

(a) Log file after the flushing operation

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

(b) Compacted log file

Figure 24 – Illustrating how the restart operation may also compact the log file.

the *Write Buffer Table*. The cost of this loop is defined by the cost function $\mathscr{C}_{aml}$ (Equation 6.12). Its input is a log entry L. $\mathscr{C}_{aml}$ has the fixed cost $\mathscr{W}$ of writing a log entry in the new log file. If the *type_mod* of L is DEL, the cost $\mathscr{F}$ of freeing modifications of L in the *Write Buffer Table* is added to $\mathscr{C}_{aml}$. Otherwise, $\mathscr{C}_{aml}$ adds the cost $\mathscr{T}(L)$ of inserting the modification of L in the *Write Buffer Table*. In addition to the cost of these two loops, $\mathscr{C}_{alg7}$ also considers the cost $\mathscr{E}$ of erasing the old version of the log file. Let n be the number of log entries stored in the log file, and m be the number of stacked log entries. We formally define $\mathscr{C}_{aml}$ and $\mathscr{C}_{alg7}$ respectively as follows:

$$\mathscr{C}_{aml}(L) = \begin{cases} \mathscr{F} + \mathscr{W} & \text{if } L \text{ has the } \textit{type\_mod} \text{ DEL} \\ \mathscr{T}(L) + \mathscr{W} & \text{otherwise} \end{cases} \tag{6.12}$$

$$\mathscr{C}_{alg7} = \sum_{i=1}^{n} (\mathscr{K} + \mathscr{R}) + \sum_{j=1}^{m} (\mathscr{C}_{aml}(L_j)) + \mathscr{E} \tag{6.13}$$

6.5 Experimental Evaluation

In this section, we measure the performance gains of eFIND against the state of the art. Section 6.5.1 describes the experimental setup employed in the experiments. Section 6.5.2 discusses results related to the construction of spatial indices, while Section 6.5.3 reports results for executing spatial queries.

6.5.1 Experimental Setup

Dataset. We used a real spatial dataset from the OpenStreetMap[1], which consisted of 1,485,866 complex regions possibly with holes representing the buildings of Brazil like hospitals, universi-

[1] <http://www.openstreetmap.org/>

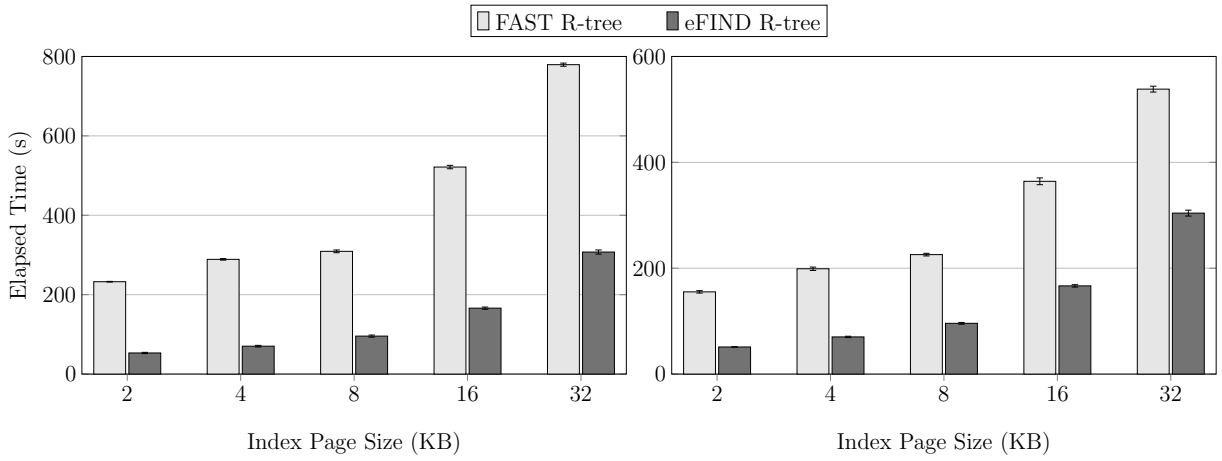
ties, schools, houses, and stadiums. Geometric and statistical descriptions of this dataset can be found in Carniel, Ciferri and Ciferri (2017c) through the name *brazil_buildings2017_v2*.

Configurations. We compared two configurations: (i) the *FAST R-tree*, and (ii) the *eFIND R-tree*. These configurations applied the quadratic split algorithm of the R-tree, as well as had a buffer of 512KB and log capacity of 10MB. We used a fixed buffer size because our goal is to analyze the performance behavior of these configurations, which is usually similar for different sizes of the buffer (CARNIEL; CIFERRI; CIFERRI, 2016c; CARNIEL; CIFERRI; CIFERRI, 2017a; SARWAT *et al.*, 2013; JIN *et al.*, 2015). For the FAST R-tree, we used the FAST* flushing policy, which provided the best results according to Sarwat *et al.* (2013). For the eFIND R-tree, we employed the best parameter values based on the analysis conducted in Section 6.6: FP_{mh} as the flushing policy, the value of 60% for p, the allocation of 20% of the buffer for the read buffer, and the simplified 2Q as the read buffer replacement policy. We considered FAST as the state of the art because it provides the best characteristics among the existing flash-aware spatial indices, as detailed in Section 6.7. We did not compare the native (plain) R-tree (GUTTMAN, 1984) because it requires a prohibitive number of writes, reads, and erases, not being suitable for SSDs, as shown in Jin *et al.* (2015) and Carniel *et al.* (2017).

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

Workloads. We executed two workloads: (i) index construction, and (ii) execution of 300 intersection range queries (IRQ) (GAEDE; GÜNTHER, 1998). Three different sets of query windows were used. These sets were respectively composed of 100 query windows with 0.001%, 0.01%, and 0.1% of the area of the total extent of Brazil. Considering that the selectivity of a spatial query is the ratio of the number of returned objects and the total objects, these sets of query windows form spatial queries with low, medium, and high selectivity, respectively. For each configuration, we executed the workloads as a sequence, that is, the index construction followed by the processing of IRQs. Each sequence was executed 5 times. We flushed the system cache after the execution of each sequence and calculated the elapsed time as follows. For the first workload, we collected the average elapsed time. For the second workload, we calculated the average elapsed time to execute each set of query windows.

Running Environment. We employed FESTIval (CARNIEL; CIFERRI; CIFERRI, 2016a), an open-source PostgreSQL extension that benchmarks spatial indices. The source code and the documentation of FESTIval (and eFIND) are publicly available at <https://github.com/accarniel/festival>. We performed the tests locally to avoid network latency. The experiments were conducted on a local server equipped with an Intel Core® i7-4770 with a frequency of 3.40GHz, 32GB of main memory, and two SSDs: (i) one Kingston V300 of 480GB, and (ii) one Intel Series 535 of 240GB. The Intel SSD is a high-end SSD that provides faster reads and



(a) Executed on the Kingston SSD **(b)** Executed on the Intel SSD

Figure 25 – The eFIND R-tree showed expressive performance gains when building spatial indices on both SSDs.

writes than the low-end Kingston SSD. This allowed us to measure the performance of eFIND by considering different architectures of SSDs. The software we used was Ubuntu Server 14.04 64 bits, PostgreSQL 9.5, and PostGIS 2.2.

6.5.2 Index Construction

As shown in Figure 25, the eFIND R-tree overcame the FAST R-tree on both SSDs and for all employed page sizes. Its performance gains were very expressive, ranging from 60% to 77% for the Kingston SSD (Figure 25a), and ranging from 43% to 67% for the Intel SSD (Figure 25b).

The eFIND R-tree exploited the benefits of the SSDs because it is based on the design goals defined in Section 6.3. The contribution of the read buffer to obtain these results was significantly relevant even using a relatively small percentage of the buffer size, as discussed in Section 6.6.2. Another important contribution was the use of the temporal control, which guaranteed that frequently accessed index pages were stored beforehand in the read buffer. Further, eFIND improved the space utilization of the write buffer. Instead of using a list that allows repeated elements, eFIND leverages efficient data structures to manage index modifications. This led to the faster retrieval of index pages, reflecting in the elapsed time when building spatial indices in the SSDs.

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

6.5.3 Spatial Query Processing

For the high-end Intel SSD, the eFIND R-tree always provided the best performance, but only slightly overcame its competitor (Figures 26b, d, and f). Its performance gains ranged from 4% to 6%. The internal characteristics of the Intel SSD like its fast read performance contributed to these results. As for the low-end Kingston SSD (Figures 26a, c, and e), the eFIND R-tree provided the best performance only for larger pages but delivered better performance gains. Its gains were of 22% and 23% for the index pages of 16KB and 32KB respectively. In both SSDs, we believe that the performance gains were not more expressive because of the high cost of loading index pages read from the SSD to the main memory. This is further analyzed in Section 6.6.3.

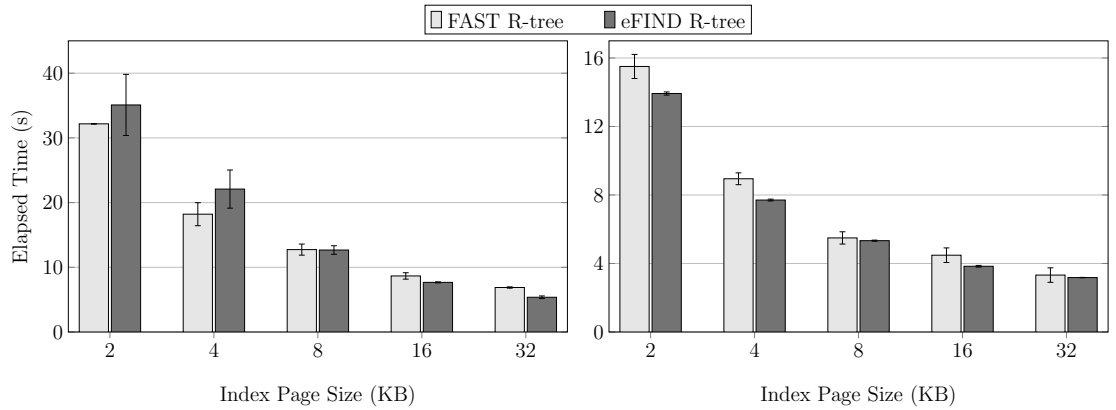
The time spent by the configurations to process the IRQs decreased as the index page size increased because a large page size allows that more entries be loaded into the main memory, requiring fewer reads. Hence, the index page size of 32KB provided the best results for both configurations and SSDs. For this page size, the eFIND R-tree was better than the FAST R-tree for all selectivity levels. The IRQs using query windows with 0.001% (Figures 26a and b) required less time to be executed than the other query windows because of their low selectivity. The IRQs with medium and high selectivity (Figures 26c and d, and Figures 26e and f) required more elapsed time because of the high number of random reads performed on the SSD together with the traversal of several entries in the main memory.

6.6 Experimental Analysis of the Design Goals

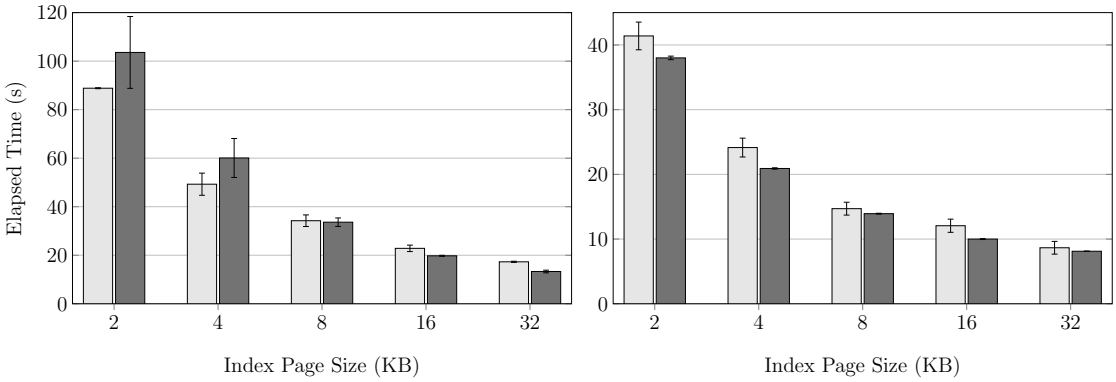
In this section, we evaluate the effect of each design goal in the performance behavior of eFIND. For these experiments, we used the Intel SSD to execute the same workloads from Section 6.5.1. Specifically here, in addition to index construction, we report the results only for IRQs with low selectivity because we gathered similar performance behavior for the other selectivity levels. Throughout the discussions conducted in this section, we show the results in two different ways of visualization. The first one is an overview that shows the performance behavior of the eFIND R-tree for all tested configurations. The second one applies a zoom in the overview by considering the index page sizes that demonstrated the best performance results: 2KB and 4KB for building indices, and 16KB and 32KB for processing IRQs. This zooming strategy allows us to better visualize and compare the differences in the results.

6.6.1 Effect of the Flushing Operation on the Write Buffer

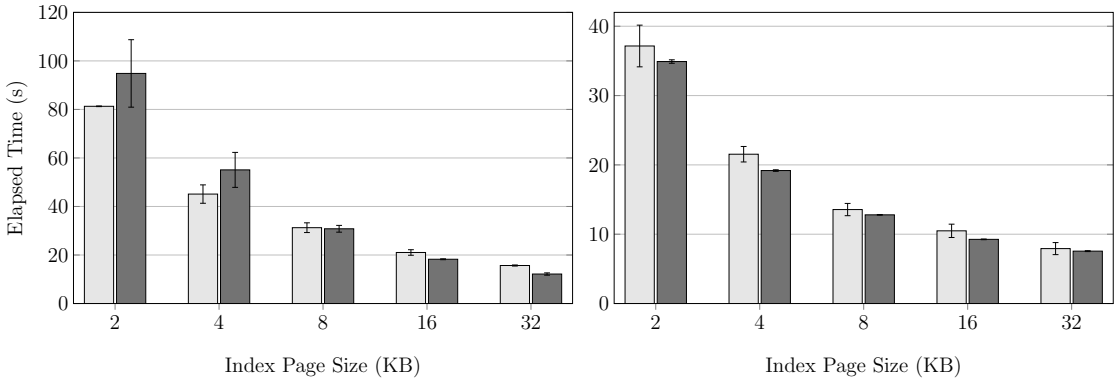
In this section, we analyze the effect of parameters related to the write buffer (Goal 1), which also includes the flushing operation (Goal 2). We varied two tuning parameters of Algorithm 6: (i) the function employed by the flushing policy to calculate the degree of a flushing unit, and (ii) the percentage value p. Since p determines the amount of index page identifiers that



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

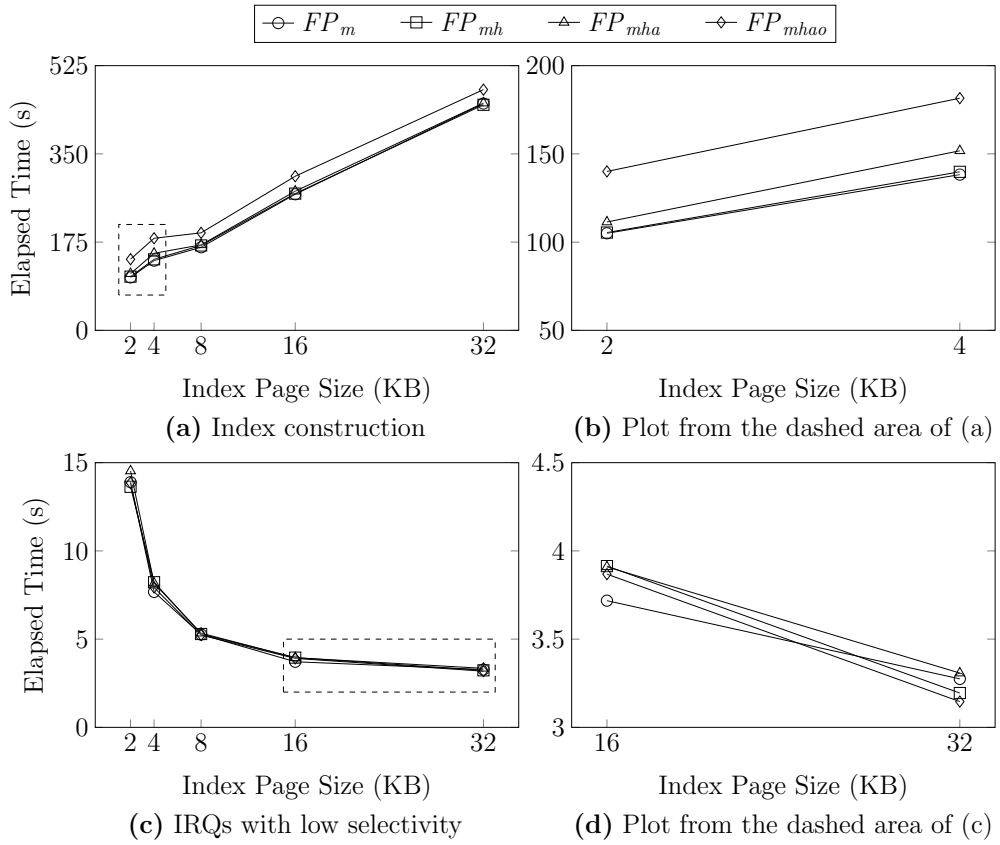


(c) IRQs with medium selectivity on the Kingston SSD **(d)** IRQs with medium selectivity on the Intel SSD



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

Figure 26 – The configurations showed the best performance to process the IRQs when using large index page sizes. For these cases, the eFIND R-tree outperformed the FAST R-tree for all selectivity levels and on both SSDs.



