UNIVERSITY OF THESSALY

DOCTORAL THESIS

Data Structuring Techniques for Non-Volatile Memories

Author:
Athanasios FEVGAS

Supervisor:
Prof. Panayiotis BOZANIS

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Department of Electrical and Computer Engineering

July, 2019

UNIVERSITY OF THESSALY

DOCTORAL THESIS

Data Structuring Techniques for Non-Volatile Memories

Author:
Athanasios FEVGAS

Supervisor:
Prof. Panayiotis BOZANIS

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Department of Electrical and Computer Engineering

Doctoral Committee
Prof. Panayiotis Bozanis
Prof. George Stamoulis
Prof. Spyridon Sioutas
Prof. Michalis Vasilakopoulos
Prof. Christos Makris
Prof. Charalampos Konstantopoulos
Prof. Stamatia Bibi

Declaration of Authorship

I, Athanasios FEVGAS, declare that this thesis titled, "Data Structuring Techniques for Non-Volatile Memories" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“*It does not matter how slowly you go as long as you do not stop.*”

Confucius

UNIVERSITY OF THESSALY

Abstract

School of Engineering
Department of Electrical and Computer Engineering

Doctor of Philosophy

Data Structuring Techniques for Non-Volatile Memories

by Athanasios FEVGAS

Non-Volatile Memories (NVMs) have revolutionize data storage replacing traditional magnetic HDDs in both consumer and enterprise computer systems. Flash based Solid State Drives (SSDs) have become the storage medium of choice for many applications thanks to their high throughput/low latency, shock resistance and low power consumption. Lately, the advent of 3DXPoint with even better properties introduced a new breakthrough for storage systems. Data indexing has been significantly influenced by these advances. Indexes are special purpose data structures, designed to provide fast access to large data collections. They have been extensively studied in DBMSes assuming HDDs as the underlying storage. However, treating SSDs as simply another category of block devices ignoring their inherent idiosyncrasies (e.g. erase-before-write, wear-out, asymmetric read/write speed) and their assets (e.g. their ability to process more than one I/O requests in parallel) may lead to poor performance.

In this dissertation we survey the most important flash-aware indexes [37] and we present new indexing techniques for the NVM based storage. Briefly, in the first part of our research we study flash aware spatial indexes, while in the second we follow a different roadmap exploiting both flash and 3DXPoint technologies. Thus, we introduce GFFM [34] and LB-Grid [35], two variants of Grid File for flash SSDs. GFFM utilizes a buffering strategy that exploits batch writes, while LB-Grid uses logging to reduce small random writes at the buckets' level. We present flash efficient algorithms for range, kNN and group point queries for both LB-Grid and GFFM. We also discuss our contribution in the development of flash efficient bulk-loading, bulk-insertion and querying algorithms for the XBR$^+$-tree [109, 107]. In the sequel, we examine hybrid storage configurations that comprise flash and 3DXPoint SSDs. We propose the H-Grid [33], a spatial index structure that utilizes flash SSDs as mass storage tier and 3DXPoint ones as performance tier. We examine R-tree efficiency on a 3DXPoint SSD as well, and we present the sHR-Tree [36] an initial, yet illuminating effort to develop a hybrid index based on R-tree. Finally, we assay the latest advances in the SSDs' architecture, the programming models and upcoming NVM technologies and we discuss the future trends and new lines of research related to this field.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διδακτορικό Δίπλωμα

Τεχνικές Δόμησης Δεδομένων για μη-Πτητικές Μνήμες

του Αθανάσιου Φεύγα

Οι μη-πτητικές μνήμες έχουν συνεισφέρει σημαντικά στην εξέλιξη των συστημάτων αποθήκευσης. Οι δίσκοι στερεάς κατάστασης τεχνολογίας flash έχουν αντικαταστήσει τους παραδοσιακούς μαγνητικούς δίσκους σε μια μεγάλη γκάμα εφαρμογών, καθώς προσφέρουν υψηλές επιδόσεις, χαμηλή κατανάλωση, οικονομία χώρου και μεγάλη αντοχή σε καταπονήσεις. Μια νέα τεχνολογία μη-πτητικών μνημών με ακόμη καλύτερες ιδιότητες, η 3DXPoint, φιλοδοξεί να αποτελέσει εφαλτήριο για ακόμη μεγαλύτερες επιδόσεις.

Η πρόοδος που έχει συντελεστεί τα τελευταία χρόνια στα συστήματα αποθήκευσης έχει επηρεάσει σημαντικά την ευρετηρίαση δεδομένων. Οι δείκτες είναι ειδικές δομές δεδομένων που αποσκοπούν στο να παρέχουν γρήγορη πρόσβαση σε μεγάλες συλλογές δεδομένων και έχουν μελετηθεί εκτενώς για την περίπτωση που ένας μαγνητικός δίσκος χρησιμοποιείται ως αποθηκευτικό μέσο. Ωστόσο, η απευθείας χρήση δομών που έχουν σχεδιαστεί για τους μαγνητικούς δίσκους σε συστήματα αποθήκευσης που χρησιμοποιούν μη πτητικές μνήμες δεν παρέχει ικανοποιητικά αποτελέσματα. Οι δίσκοι στερεάς κατάστασης συγκεντρώνουν ένα αριθμό από ιδιαιτερότητες, που μπορεί να οδηγήσουν σε χαμηλή απόδοση. Συνοπτικά, οι ταχύτητες ανάγνωσης, εγγραφής, διαγραφής δεδομένων διαφέρουν, η εγγραφή νέων δεδομένων απαιτεί την πρότερη διαγραφή τυχόν υπαρχόντων, ενώ μεγάλος αριθμός εργασιών εγγραφής/διαγραφής προκαλεί φθορά της μνήμης. Από την άλλη, η αποδοτική εκμετάλλευση του εσωτερικού παραλληλισμού που διαθέτουν οι δίσκοι στερεάς κατάστασης είναι άρρηκτα συνδεδεμένη με υψηλές αποδόσεις.

Στα πλαίσια αυτής της διατριβής μελετήσαμε εκτενώς την βιβλιογραφία και παρουσιάσαμε μια κριτική καταγραφή των σημαντικότερων δομών ευρετηρίασης μονοδιάστατων και πολυδιάστατων δεδομένων που έχουν αναπτυχθεί για τους δίσκους στερεάς κατάστασης. Σχεδιάσαμε και αναπτύξαμε τα GFFM και LB-Grid, δύο δομές ευρετηρίασης χωρικών δεδομένων που στηρίζονται στο Grid File. Το GFFM αξιοποιεί μαζικές εγγραφές και αναγνώσεις, ενώ το LB-Grid χρησιμοποιεί μια τεχνική logging για να μειώσει των αριθμό των τυχαίων εγγραφών. Ταυτόχρονα, αναπτύξαμε αλγορίθμους ερωτημάτων που εκμεταλλεύονται τα χαρακτηριστικά των δίσκων στερεάς κατάστασης. Συμμετείχαμε στην σχεδίαση τεχνικών μαζικού χτισίματος, μαζικής εισαγωγής νέων δεδομένων και μαζικής επεξεργασίας ερωτημάτων για το XBR+tree. Μελετήσαμε το πρόβλημα της ευρετηρίασης χωρικών δεδομένων σε υβριδικά συστήματα αποθήκευσης που αποτελούνται από δίσκους στερεάς

κατάστασης τεχνολογίας flash και 3DXPoint. Αναπτύξαμε το H-Grid και το sHR-tree, δύο υβριδικούς δείκτες, που εκμεταλλεύονται τις ιδιότητες των δύο τεχνολογιών μη-πτητικών μνημών. Τέλος, με βάση τις πιο πρόσφατες εξελίξεις στις τεχνολογίες των μη πτητικών μνημών και των δίσκων στερεάς κατάστασης αναδείξαμε νέες ερευνητικές ευκαιρίες στο χώρο της ευρετηρίασης δεδομένων.

Acknowledgements

I would like to express my gratitude to my advisor and my friend Prof. Panayiotis Bozanis for giving me the opportunity to be his student, for his help and his guidance all these years. I also thank him because he never hesitated to give me responsibilities making me a better professional.

I would like to thank the other two members of the advisory committee Prof. George Stamoulis and Prof. Michalis Vassilakopoulos for their valuable advices.

I would like to express my gratitude to Prof. Michalis Vassilakopoulos, Prof. Antonio Corral, Prof. Yannis Manolopoulos, Prof. Panagiota Tsompanopoulou, Prof. Miltiadis Alamaniotis, Dr. George Roumelis, Dr. Leonidas Akritidis and Dr. K Daloukas for giving me the opportunity to work with them.

I also feel obligated to thank the members of the Doctoral Committee by their names, Prof. Panayiotis Bozanis, Prof. George Stamoulis, Prof. Spyridon Sioutas, Prof. Michalis Vasilakopoulos, Prof. Christos Makris, Prof. Charalampos Konstantopoulos and Prof. Stamatia Bibi for reading and commenting this dissertation, as well as, the anonymous reviewers of the research papers that I have published. Their comments and suggestions undoubtedly improved the quality of my work.