(a) Index construction **(b)** Plot from the dashed area of (a)

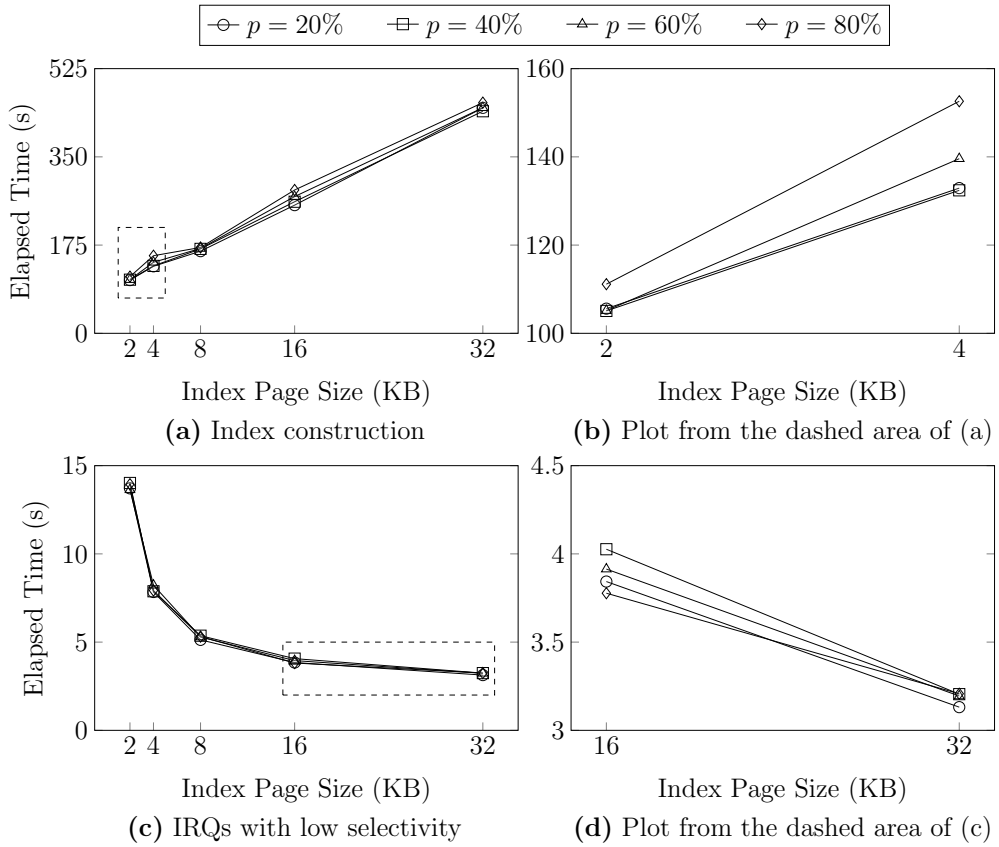
(c) IRQs with low selectivity **(d)** Plot from the dashed area of (c)

Figure 27 – Effect of different flushing policies. The flushing policy FP_{mh} showed the best results when building eFIND R-trees (a) and (b). These flushing policies did not significantly affect the query performance (c) and (d).

the flushing policy must handle, we first varied the flushing policies and fixed p to 60%. Then, we fixed a flushing policy and varied p. For all experiments of this section, we used the size of log equal to 10MB and turned off the support for the read buffer and the temporal control.

Varying the flushing policy. We evaluated the set $\mathscr{P} = \{FP_m, FP_{mh}, FP_{mha}, FP_{mhao}\}$ of flushing policies (Section 6.4.4). Figure 27 depicts the performance results of each flushing policy in $\mathscr{P}$. The flushing policies FP_m and FP_{mh} showed the best elapsed times when building indices (Figures 27a and b). Considering the number of reads and writes made by them, FP_{mh} required up to 3% less operations than FP_m because it prioritizes the nodes in the highest levels of the index, which are not so frequently updated. The use of other characteristics like the modified and overlapping areas of the modifications did not improve the performance of the index construction because of their computational costs. The flushing policies FP_{mha} and FP_{mhao} showed the worst results with losses up to 34% compared to the FP_{mh}.

With respect to the execution of IRQs, the flushing policies showed similar elapsed times to process them (Figures 27c and d). That means the order in which the index pages were written to the SSD did not impair the performance of the IRQs. Because of these results, we assume the flushing policy FP_{mh} in the remainder of the experiments.



(a) Index construction **(b)** Plot from the dashed area of (a)

(c) IRQs with low selectivity **(d)** Plot from the dashed area of (c)

Figure 28 – Effect of the percentage value p used in the first line of Algorithm 6. Although $p = 20\%$ showed better elapsed times in the index construction (a) and (b), it required more writes than $p = 60\%$, which in turn showed a better performance than $p = 80\%$. Varying p did not significantly affect the query performance (c) and (d).

Varying p. We now analyze the impact of the tuning parameter p, which determines the number of index pages to be considered by the flushing policy based on the balance between the recency of modifications and number of modifications. We evaluated each value in $\{20\%, 40\%, 60\%, 80\%\}$ as shown in Figure 28. Our analysis focus on the index construction since the experiments for processing the IRQs (Figures 28c and d) showed similar elapsed times than those depicted in (Figures 27c and d), respectively.

For building indices, the best balance was obtained for p equal to 60% (Figures 28a and b). If p is small (i.e., 20%), the flushing algorithm picks index pages containing only a few modifications, reducing the available space of the write buffer after the flushing operation. In this case, the number of flushing operations needed to create space for storing new modifications is increased. $p = 20\%$ required 7% more writes than $p = 60\%$. If p is very large (i.e., 80%), index pages frequently modified are flushed multiple times, decreasing the performance of the index construction. $p = 60\%$ outperformed $p = 80\%$ by showing reductions up to 5%.

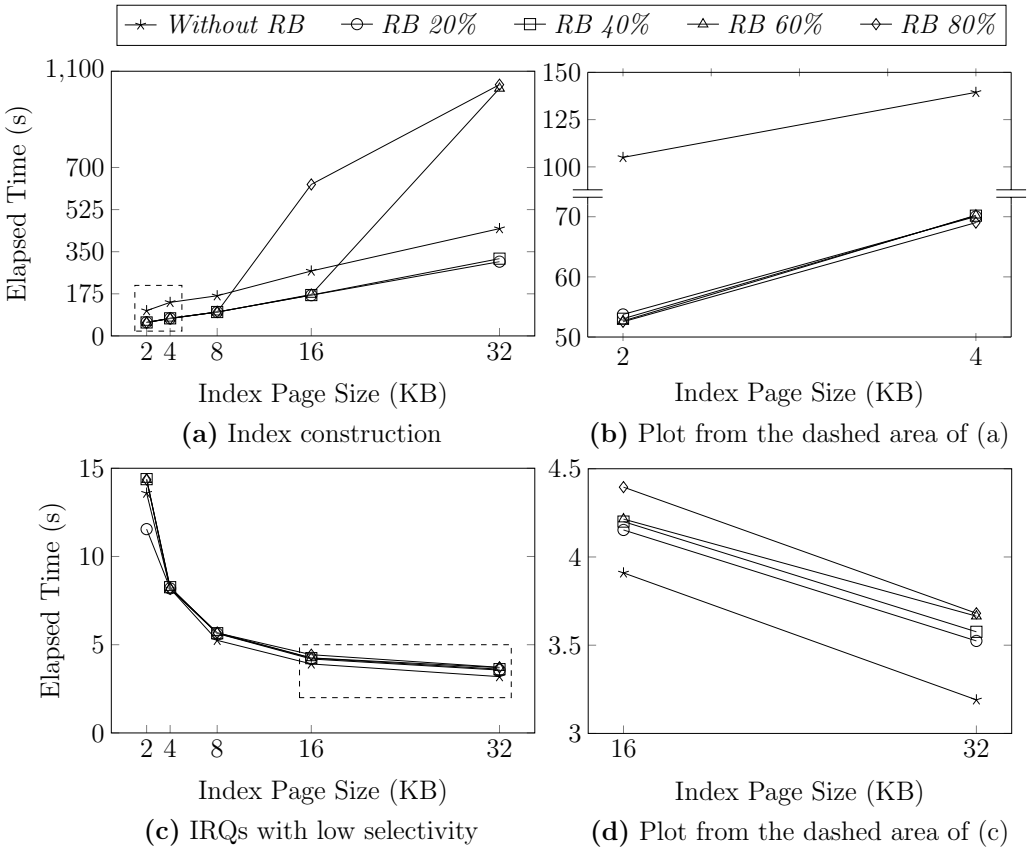


Figure 29 – Effect of allocating a part of the buffer for the read buffer. The read buffer showed to be very promising to improve the performance of the index construction (a) and (b). But, this was not the case for the execution of IRQs (c) and (d) because of the cost of loading pages from the SSD.

6.6.2 Effect of the Read Buffer

In this section, we analyze the effect of allocating a percentage of the buffer for the read buffer (Goal 3). We varied this percentage in 0%, 20%, 40%, 60%, and 80%, forming the configurations *Without RB*, *RB 20%*, *RB 40%*, *RB 60%*, and *RB 80%*, respectively. As a result, each configuration divided the buffer of the eFIND R-tree (i.e., 512KB) into two parts, the read buffer, and the write buffer. Since our focus is to analyze the balance between the sizes of the read buffer and the write buffer, we did not vary the read buffer replacement policy by considering only the LRU. For all experiments of this section, we used the flushing policy FP_{mh}, $p = 60\%$, and log size of 10MB. The support for the temporal control was turned off.

In most cases, the use of the read buffer, independently of its dedicated percentage, greatly improved the elapsed time for building indices, if compared to *Without RB* (Figures 29a and b). The best configuration was *RB 20%*. Compared to *Without RB*, it showed reductions between 31% to 49%. Compared to *RB 40%*, *RB 60%*, and *RB 80%*, *RB 20%* showed reductions in the number of writes varying from 2% to 99%. If larger allocations for the read buffer are used, fewer modifications can be stored in the write buffer leading to an increasing number of

flushing operations, especially for large sizes of the index page. For instance, *RB 80%* showed the worst performance results for the index page sizes equal to 16KB and 32KB. In general, the effectiveness of the read buffer can be also improved by prefetching frequently accessed nodes, as discussed in Section 6.6.3.

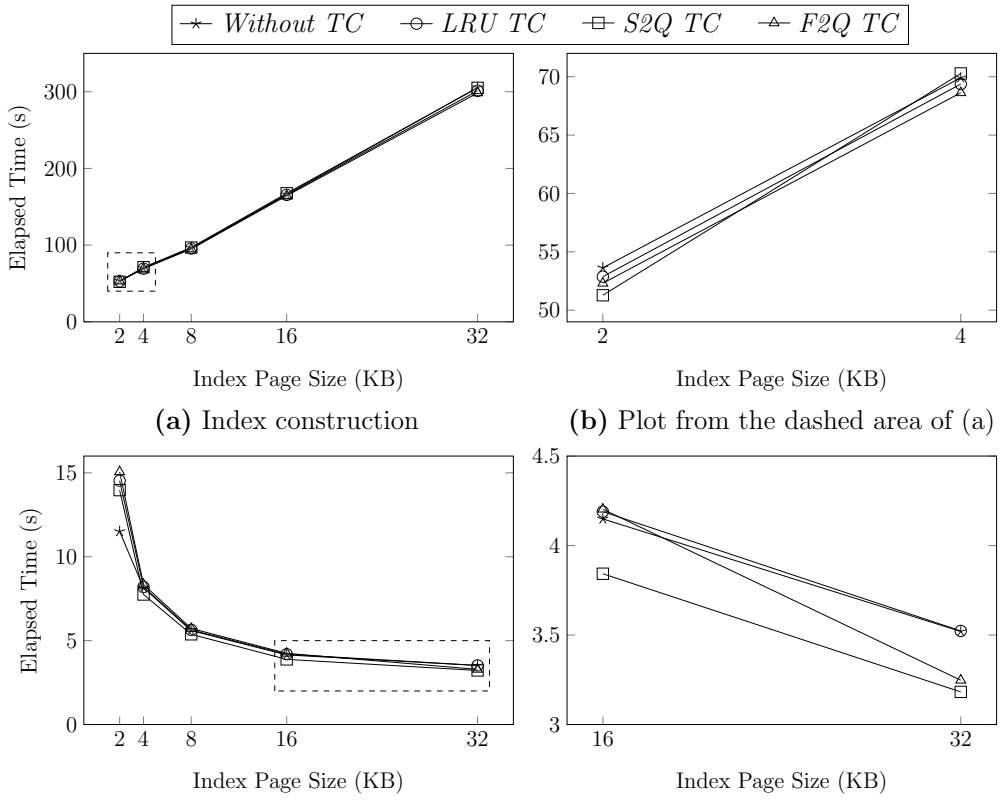
As for the processing of IRQs (Figures 29c and d), allocating space for the read buffer did not improve this processing. This occurred even when a large allocation for the read buffer was used (i.e., 80%). Because of the cost of loading the index pages read from the SSD to the main memory, the read buffer impaired the performance of the IRQs. Another fact that contributed to these results was the high cost of processing spatial data in the main memory because of the traversal of multiple paths of the tree to answer an IRQ.