I would like to acknowledge all the members of the Department of Electrical and Computer Engineering of University of Thessaly and especially Profs. Eleftherios Tsoukalas and Panayiotis Bozanis, Heads of the Department, for providing me equipment and financial support through the years of my study. Special thanks to administrative staff of the Department and Mrs. Maria Karasimou, director of administration, for their apt response in any problem I faced during my study.

I would like to thank my colleagues in the ECE Department, Apostolos Tsiovoulos and Nikolaos Fraggogiannis who replenished my duties during my sabbatical leave.

I would like to thank the team members of the research projects that I participated all the lasts years from University of Thessaly, Aristotle University of Thessaloniki, Technical University of Crete and University of Patras. It was a great experience to work with and learn from them.

I would like to thank my mother Apostolia and my father Yiannis for their efforts to support my studies.

Last but not least, I would like express my sincere gratitude to my wife Yota and my daughter Olia for their love and understanding. Without their encouragement and support it would not be possible to finish this work.

Contents

List of Figures

List of Tables

Dedicated to the women of my life

1 | Introduction

1.1 Motivation

Since its introduction, flash memory was successfully applied in various cases, ranging from embedded systems of limited resources to large scale data centers. This was a natural consequence of its appealing features: high read/write speeds, low power consumption, small physical size, shock resistance, and absence of any mechanical parts. Thus, NAND flash based solid state drives (SSDs) are gradually replacing their magnetic counterparts in personal as well as in enterprise computing systems.

This evolution also reached DBMSes, creating interesting lines of research in various topics for many researchers. Since the efficient access to the contents of a file is of paramount importance, the study of indexing was not an exception; indexes are special purpose data structures, designed to speed up data retrieval. Indexing was extensively investigated in the context of (magnetic) hard disk drives (HDDs). There exist numerous proposals, the majority of which are based on the popular B-trees [8], R-trees [44], Linear [75] and Extendible Hashing [32]. The direct usage of these data structures on flash based storage devices leads to inadequate performance due to several remarkable characteristics that differentiate it from the magnetic disk. Namely:

1. Asymmetric read/write cost: reads are faster than writes;

2. Erase-before-write: a page write ("program") can be performed only after erasing the block (a set of pages) it belongs. This means that in-place updates, a standard feature of all HDD indexes, triggers a number of reads and writes in flash devices;

3. Limited life span: every flash cell exhibits a high bit error rate (wear-out) after a certain number of program/erase cycles. Consequently, the uneven writing of the pages may render whole areas completely unreliable or unavailable.

To hide or even alleviate some of these peculiarities, SSDs adopt a sophisticated firmware, known as Flash Translation Layer (FTL), which runs on the device controller. FTL employs an out-of-place policy, using logical-to-physical address mapping, and takes care of wear leveling, space reclamation, and bad block management. Thus, SSDs operate as block devices through FTL. However, traditional indexes will still perform poorly if they are directly applied to SSDs. For example, B-trees during updates generate small random writes, which may lead to long write times.

The latter, and many others, are due to the intrinsic SSD internals, which cannot be described adequately by a model, like, e.g. the successful I/O model of HDDs [1] and, therefore, should be considered during the index design; otherwise, the performance will be rather unsatisfactory. For instance, a robust algorithm design should not mix reads and writes or should batch (parallelize) I/O operations. All these are general indications, deducted through extensive experimental analysis of the flash devices, since manufacturers avoid unveiling their design details. During the last years, many researchers have proposed flash efficient access methods. The vast majority of these works concern one- and multi-dimensional data, mainly adapting B-trees and R-trees, respectively. Moreover, there exist a few solutions dealing with the problems of set membership and document indexing.

3DXPoint is new non-volatile memory (NVM) that introduced by Intel and Micron in 2015; and the first storage devices (SSDs) based on it became available to the market in 2017 by Intel, under "Optane" brand name. 3DXPoint supports individual addressing of each memory cell. A key feature of this new NVM is its low latency. 3DXPoint based SSDs can deliver high IOPS even when a small number of concurrent outstanding I/O is used (small queue depth), while their flash counterparts are more efficient under large batched I/O [61, 46].

Motivated by previous research, as well as the latest advances in SSD technology (e.g. NVMe) and in non-volatile memories (e.g. 3DXPoint) we introduce new indexing techniques for spatial data.

The reminder of this Chapter we shortly describe our contributions and we provide the necessary background on NMVs and SSDs.

1.2 Contributions of this dissertation

In this section we summarize the main contributions of this dissertation.

GFFM. The GFFM [34] is the first study on the performance of the two-level Grid File in flash-based storage. GFFM utilizes a buffering strategy that evicts the coldest pages first.

The dirty evicted pages are gathered into a write buffer. The buffer is persisted to the SSD at once, reducing the interleaving between read and write operations, and exploiting the internal parallelism of contemporary SSDs. Increasing the size of the write buffer, and consequently the number of outstanding I/O operations, leads to remarkable performance gains. Regarding spatial query execution we introduce and evaluate flash efficient algorithms for range, kNN and group point queries in [35].

LB-Grid. The LB-Grid [35] is another Grid File variant for flash storage. It utilizes logging to reduce small random writes at the buckets' level; and it uses a memory resident data structure (BTT) to associate data buckets with their corresponding flash pages. LB-Grid alleviates the increased reading cost of logging, with efficient algorithms for single point retrieval, range, kNN, and group point queries. The introduced algorithms exploit the high IOPS, the internal parallelism of SSDs, and the efficiency of the NVMe protocol. We evaluate LB-Grid against GFFM, R*-tree and FAST R-tree. LB-Grid outperforms its competitors in all update intensive workloads, while it presents adequate performance in the read dominated ones. The average cost of a single point search in LB-Grid is $1+c/\varpi$ reads, c the average length of pages' lists in the BTT and ϖ the average gain due to SSD's internal parallelization. Regarding to the average insertion cost, it demands $1+S_r*P_{sm}*\varpi^{-1}$ reads and $S_w*P_{sm}*\varpi^{-1}$ writes, S_r, S_w the average number of page reads and writes, respectively, caused by splits, and P_{sm} the probability of a split operation.

xBR$^+$-tree. The XBR$^+$-tree [110] is a balanced index that belongs to the Quadtree family [38]. We have contributed in the design of flash efficient algorithms for spatial query processing for the XBR$^+$-tree. [107] presents algorithms for bulk-loading and bulk-insertions that outperform the previous HDD based proposals by a significant margin. Similarly, [109] introduces flash efficient batch processing algorithms for point location, window, and distance range queries. The proposed algorithms achieve remarkable performance gains over the original ones, even if the later are assisted by LRU buffering. The greatest improvement is observed in experiments that utilize large datasets.

H-Grid. The emergence of 3DXPoint, a new non-volatile memory, sets new challenges for data indexing. Although 3DXPoint SSDs feature high performance their cost is significantly higher than that of flash-based devices. This fact renders hybrid storage systems a good alternative. Thus, we introduce the H-Grid [33], a variant of Grid-File for hybrid storage. H-Grid uses a flash SSD as main store and a small 3DXPoint device to persist the hottest data. The performance of the proposed index is experimentally evaluated, comparing it against GFFM. The results show that H-Grid is faster than GFFM execution on a flash SSD, reducing the single point search time from 35% up to 43%.

sHR-tree Motivated by the gained results in H-Grid, we investigate R-tree performance in a hybrid storage which is composed by a flash and a 3DXPoint SDD [36]. Furthermore, we introduce the sHR-tree a simple, yet informative approach towards a hybrid R-tree. We experimentally evaluate three different cases: i) R-tree on flash, ii) R-tree on 3DXPoint, and iii) sHR-tree. The experimental results support our design hypothesis. Specifically, the R-tree execution exclusively on a 3DXPoint device improves index construction up to 69%. Similarly, the sHR-tree improves up to 24%. Regarding range queries, a gain up to 82% is achieved when 3DXPoint is the sole storage. However, the gain is marginal for the sHR-tree, since only a small number of nodes is persisted in the fast storage.

1.3 Background

In this section we review non-volatile memory (NVM) technologies, emphasizing on flash memories and SSDs based on NAND flash chips. Understanding the basic functionality, as well as the weaknesses of the medium and how they are tackled, gives the first insights in its appropriate usage for efficient data manipulation.

1.3.1 Non-volatile Memories

Flash Memory. NAND flash has been introduced in 1999 by Toshiba as non-volatile storage technology. Flash cells store information by trapping electrical charges. The first NAND cells could use only a single voltage threshold storing one bit (SLC). Since then, MLC (multi level) and TLC (triple level) cells were developed, which are able to store two and three bits per cell, respectively, exploiting multiple voltage thresholds, 3 for MLC and 7 for TLC [86, 85]. Recently, products that use QLC (quadruple level) NAND flash chips were introduced to the market. QLC cells can store four bits per cell using sixteen different charge levels (15 thresholds). QLC devices are slower and less durable than MLC and TLC. However, they provide higher capacities targeting to read intensive applications. Another method to increase flash storage density is to stack flash cells in layers, one over the other. This type of memory is called 3D NAND or vertical NAND (V-NAND). Today, up to 64-Layer chips are widely available, while some products based on TLC 96-Layer NAND have been recently introduced.

Two or more NAND flash memory chips (dies) can be gathered into the same IC package. Fig. 1.1 depicts a typical NAND flash package containing two dies. Dies within a package operate independently of each other, i.e. they are able to execute different commands (e.g. read, program, erase) at the same time. They are further decomposed into planes which are assembled by several blocks (e.g. 2048 blocks) and registers. The same operation can be performed by multiple planes of the same die concurrently [26, 48]. Blocks are the smallest erasable units in NAND flash, whereas pages are the smallest programmable ones. Each page has a data region and an additional area for error correction data and metadata [85]. The

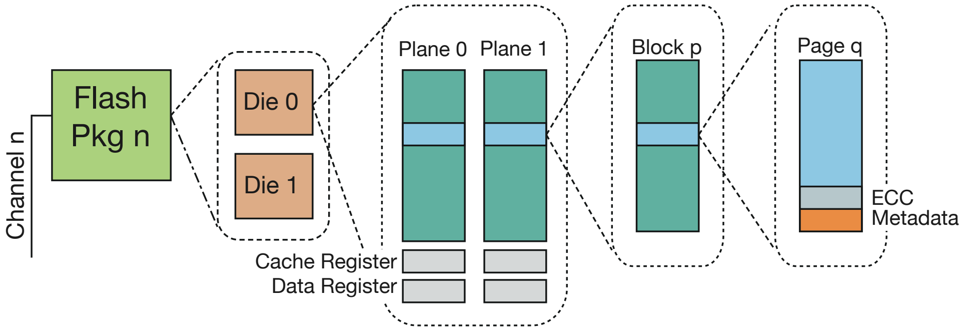


FIGURE 1.1: The anatomy of a flash package

sizes of pages and blocks are usually unknown. However, typical values are 2–16KB and 128–512KB, respectively.

Continuous program/erase (PE) operations cause flash cells to gradually lose their capability to retain electric charge (wearing-out). Thus, every flash cell may undergo a certain number of PE cycles before it starts to present high bit error rate. The lifespan of SLC NAND is around 10^5 PE cycles, for MLC NAND is around 10^4, whereas for TLC is 10^3. QLC is expected to have 10^2, hopefully 10^3 PE cycles.

NAND flash memories are prone to different types of errors, like failure to reset all the cells of an erase block during an erase operation, charge retention errors, and read or write disturbance errors [17, 83, 90, 115]. The most important reasons causing increased error rates are wearing-out, aging, and exposure to high temperatures [83, 115]. Latest storage systems utilized sophisticated error correction codes (ECC), such as BCH and LDPC, to decrease bit errors [85]. However, when a block presents an unrecoverable error [115], or its error rate exceeds the average [17], it is marked as bad and it is not used anymore.

Another type of flash memory is NOR, which was introduced by Intel in 1998. NOR flash supports reads and writes at byte level. However, erases are performed at block level. It enables in-place execution of machine code (XIP), avoiding prior copy of the program code to RAM. NOR is usually used for storing small amounts of hot data or code in embedded systems.

Other Non-volatile Memories. The emergence of new NVM technologies is promising to change memory hierarchy in the future computer systems. 3DXPoint, Phase Change Memory (PCM), Resistive RAM (ReRAM), Magneto-resistive RAM (MRAM) and Spin transfer Torque RAM (STT-RAM) are NVM technologies at different stages of development, which promise very low read and write latencies similar to DRAM's, high density, low power consumption and high endurance [87]. Several studies propose different approaches for integrating the upcoming NVMs into the memory hierarchy [87]. However, their integration into the memory system dictates changes to both hardware and software [62]. For example, data consistency guarantees after a power failure are required in case that NVMs will be adopted as main memory instead of DRAM, or as low-cost extension of DRAM [24, 132].

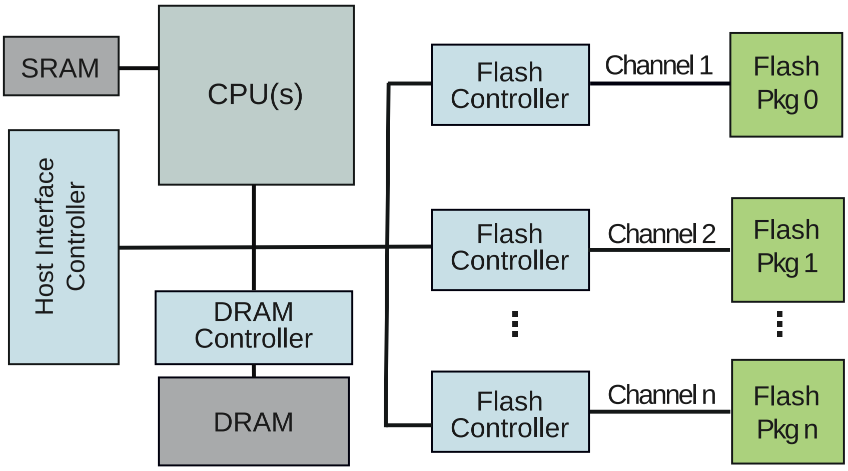


FIGURE 1.2: Overview of the architecture of a solid-state drive

3DXPoint in one of the most promising NVMs today. Intel and Micron announced 3DXPoint in 2015, and Intel started to ship block devices based on 3DXPoint two years later. In 2019 the first products for the memory bus became available. Its architecture is based on stackable arrays of 3DXPoint memory cells. 3DXPoint supports bit addressability and allows in-place writes. It provides better performance and greater endurance than NAND flash (up to 10^3 times), while its density is 10 times higher than that of DRAM [84]. 3DXPoint SSDs provide 10 times lower latency compared to their NAND flash counterparts, while they can achieve high performance at small queue depths [46, 61]. [46] discuses three different models of adopting 3DXPoint within computer systems, particularly as: i) persistent storage, ii) as extension of DRAM, and iii) as persistent main memory.

1.3.2 Solid State Drive Technology

Solid State Drives are met in most consumer computer systems that are sold today, while they are gradually replacing HDDs in big data centers. SSDs employ non-volatile memories (usually NAND flash) to store data, instead of spinning magnetic-plates that HDDs use.

SSD Architecture. Uncovering an SSD device (Fig. 1.2), we can see roughly a number of NAND flash memory chips, controllers for the flash and the interconnection interface, and an embedded CPU.

The most important component of an SSD is certainly the flash memory. Each drive incorporates from a small number up to tens of IC packages, reaching storage capacities of tens of terabytes. Two or more NAND chips (dies) are connected to a flash controller by a communication bus, which is called channel. All commands (I/O or control) are transmitted through the channels, which are 8 or 16 bits wide. Low-end devices incorporate 4 to 10 channels, while enterprise ones use many more. The flash controller pipelines data and commands to NAND chips, translates CPU commands to low level flash commands, and manages error correction. The existence of multiple communication channels and the interleaving of operations within each channel allows the execution of many I/O commands simultaneously, which is known as internal parallelization.

The embedded CPU is an essential part for each SSD, because along with static RAM (SRAM) provides the execution environment for the firmware that controls the operation of the device. A multi-core embedded processor (e.g. 32-bit ARM) provides the required processing capacity. A noteworthy amount of DRAM, varying from several MBs to a few GBs, is also incorporated into the SSDs, storing metadata, such as the physical-to-virtual address mapping table, or temporary user data. In some cases, consecutive update operations to the same flash page are performed into DRAM, preserving medium's endurance and saving bandwidth. The Host Interface Controller interconnects the device with the host system [85]. All incoming or outgoing data are delivered through this controller. The entry level consumer devices use the SATA interface, while the more advanced and the enterprise ones use faster interfaces such as PCIe and SAS.

Storage Protocols. Storage protocols determine the connection and communication of the storage devices with the host system. Both consumer and enterprise SSDs relied for a long time on the SATA protocol, while SAS used to hold also a share of the enterprise market [29]. The advancement of SSDs made SATA and SAS inadequate to support their high efficiency as they have been developed for magnetic disks [119]. Therefore, SSDs for the PCIe bus were introduced.

The first SSD devices for the PCIe bus used to employ proprietary protocols or AHCI. The demand for interoperability among different platforms and manufacturers led to the introduction of the NVMe standard. NVMe is designed to exploit the internal parallelism of modern SSDs, aiming for high delivery capability and low latency. It targets to the high demanding applications with even real-time data processing requirements. To achieve these goals, NVMe supports up to 64K I/O queues with up to 64K commands per queue, message signaled interrupts (MSI-X), and a minimal instruction set of 13 commands [126]. Each application has its own submission and completion queue which both run into the same CPU core. On the opposite, the legacy SATA and SAS protocols support only single queues with up to 32 and 256 commands, respectively. NVMe enables hosts to fully utilize the NVM technologies providing outstanding performance. As a result, NVMe storage devices can get over 1M IOPS, while the SATA ones cannot exceed 200K IOPs. NVMe enhances performance and improves scalability [119]; it also reduces the overhead of system's software up to four times [134]. The NVMe protocol has contributed to the acceptance of SSDs in enterprise storage systems.