6.6.3 Effect of the Temporal Control

In this section, we analyze the use of the temporal control to avoid interleaved reads and writes (Goal 4). In addition, we evaluate how to improve the cost of loading index pages read from the SSD to the main memory. For this, we employed 2Q as the read buffer replacement policy because of its good performance on newer memories (LERSCH *et al.*, 2017). There are two versions of 2Q: (i) the simplified version, and (ii) the full version. The simplified version of 2Q, namely *S2Q*, caches the most recent accessed pages in an LRU queue, and stores identifiers of frequently accessed pages in a FIFO queue. Since this FIFO queue is equivalent to the read queue *RQ* of eFIND (Section 6.4.2), S2Q can employ our *RQ* for its management. The full version of 2Q, namely *F2Q*, caches frequently accessed pages in a FIFO queue, stores recently accessed pages in an LRU queue, and maintains the identifiers of pages with at least two accesses in a FIFO queue. The later FIFO queue is also compatible with *RQ* of eFIND.

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

Figure 30 shows that the combination of 2Q algorithms with the temporal control improved the performance not only when building indices (Figures 30a and b), but also when processing IRQs (Figures 30c and d). Regarding index construction, *S2Q TC* and *F2Q TC* showed the best elapsed times. *Without TC* showed to be inefficient because of interleaved reads and writes. Although *LRU TC* improved the performance of *Without TC*, the use of LRU did not show the best results because it did not provide a full integration with the temporal control. Considering the index pages of 2KB and 4KB (Figure 30b), the performance gains of *S2Q TC* and *F2Q TC* over *LRU TC* were between 2% and 4%, respectively. This means that the temporal control using both versions of 2Q showed best performance because it prefetches frequently accessed nodes and provides specific write patterns to improve index performance on SSDs.



(a) Index construction (b) Plot from the dashed area of (a)

(c) IRQs with low selectivity (d) Plot from the dashed area of (c)

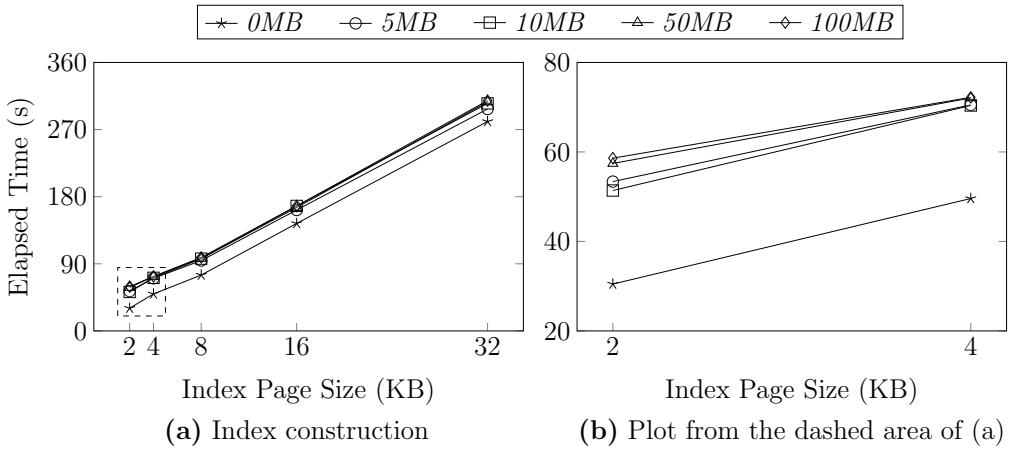
Figure 30 – Effect of the temporal control when building eFIND R-trees (a) and when executing IRQs with low selectivity (c). The temporal control with the S2Q, *S2Q TC*, showed the best results as visualized in the zoomed areas (b) and (d).

As for the execution of IRQs (Figures 30c and d), *S2Q TC* showed the best elapsed times when considering the index pages of 16KB and 32KB (Figures 30d). While LRU faced problems regarding the cost of loading objects from the main memory, F2Q required complex management of the cached objects. Hence, *S2Q TC* guaranteed reductions of 8% and 10% over *LRU TC*, and reductions of 2% and 9% over *F2Q TC*.

6.6.4 Effect of the Log Size

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

Compared to the non-support for data durability (i.e., *0MB*), compacting the log clearly required extra time (Figure 31a). But, this extra time decreased as the size of the index page increased. This is related to the efficiency of the compaction, which refers to the number of log entries that can be discarded. A log entry is only discarded if its index page was already flushed.



(a) Index construction (b) Plot from the dashed area of (a)

Figure 31 – Effect of the log size when building eFIND R-trees (a). Although the additional cost, compacting the log is needed to better use the space allocated to provide data durability. The log size *10MB* showed the best results for small index pages (b).

For small index pages, a low number of log entries is discarded because of the high range of created pages. In this case, the compaction was inefficient and was executed multiple times. On the other hand, the range of created pages was low for large index pages, increasing the number of discarded log entries and improving the compaction of the log. Hence, for the page size of 32KB, compacting the log required the lowest overhead, varying from 6% to 8%.

The size of the log also impacts on the performance of its compaction. In general, building indices took more time for large log files (i.e., *50MB* and *100MB*) since the compaction log algorithm traverses all log entries to compact the log. But, the size of the log cannot be very small; otherwise, the compaction is inefficient. Our experiments showed best elapsed times for the log sizes of *5MB* and *10MB* (Figure 31b), showing reductions from 3% to 12% compared to *50MB* and *100MB*.

Another aspect that affects the efficiency of the compaction log is the size of the write buffer. If the write buffer is very large, the compaction of the log file is not so efficient because only a few number of flushing operations is made. Further, if the log size is also very small, probably the log file cannot be compacted because flushing operations were not executed. In this case, the compaction algorithm should force the execution of some flushing operations to compact the log file, if space is critical for the application. These aspects are also discussed in Sarwat *et al.* (2013).

6.6.5 Summary of the Effects of the Design Goals

Throughout Section 6.6, we have evaluated the effects of each design goal by employing an incremental strategy. That is, we first discovered the best parameters related to the write buffer and the flushing operation without considering the read buffer and temporal control. Then, we added the support for the read buffer, and subsequently, considering the best configuration of

the read buffer, we added the support for the temporal control. Finally, we varied the log size to analyze its effect. This incremental strategy allowed us to understand the effects of each design goal isolatedly. The summary of the main findings are detailed as follows.

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

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

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

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

6.7 Related Work

Since many characteristics of flash-aware spatial indices are inspired by buffer managers and unidimensional indices for flash memory, in Sections 6.7.1 and 6.7.2 we present overviews

of *flash-aware buffer managers* and *flash-aware unidimensional indices* respectively. Next, in Section 6.7.3 we detail existing flash-aware spatial indices and compare them according to our design goals.

6.7.1 An Overview of Flash-Aware Buffer Managers

Database buffer management is a traditional field that provides general solutions to speed up accesses to storage devices (EFFELSBERG; HAERDER, 1984). Commonly, a page replacement algorithm is employed by a buffer to decide which pages should be maintained in the main memory. For HDDs, we can cite the classical Least Recently Used (LRU) replacement algorithm (DENNING, 1980), the two versions of the 2Q replacement algorithm (JOHNSON; SHASHA, 1994), and the Adaptive Replacement Cache (ARC) (MEGIDDO; MODHA, 2003) as examples. There are also general buffers proposed to deal with the intrinsic characteristics of flash memory. In this section, we provide the central idea of some flash-aware buffer managers.

Many flash-aware buffer managers are based on the LRU. The *Clean-First LRU* (*CFLRU*) (PARK *et al.*, 2006) divides the LRU into two regions, the working region that stores recently used pages, and the clean-first region that stores candidates for eviction. CFLRU prioritizes clean pages, stored in the clean-first region, to be evicted. The *Clean-First Dirty-Clustered* (*CFDC*) (OU; HÄRDER; JIN, 2010) improves the page replacement of CFLRU by dividing the clean-first region into two other regions. CFDC also changes the priority of the candidates for eviction by considering their locality in the flash memory. The *Flash-based Operation-aware buffer Replacement*$^+$ (*FOR*$^+$) (LV *et al.*, 2011a) is also based on the LRU and divides it into two regions, hot and cold, according to the access frequency. Each page cached in the FOR$^+$ has a weight that is based on region membership, which is used for eviction. Finally, the *Flash Based-ARC* (*FBARC*) (DUBS *et al.*, 2013) is based on the ARC. FBARC distinguishes itself by creating a write list that uses the locality of evicted pages as a temporal control to produce semi-sequential write patterns.

Commonly, these general-purpose buffer managers do not use special knowledge from the data structure in their page replacement algorithms. That is, they are not specially designed to deal with index structures. As we discuss in Sections 6.7.2 and 6.7.3, proposals of unidimensional and spatial indices usually adapt well-known buffer managers to include specific characteristics of index structures. For instance, instead of storing an entire modified index page in an in-memory buffer, flash-aware indices often store modified entries only, improving memory management. Another example is the development of specialized flushing algorithms that consider both intrinsic characteristics of SSDs and index structures.

6.7.2 An Overview of Flash-Aware Unidimensional Indices

Unidimensional indices manipulate alphanumeric data in order to deliver fast search operations. For HDDs, we can cite the traditional B-tree and its variants, the B+-tree and the B*-tree, as examples (CORMER, 1979). Motivated by the positive characteristics of SSDs, researchers proposed adaptations for the B-tree and its variants (WU; KUO; CHANG, 2007; JIN *et al.*, 2016) or even new tree structures (BYUN; HUR, 2009; AGRAWAL *et al.*, 2009; LI *et al.*, 2010) for indexing alphanumeric data on SSDs efficiently. Our goal here is not to describe in detail all flash-aware unidimensional indices, but show their characteristics to index data on SSDs.

The main focus of the flash-aware unidimensional indices is to deal with the poor performance of random writes. The first strategy employed to this end is to possibly execute an excessive number of sequential writes and random reads to avoid random writes as much as possible. Examples of flash-aware unidimensional indices that employ this strategy are the *CHC-tree* (BYUN; HUR, 2009) and the *Lazy-Adaptive tree* (AGRAWAL *et al.*, 2009).

Another strategy is to store index modifications in an in-memory buffer and flush them in batch when space is needed. Existing flash-aware unidimensional indices mainly differ on how the write buffer is managed. The *B-tree over the FTL* (WU; KUO; CHANG, 2007) makes use of logical addresses of the FTL to organize the write buffer and packs the modifications into logical blocks of the FTL in a flushing operation. The *FD-tree* (LI *et al.*, 2010) focuses on large-capacity SSDs and organizes the write buffer in different levels of the tree, respecting ascending order. The *read/write optimized B+-tree* (JIN *et al.*, 2016) allows overflowed nodes to reduce random writes and leverages Bloom filters to reduce extra reads to these overflowed nodes.

6.7.3 Flash-Aware Spatial Indices

There are few flash-aware spatial indices proposed in the literature. The *R-tree over FTL* (*RFTL*) (WU; CHANG; KUO, 2003), is a straightforward extension of the R-tree. It does not change the structure of the R-tree and only employs a write buffer to deal with the well-known poor performance of random writes of SSDs. The RFTL faces two main problems. First, the write buffer does not store the results of the modifications of nodes, but only how they must be performed. This means that to retrieve a node, the RFTL has to remake the modifications on this node, degrading search performance. Second, the flushing operation is an expensive operation since it writes all modifications stored in the write buffer.

The *Log-Compact R-tree* (*LCR-tree*) (LV *et al.*, 2011b) emerged to improve the flushing and search operations of the RFTL by using a log-structured format to store the modifications in its write buffer. However, the management of this write buffer requires an additional computational cost to keep the log-structured format. Another problem is the lack of a flushing policy for the flushing operation, which still requires long times to write all buffered modifications. In

addition, its write buffer does not store the results of the modifications.

The *aggregated RFTL* (*aRFTL*) (PAWLIK; MACYNA, 2012) was proposed to be used in decision making systems. It stores numeric values together with the MBR of the nodes to represent the different aggregations of a system. Since the aRFTL is only a simple extension of the RFTL, it does not advance on solving its aforementioned problems.

The *Framework for Search Trees* (*FAST*) (SARWAT *et al.*, 2013) is an upper-level solution that transforms any disk-based hierarchical index into a flash-aware index. To this end, FAST generalizes the write buffer to be applied to any hierarchical index, which includes unidimensional and spatial indices, such as the creation of the FAST R-tree from the R-tree. This buffer also stores the results of the modifications, improving search performance. On such buffer, FAST applies a specialized flushing algorithm to create space for new modifications. Another characteristic of FAST is its support for data durability. Despite the aforementioned positive characteristics, FAST faces the following problems. First, it can write a flushing unit containing a node without modification, resulting in unnecessary writes to the SSD. This is due to the static creation of flushing units as soon as nodes are created in the index. Second, the modifications of a node are stored in a list that allows repeated elements. That means this list can store the result of old modifications and a full scan is needed to retrieve the most recent version of a node.

In this article, we consider FAST as the state of the art and empirically compare it against eFIND in Section 6.5. FAST has the same goal of eFIND in terms of porting any tree structure for SSDs. However, eFIND distinguishes from FAST because it employs efficient in-memory data structures for the write buffer in order to store only the latest version of modified entries. Further, eFIND employs a flushing algorithm that does not lead to unnecessary writes to the SSD since it considers only the modifications stored in the write buffer. Finally, eFIND provides a read buffer and a temporal control to deal with other intrinsic characteristics of SSDs that FAST does not consider (see Table 15). These advantages contributed to the performance gains of eFIND, as detailed in Section 6.5.

The *Flash-Optimized R-tree* (*FOR-tree*) (JIN *et al.*, 2015) improves the flushing algorithm of FAST by dynamically creating flushing units with only the modifications stored in the write buffer. It also abolishes splitting operations of full nodes by allowing overflowed nodes. When a specific number of accesses in an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting elements in its parent, growing up the tree if needed. However, the number of accesses of an overflowed root node is never incremented in an insertion operation. As a consequence, spatial objects are stored in the overflowed root node in a sequential form when building an index. This critical problem disallowed us to execute our experiments since the FOR-tree failed to construct the index over our dataset.

There are also flash-aware spatial indices for multidimensional points (LIN *et al.*, 2006; LI *et al.*, 2013; FEVGAS; BOZANIS, 2015). A common limitation of these indices is that they mainly focus on two-dimensional points only. *MicroHash* and *MicroGF* (*MH & MGF*) (LIN

Table 15 – Comparison of flash-aware spatial indices according to our design goals (Section 6.3).

	Write buffer	*Specialized flushing algorithm*	*Read buffer*	*Temporal control*	*Data durability*
RFTL (WU; CHANG; KUO, 2003)	✓				
LCR-tree (LV *et al.*, 2011b)	✓				
aRFTL (PAWLIK; MACYNA, 2012)	✓				
FAST (SARWAT *et al.*, 2013)	✓	✓			✓
FOR-tree (JIN *et al.*, 2015)	✓	✓			
MH & MGF (LIN *et al.*, 2006)	✓				
F-KDB (LI *et al.*, 2013)	✓	✓			
GFFM (FEVGAS; BOZANIS, 2015)	✓	✓			
eFIND	✓	✓	✓	✓	✓

et al., 2006) are index structures to execute spatial queries on flash-based sensor devices with very limited main memory and low processing capabilities. Since these indices are designed to sensor devices, they employ write buffers only. The *K-D-B-tree over flash memory* (*F-KDB*) (LI *et al.*, 2013) adapts the K-D-B-tree (ROBINSON, 1981) to be implemented over the FTL. It employs a write buffer that stores modified entries of the K-D-B-tree, called logging entries. When its write buffer is full, a flushing policy selects some logging entries to be written. The main problem of F-KDB is that retrieving a node is a complex operation since logging entries of a node might be scattered over different flash pages. Finally, the *Grid file for flash memory* (*GFFM*) (FEVGAS; BOZANIS, 2015) employs a buffer strategy based on the LRU to cache modifications of the grid file (NIEVERGELT; HINTERBERGER; SEVCIK, 1984). A flushing operation only writes to the SSD those index pages that are classified as cold pages. However, the number of modifications is not taken into account, leading to a possibly high number of flushing operations.