Flash Translation Layer. In the previous sections we analyzed the features of NAND memory and presented the architecture of SSD devices, which aspires to obscure the idiosyncrasies of flash, providing a standard interface with the host system. To achieve this, SSDs run sophisticated firmware code which is known as Flash Translation Layer (FTL). The main functions of FTL are address mapping, wear levelling, garbage collection, and bad block management.

FTL implements an out-of-place update policy, since flash cells cannot be updated in-place. It marks old pages as invalid and writes the updated data into clean locations. It hides the complexity of programming the flash cells, maintaining a logical to physical address mapping mechanism. Three different mapping methods have been proposed which are distinguished, based on the mapping granularity, into page-level, block-level and hybrid [65].

Out-of-place updates leave flash pages invalidated; thus FTL initiates a garbage collection procedure to reclaim free space for incoming data [31]. Garbage collection is invoked when the drive is idle or runs out of free pages. It exploits the utilization index $(\frac{\text{invalid pages}}{\text{total pages}})$ to select the proper blocks [65]. Next, it copies any valid pages of the candidate blocks into new positions, and performs the erases.

Continuous erase operations wear flash memory cells. Therefore, FTL provides a wear levelling mechanism to protect them from premature aging. It distributes the erasures uniformly across all medium, utilizing an erasure counter for each erase block.

However, SSDs contain bad blocks, which occurred during operation, or even pre-existed as a result of the manufacturing process. FTL handles defective blocks through address mapping [29, 94]. Thus, when a block is recognized as faulty, its valid data (if any) are copied to another functional block. Then, it is marked as bad, prohibiting its use in the future.

1.3.3 Optimizing I/O for Flash SSDs

Flash based solid state drives can deliver hundreds of thousands IOPs due to their internal parallelism and the high speed interface interconnects. As a result, modern SSDs can execute multiple I/O commands simultaneously. However, the full utilization of SSDs' performance capabilities is tied up with the I/O software subsystem. A single application should be able to submit many I/O requests in parallel in order to fully utilize SSD performance [117]. A well known path to deliver multiple I/O commands to an SSD at once, is by utilizing asynchronous I/O [100, 102, 117, 55, 64, 76]. Using asynchronous I/O, applications may pack several I/O operations together and submit them with one system call. Linux operating system supports asynchronous I/O through kernel AIO. A user application can take advantage of the kernel asynchronous I/O through the *libaio* library which provides helper functions and hooks to system calls. In more detail, the functions *io_prep_pread()* and *io_prep_pwrite()* are used to initialize a read or write operation respectively, whereas *io_submit()* sends all prepared I/O requests to the storage device at once. Completions are picked with *io_getevents()*; by calling this function, an application is notified about the completion of submitted requests or even can block, waiting for the completion. In this paper we exploit kernel AIO to pack many I/O operations together and issue them at once.

POSIX AIO is a user space library that exploits threads to perform multiple I/O requests asynchronously. However, it is less attractive since maintaining multiple threads induces additional overhead.

On Windows operating systems family, asynchronous I/O functionality is supported through the Overlapped I/O API.

2 | Literature Review

2.1 Design Considerations

The I/O Behavior of Flash Devices. Algorithm and data structure developers utilize theoretical models to predict the efficiency of their designs. The external memory model [1], also known as I/O model, is the most widely used for the analysis of algorithms and data structures operating on magnetic disks. The I/O model adopts uniform costs for both reads and writes, and measures the performance based on the number of I/O operations executed. The emergence of flash memory motivated researchers to study the efficiency of external memory algorithms and data structures on this new medium. However, as mentioned, flash memory exhibits intrinsic features that also affect the performance characteristics of flash devices, rendering the HDD models unsuitable for them. Thus, several studies tried to elucidate the factors that influence the efficiency of SSDs, since manufacturers avoid disclosing the details of their designs.

Early proposals of SSD theoretical models are described in [5, 4, 104]. In particular, [4] presents a rather simplistic model which adopts different costs for reads and writes, whereas the approach in [104] requires bit level access to flash pages. [5] presents two distinct models which assume different block sizes for reads and writes. When these models are applied, they may result in limited deductions, since they are confined to counting the numbers of read and write operations, overlooking the ability of SSD devices to serve multiple concurrent I/O [97], as well as the effects of internal processes, like garbage collection and wear levelling, run by proprietary FTL firmware. Thus, they do not provide solid design guidelines.

For these reasons, a number of works tried to uncover the internals of SSDs through careful experimentation. Some of them utilized real devices [26, 28], whereas others employed simulation [3, 15] to investigate methods for better utilization. In all these approaches, SSD internal parallelism was acknowledged as the most important factor for their ability to deliver high IOPS and throughput.

As mentioned in Section 1.3.2, the SSD internal parallelization stems from the presence of multiple flash chips, controllers and communication channels in each device. Specifically, four different types of parallelism are described in the literature: channel, package, die and plane level parallelism. Channel level parallelism is exploited when several I/O requests are issued simultaneously to different channels. The package level parallelism results from attaching more than one packages to the same channel. The I/O commands are interleaved in the channels, allowing packages to operate independently from each other. The die-level parallelism is based on the ability of the dies (within a package) to perform I/O operations independently. Finally, plane level parallelism refers to the execution of the same command on multiple planes of the same die. The efficient utilization of all parallel units is tightly associated with good performance.

Nevertheless, the internal parallelization of SSDs cannot be effectively exploited in all cases [26]. Actually, access conflicts to the same resources between consecutive I/O operations may prevent the full utilization of parallelism [26, 40]. Once such a conflict occurs, then the first I/O command will block all the following, forcing them to wait for the same parallel units. [40] distinguishes three different types of conflicts (read-read, read-write, and write-write), which prevent the fully fledged utilization of SSD parallelism. Moreover, any benefits that may arise from the internal parallelization are highly depended on the firmware code (FTL) as well. FTL maps physical block addresses (PBA) of flash pages to logical ones (LBA) viewed by the operating system, using a mapping table. The fragmentation of this table is correlated with low performance: high volumes of random write operations may fragment the mapping table; in contrast, sequential entries lead to a more organized and thus faster mapping table [58].

Furthermore, the experimental evaluation of SSD devices has provided very interesting conclusions about their efficiency [26, 28]. We may sum up them as follows:

- Parallelizing data access provides significant performance gains; however, these gains are highly dependent on the I/O pattern.

- Parallel random writes may perform even better than reads, when the SSD has no write history. This happens because conflicts are avoided and, thus, the internal parallelization is more efficiently exploited. However, in the long run (steady state) triggered garbage collection may lower performance.

- Small random writes can also benefit from the large capacities of DRAM that equips contemporary SSDs.

- Mixing reads and writes leads to unpredictable performance degradation, which is exaggerated when the reads are random. [40] presented a generic solution for minimizing the interference between reads and writes, using a host-side I/O scheduler that detects conflicts among I/O operations and resolves them by separating operations in different batches.

- Regarding read performance, [26] suggests issuing random reads as quick as possible in order to better utilize parallel resources, since random reads can be as efficient as the sequential ones with the right parallelization of I/O.
- With respect to data management systems, [26] recommends either parallelizing small random reads (e.g., in indexing) or using pre-fetching (that is, sequential reads).

Additionally, based on our experience in the development of flash-aware indexes for flash storages [34, 35, 109, 107], we recommend the utilization of large batches of parallel I/Os to ensure that the internal parallelization is fully exploited. With this type of I/O, adequate workload supply in all parallel units of the device is achieved. The latter is congruent with [26].

Indexing Design Techniques. The above discussion suggests that software programs should carefully consider the way they compose and issue I/O requests to achieve the maximum performance gains. Next, we overview how this is accomplished in flash-aware indexing.

Based on the type of flash memory device they employ, the proposed methods can be categorized into two broad groups: FTL-based indexes exploit the block device character of SSDs, while the raw flash ones are optimized for small, raw flash memories. The indexes of the first group rely on the FTL firmware. Thus, their design must comply with the performance behavior of SSDs, determined by the specific (usually unknown) FTL algorithms they employ; otherwise, the indexes may suffer degraded performance, as indicated in the above discussion. On the contrary, the indexes of the second group handle directly the flash memories. This surely provides more flexibility. However, they have to tackle burdens like wear levelling, garbage collection, page allocation, limited main memory resources etc. In both categories, one can discern the following techniques, used either stand-alone or in combination:

In-memory Buffering. An area of the main memory is reserved for buffering the update operations, using usually the so called index units (IUs), i.e. special log entries that fully describe the operations. When the buffer overflows, some or all of the entries are grouped together, employing various policies, and batch-committed to the original index. Delaying updates in such a way reduces the number of page writes and spares block erases, since it increases the number of updates executed per page write. Buffering introduces main memory overheads and may cause data loses in case of power failures.

Scattered Logging. An in-memory (write) buffer is reserved for storing IUs. In case of overflow, IUs are grouped under some policy and flushed. Since IU entries are associated with certain index constructions, such as tree nodes, an in-memory table performs the mapping during query executions. Scattered logging trades-off reads for writes, trying to exploit the asymmetry between them. It constitutes one of the earliest techniques employed. However, stand-alone can be considered outdated, since saving reads has been proven to be also beneficial for index performance. So, some recent works exploit batch reads to alleviate it. It also reserves an amount of main memory for the mapping table.

Flash Buffering. One or multiple buffer areas in flash are used complementary to an in-memory (write) buffer. During overflows, buffered entries are moved to/between the buffers. This way, any changes to the index are gradually incorporated, incurring thus an increased number of batch reads and writes.

In-memory Batch Read Buffering. It is utilized for buffering read requests. Its purpose is two-fold: on one hand, batch read operations are enabled, on the other, it limits read-write mingling, exploiting the best I/O performance of contemporary SSDs.

Structural Modification. Certain index building elements, like tree nodes, are modified to delay or even eliminate costly structural changes that result in a number of small, random accesses. For example, "fat" leaves or overflow nodes are used in the case of B-tree-like indexes to postpone node splitting; or tree nodes are allowed to underflow, thus node merging or item redistribution is eliminated.

Since parallel I/O deploys most of the flash devices full potential, the usage of both read and write buffers must be considered mandatory.

Next, we present representative one- and multi- dimensional data indexes that exploit the aforementioned techniques.

2.2 One-dimensional Indexes

2.2.1 B-tree Indexes

B-trees are generalization of Binary Search Trees (BST) [8]. With the exception of the root, every node can store at least m and at most M linearly ordered keys and at least $m+1$ and at most M+1 pointers. The root can accommodate at least 1 and at most M linearly ordered keys and at least 2 and at most $M+1$ pointers. Each key is an upper bound for the elements of the subtree pointed by the respective pointer. All leaves are at the same level (have equal depths); this guarantees that the length of all paths from the root to any leaf is logarithmic in the number of nodes. Searches are processed top-down, starting from the root node and following proper pointers, until the key sought is found or declared missing.

TABLE 2.1: Design techniques of flash-aware B-trees.

	Scattered Logging	**Node/Bucket Modification**	**In Memory Buffering**	**Flash Buffering**	**In Memory Batch Read Buffering**
BFTL	*				
IBSF			*		
RBFTL			*		
PIO B-tree		*	*		*
AS B-tree			*		
FD-tree				*	
Bloom tree		*		*	
BF-tree		*			
HybridB tree		*			
LA-tree				*	
μ-tree		*	*		

Adapting B-trees for the Flash Memory. When a B-tree must be stored in a flash device, one has to deal with the peculiarities of the storage medium, i.e., out-of-place updates, asymmetric read/write times and wear-off. For example, naively programming a B+tree on a raw flash device, like the one used in embedded systems, will face the following problem: just one update in the right-most leaf results in updating all tree nodes. This happens since modifying the leaf in question dictates modifying both its father and its sibling to the left (since all leaves are forming a linked list). These nodes, in their turn, will modify their fathers and the sibling leaf node to the left, and so on, all the way up to the root. This effect is time consuming and devastating, in the long run, for the device lifespan.

Things are better in flash devices with an FTL, since the mapping of logical to physical pages confines the described write propagation effect. However, still one has to address the large number of frequent (small) random writes of the original B-tree insertion and deletion algorithms. Small random writes, apart from being slow compared to reads or batch writes, they quickly deplete free space, since they are served by out-of-place writes. Thus, they cause frequent garbage collection (i.e. space reclamation), as well as endurance degradation due to the induced recurring block erases.

In the sequel, we present flash-aware versions of the original B-tree. There are two broad categories of approaches: the first one refers to devices equipped with an FTL, whereas the latter concerns raw flash without such functionality. In both categories, there exist schemes that mainly employ the techniques of buffering, either in main memory or in flash, scattered logging, and original B-tree node structure modification, or a combination of the previous, aiming to delay updates, turning small random writes into sequential or batch ones. Table 2.1 summarizes the techniques employed by each flash-aware B-tree.

FTL-based Indexes

BFTL. Wu et al. [127, 129] were the first researchers that introduced a B-tree variant especially designed for SSDs, named *BFTL*. BFTL employs log-structured nodes and a main memory write buffer, called *reservation buffer*. Each operation is described in the form of IU records, containing all necessary information like the type of operation, the key and the node involved. IUs are accommodated into the reservation buffer. When the reservation buffer overflows, it is reorganized: all IUs belonging to same node are clustered together and gathered into pages using the approximation algorithm First-Fit. Then, these pages are flushed to the SSD device.

The packing procedure may result in writing IUs belonging to different nodes into the same page. Thus, the original B-tree nodes may be scattered into several physical pages. Therefore, they must be reconstructed before accessing them. The necessary mapping between nodes and the corresponding pages where their IU records reside in is stored in an auxiliary data structure, called *node translation table (NTT)*. Each entry of the NTT corresponds to one tree node and is mapped to a list of physical pages where its items are stored. Since the NTT is memory-resident, the scheme depends on main memory, which means potential data lost in case of power failure. Also, the NTT and the tree nodes must be reorganized when page list lengths exceed a pre-defined threshold c.

A search operation costs $h * c$ reads, where h is the B-tree height, and $2(\frac{1}{M-1} + \tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$ amortized writes per insertion/deletion, where $\tilde{N}$ denotes the amortized number of respective structural operations per insertion/deletion. Space complexity is not analyzed, but from the discussion it can deduced that is upper-bounded by $n * c + B$, n the number of nodes and B the size of the reservation buffer. The authors presented experiments with SSD, comparing their approach with B-trees, where it is shown that increased reads are traded-off for small random writes. In a nutshell, nowadays this is completely inadequate with modern SSDs which exhibit far more complex performance characteristics.

IBSF. [66] tries to improve BFTL by employing a better buffer management policy, termed IBSF, while avoiding the distribution of node contents into several pages. Thus, an in-memory buffer for logging operations in the form of IUs is employed, as in BFTL. Additionally, each node occupies one page on SSD and, potentially, a number of IUs into the write buffer. When an IU is inserted into the buffer, it is checked whether it invalidates an IU concerning the same key. This way, only the latest IUs are kept into the buffer, delaying its overflow. When finally the buffer overflows, then the first IU is chosen (FIFO policy), whereas the remaining IUs of the same node are collected. This set of entries is merged with the node contents of the respective SSD page and, after eliminating redundant entries, the new node is flushed to the SSD device. In comparison to BFTL, IBSF succeeds in reducing the number of reads and writes. Specifically, searching is accomplished with h reads, while $\frac{1}{n_{\text{iu}}}(\tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$ amortized writes are necessary per insertion/deletion, where $1 \leq n_{\text{iu}} \leq B$ is the average number of IUs involved in a commit operation. However, since each node is stored in only one page, frequent garbage collection activation may be caused due to occurring page writes.

RBFTL. RBFTL was introduced in [131] as a reliable version of IBSF. Specifically, it uses (alternatively) two small NOR flash devices for backing-up the IUs, before they are inserted into the in-memory buffer. As a last step, it writes the dirty record to NAND flash using FTL. When the buffer overflows, RBFTL commits all its IUs to the SSD. Additionally, it logs into the NOR flash in use the beginning and the ending of the flushing process. Whenever a crash takes place, the IUs or the records are restored into the flash according to the NOR flash logs. When the NOR device in use overflows, the other one replaces it, while the former is erased synchronously. In this way, the slow erase speed of NORs do not degrade the index performance. RBFTL is evaluated only experimentally against standard B-tree and found to have better performance, at the expense of using extra NOR flash memories.

PIO B-tree. PIO B-tree [101, 103] tries to exploit the internal parallelism of the flash devices. Specifically, it uses Psync I/O to deliver a set of I/Os to the flash memory and waits until the requests are completely processed. Psync I/O exploits Kernel AIO to submit and receive multiple I/O requests from a single thread. Based on Psync, a set of search requests can be served simultaneously, using the multi-path search algorithm (MP-search), which descends the tree index nodes like a parallel DFS; at each level, a predefined maximum number of outstanding I/Os are generated to limit main memory needs. Updates are also executed in groups. Firstly, they are buffered in main memory, sorted by key. When it is decided, they are executed in batch, descending and ascending the tree, like MP-search does. Additionally, PIO B-tree leaves are "fat", comprising of multiple pages. Updates are appended to the end of the tree, to save I/Os (only one is needed). When a leaf overflows, operations referred to the same keys are cancelled out and splitting or merging/redistribution is conducted in case of overflow or underflow, respectively. Lastly, the sizes of leaves, nodes, and buffers are determined by a cost model. The authors show that the average cost of search is $h-1+t_L$, t_L the time to read a fat leaf of size L. Experiments conducted on SSDs show the superiority of PIO B-tree over BFTL, FD-tree, and B+tree.