To the best of our knowledge, there is no flash-aware spatial index that fulfills all design goals of Section 6.3, as shown in Table 15. Existing flash-aware spatial indices do not improve the performance of reads and do not avoid interleaved reads and writes. Among them, FAST provides the best characteristics. Hence, we consider FAST as the state of the art in spatial indexing for SSDs by comparing it in our experiments.

On the other hand, eFIND consists of sophisticated managers that fulfill all design goals. In this article, we extend the first version of eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b) as follows. First, we leverage efficient data structures to manipulate the write buffer and improve the computational complexity of the flushing algorithm. Second, we generalize the read buffer to accept a read buffer replacement policy as a parameter. Third, we refine the temporal control algorithms of eFIND and link them to the read buffer. Finally, we specify an algorithm to compact the log used for guaranteeing data durability. Another important extension refers to the performance tests. Here, we measure the efficiency of eFIND by considering two SSDs with different characteristics and by analyzing the impact of each design goal of eFIND.

6.8 Conclusions and Future Work

This article proposes eFIND, a new generic and efficient framework that transforms disk-based spatial indices into flash-aware spatial indices. eFIND is generic because it can be applied in a wide range of spatial indices, such as the R-tree (GUTTMAN, 1984), the R*-tree (BECKMANN *et al.*, 1990), the Hilbert R-tree (KAMEL; FALOUTSOS, 1994), and the xBR+-tree (ROUMELIS *et al.*, 2017), without changing original algorithms of the underlying index. Instead, it only changes the way in which reads and writes are performed on the SSD. This allows us to integrate eFIND into spatial database systems with a low-cost implementation.

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

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

Future work will deal with a number of topics. Currently, eFIND does not provide support for parallel transactions and concurrency control. Hence, a future topic is to include this support by also considering the ACID properties (HARDER; REUTER, 1993). Another future topic is to evaluate eFIND by using other underlying spatial indices, such as the R*-tree, the Hilbert R-tree, and the xBR+-tree. In addition, the internal structure of these indices might be changed, allowing us to analyze the performance of different spatial organizations on SSDs. To this end, we aim to execute workloads containing different types of spatial queries (e.g., point query and containment range query) on large spatial datasets with different types of spatial data (e.g., points, lines, and regions). We further plan to execute these extended workloads by employing *flash simulators* (SU *et al.*, 2009; KIM *et al.*, 2009; CARNIEL; SILVA; CIFERRI, 2018). The objective is to evaluate the performance of eFIND-based indices when using different configurations of SSDs.

Finally, by considering the emerging of *non-volatile main memories* (NVMM) like ReRAM, STT-RAM, and PCM (MITTAL; VETTER, 2016; ZHANG; SWANSON, 2015), we

also aim to port eFIND for these memories. Although many characteristics of NVMMs are similar to SSDs (e.g., asymmetric costs of reads and writes), eFIND would not exploit all the advantages of NVMMs. The main reason is that NVMMs are byte-addressable, allowing to access persistent data with CPU load and store instructions. Therefore, our last future topic is to study how eFIND should be extended to deal with intrinsic characteristics of NVMMs.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil (CAPES) - Finance Code 001. This work has also been supported by the Brazilian federal research agency CNPq, as well as by the São Paulo Research Foundation (FAPESP). Anderson C. Carniel has been supported by the grant #2015/26687-8, FAPESP. Ricardo R. Ciferri has been supported by the grant #311868/2015-0, CNPq. Cristina D. A. Ciferri has been supported by the grant #2018/22277-8, FAPESP.

CHAPTER

7

SPATIAL INDEXING ON FLASH-BASED SOLID STATE DRIVES

This Chapter attaches the following paper (CARNIEL, 2018):

- Carniel, A. C. Spatial Indexing on Flash-based Solid State Drives. In Proceedings of the VLDB 2018 PhD Workshop (VLDB 2018), p. 1-4, 2018.

This paper extends a previous paper that was also discussed in another forum for PhD students (CARNIEL; CIFERRI, 2016):

- Carniel, A. C.; Ciferri, C. D. A. Spatial Indexing in Flash Memories: Proposal of an Efficient and Robust Spatial Index with Durability. In Proceedings of the Satellite Events of the 31st Brazilian Symposium on Databases (SBBD 2016), p. 66-73, 2016.

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%.

7.1 Introduction

Spatial database systems largely employ spatial indices to process spatial queries (GAEDE; GÜNTHER, 1998), 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) (EMRICH *et al.*, 2010; CARNIEL; CIFERRI; CIFERRI, 2017a). 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 (JUNG; KANDEMIR, 2013). 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 (WU; CHANG; KUO, 2003; LV *et al.*, 2011b; SARWAT *et al.*, 2013; JIN *et al.*, 2015).

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 (JUNG; KANDEMIR, 2013). Third, there is no special treatment to minimize the effects of interleaved reads and writes, which also negatively impact SSD performance (JUNG; KANDEMIR, 2013). 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 (CARNIEL; CIFERRI; CIFERRI, 2017a). 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) (CARNIEL; CIFERRI; CIFERRI, 2017b). 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.

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 7.2 surveys related work. Section 7.3 summarizes our studies on the impact of SSDs on the spatial indexing. Section 7.4 introduces our design goals and eFIND. Finally, Section 7.5 concludes the paper and presents future work.

7.2 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 Emrich *et al.* (2010), 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* (AGRAWAL *et al.*, 2009)) 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* (WU; CHANG; KUO, 2003) 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* (LV *et al.*, 2011b) 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 (SARWAT *et al.*, 2013) 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 Carniel, Ciferri and Ciferri (2017b).

The *FOR-tree* (JIN *et al.*, 2015) 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.

7.3 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 (CARNIEL; CIFERRI; CIFERRI, 2017a). Considering a dataset extracted from the OpenStreetMap containing 534,926 regions (specified

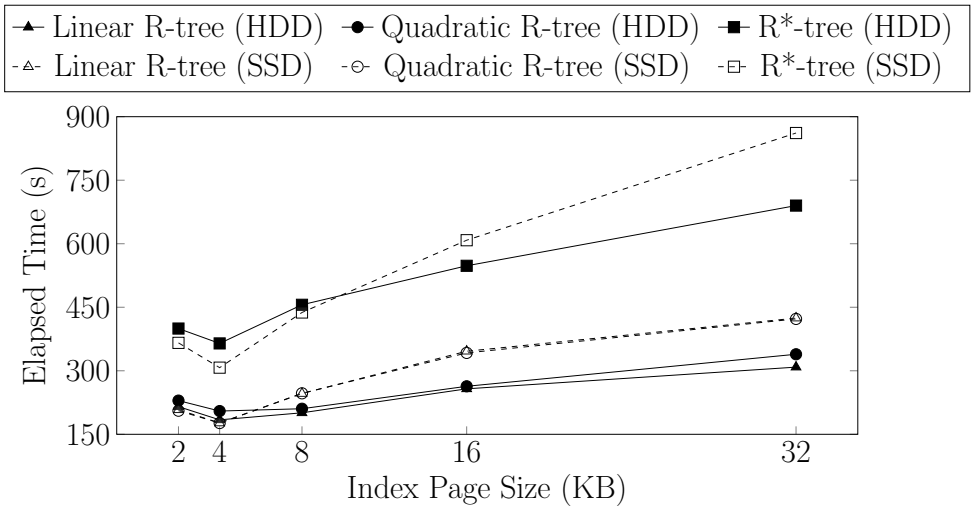


Figure 32 – Performance results when creating R-trees and R*-trees on an HDD (denoted by filled marks) and on an SSD (denoted by empty marks).

in Carniel, Ciferri and Ciferri (2017c)), here we report the elapsed time when creating R-trees (varying the split algorithm) and R*-trees on an HDD and SSD (Figure 32). 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 Jung and Kandemir (2013). 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 7.4.

7.4 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 7.4.1), and by proposing eFIND (Section 7.4.2).

7.4.1 Design Goals

Our five design goals are inspired by analysis of experimental evaluations on SSDs (Section 7.3) and techniques used by existing indices for flash memory (Section 7.2). 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 (JUNG; KANDEMIR, 2013). 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 (SARWAT *et al.*, 2013). 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 (JUNG; KANDEMIR, 2013). 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 (JUNG; KANDEMIR, 2013). 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 7.2). Therefore, we consider FAST as the state of the art in spatial indexing for SSDs by comparing it in our experiments.

7.4.2 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 7.4.1. More details about eFIND are given in Carniel, Ciferri and Ciferri (2017b).

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 Carniel, Ciferri and Ciferri (2017c)). 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 (GAEDE; GÜNTHER, 1998). 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.

Index Construction. As shown in Figure 33a, 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

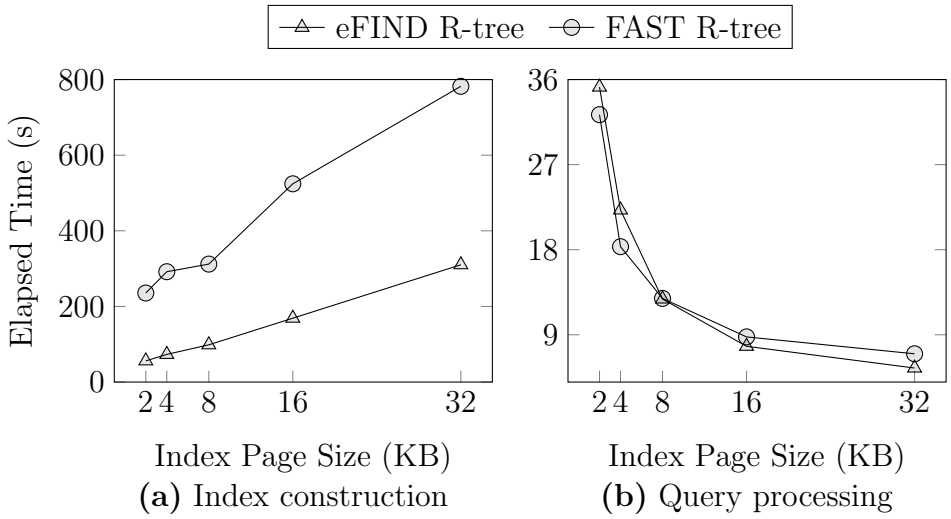


Figure 33 – The eFIND R-tree showed expressive performance gains when building spatial indices (a). It also showed the best performance to process the IRQs when using large index page sizes (b).

goals defined in Section 7.4.1. 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 33b 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.

7.5 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.

Acknowledgments

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

CHAPTER

8

AN EFFICIENT FLASH-AWARE SPATIAL INDEX FOR POINTS

This Chapter attaches the following paper (CARNIEL *et al.*, 2018):

- Carniel, A. C.; Roumelis, G.; Ciferri, R. R.; Vassilakopoulos, M.; Corral, A.; Ciferri, C. D. A. An Efficient Flash-aware Spatial Index for Points. In Proceedings of the XIX Brazilian Symposium on GeoInformatics (GEOINFO 2018), p. 68-79, 2018.

Abstract

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

8.1 Introduction

The use of a spatial index is essential for processing spatial queries because the search space is greatly reduced (GAEDE; GÜNTHER, 1998). The main assumption of several spatial

indices is that the spatial objects are stored in magnetic disks (i.e., *Hard Disk Drives* - HDDs). Hence, they often consider the slow mechanical access and the high cost of search and rotational delay of disks in their design. We term spatial indices designed for magnetic disks as *disk-based spatial indices*.

A wide range of disk-based spatial indices has been proposed in the literature (GAEDE; GÜNTHER, 1998). The R-tree and its variants, such as the R^+-tree and the R*-tree, are well-known spatial indices. The efficient indexing of multidimensional points has been a main focus of several indices because of the use of points in real spatial database applications (GAEDE; GÜNTHER, 1998). Among the existing disk-based spatial indices, we highlight the xBR^+-tree (ROUMELIS *et al.*, 2015), which provides data structures and algorithms for handling points efficiently. In fact, extensive experimental evaluations (ROUMELIS *et al.*, 2017) showed that the xBR^+-tree outperforms variants of the R-tree (the R*-tree and the R^+-tree) when processing different types of spatial queries.

On the other hand, advanced database applications are interested in using modern storage devices like *flash-based Solid State Drives* (SSDs) (MITTAL; VETTER, 2016). This includes spatial database systems that employ spatial indices to efficiently retrieve spatial objects stored in SSDs (CARNIEL; CIFERRI; CIFERRI, 2017a). The main reason of this interest is because SSDs, in contrast to HDDs, have smaller size, lighter weight, lower power consumption, better shock resistance, and faster reads and writes.

However, SSDs have introduced a new paradigm in data management because of their intrinsic characteristics (JUNG; KANDEMIR, 2013; MITTAL; VETTER, 2016). A well-known characteristic is the asymmetric cost of reads and writes, where a write requires more time and power consumption than a read. Further, SSDs are able to write data to empty pages only, which means that updating data in previously written pages requires an erase-before-update operation. Other factors that impact on SSD performance are the processing of interleaved reads and writes, and the execution of reads on frequent locations. These factors are related to the internal controls of SSDs, such as the internal buffers and the read disturbance management (JUNG; KANDEMIR, 2013).

To deal with the intrinsic characteristics of SSDs, spatial indices specifically designed for SSDs have been proposed in the literature. However, designing spatial indices for SSDs, termed here as *flash-aware spatial indices*, has been a challenging task. A common strategy is to mitigate the poor performance of random writes by storing index modifications in a write buffer. Whenever this buffer is full, a flushing operation is performed. Among existing flash-aware spatial indices proposed in the literature (see Section 8.2), FAST-based indices (SARWAT *et al.*, 2013) and eFIND-based indices (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018) distinguish themselves. FAST and eFIND are generic frameworks that transform disk-based hierarchical indices into flash-aware hierarchical indices. They also provide support for data durability by using a log-structured approach that allows to recover its write buffer after

a fatal problem (e.g., power failure). Comparing FAST to eFIND, the former does not fully exploit SSD performance because it does not consider several intrinsic characteristics of SSDs. On the other hand, eFIND contains managers based on a set of design goals that are developed to fully take into account the intrinsic characteristics of SSDs. Hence, we consider eFIND as the state-of-the-art method for porting disk-based spatial indices to SSDs.

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

The rest of this paper is organized as follows. Section 8.2 surveys related work. Section 8.3 presents the eFIND xBR$^+$-tree. Section 8.4 discusses the conducted experiments. Finally, Section 8.5 concludes the paper and presents future work.

8.2 Related Work

A few flash-aware spatial indices have been proposed in the literature. In this section, we summarize the characteristics of the main flash-aware spatial indices as follows.

The *RFTL* (WU; CHANG; KUO, 2003) ports the R-tree to SSDs using a write buffer to avoid random writes. The main problem of RFTL is the flushing operation because it flushes all modifications stored in the write buffer, requiring high elapsed times. Another problem is related to the data durability. This means that the modifications stored in the write buffer are lost after a system crash or power failure.

FAST (SARWAT *et al.*, 2013) distinguishes itself because it generalizes the write buffer to store modifications of any hierarchical index. Hence, it transforms any disk-based hierarchical index into a flash-aware index. Further, FAST provides a specialized *flushing algorithm* that picks only a set of nodes, termed *flushing unit*, to be written to the SSD instead of writing all modifications contained in the write buffer. FAST also provides support for data durability. However, FAST faces several problems. First, its flushing algorithm might pick nodes without modifications, resulting in unnecessary writes to the SSD. This is due to the static creation of flushing units as soon as nodes are created in the index. Second, its write buffer stores the modifications in a list possibly containing repeated entries, impacting negatively the performance

of retrieving modified nodes. Finally, FAST does not improve the performance of reads.

The *FOR-tree* (JIN *et al.*, 2015) improves the flushing algorithm of FAST by dynamically creating flushing units containing modified nodes only. It also abolishes splitting operations by allowing overflowed nodes. Whenever a specific number of accesses in an overflowed node is reached, a merge-back operation is invoked. This operation eliminates overflowed nodes by inserting them into the parent node, growing up the tree if needed. However, the number of accesses of an overflowed root node is never incremented in an insertion operation. As a consequence, the construction of a FOR-tree, inserting one spatial object by time, forms an overflowed root node instead of a hierarchical structure. This critical problem disallowed us to create spatial indices over large and medium spatial datasets.

Specific flash-aware spatial indices for points have also been developed. *MicroHash* and *MicroGF* (LIN *et al.*, 2006) are data structures for flash-based sensor devices. Due to the low processing capabilities of sensor devices, they deploy write buffers only. The *F-KDB* (LI *et al.*, 2013) employs a write buffer that stores modified entries of the K-D-B-tree, called logging entries. Its main problem is the complex operation to retrieve nodes because the entries of a node might be stored in different flash pages. Finally, the *Grid file for flash memory* (FEVGAS; BOZANIS, 2015) employs a buffer strategy based on the LRU to cache modifications of the grid file. A flushing operation writes to the SSD only those index pages that are classified as cold pages. However, the quantity of modifications is not considered, leading to a possibly high number of flushing operations.

eFIND (CARNIEL; CIFERRI; CIFERRI, 2017b; CARNIEL; CIFERRI; CIFERRI, 2018) is a generic framework that efficiently transforms any disk-based spatial index into a flash-aware spatial index. It is based on distinct design goals that considers the intrinsic characteristics of SSDs. eFIND employs efficient in-memory data structures to handle index modifications and a specialized flushing operation that smartly picks a number of nodes to be written to the SSD. Further, eFIND prevents reads on frequent locations and avoids interleaved reads and writes. Due to its advantages, we consider eFIND as the state-of-the-art method for porting disk-based spatial indices to SSDs. Hence, we employ eFIND to port the xBR$^+$-tree to SSDs.

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

8.3.1 The Tree Structure

eFIND does not change the underlying tree structure of the ported index. Hence, the tree structure of the eFIND xBR$^+$-tree is the same as the xBR$^+$-tree. The xBR$^+$-tree is a hierarchical index based on the regular decomposition of space of Quadtrees (GAEDE; GÜNTHER, 1998) able to index multidimensional points. Hence, it is a *space-driven access method*. For

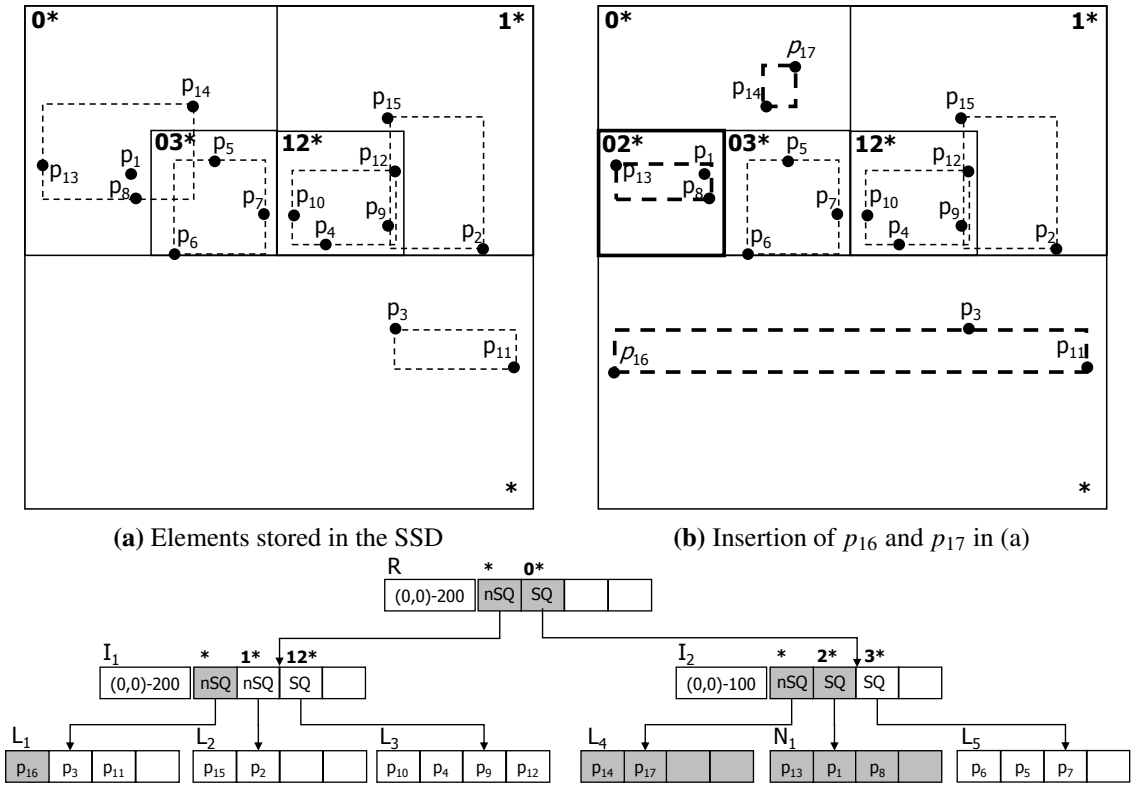
bidimensional points, the xBR$^+$-tree decomposes recursively the space by 4 equal quadrants, called *sub-quadrants*. Figure 34a depicts an example of an eFIND xBR$^+$-tree that indexes 15 points (i.e., p_1 to p_{15}) and is whole stored in the SSD. Figure 34b shows the eFIND xBR$^+$-tree with a set of adjustments, represented by thick lines, after the insertion of two new points, p_{16} and p_{17}. These points and the resulting adjustments are modifications stored in the main memory (Section 8.3.2) and are also highlighted in the hierarchical representation of Figure 34c. We detail the structure of this eFIND xBR$^+$-tree as follows.

There are two types of nodes, internal nodes and leaf nodes. Internal nodes consist of entries in the following format $(p, DBR, qside, shape)$. Each entry of an internal node refers to a child node that is pointed by p and represents a sub-quadrant of the original space. *DBR* refers to the data bounding rectangle that minimally encompasses the points stored in such sub-quadrant. *qside* stores the side length of the sub-quadrant corresponding to the child node's entry. Finally, *shape* is a flag that indicates if the sub-quadrant is either a complete square or a non-complete square. The entries of an internal node are also sorted by their *addresses*. Each address is calculated by using *qside* and *DBR*, and consists of a sequence of *directional digits* representing a sub-quadrant. The directional digits 0, 1, 2, and 3 respectively symbolize the NW, NE, SW, and SE sub-quadrants of a relative space. Hence, it follows the Z-order.

Figure 34c depicts a tree with 3 internal nodes, R, I_1, and I_2. Each internal node has also a header containing data about its sub-quadrant. For instance, the origin point of the sub-quadrant of R is $(0,0)$ with a side length of 200. The address of each entry of an internal node is showed in bold (but, this is not actually stored). For instance, the right child of R that points to I_2 is the NW quadrant of the original space, denoted as *0** (* is used to mark the end of the address). Further, it represents a complete square (i.e., *SQ*). Its *DBR* consists of a minimum bounding rectangle containing the points p_5 to p_8, p_{14}, p_{17}, p_{13}, and p_1. The left child of R represents a region derived from the spatial difference between the original space and the region of the NW quadrant. Hence, it has address equal to * (i.e., empty) and represents a non-complete square (i.e., *nSQ*). Finally, addresses of entries of internal nodes determine a sub-quadrant in relation to the region of their node. For instance, the address *3** (in node I_2 of Figure 34c) represents the SE sub-quadrant of the NW sub-quadrant of the original space (the region of I_2, denoted by *0** in R of Figure 34c).

Leaf nodes contain entries in the format (p, id), where p is the multidimensional point and *id* is a pointer to the register of p. These entries are sorted by X-axis coordinates of the points, allowing the use of the *plane sweep technique* in specific spatial query types. For instance, the leaf node L_1 in Figure 34c contains the points p_{16}, p_3, and p_{11}, which are sorted by their X-axis coordinates depicted in Figure 34b. The pointers to the registers of these points are omitted.

When the capacity of a leaf or internal node is achieved, the quadrant encompassing the overflowed node is partitioned into two sub-quadrants according to a Quadtree-like hierarchical decomposition. Different criteria for this partitioning are conceivable, as discussed in Roumelis



(a) Elements stored in the SSD

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

(c) Hierarchical representation of (b)

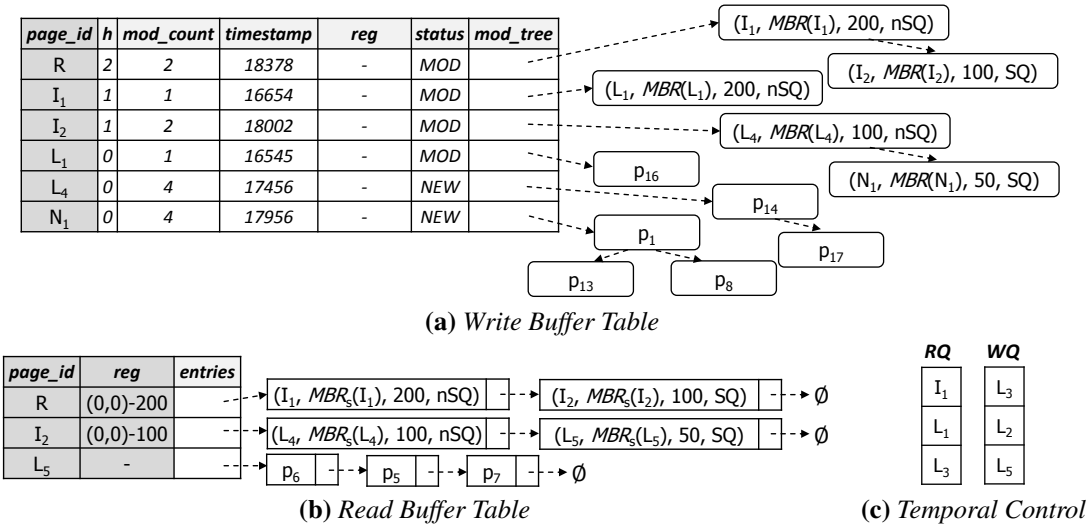
Figure 34 – An example of an eFIND xBR$^+$-tree.

et al. (2017). For instance, Figure 34b depicts the creation of a new sub-quadrant with address *02** (i.e., node N_1 in Figure 34c) resulting from a splitting operation after inserting p_{17}.

8.3.2 Employed Data Structures

eFIND provides specific data structures to fulfill its design goals (CARNIEL; CIFERRI; CIFERRI, 2018); they are: (i) a write buffer, (ii) a read buffer, (iii) a log file, and (iii) read and write queues. To deal with the xBR$^+$-tree, we extend the eFIND's data structures as follows: (i) we adapt the write and read buffers to store specific data related to internal nodes, (ii) we generalize the storage of index modifications according to the sorting properties of internal and leaf nodes (Section 8.3.1), and (iii) we adjust the structure of log entries to recover the write buffer after a system crash. We detail these extensions as follows.

The write buffer is implemented as a hash table named *Write Buffer Table* and stores the modifications of nodes that were not applied to the SSD yet. Its main goal is to avoid random writes to the SSD. The key of this hash table is the identifier of a node (*page_id*) and its value stores modifications in the format (*h*, *mod_count*, *timestamp*, *reg*, *status*, *mod_tree*). Here, *h* stores the height of the modified node; *mod_count* is the quantity of in-memory modifications; *timestamp* informs when the last modification was made; *reg* is the sub-quadrant of a newly created internal node; and *status* is the type of modification made and can be NEW, MOD,



(a) *Write Buffer Table*

(b) *Read Buffer Table* **(c)** *Temporal Control*

Figure 35 – Data structures to handle the eFIND xBR$^+$-tree of Figure 34.

or DEL for representing newly created nodes in the buffer, nodes stored in the SSD but with modified entries, and deleted nodes, respectively. If *status* is equal to DEL, *mod_tree* is null. Otherwise, it is a red-black tree containing the most recent version of modified entries. Each element of this red-black tree has the format (*e*, *mod_result*), where *e* is the key and corresponds to the unique identifier of an entry and *mod_result* stores the latest version of an entry, assuming null if *e* was removed. The comparison function to determine the order of the elements in the red-black tree is defined to deal with the specific sorting of entries of internal and leaf nodes (Section 8.3.1). This is important when retrieving nodes (Section 8.3.3).

Figure 35a shows the *Write Buffer Table* for the eFIND xBR$^+$-tree of Figure 34b. In this figure, *MBR* means the rectangle that encompasses all points of a sub-quadrant considering the modifications stored in the write buffer. The elements of the *mod_tree* employ the same format as an entry of the underlying index. For instance, the first line of the hash table in Figure 35a shows that *R*, located in the *height* 2, has the *status* MOD, and stores 2 in-memory modifications in the *mod_tree*. Hence, the most recent version of the two entries of *R* are now the entries of the red-black tree, i.e., $(I_1, MBR(I_1), 200, nSQ)$ and $(I_2, MBR(I_2), 100, SQ)$.

The read buffer is implemented as another hash table named *Read Buffer Table* and caches nodes stored in the SSD that are frequently accessed. The key of this hash table is the unique node identifier (*page_id*) and its value stores a list of entries of the node (*entries*) and its sub-quadrant, if it is an internal node (*reg*). Figure 35b depicts that *R*, I_2, and L_5 are cached in the *Read Buffer Table*. In this figure, MBR_S refers to the stored data bounding rectangle of a child node. For instance, the entries of the cached version of I_2 consists of two entries, even after the creation of N_1.

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

system crash. The cost of keeping the log of the modifications is very low because it requires sequential writes only (SARWAT *et al.*, 2013; CARNIEL; CIFERRI; CIFERRI, 2018).

The temporal control of eFIND remains unchanged. The read and writes queues, named *RQ* and *WQ*, are employed to provide the temporal control of eFIND. Each queue is a First-In-First-Out data structure. *RQ* stores identifiers of the nodes read from the SSD, while *WQ* keeps the identifiers of the last nodes written to the SSD. Figure 35c shows that the last read nodes are I_1, L_1, and L_3, and the last flushed nodes are L_3, L_2, and L_5.

8.3.3 Methods for Handling the Index Operations

eFIND provides generic algorithms to execute the following operations: (i) maintenance operation, which is responsible for reorganizing the index whenever modifications are made on the underlying spatial dataset (i.e., insertions, deletions, and updates); (ii) search operation, which is responsible for executing spatial queries; (iii) flushing operation, which selects a set of modifications stored in the write buffer to be written to the SSD according to a flushing policy; and (iii) restart operation, which rebuilds the write buffer after a fatal problem and compacts the log file. To deal with the xBR$^+$-tree, we extend eFIND as follows: (i) we generalize the retrieval algorithm of eFIND to return valid internal and leaf nodes, respecting their sorting properties (Section 8.3.1), and (ii) we detail the management of splits, which improves the space utilization of the write buffer.

To retrieve a node N, the eFIND xBR$^+$-tree takes two sorted lists as input: (i) the modified entries stored in the *Write Buffer Table*, and (ii) the entries stored in the SSD. The former is empty if N has not modifications, while the latter is empty if there exists a hash entry of N in the *Write Buffer Table* with *status* equal to NEW. If one list is empty, the other non-empty list is directly returned. The second list is always sorted because its first flushing happens when its status in the *Write Buffer Table* is equal to NEW.