AS B-tree. Always Sequential B-tree (AS B-tree) [99] follows an append-to-the-end technique, combined with buffering: an updated or newly inserted B-tree node is always placed at the end of file. Thus, the nodes are placed in sequential (logical) pages. Particularly, updated/inserted nodes are firstly placed into a write buffer. When the write buffer overflows, all nodes are sequentially appended to the flash resident file. Since the (logical) pages containing the nodes are not overwritten, but each node is written to a new place, the AS B-tree maintains a mapping table, that maps node ids to logical pages. To collect the invalidated pages, the index is written to several subfiles of fixed size. These subfiles are periodically garbage collected, since the active nodes are mainly located in the most recent generated ones. AS B-tree successfully employs sequential writes; however, it suffers from increased time and space overhead to handle the structure allocation. Experiments conducted against

B+tree, BFTL, LA-tree (described below), and FD-tree showed that AS B-tree has favorable performance in search oriented workloads.

FD-tree. FD-tree is a two-part index [71, 72]. The top part consists of a main memory B+tree, termed the *Head tree*. The bottom part comprises $L-1$ sequential, sorted files (sorted runs-levels) of exponentially increasing size, which are stored on flash disk, according to the logarithmic method [10]. That is, their sizes are exponentially increasing from level to level. Between adjacent files, fences (i.e., pointers) are inserted to speedup searching, employing the algorithmic paradigm of fractional cascading [25, 82]. The fences permit the continuation of the search in the next level, without the need of scanning or binary searching it each time from scratch.

The search operation starts at the Head tree and, if necessary, continues in the sorted files, following proper fences. Regarding updates, deletions are handled by inserting appropriate deletion records to the bottom part when they are not (physically) served by the upper B+tree. Insertions are handled by the Head tree. When it overflows, recursive merging of adjacent levels is conducted according to the logarithmic method. These merges are using only sequential reads and writes, and are performed in a multi-pass way. So, random writes are confined to the upper part only; the vast majority of them is transformed into sequential reads and writes, which are issued in batches. The authors provided theoretical analysis, however, they do not discuss performance in the presence of deletions. Specifically, an FD-tree indexing n records supports searches in $O(\log_k n)$ random reads, k the logarithmic size ratio between adjacent levels, and inserts in $O(\frac{k}{f-k}\log_k n)$ sequential reads and writes, f the number of entries in a page; k can be determined according to the percentage of insertions and deletions in the workload. The FD-tree is compared with B+tree, BFTL, and LSM-tree and found to have the best overall performance. Its search times are comparable or worse than those of B+tree, since the fences result in smaller fanouts and thus increased height. The multi-pass merge procedure may incur an increasing number of writes compared to a single-pass solution. Finally, the insertion behavior is similar to that of LSM-tree, since they both limit significantly the need for random writes.

Bloom tree. Bloom tree [54] is a probabilistic index, introduced to optimize both reads and writes. Specifically, it is a B+tree variant, with three types of leaves, namely, regular, overflow and bloom filter. Each (regular) leaf node, when overflows, turns into an overflow leaf, consisting of at most three overflow pages. This way, the scheme defers node splitting. When an overflow leaf exceeds its size limit, it is converted into a Bloom Filter leaf, which is a tree of height one. The root in this tree comprises a number of Bloom Filters (BFs), which guide the search to its more than three overflow child nodes, with the exception of one child which has empty BF and is characterized as *active*; the others are termed as *solid*. A BF is rebuilt only after a predefined number of deletions is performed to its child node.

New insertions to a Bloom Filter leaf are always accommodated into the active leaf. When the active leaf becomes full, it acquires a BF in the root and it is turned into a solid one. If there is no space for new active leaf, then the solid leaf with the least number of entries is turned active and its BF is erased. When no solid leaf can be switched to active, the Bloom Filter leaf overflows; it is turned into a set of d normal nodes, which are inserted into the parent node. Thus, $2*d-4$ writes are saved w.r.t. the standard B+tree. Bloom tree also uses an in-memory write buffer to group updates. A search operation starts from the root, and depending on the type of leaf it ends, it will employ one, at most 3, or at most $p_{fp}*d+2$ extra reads, respectively.

BF-tree. Bloom Filter tree (BF-tree) [6] is an approximate tree index, which aims to minimize space demands, sacrificing search accuracy. Essentially, it is a B+tree with leaf nodes associated with a set of consecutive pages and a key range. This means that the data must be ordered or partitioned on key. Each leaf node comprises a number of Bloom Filters [13] (BFs), each one indexing a page or a page range. Thus, searching for a key in a BF-tree is conducted in two parts. The first part involves the same steps as searching in a B+tree and leads to a leaf. Inside the leaf all BFs are prompted for key membership and thus the result has false positive probability. It may also involve reading several pages. The insertion procedure is very costly, since adding a new key in a BF may violate the desired false positive probability p_{fp}. In this case, the leaf node is split. This means that every key belonging to the leaf key range must be probed for membership to gather the true members of the leaf and build two new leaf structures. On the other hand, bulk-loading the entire index is simple; after scanning the pages, the leaf nodes are formed and then a B+tree is built on top of them. For this reason, BF-tree is mainly used as a bulk-loaded index, tolerating a small percent of updates. The search cost is estimated as h random reads and $p_{fp}*n_{pl}$ sequential reads, n_{pl} the number of pages per leaf node. BF-tree was experimentally investigated against standard B+tree, FD-tree, and in-memory hashing, exhibiting significant space savings.

HybridB tree. HybridB tree [50] is a B+tree which tries to capitalize on the advantages of both HDDs and SSDs in hybrid systems. Specifically, since internal nodes are updated less frequently, each of them is stored in one page of a SSD device. The leaves are huge and constitute a height-one tree of several pages, distributed between SSD and HDD: the leaf-head page is located in SSD, and directs requests to several lower leaf-leaf nodes. A leaf-leaf node is stored either on HDD, when it is not full (its state is characterized as *change*), or on SSD when it is full (and thus it is in *solid* state). Updates and deletes to a solid leaf are accommodated by the associated leaf-log pages, stored on HDD; if necessary, a solid node acquires a leaf-log. An update does not change the state of a leaf-node, while a deletion turns a solid node into an *after-solid* one. After-solid nodes continue to reside in the SSD. When a solid leaf, not having an associated leaf-log, receives an insertion request, it is split. When a leaf-log overflows, or it is full and its solid leaf must accommodate an insertion, a sync operation merges it with its solid-leaf, taking care of various cases of state combinations. A search operation costs at most h SSD reads and one HDD read, an insertion needs additionally at most 3 SSD writes and 2 HDD writes, and a deletion incurs one extra HDD write. The cost of the update operation is higher, demanding at most most 3 SSD writes and 4 HDD writes. HybridB tree was experimentally compared against a B+tree fully located on HDD, and a hybrid B+tree with internal nodes on SSD and leaves on HDD, and has been proven to achieve the best performance.

Raw Flash Indexes

LA-tree. Lazy Adaptive tree (LA-tree) [2] is a B+tree, augmented with flash-resident buffers to hold updates. These buffers are associated with nodes at every k-th level, starting from the root. Update operations are served by appending appropriate log records to the root buffer. When a buffer overflows, or when buffer flushing is beneficial for look-ups, then its contents are sorted and recursively batch-appended to lower level buffers, using the B+tree infrastructure for distributing them at proper descendants. The cost of emptying an non-leaf subtree buffer is linear in the size of the buffer, while a leaf subtree buffer needs $n_{sl}+h+\frac{8B}{M}$ writes (M is the fanout), and linear in the number of subtree nodes n_{sn} and the size of buffer B number of reads. Searches are performed in a top-down fashion. Since the buffers hold items that did not yet find their way to the leaves, they must be checked during descending. LA-tree employs *adaptive buffering*; i.e., it uses an online algorithm, based on sky rental, which decides the optimal size independently for each buffer, relying on the difference between scanning and flushing of past look-ups. Most of the main memory is used for buffers, which, when flushed, are linked together. Since LA-tree is designed for raw flash devices, it implements out-of-place writes with a proper mapping table. Overall, the scheme trades-off reduced update time for increased look-up time. Although it assumes byte addressable raw flash, it has been adapted for SSDs as well.

μ-tree. In [67] μ-tree was introduced as a flash-aware version of B+trees. It is based on the following idea: when an update takes place, all updated nodes along the path are stored in a single page, where the leaf occupies half of the space, whereas the other half is shared among the remaining nodes. As a result, a node has varying fanout which depends on the tree height and its level. Additionally, all nodes at the same level have the same fanout, whereas the root has the same size with its children. The search process basically follows that of B+trees. During an insertion, a new page is allocated to store the updated path. In case the tree height increases by one, node sizes are halved. Deletion is implemented without the sharing/merging policies of the original B+tree. Additionally, μ-tree employs a write buffer, acting according to the allocation order. When the write buffer overflows, all its contents are flushed. Since this index pertains to raw flash, recycling must be explicitly

TABLE 2.2: Design techniques of flash-aware multidimensional indexes and generic frameworks

	Scattered Logging	**Node/Bucket Modification**	**In Memory Buffering**	**Flash Buffering**
F-KDB	*			
MicroGF		*	*	
RFTL	*			
LCR-tree				*
Flash aR-tree	*	*		
FOR-tree		*	*	
FAST			*	
eFind			*	

implemented. Notably, it occupies more space than a B+tree due to the strict page structure. All experimental results against the original B+tree are based on simulation.

2.3 Multidimensional Indexes

Multidimensional indexes refer to data structures designed to enable searches in high-dimensional spaces. Their origins are located in the management of objects with spatial features (spatial data management). Multidimensional access methods are distinguished into two major classes: point access methods (PAMs) manipulate multidimensional data points, whereas spatial access methods (SAMs) deal with more complex geometrical objects, like segments, rectangles etc. [39].

Challenges in Flash-aware Spatial Indexing Design. Consecutive insertions of spatial objects in leaf nodes (or data buckets) force re-construction of the spatial indexes that may be propagated up to the root. This imposes a considerable amount of small random operations that degrade the performance and limit the lifespan of SSDs. Logging and write buffering are widely used methods to counter the small random I/O burden, especially in the early works. Our work is targeting to the spatial queries' efficiency also, exploiting in-memory read buffers and batch read operations. Table 2.2 sums up the design techniques for developing flash efficient spatial data access methods. It also includes two generic framework that are described in the next section (§2.4). Next, we present several works concerning flash-aware spatial indexes that fall in the aforementioned classes.

2.3.1 Point Access Methods

PAMs have been designed to enable spatial searches for multidimensional points. K-D-B-tree, Grid File and XBR$^+$-tree are the secondary storage PAMs that attracted the interest of researchers to study their efficiency in flash SSDs. K-D-B-tree [98] is an extended B-tree to support high dimensional data, which retains its balance while adapting well to the shape of data. Grid File [91] is a multidimensional hashing structure that partitions space using an orthogonal grid. Finally, the XBR$^+$-tree [110] is a balanced index belonging to the Quadtree family [38].

FTL-based Indexes

F-KDB. F-KDB [69] is a flash aware variant of K-D-B-tree that aims to avoid random writes applying a scattered logging method. It represents K-D-B-tree as a set of log-records that are held in an in-memory buffer. Two different data structures are used to represent points and regions, respectively. Each flash page may contain records that belong to different nodes. Therefore, an NTT is used to associate each tree node with the flash pages that store its entries. An on-line algorithm decides when a node will be merged into a new flash page to improve read performance. The cost of a search operation is $h * c$, h the height of the tree and c the maximum NTT page list length. Similarly, the cost of an insertion is $\tilde{N}_{\text{split}} + 2/n_{log}$, $\tilde{N}_{\text{split}}$ the number of node splits and n_{log} the number of log records in a flash page. F-KDB, contrary to almost all other studies [19, 53, 113, 128], does not exploit any batch write technique to persist the contents of the in-memory buffer; it simply flushes a single page each time. So, this approach does not exploit the high throughput of modern SSDs efficiently and imposes interleaving of read and write operations that, as we earlier discussed, degrades performance. F-KDB outperforms K-D-B-tree in all test cases. However, the evaluation was performed with datasets of small size only.

Raw-Flash Indexes

MicroGF. MicroGF [73] is the multidimensional generalization of MicroHash, bearing a resemblance with the Grid File. Namely, in two dimensions, the directory represents a grid of n^2 square cells, each associated with the address of the latest index page belonging to this region. All index pages, related to a cell, are forming a chain. Inside the index page, each record (and thus the cell) is divided into four equal quadrants. Each quadrant maintains up to K records. During an insertion, if the write buffer overflows, then the relevant index records are created and grouped into index pages, which are associated with the pertinent grid cell. In case a quadrant overflows, then the records are offloaded to a neighboring empty quadrant, termed borrowing. When a borrowing quadrant overflows or there is no such quadrant, then the original quadrant is further divided into four sub-quadrants, and the new index record is inserted into the index page, if there is enough space. Otherwise, it is inserted into a new index page, which is linked to the respective cell as the newest one.

MicroGF can serve range queries. Specifically, after locating the pertinent grid cell, the respective index pages are scanned and, for each index record, the overlapping quadrants are checked, along with their borrowing quadrant and their sub-quadrants. The scheme is experimentally evaluated against the original one-level Grid File and Quadtree for 2-dimensional point sets under the specifications of the sensor employed. The experiments were conducted through a simulation environment for wireless sensor applications. The authors consider the one-level Grid File [91] in their analysis and experimentation. However, a two-level approach [47] would be more appropriate.

2.3.2 Spatial Access methods

R-tree [44] and its variants are the most popular SAMs, utilized in a wide range of data management applications [81]. R-trees are general-purpose height-balanced trees similar to B+trees. Namely, R-tree leaves store minimum bounding rectangles (MBRs) of geometric objects (one MBR for each object), along with a pointer to the address where the object actually resides. Each internal node entry is a pair (pointer to a subtree T, MBR of T). The MBR of a tree T is defined as the MBR that encloses all the MBRs stored in it. Similarly to B-trees, each R-tree node accommodates at least m and at most M entries, where $m \leq M/2$. Searching starts from the root and moves towards the leaves, and may follow several paths. Thus, the cost of retrieving even a few objects may be linear in size of data in the worst case.

FTL-based Indexes

RFTL. RFTL [128] is a flash efficient R-tree implementation, equivalent to BFTL [127, 129]. All node updates are kept into an in-memory (reservation) buffer in the form of IUs. Once the reservation buffer gets full, the IUs are packed into groups, using a First Fit policy: the IUs of a certain node are stored to the same flash page. Please note that each node can occupy only one physical page; however, one page may contain IUs of several nodes. A NTT is used to associate each tree node with its corresponding pages into the flash storage. This approach aims to reduce slow random write operations with a penalty of extra reads. To keep balance between the two, a compaction process is introduced. The threshold of 4 pages that is used in the presented experiments may not be efficient for the present day devices, which are characterized by high bandwidth and IOPS. The effects of spatial locality of inserted objects to the efficiency of the compaction process has been also studied. The cost of search operation is $h*c$ reads, h the height of the tree and c the size of biggest list in the NTT. An insertion needs $\frac{2}{M-1}+\tilde{N}_{\text{split}}$ writes, in the amortized case, $\tilde{N}_{\text{split}}$ the amortized number of splits per insertion. Like in BFTL, we deduce that the space complexity is bounded by $n*c+B$, n the number of nodes and B the size of the reservation buffer. Through experimentation, it is found that the spatial locality of inserted objects influences the compaction efficiency. The experimental results show that RFTL reduces the number of page writes, and the execution time as well, compared to the original R-tree. RFTL targets to alleviate the wide gap between read and writes speeds of the first SSDs, neglecting the search performance.

LCR-tree. LCR-tree [80] aims to optimize both reads and writes. It uses logging to convert random writes into sequential ones. However, it retains the original R-tree structure in the disk as is, while it exploits a supplementary log section, which stores all the deltas. Whenever the log area overflows, a merge process is initiated, merging the R-tree with the deltas. Contrary to RFTL, LCR-tree compacts all log records for a particular node into a single page in the flash memory. This way, it guarantees only one additional page read to retrieve a tree node, with the penalty of re-writing the log page (in a new position) in each node update. The LCR-tree maintains an in-memory buffer for the log records, and an index table associating each tree node with its respective log page. The insert operation updates the log records of the affected existing nodes, and stores the new added nodes as deltas in the log section. LCR-tree presents better performance than the R-tree and RFTL in mixed search/insert workloads. An additional advantage of this proposal is that it can be used as is with any R-tree variant. LCR-tree does not exploit any particular policy for flushing updates to the SSD, neither exploits any of SSDs' high performance assets during read operations.

Flash-aware aR-tree. [96] presents a flash-aware variant of Aggregated R-tree (aR-tree) that is based on RFTL. The proposed index, similar to RFTL, employs the concepts of IUs, reservation buffer and node translation table. However, the aggregated values are stored separately from the R-tree nodes, since they are updated more frequently. The aggregated values for a certain node may span to several physical pages. To this end, an index table is maintained to facilitate the matching of R-tree nodes with their respective aggregated values. Therefore, the retrieval of a node along with its aggregated values requires scanning of the reservation and the index tables and fetching all the corresponding pages. The authors provide the cost for reading and updating the aggregated values. Particularly, a search operation costs $2*h*(c+1)$ reads, h the height of the tree and c the number of flash pages accommodating aggregated values of a particular node. Similarly, the amortized number of the necessary writes per update is $\frac{2*h}{r}$, with r denoting the number of records that fit in a flash page. The evaluation of the proposed index is performed against RFTL-aRtree, an aR-tree implementation which is directly derived by RFTL (the aggregated values are stored inside the IUs). The presented experimental results show that the proposed aR-tree variant reduces writes and achieves better execution times.