The following merge operation should be performed if these lists are not empty. It is based on the classical merge operation between sorted files (FOLK; ZOELLICK; RICCARDI, 1997). Let i, j be two integer values, where i indicates the position in the first list and j indicates the position in the second list. A loop is then processed, starting with $i = j = 0$. If the element in the position i on the first list, called E_a, goes before the element in the position j on the second list, called E_b, this means that the merge operation appends E_a to N and increments i by 1 since an element of the first list has been processed. If the inverse happens, i.e., E_b goes before E_a, the merge operation appends E_b to N and increments j by 1. Evaluating the order of two node entries requires the execution of the same comparison function employed by the red-black trees (Section 8.3.1). If E_a and E_b point to the same entry (i.e., their unique identifier are equal), the merge operation appends only E_a to N if its value (i.e., *mod_result* in the *mod_tree*) is different to null and increment both i and j by 1. This is done because the result should only maintain the latest version of the entry and non-null entries. The loop is finished if i (j) is equal to the number

of entries in the first (second) list. Finally, the entries that were not evaluated by the loop are appended to N, which is returned as the final step of the merge operation.

The merge operation requires a cost of $\mathcal{O}(n+m)$, where n is the number of elements in the first list and m is the number of elements in the second list. The use of a red-black tree for storing modified entries is essential for the merge operation and represents a main advantage compared to FAST. First, it guarantees the order between the entries stored in the write buffer. Hence, the resulting node is valid. Second, it has an amortized cost of inserting and updating entries stored in the main memory. Finally, the space allocated in the main memory is better managed because it does not allow repeated elements. All these factors combined to the other eFIND's managers lead to a better performance compared to porting the xBR$^+$-tree using FAST, as reported in our experiments (Section 8.4).

Handling splitting operations in the write buffer is performed as follows. Let A be an overflowed node. First, if A has a hash entry in the *Write Buffer Table*, it assumes *status* equal to DEL, deleting previous modifications of A and thus freeing some space in the write buffer. Otherwise, a new hash entry, with *status* equal to DEL, in the *Write Buffer Table* is created. Then, after completing the splitting operation in the main memory, A has a new set of entries and a new node, called B, is created. Hence, the hash entry of A in the *Write Buffer Table* becomes NEW and the entries of A are added in its corresponding *mod_tree*. A similar procedure for B is employed. This strategy for handling splitting operations is important because of the management of the write buffer space. An example of handling of a splitting operation is depicted in Figure 34c, after inserting p_{17}. As a result, L_4 has 4 modifications (fifth line in the *Write Buffer Table* of Figure 35a), where one modification is related to its deletion, another modification for its creation, and then two modifications for inserting its two entries. Further, N_1 is newly created in the write buffer (last line in the *Write Buffer Table* of Figure 35a).

8.4 Experimental Evaluation

8.4.1 Experimental Setup

Datasets. We used two spatial datasets. The first one is a real spatial dataset, called *brazil_-points2017*, containing 770,842 points that represent geographical locations of Brazil like public telephones, ATMs, and towers. This dataset was extracted from the OpenStreetMap and its statistical description can be found in Carniel, Ciferri and Ciferri (2017c). The second one is a synthetic dataset containing 1,000,000 points equally distributed in 125 clusters uniformly distributed in the range $[0,1]^2$. The points in each cluster (i.e., 8,000 points) were located around the center of each cluster, according to Gaussian distribution.

Configurations. We compared two configurations: (i) the *FAST xBR$^+$-tree*, which is our closest competitor (Section 8.2), and (ii) the *eFIND xBR$^+$-tree*, which is our proposed index. We

created the FAST xBR$^+$-tree by extending FAST in an analogous way to the extensions we performed to eFIND. However, due to space limitations, this extension is not presented here. Both configurations had a buffer of 512KB, log capacity of 10MB, and employed index page sizes (i.e., node sizes) from 4KB to 32KB. For the FAST xBR$^+$-tree, we used the FAST* flushing policy, which provided the best results according to Sarwat et al. 2013. For the eFIND xBR$^+$-tree, we employed the best parameter values according to our experiments (CARNIEL; CIFERRI; CIFERRI, 2018): the use of 60% of the oldest modified nodes to create flushing units, a flushing policy using the height of nodes as weight to choose one flushing unit to be written, and the allocation of 20% of the buffer for the read buffer. Finally, both configurations employed the flushing unit size equal to 5 since this value commonly provide good results for FAST and eFIND (CARNIEL; CIFERRI; CIFERRI, 2018).

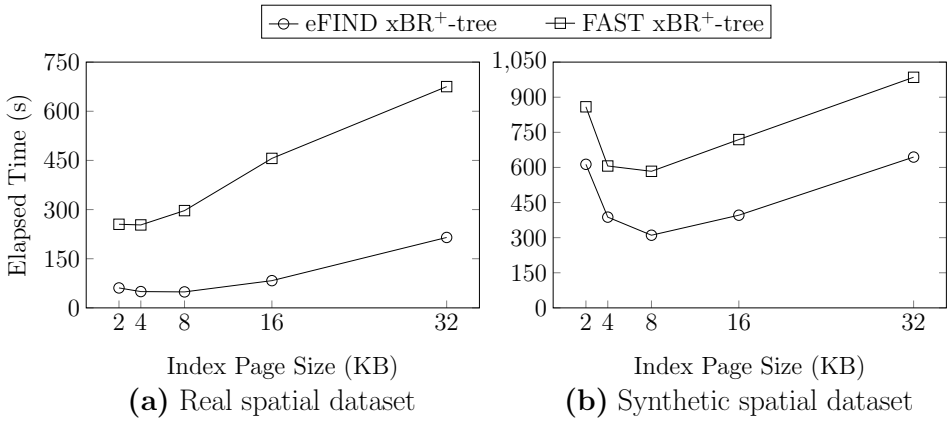
Workloads. We executed two workloads: (i) index construction, and (ii) execution of 300 intersection range queries (IRQs). An IRQ retrieves the points contained in a given rectangular query window, including its borders. Three different sets of query windows were used, representing respectively 100 rectangles with 0.001%, 0.01%, and 0.1% of the area of the total extent of the dataset being used by the workload. We generated different query windows for each dataset using the algorithms described in Carniel, Ciferri and Ciferri (2017c). This method allows us to measure the performance of spatial queries with distinct selectivity levels. We consider the selectivity of a spatial query as the ratio of the number of returned objects and the total objects; thus, the three sets of query windows built IRQs with low, medium, and high selectivity, respectively. We executed the workloads as a sequence, that is, the index construction followed by the execution of IRQs. For each configuration and dataset, this sequence was executed 5 times. We avoided the page caching of the system by using direct I/O. For the first workload, we collected the average elapsed time. For the second workload, we calculated the average elapsed time to execute each set of query windows.

Running Environment. We employed a server equipped with an Intel Core® i7-4770 with a frequency of 3.40GHz, 32GB of main memory, and the SSD Kingston V300 of 480GB. The operating system used was Ubuntu Server 14.04 64 bits.

8.4.2 Performance Results

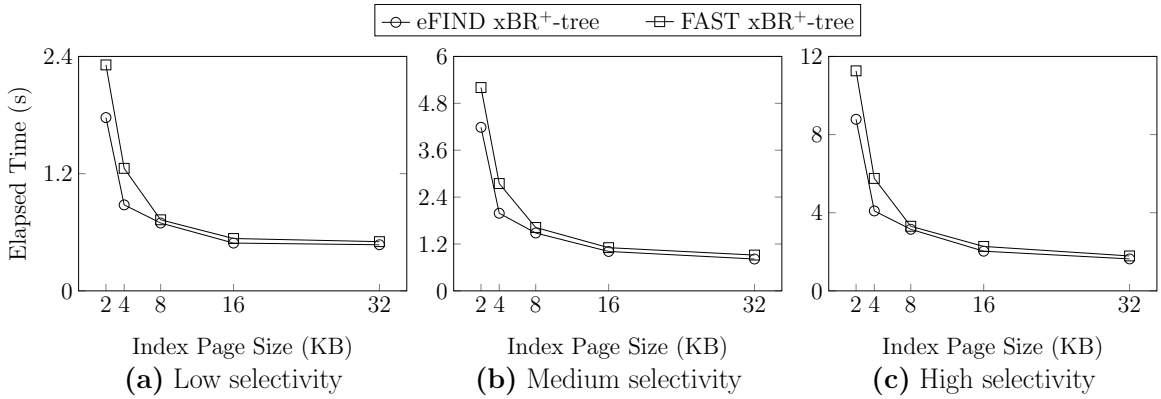
Index Construction. Figure 36 depicts that the eFIND xBR$^+$-tree overcame the FAST xBR$^+$-tree for both spatial datasets. The performance gains of the eFIND xBR$^+$-tree ranged from 68.1% to 83.5% for the real spatial dataset (Figure 36a) and from 28.4% to 46.5% for the synthetic spatial dataset (Figure 36b). A performance gain shows how much a configuration reduced the elapsed time from another configuration.

The eFIND xBR$^+$-tree exploited the benefits of the SSD because it leverages specific data structures and sophisticated methods that take into account the intrinsic characteristics of



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

Figure 36 – The eFIND xBR$^+$-tree showed the fastest elapsed times when building spatial indices over both spatial datasets.



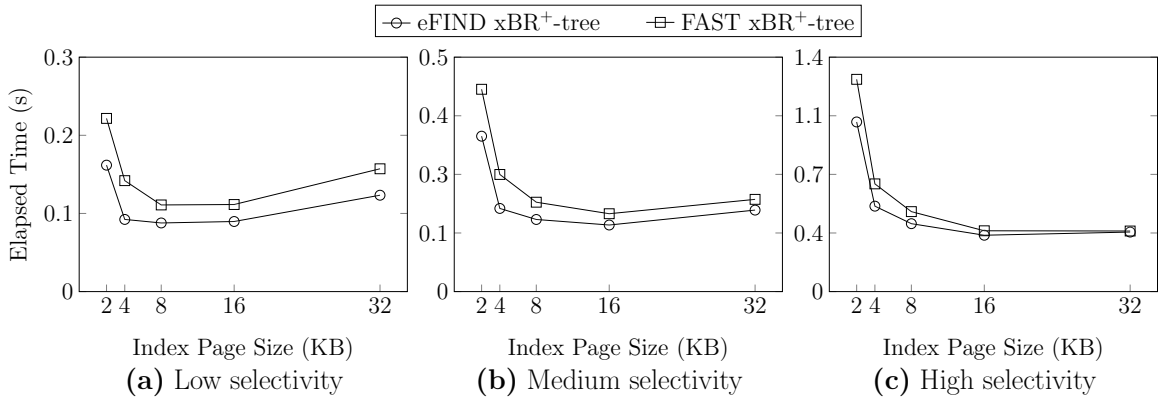
(a) Low selectivity (b) Medium selectivity (c) High selectivity

Figure 37 – Performance results when processing IRQs on the real spatial dataset. The eFIND xBR$^+$-tree outperformed the FAST xBR$^+$-tree for all selectivity levels, showing expressive performance gains.

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

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

Spatial Query Processing. Figures 37 and 38 depict that the eFIND xBR$^+$-tree always provided the best performance results when processing all selectivity levels of IRQs. For the real spatial



(a) Low selectivity (b) Medium selectivity (c) High selectivity

Figure 38 – Performance results when processing IRQs on the synthetic spatial dataset. The eFIND xBR$^+$-tree showed better elapsed time than the FAST xBR$^+$-tree for all selectivity levels.

dataset (Figure 37), the eFIND xBR$^+$-tree showed performance gains up to 29.4%, 27.2%, and 28.5% for the high, medium, and high selectivity levels, respectively. For the synthetic spatial dataset (Figure 38), it showed performance gains up to 34.6%, 28.6%, and 20.2% for the high, medium, and high selectivity levels, respectively. Similarly to our previous discussions, these performance gains were obtained thanks to the effective use of the merge operation and read buffer.

Processing IRQs over the synthetic dataset required much less time than processing IRQs over the real dataset because of its specific spatial distribution. In most of the cases, better elapsed times were obtained by using large page sizes (i.e., 16KB and 32KB) because more entries are loaded into the main memory with a few reads. IRQs returning more points (i.e., with high selectivity) exhibited higher elapsed times. This is due to the traversal of multiple large nodes in the main memory, requiring more CPU time than queries with low selectivity. This fact also contributed to a similar time among the configurations when processing IRQs with high selectivity using the page size of 32KB.

8.5 Conclusions and Future Work

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

The eFIND xBR$^+$-tree has empirically evaluated against the FAST xBR$^+$-tree, which employed FAST to port the xBR$^+$-tree to SSDs. The eFIND xBR$^+$-tree provided performance gains from 28.4% to 83.5% when building spatial indices and up to 34.6% when processing IRQs. In general, the page size of 16KB was the best configuration. Although this page size

required more time to build an index compared to smaller page sizes, it provided the best results to execute the IRQs. Hence, the cost of its construction can be suppressed by its efficiency when processing spatial queries.

The efficiency of the eFIND xBR$^+$-tree is obtained mainly because of two reasons. First, the internal structure of the xBR$^+$-tree was completely integrated to eFIND, guaranteeing all the properties of the xBR$^+$-tree that offer good spatial indexing performance. Second, eFIND is based on distinct design goals that fully exploit SSD performance.

Our future work includes to evaluate the eFIND xBR$^+$-tree against other spatial organizations, such as the data partitioning strategy of eFIND R-trees (CARNIEL; CIFERRI; CIFERRI, 2018). Another future work is to extend our experiments to consider workloads that mix insertions and other types of queries, such as point queries.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work has also been supported by CNPq and by the São Paulo Research Foundation (FAPESP). Anderson C. Carniel has been supported by the grants #2015/26687-8 and #2018/10687-7, FAPESP. Ricardo R. Ciferri has been supported by the grant #311868/2015-0, CNPq. Cristina D. A. Ciferri has been supported by the grant #2018/22277-8, FAPESP.

CHAPTER

9

CONCLUSIONS AND FUTURE WORK

This Chapter is organized as follows. Section 9.1 presents the final remarks of this PhD work, while Section 9.2 outlooks future work.

9.1 Conclusions

Spatial database systems and Geographic Information Systems (GIS) provide the foundation for applications managing spatial data (RIGAUX; SCHOLL; VOISARD, 2001; OOSTEROM, 2005). These systems widely employ spatial indexing structures to speed up the spatial query processing (GAEDE; GÜNTHER, 1998). Several spatial indices have been proposed in the literature assuming that magnetic disks (e.g., Hard Disk Drives - HDDs) are the underlying storage system. Hence, these indices, termed *disk-based spatial indices*, consider the slow mechanical access and the high cost of search and rotational delay of disks in their design. Examples of disk-based spatial indices include the well-known R-tree (GUTTMAN, 1984) and its variants, the R*-tree (BECKMANN *et al.*, 1990) and the Hilbert R-tree (KAMEL; FALOUTSOS, 1994), as well as Quadtree-based indices (SAMET, 1984) like the xBR$^+$-tree (ROUMELIS *et al.*, 2015).

Motivated by the interest of the spatial database and GIS communities on using flash-based Solid State Drives (SSDs), this PhD thesis focused on understanding and improving aspects related to the spatial indexing on SSDs. This is a challenging task since SSDs have intrinsic characteristics that must be take into account to exploit the benefits of these storage devices (AGRAWAL *et al.*, 2008; BOUGANIM; JÓNSSON; BONNET, 2009; CHEN; KOUFATY; ZHANG, 2009; JUNG; KANDEMIR, 2013; CHEN; HOU; LEE, 2016). For instance, a write requires more time and power consumption than a read. Other characteristics are related to the internal management of SSDs, such the disturbance of reads on frequent locations and the interference between reads and writes.

The first goal of this PhD work was to understand the impact of SSDs on the spatial

indexing context. To this end, extensive experimental evaluations was conducted aiming at analyzing the performance behavior of spatial indices on different storage devices. Such evaluations consistently employed real spatial datasets extracted from the OpenStreetMaps and search objects created by using a systematic methodology. In addition, FESTIval was developed as a solution for conducting experimental evaluations in a unique environment. As a result, these experiments can be reproduced and even extended by other researchers.

Initially, two main experiments were conducted with different perspectives: (i) to understand the performance relation of spatial indices managed on HDDs and SSDs, and (ii) to understand the applicability of flash simulators. As for the first perspective, the deficiencies of disk-based spatial indices have been observed by comparing their performance on different storage devices. The main finding was that the direct use of disk-based spatial indices on SSDs did not always guarantee good performance results. This was the case when the workload mix reads and writes. Hence, the design of the spatial indices should consider the intrinsic characteristics of SSDs, such as the poor performance of random writes and the interference between reads and writes. As for the second perspective, there is a correlation between the results obtained from the flash simulator and the results obtained from SSDs. We empirically observed that the index configurations that show the worst performance on the flash simulator in terms of number of reads, writes, and erases internally and actually performed on the emulated flash memory, tend to also report the worst performance result in the SSD. This means that we can reduce the number of configurations to be actually evaluated in SSDs based on the results of the flash simulator. However, the observational analysis exclusively based on the results of the flash simulator is not enough to determine the best index configuration.

Based on the lessons learned from the experiments and correlating them to the intrinsic characteristics of SSDs, eFIND is proposed. eFIND represents the main contribution of this PhD thesis. It is a new generic and efficient framework that transforms disk-based spatial indices into flash-aware spatial indices. It is generic because it can be applied in a wide range of spatial indices, such as the variants of the R-tree. The integration of eFIND in such indexing structures is very low because eFIND does not change the original data structures and algorithms of the underlying index. Instead, eFIND changes the way in which reads and writes are performed on the SSD in order to exploit its performance. eFIND is efficient because it is based on distinctive design goals that exploit the benefits of SSDs. The first design goal refers to the use of efficient in-memory data structures to manage the write buffer, which avoids random writes to SSDs. The second design goal is related to the specification of a flushing algorithm that employs a flushing policy to smartly pick a set of modified index pages to be sequentially written to the SSD. The third design goal includes a read buffer to improve the performance of random reads. The fourth design goal specifies data structures and algorithms to implement a temporal control for avoiding interleaved reads and writes. Finally, the last design goal refers to the guarantee of data durability by using a log-structured approach combined to the modifications stored in the write buffer.

The effect of each design goal was also studied in workloads that build indices and execute intersection range queries. The most significant impact is the use of the read buffer together with the temporal control, reducing the overhead in the internal management of SSDs. eFIND distinguishes from its closest competitor FAST (SARWAT *et al.*, 2013), which also ports disk-based spatial indices to SSDs, because of its efficiency. eFIND has been firstly applied to the R-tree, creating the eFIND R-tree, which outperformed the state-of-the-art FAST R-tree. Regarding the index construction, eFIND showed expressive performance gains that ranged from 43% to 77%. As for the query processing, eFIND showed performance gains from 4% to 23%.

eFIND was also applied to porting the xBR$^+$-tree to SSDs, creating a novel flash-aware spatial index for points, the eFIND xBR$^+$-tree. The data structures of eFIND allowed to be efficiently combined to the properties of the xBR$^+$-tree, such as the sorting of its internal and leaf nodes. Similarly to the porting of the R-tree, eFIND also showed expressive performance reductions compared to its closest competitor, FAST. The eFIND xBR$^+$-tree provided performance gains from 28.4% to 83.5% when building spatial indices and up to 34.6% when processing spatial queries.

9.2 Future Work

This PhD has opened new research topics that we plan to work. They are:

Porting disk-based spatial indexing structures to SSDs. Although the advantages of eFIND, designing an efficient flash-aware spatial index remains a challenging task yet. In fact, there are three open problems. First, since eFIND has been applied only to the R-tree and xBR$^+$-tree, it is still unclear how to adequately port other disk-based spatial indices to SSDs in a way that they exploit the advantages of SSDs. This leads to the second problem, how in-memory structures of eFIND should be adapted in order to fit well with the structure of the underlying index, which might be a data- or space-driven access method. Finally, the third problem refers to the lack of an extensive performance study that identifies the best index to handle spatial objects on SSDs. That is, identifying the best hierarchical structure for building indices and for processing spatial queries on spatial datasets storing point, line, and region objects. To solve these problems, the first future work topic includes the proposal of a *systematic methodology* for porting disk-based spatial indices to SSDs. This methodology should be derived from a combination of typical operations on index pages with the eFIND's data structures and algorithms, which in turn should be adapted and generalized to deal with the different indexing strategies. It is expected that, by using this methodology, we should be able to port other spatial indices to SSDs and perform an extensive performance evaluation, considering different spatial datasets, workloads, and spatial queries.

The proposal of a novel flash-aware spatial index. eFIND does not change the structure and algorithms of the underlying index being ported. This characteristic is very good for reducing

the cost of integrating eFIND into existing spatial database systems and GIS. However, a hypotheses is that if the structure of the spatial index also considers the design goals of eFIND, the performance of index operations can be improved. 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. This means that if we reduce the number of writes in the bottom levels by allowing overflowed nodes, we could reduce the propagation of modifications along the height of the tree without impairing the spatial organization of the index. Further, the buffer replacement policy of the read buffer should now prioritize those nodes in the highest levels of the tree. Hence, the second future work includes the *proposal of a novel flash-aware spatial index* that has its internal structure correlated to the design goals of eFIND. Such a new spatial index should be evaluated by means of experiments, considering disk-based spatial indices ported using eFIND, such as the eFIND R-tree and the eFIND xBR$^+$-tree.

A generic and efficient model to evaluate algorithms on flash memory. The use of flash simulators on the evaluation of spatial indices is a promising approach to reduce the time spent for conducting experiments and to minimize the overhead on the SSDs. However, this applicability has to be better studied since only observational analysis was conducted. The analysis can be improved if data mining and machine learning algorithms are also considered to discover patterns between the performance of a spatial index using a flash simulator and its performance on the SSD. For instance, the work in Carniel, Silva and Ciferri (2018) employs *associate rules* in order to identify the situations where a spatial index certainly show the worst performance on the SSD, using as a basis the performance results obtained from the flash simulator. Hence, the third future work includes the proposal of a methodology that integrates flash simulators in the performance evaluation of spatial indices. Such a methodology can be based on *machine learning models* and then be validated by using the experiments already conducted during this PhD work.

Spatial indexing on non-volatile main memories. Finally, the last future work considers the emerging of *non-volatile main memories* (NVMM) like ReRAM, STT-RAM, and PCM (MITTAL; VETTER, 2016; ZHANG; SWANSON, 2015). Memories have been constantly improved in the academia and industry. Hence, the concepts developed in the PhD work should be extended to deal with these NVMMs. Such storage devices have similar characteristics as the SSDs; but they are byte-addressable, permitting the direct access of persistent data by using CPU load and store instructions. Hence, a new paradigm is created, requiring to rethink how the spatial indexing can exploit the benefits of NVMMs. For instance, a question is how eFIND should be modified and extended to deal with NVMMs.

BIBLIOGRAPHY

AGRAWAL, D.; GANESAN, D.; SITARAMAN, R.; DIAO, Y.; SINGH, S. Lazy-adaptive tree: An optimized index structure for flash devices. **VLDB Endowment**, v. 2, n. 1, p. 361–372, 2009. Citations on pages 134 and 141.

AGRAWAL, N.; PRABHAKARAN, V.; WOBBER, T.; DAVIS, J. D.; MANASSE, M.; PANIGRAHY, R. Design tradeoffs for SSD performance. In: **USENIX 2008 Annual Technical Conference**. [S.l.: s.n.], 2008. p. 57–70. Citations on pages 24, 44, 103, 104, and 163.

ANG, C.-H.; TAN, T. C. New linear node splitting algorithm for R-trees. In: **International Symposium on Advances in Spatial Databases**. [S.l.: s.n.], 1997. p. 339–349. Citation on page 49.

BATTY, M. Urban modeling in computer-graphic and geographic information system environments. **Environment and Planning B: Planning and Design**, v. 19, n. 6, p. 663–688, 1992. Citation on page 23.

BECKMANN, N.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. The R*-tree: An efficient and robust access method for points and rectangles. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 1990. p. 322–331. Citations on pages 24, 32, 44, 68, 71, 75, 137, and 163.

BOUGANIM, L.; JÓNSSON, B.; BONNET, P. uFLIP: Understanding flash IO patterns. In: **Fourth Biennial Conference on Innovative Data Systems Research**. [S.l.: s.n.], 2009. Citations on pages 24, 25, 104, and 163.

BRAYNER, A.; FILHO, J. M. M. Hardware-aware database systems: A new era for database technology is coming - vision paper. In: **Brazilian Symposium on Databases**. [S.l.: s.n.], 2016. p. 187–192. Citations on pages 24 and 88.

BYUN, S.; HUR, M. An index management using CHC-cluster for flash memory databases. **Journal of Systems and Software**, v. 82, n. 5, p. 825–835, 2009. Citation on page 134.

CANADA, G. of. **Geospatial Products - Agriculture and Agri-Food Canada**. 2018. Available at <http://www.agr.gc.ca/eng/?id=1343066456961> [Accessed November 2018]. Citation on page 23.

CARNIEL, A. C. **Incorporando Dados Espaciais Vagos em Data Warehouses Geográficos: A Proposta do Tipo Abstrato de Dados VagueGeometry**. Master's Thesis (Master's Thesis) — Universidade Federal de São Carlos, São Carlos, SP, Brasil, 2014. Citation on page 27.

CARNIEL, A. C. Spatial indexing on flash-based solid state drives. In: **Proceedings of the VLDB 2018 PhD Workshop**. [S.l.: s.n.], 2018. p. 1–4. Citations on pages 29, 60, 66, and 139.

CARNIEL, A. C.; CIFERRI, C. D. A. Spatial indexing in flash memories: Proposal of an efficient and robust spatial index with durability. In: **Proceedings of the Brazilian Symposium on Databases - Workshop of Thesis and Master Dissertations in Databases**. [S.l.: s.n.], 2016. p. 66–73. Citations on pages 29 and 139.

CARNIEL, A. C.; CIFERRI, R. R.; CIFERRI, C. D. A. An abstract data type to handle vague spatial objects based on the fuzzy model. In: **Brazilian Symposium on GeoInformatics**. [S.l.: s.n.], 2015. p. 210–221. Citation on page 28.

____. Embedding vague spatial objects into spatial databases using the vaguegeometry abstract data type. In: **Brazilian Symposium on GeoInformatics**. [S.l.: s.n.], 2015. p. 233–244. Citation on page 28.

____. Experimental evaluation of spatial indices with FESTIval. In: **Satellite Events of the Brazilian Symposium on Databases - Demonstration Track**. [S.l.: s.n.], 2016. p. 123–128. Citations on pages 26, 28, 40, 43, 66, 74, 92, and 122.

____. Handling fuzzy points and fuzzy lines using the FuzzyGeometry abstract data type. **Journal of Information and Data Management**, v. 7, n. 1, p. 35–51, 2016. Citation on page 28.

____. The performance relation of spatial indexing on hard disk drives and solid state drives. In: **Brazilian Symposium on GeoInformatics**. [S.l.: s.n.], 2016. p. 263–274. Citations on pages 24, 26, 28, 33, 40, 44, 60, 64, 66, 67, 71, 74, 91, 102, and 122.

____. The VagueGeometry abstract data type. **Journal of Information and Data Management**, v. 7, n. 1, p. 18–34, 2016. Citation on page 28.

____. Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives. **Journal of Information and Data Management**, v. 8, n. 1, p. 34–49, 2017. Citations on pages 24, 26, 28, 44, 60, 64, 66, 67, 102, 122, 140, 142, and 150.

____. A generic and efficient framework for spatial indexing on flash-based solid state drives. In: **European Conference on Advances in Databases and Information Systems**. [S.l.: s.n.], 2017. Citations on pages 27, 29, 32, 33, 40, 44, 49, 62, 64, 66, 99, 101, 103, 136, 141, 142, 145, 150, and 152.

____. Spatial datasets for conducting experimental evaluations of spatial indices. In: **Satellite Events of the Brazilian Symposium on Databases - Dataset Showcase Workshop**. [S.l.: s.n.], 2017. p. 286–295. Citations on pages 25, 28, 31, 50, 58, 62, 122, 143, 145, 157, and 158.

____. A generic and efficient framework for flash-aware spatial indexing. **Information Systems**, 2018. Available: <https://doi.org/10.1016/j.is.2018.09.004>. Citations on pages 27, 29, 44, 49, 62, 64, 66, 101, 150, 152, 154, 156, 158, 159, and 161.

CARNIEL, A. C.; ROUMELIS, G.; CIFERRI, R. R.; VASSILAKOPOULOS, M.; CORRAL, A.; CIFERRI, C. D. A. An efficient flash-aware spatial index for points. In: **Brazilian Symposium on GeoInformatics**. [S.l.: s.n.], 2018. p. 68–79. Citations on pages 27, 29, and 149.

CARNIEL, A. C.; SCHNEIDER, M. A conceptual model of fuzzy topological relationships for fuzzy regions. In: **IEEE International Conference on Fuzzy Systems**. [S.l.: s.n.], 2016. p. 2271–2278. Citation on page 28.

____. Coverage degree-based fuzzy topological relationships for fuzzy regions. In: **International Conference on Flexible Query Answering Systems**. [S.l.: s.n.], 2017. p. 112–123. Citation on page 28.

____. Fuzzy inference on fuzzy spatial objects (fifus) for spatial decision support systems. In: **IEEE International Conference on Fuzzy Systems**. [S.l.: s.n.], 2017. p. 1–6. Citation on page 28.

____. Spatial plateau algebra: An executable type system for fuzzy spatial data types. In: **IEEE International Conference on Fuzzy Systems**. [S.l.: s.n.], 2018. p. 1–8. Citation on page 28.

CARNIEL, A. C.; SCHNEIDER, M.; CIFERRI, R. R. FIFUS: A rule-based fuzzy inference model for fuzzy spatial objects in spatial databases and GIS. In: **ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**. [S.l.: s.n.], 2015. p. 529–532. Citation on page 28.

CARNIEL, A. C.; SCHNEIDER, M.; CIFERRI, R. R.; CIFERRI, C. D. A. Modeling fuzzy topological predicates for fuzzy regions. In: **ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**. [S.l.: s.n.], 2014. p. 529–532. Citation on page 28.

CARNIEL, A. C.; SILVA, T. B.; BONICENHA, K. L. S.; CIFERRI, R. R.; CIFERRI, C. D. A. Analyzing the performance of spatial indices on flash memories using a flash simulator. In: **Brazilian Symposium on Databases**. [S.l.: s.n.], 2017. p. 40–51. Citations on pages 26, 29, 33, 40, 49, 60, 66, 87, and 122.

CARNIEL, A. C.; SILVA, T. B.; CIFERRI, C. D. A. Understanding the applicability of flash simulators on the experimental evaluation of spatial indices. In: **9th Annual Non-Volatile Memories Workshop**. [S.l.: s.n.], 2018. p. 1–2. Citations on pages 26, 29, 60, 66, 137, and 166.

CASTRO, J. P. C.; CARNIEL, A. C.; CIFERRI, C. D. A. A user-centric view of distributed spatial data management systems. In: **Brazilian Symposium on GeoInformatics**. [S.l.: s.n.], 2018. p. 80–91. Citation on page 28.

CASTRO, P. S.; ZHANG, D.; CHEN, C.; LI, S.; PAN, G. From taxi gps traces to social and community dynamics: A survey. **ACM Computing Surveys**, v. 46, n. 2, 2013. Citation on page 23.

CHEN, F.; HOU, B.; LEE, R. Internal parallelism of flash memory-based solid-state drives. **ACM Transactions on Storage**, v. 12, n. 3, p. 13:1–13:39, 2016. Citations on pages 24, 44, 104, 106, and 163.

CHEN, F.; KOUFATY, D. A.; ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: **ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems**. [S.l.: s.n.], 2009. p. 181–192. Citations on pages 24, 25, 44, 55, 68, 72, 77, 102, 104, 106, and 163.

CHUNG, T.-S.; PARK, D.-J.; PARK, S.; LEE, D.-H.; LEE, S.-W.; SONG, H.-J. A survey of flash translation layer. **Journal of Systems Architecture**, v. 55, n. 5, p. 332–343, 2009. Citations on pages 73, 88, 90, and 104.

CORMER, D. Ubiquitous B-tree. **ACM Computing Surveys**, v. 11, n. 2, p. 121–137, 1979. Citation on page 134.

DENNING, P. J. Working sets past and present. **IEEE Transactions on Software Engineering**, SE-6, n. 1, p. 64–84, 1980. Citations on pages 49, 106, and 133.

DONG, X.; XU, C.; XIE, Y.; JOUPPI, N. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. **IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems**, v. 31, n. 7, p. 994–1007, 2012. Citation on page 89.

DUBS, P.; PETROV, I.; GOTTSTEIN, R.; BUCHMANN, A. FBARC: I/O asymmetry-aware buffer replacement strategy. In: **Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures**. [S.l.: s.n.], 2013. p. 58–69. Citation on page 133.

EFFELSBERG, W.; HAERDER, T. Principles of database buffer management. **ACM Transactions on Database Systems**, v. 9, n. 4, p. 560–595, 1984. Citation on page 133.

EMRICH, T.; GRAF, F.; KRIEGEL, H.-P.; SCHUBERT, M.; THOMA, M. On the impact of flash SSDs on spatial indexing. In: **International Workshop on Data Management on New Hardware**. [S.l.: s.n.], 2010. p. 3–8. Citations on pages 24, 32, 44, 69, 70, 91, 102, 140, and 141.

FEVGAS, A.; BOZANIS, P. Grid-file: Towards to a flash efficient multi-dimensional index. In: **International Conference on Database and Expert Systems Applications**. [S.l.: s.n.], 2015. p. 285–294. Citations on pages 69, 70, 102, 135, 136, and 152.

FOLK, M. J.; ZOELLICK, B.; RICCARDI, G. **File Structures: An Object-Oriented Approach with C++**. 3rd. ed. [S.l.]: Addison Wesley, 1997. Citation on page 156.

GAEDE, V.; GÜNTHER, O. Multidimensional access methods. **ACM Computing Surveys**, v. 30, n. 2, p. 170–231, 1998. Citations on pages 23, 24, 32, 34, 44, 53, 68, 70, 71, 72, 75, 88, 92, 99, 102, 118, 122, 140, 145, 149, 150, 152, and 163.

GREENE, D. An implementation and performance analysis of spatial data access methods. In: **International Conference on Data Engineering**. [S.l.: s.n.], 1989. p. 606–615. Citations on pages 32 and 49.

GURRET, C.; MANOLOPOULOS, Y.; PAPADOPOULOS, A.; RIGAUX, P. The BASIS system: A benchmarking approach for spatial index structures. In: **International Workshop on Spatio-Temporal Database Management**. [S.l.: s.n.], 1999. p. 152–170. Citation on page 45.

GÜTING, R. H. An introduction to spatial database systems. **The VLDB Journal**, v. 3, n. 4, p. 357–399, 1994. Citations on pages 32, 68, and 88.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 1984. p. 47–57. Citations on pages 24, 32, 44, 49, 68, 71, 102, 122, 137, and 163.

HARDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM Computing Surveys**, v. 15, n. 4, p. 287–317, 1993. Citations on pages 55 and 137.

JIN, P.; XIE, X.; WANG, N.; YUE, L. Optimizing R-tree for flash memory. **Expert Systems with Applications**, v. 42, n. 10, p. 4676–4686, 2015. Citations on pages 32, 44, 49, 69, 70, 71, 88, 89, 91, 102, 122, 135, 136, 140, 142, and 152.

JIN, P.; YANG, C.; JENSEN, C. S.; YANG, P.; YUE, L. Read/write-optimized tree indexing for solid-state drives. **The VLDB Journal**, v. 25, n. 5, p. 695–717, 2016. Citation on page 134.

JOHNSON, T.; SHASHA, D. 2Q: A low overhead high performance buffer management replacement algorithm. In: **International Conference on Very Large Data Bases**. [S.l.: s.n.], 1994. p. 439–450. Citations on pages 49, 106, and 133.

JUNG, M.; KANDEMIR, M. Revisiting widely held SSD expectations and rethinking system-level implications. In: **ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems**. [S.l.: s.n.], 2013. p. 203–216. Citations on pages 24, 25, 44, 55, 68, 72, 88, 90, 102, 103, 104, 106, 140, 143, 144, 150, and 163.

KAMEL, I.; FALOUTSOS, C. Hilbert R-tree: An improved R-tree using fractals. In: **International Conference on Very Large Data Bases**. [S.l.: s.n.], 1994. p. 500–509. Citations on pages 24, 44, 85, 137, and 163.

KIM, Y.; TAURAS, B.; GUPTA, A.; URGAONKAR, B. FlashSim: A simulator for NAND flash-based solid-state drives. In: **International Conference on Advances in System Simulation**. [S.l.: s.n.], 2009. p. 125–131. Citations on pages 89 and 137.

KOLTSIDAS, I.; VIGLAS, S. D. Data management over flash memory. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2011. p. 1209–1212. Citation on page 24.

_____. Spatial data management over flash memory. In: **International Conference on Advances in Spatial and Temporal Databases**. [S.l.: s.n.], 2011. p. 449–453. Citations on pages 24, 44, and 102.

KORNACKER, M.; SHAH, M.; HELLERSTEIN, J. M. AMDB: An access method debugging tool. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 1998. p. 570–571. Citation on page 45.

KRYDER, M. H.; KIM, C. S. After hard drives–what comes next? **IEEE Transactions on Magnetics**, v. 45, n. 10, p. 3406–3413, 2009. Citations on pages 68 and 72.

KWON, S. J.; RANJITKAR, A.; KO, Y.-B.; CHUNG, T.-S. FTL algorithms for NAND-type flash memories. **Design Automation for Embedded Systems**, v. 15, n. 3-4, p. 191–224, 2011. Citations on pages 46 and 104.

LEE, S.-W.; MOON, B. Design of flash-based DBMS: An in-page logging approach. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2007. p. 55–66. Citation on page 77.

LERSCH, L.; OUKID, I.; SCHRETER, I.; LEHNER, W. Rethinking DRAM caching for LSMs in an NVRAM environment. In: **European Conference on Advances in Databases and Information Systems**. [S.l.: s.n.], 2017. p. 326–340. Citation on page 129.

LI, G.; ZHAO, P.; YUAN, L.; GAO, S. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. **The Journal of Supercomputing**, v. 64, n. 3, p. 1055–1074, 2013. Citations on pages 102, 135, 136, and 152.

LI, Y.; HE, B.; YANG, R. J.; LUO, Q.; YI, K. Tree indexing on solid state drives. **VLDB Endowment**, v. 3, n. 1-2, p. 1195–1206, 2010. Citation on page 134.

LIN, S.; ZEINALIPOUR-YAZTI, D.; KALOGERAKI, V.; GUNOPULOS, D.; NAJJAR, W. A. Efficient indexing data structures for flash-based sensor devices. **ACM Transactions on Storage**, v. 2, n. 4, p. 468–503, 2006. Citations on pages 102, 135, 136, and 152.

LIU, Q.; NIE, H.; BU, K.; LIU, H.; SUN, Z.; LI, M.; XIE, Q. An efficient flash-based remote sense image storage approach for fast access geographic information system. In: **International Conference on Digital Manufacturing Automation**. [S.l.: s.n.], 2012. p. 175–178. Citations on pages 44 and 102.

LUO, L.; WONG, M. D. F.; LEONG, L. Parallel implementation of r-trees on the gpu. In: **Asia and South Pacific Design Automation Conference**. [S.l.: s.n.], 2012. p. 353–358. Citation on page 32.

LV, Y.; CUI, B.; HE, B.; CHEN, X. Operation-aware buffer management in flash-based systems. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2011. p. 13–24. Citation on page 133.

LV, Y.; LI, J.; CUI, B.; CHEN, X. Log-Compact R-tree: An efficient spatial index for SSD. In: **International Conference on Database Systems for Advanced Applications**. [S.l.: s.n.], 2011. p. 202–213. Citations on pages 32, 69, 70, 71, 72, 88, 91, 102, 134, 136, 140, and 142.

MEGIDDO, N.; MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In: **USENIX Conference on File and Storage Technologies**. [S.l.: s.n.], 2003. p. 115–130. Citations on pages 106 and 133.

MITTAL, S.; VETTER, J. S. A survey of software techniques for using non-volatile memories for storage and main memory systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 27, n. 5, p. 1537–1550, 2016. Citations on pages 24, 68, 72, 73, 88, 90, 102, 137, 150, and 166.

MYLLYMAKI, J.; KAUFMAN, J. LOCUS: A testbed for dynamic spatial indexing. **IEEE Data Engineering Bulletin (Special Issue on Indexing of Moving Objects)**, v. 25, p. 48–55, 2002. Citation on page 45.

____. DynaMark: A benchmark for dynamic spatial indexing. In: **International Conference on Mobile Data Management**. [S.l.: s.n.], 2003. p. 92–105. No citation.

NIEVERGELT, J.; HINTERBERGER, H.; SEVCIK, K. C. The grid file: An adaptable, symmetric multikey file structure. **ACM Transactions on Database Systems**, v. 9, n. 1, p. 38–71, 1984. Citation on page 136.

OOSTEROM, P. V. a. N. Spatial Access Methods. In: LONGLEY, P. A.; GOODCHILD, M. F.; MAGUIRE, D. J.; RHIND, D. W. (Ed.). **Geographical Information Systems: Principles, Techniques, Management and Applications**. 2nd edition. ed. [S.l.: s.n.], 2005. p. 385–400. Citations on pages 23, 32, 44, 53, 102, and 163.

OU, Y.; HÄRDER, T.; JIN, P. CFDC: A flash-aware buffer management algorithm for database systems. In: **European Conference on Advances in Databases and Information Systems**. [S.l.: s.n.], 2010. p. 435–449. Citation on page 133.

PARK, S. yeong; JUNG, D.; KANG, J. uk; KIM, J. soo; LEE, J. CFLRU: a replacement algorithm for flash memory. In: **International Conference on Compilers, Architecture and Synthesis for Embedded Systems**. [S.l.: s.n.], 2006. p. 234–241. Citation on page 133.

PAWLIK, M.; MACYNA, W. Implementation of the aggregated R-tree over flash memory. In: **International Conference on Database Systems for Advanced Applications**. [S.l.: s.n.], 2012. p. 65–72. Citations on pages 69, 70, 71, 72, 102, 135, and 136.

RIGAUX, P.; SCHOLL, M.; VOISARD, A. **Spatial databases: with application to GIS**. 1st. ed. [S.l.]: Morgan Kaufmann, 2001. Citations on pages 23, 32, and 163.

ROBINSON, J. T. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: **ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 1981. p. 10–18. Citation on page 136.

ROUMELIS, G.; VASSILAKOPOULOS, M.; CORRAL, A.; MANOLOPOULOS, Y. Efficient query processing on large spatial databases: A performance study. **Journal of Systems and Software**, v. 132, p. 165–185, 2017. Citations on pages 137, 150, and 154.

ROUMELIS, G.; VASSILAKOPOULOS, M.; LOUKOPOULOS, T.; CORRAL, A.; MANOLOPOULOS, Y. The xBR+-tree: an efficient access method for points. In: **International Conference on Database and Expert Systems Applications**. [S.l.: s.n.], 2015. p. 43–58. Citations on pages 24, 150, and 163.

SAMET, H. The quadtree and related hierarchical data structures. **ACM Computing Surveys**, v. 16, n. 2, p. 187–260, 1984. Citations on pages 24 and 163.

SARWAT, M.; MOKBEL, M. F.; ZHOU, X.; NATH, S. Generic and efficient framework for search trees on flash memory storage systems. **GeoInformatica**, v. 17, n. 3, p. 417–448, 2013. Citations on pages 32, 44, 49, 69, 70, 72, 75, 88, 91, 92, 102, 105, 122, 131, 135, 136, 140, 142, 144, 150, 151, 156, 159, and 165.

SCHNEIDER, M.; BEHR, T. Topological relationships between complex spatial objects. **ACM Transactions on Database Systems**, v. 31, n. 1, p. 39–81, 2006. Citation on page 53.

STACKHOUSE, P. W.; BARNETT, A. J.; TISDALE, M.; TISDALE, B.; CHANDLER, W.; HOELL JR., J. M.; WESTBERG, D. J.; QUAM, B. A Beta Version of the GIS-Enabled NASA Surface meteorology and Solar Energy (SSE) Web Site With Expanded Data Accessibility and Analysis Functionality for Renewable Energy and Other Applications. **American Geophysical Union (AGU) Fall Meeting Abstracts**, 2015. Citation on page 23.

STONEBRAKER, M.; RUBENSTEIN, B.; GUTTMAN, A. Application of abstract data types and abstract indices to CAD databases. In: **ACM/IEEE Conf. on Engineering Design Applications**. [S.l.: s.n.], 1983. p. 107–113. Citation on page 51.

SU, X.; JIN, P.; XIANG, X.; CUI, K.; YUE, L. Flash-DBSim: A simulation tool for evaluating flash-based database algorithms. In: **IEEE International Conference on Computer Science and Information Technology**. [S.l.: s.n.], 2009. p. 185–189. Citations on pages 45, 49, 89, and 137.

SUI, D.; LD, M. G. The convergence of gis and social media: challenges for giscience. **International Journal of Geographical Information Science**, v. 25, n. 11, p. 1737–1748, 2011. Citation on page 23.

WU, C.-H.; CHANG, L.-P.; KUO, T.-W. An efficient R-tree implementation over flash-memory storage systems. In: **ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**. [S.l.: s.n.], 2003. p. 17–24. Citations on pages 32, 69, 70, 71, 72, 88, 91, 102, 134, 136, 140, 141, and 151.

WU, C.-H.; KUO, T.-W.; CHANG, L.-P. An efficient B-tree layer implementation for flash-memory storage systems. **ACM Transactions on Embedded Computing Systems**, v. 6, n. 3, 2007. Citation on page 134.

XIE, W.; CHEN, Y.; ROTH, P. C. ASA-FTL: An adaptive separation aware flash translation layer for solid state drives. **Parallel Computing**, v. 61, p. 3–17, 2017. Citation on page 104.

ZHANG, Y.; SWANSON, S. A study of application performance with non-volatile main memory. In: **Symposium on Mass Storage Systems and Technologies**. [S.l.: s.n.], 2015. p. 1–10. Citations on pages 137 and 166.


ICMC
USP
SÃO CARLOS