FOR-tree. FOR-tree (OR-tree) [53, 124] proposes an unbalanced structure for the R-tree, aiming to reduce the costly small random writes which are dictated by node splits. To achieve this, they attach one or more overflow nodes to the R-tree leaves. An *Overflow Node Table* keeps the association between the primary nodes and their overflow counterparts. An access counter for each leaf is also stored in it. When the number of overflow nodes increases, and thus the number of page reads conducted during searching also rises, a merging back operation takes place. The merging process for a specific leaf node is controlled by a formula that takes into account the number of accesses to the node and the costs of reading and writing a flash page. A buffering mechanism suitable for the unbalanced structure of FOR-tree is also proposed aiming to further reduce random writes. Thus, an in-memory write buffer holds all the updates to nodes (primary and overflow) using a hash table. The nodes are clustered according to increasing IDs, composing flushing units. The coldest flushing units (i.e. no recent updates) with the most in-memory updates are persisted first. The evaluation includes experiments using both simulation (Flash-DBSim) and real SSDs, and shows that FOR-tree achieves better performance against R-tree and FAST R-tree (presented bellow).

Discussion

The first flash efficient multidimensional indexes aimed to reduce the increased cost of random writes by introducing extra reads. For this reason, early works like RFTL, Flash aR-Tree and F-KDB utilized various scattered logging methods to improve the performance of the R-Tree, the Aggregated R-Tree and the KDB-Tree, respectively. In different direction, FOR-Tree uses overflow nodes to reduce the small random writes imposed by the re-balancing operations. In almost all cases we discussed, index updates are persisted in batches, turning small random writes onto sequential ones. At the same time the mingling of reads and write is avoided. As the gap between read and write speeds is reduced, researchers target to the reads' efficiency exploiting appropriate buffering policies. FOR-Tree is a representative example.

2.4 Generic Frameworks

FAST and eFind, discussed in the sequel, can be categorized as generic frameworks for database indexes. They aim to provide all required functionality for turning any existing index into a flash efficient one. They are both designed for SSDs equipped with a FTL and apply the techniques of buffering and logging.

FAST. The FAST generic framework [113, 114] has been successfully tested with both one- (B-tree) and multi-dimensional (R-tree) indexes. Although it exploits logging, it logs the result of an operation rather than the operation itself. This enables FAST to exploit the original search and update algorithms of the underlying indexes. The updates are performed in-memory and maintained with the help of a hash table, termed *tree modifications table* (TMT). Moreover, the updates are audited into a sequential written log held in flash, to facilitate recovery after a system crash. The update operations in the TMT are grouped in flush units and flushed periodically. Two different flushing policies are demonstrated; the first promotes flushing of the nodes with the larger number of updates, whereas the other gives an advantage to the least recently updated nodes that contain the most updates. The authors evaluated FAST R-tree against RFTL, studying several performance parameters, such as memory size, log size and number of updates, and was found to trigger less erase operations in all examined cases.

In [22, 21] Linear R-Tree, Quadratic R-Tree, R*-Tree, and their FAST versions were studied, on both HDDs and SDDs. Specifically, the authors compared the performance of index construction and range queries, changing parameters like page size, buffer size, flushing unit size, etc. The experiments were conducted on a local machine, as well as on Azure[1] cloud infrastructure. They found that i) FAST versions exhibit faster construction times, ii) query performance depends on the selectivity and the device employed, iii) page sizes should be decided based on query selectivity, iv) indexes have different behavior on different devices, and v) on some cases, query performance on HDDs is better than on SSDs, by increasing the page size.

eFind. Another generic framework similar to FAST is eFind [20, 19]. The authors were motivated by five design goals; four of them are based to well-known characteristics of flash-based storages, while the fifth is more generic. Specifically, they suggest one to i) avoid random writes, ii) favor sequential writes, iii) reduce the random reads with in-memory buffers, iv) prevent interleaving of reads and writes, and v) protect the data from system crashes.

The eFind framework is composed of three components that control buffering, page flushing and logging. The proposed buffering policy uses separate buffers for reading and writing. The write buffer stores node modifications to the underlying tree index, while the read buffer accommodates frequently accessed nodes, giving higher priority to nodes of the highest levels. The buffering algorithms are developed around two hashing tables, one for each buffer. Any particular record in the hash tables represents an index page. The reconstruction of an existing item may involve gathering data from the two buffers and the SSD.

The contents of the write buffer are persisted in batches of nodes (flushing units), sorted by index page IDs. The authors evaluate five different policies, considering the recency of the modifications, the node height (upper level nodes have higher priority), and the geometric characteristics of the applied operations. A log file is held on the SSD to provide index recovery after a system crash. The cost of the search operation is depended on the respective cost of the underlying index, since eFind does not modify it. eFind is evaluated against FAST, achieving better performance in index construction. Regarding the flushing operation, eFind works better with large page sizes. An initial effort to integrate XBR$^+$-tree into the eFind generic framework is discussed in [23]. So, eFind XBR$^+$-tree outperforms an implementation that employs the FAST generic framework. However, it is not evaluated aganist the original XBR$^+$-tree.

[1] https://azure.microsoft.com/en-us/

3 | Flash Efficient Point Access Methods

3.1 Introduction

A significant number of already presented research studies, targeting to flash efficient data access methods. Regarding multi-dimensional data, most researchers are aiming to the efficiency of R-Tree. Motivated by the performance of flash SSDs and the challenges imposed by their intrinsic characteristics, we study for flash efficient Point Access Methods. Particularly, we introduce GFFM and LB-Grid, two variants of Grid File for flash storage. Surpassing previous works in spatial data indexing, that focus on simple operations (e.g. insertions and searches), we propose new flash efficient algorithms for range, kNN and group point queries. Finally, we present our contribution to the development of flash efficient methods for bulk-loading, bulk-insertions and query algorithms for the XBR^{+}-tree.

3.2 Overview of Grid File

The Grid File [91] is a spatial data structure for indexing multidimensional data in disk drives. It is categorized to point access methods, but it can be used for other spatial objects as well. In the later case, each geometric object is represented by points of a higher dimensional space; e.g., rectangles are encoded as 4-dimensional points. Grid File partitions a k-dimensional space S using an orthogonal grid (Fig. 3.1). It consists of i) a dynamic k-dimensional array, which is known as the *Grid array*, and ii) k one-dimensional arrays, called *Linear scales*. The *Grid array* stores the addresses of data which are accommodated in secondary storage pages, termed as *buckets*. The *Linear scales* define the partitioning of S by storing the proper

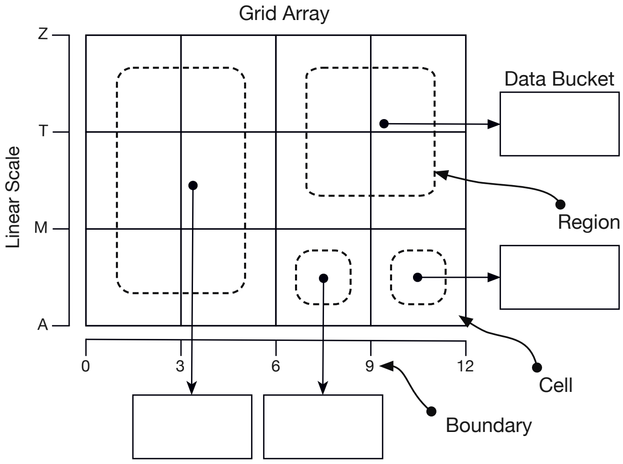


FIGURE 3.1: An instance of Grid File depicting the Grid array, Linear scales, buckets, and regions

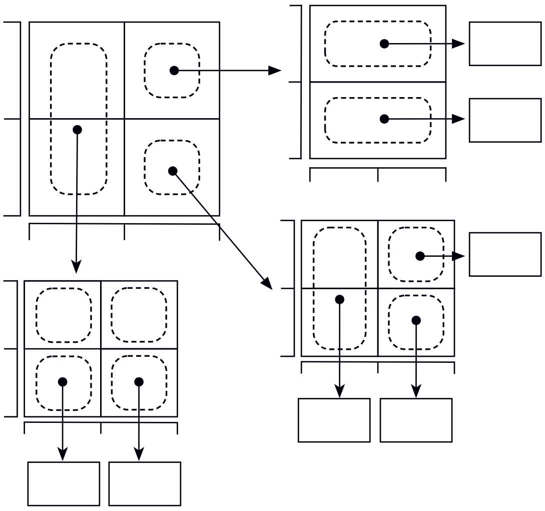


FIGURE 3.2: Two-level Grid File: Root directory/scales and sub- directories/scales

boundaries. Each boundary determines a $(k-1)$-dimensional hyperplane that partitions S. The *Linear scales* are always kept in main memory, whereas the *Grid array* is usually retained in secondary storage. To avoid low bucket occupancy, several cells may point to the same bucket, thus forming d-dimensional pairwise disjoint boxes, called *regions*. Each region $R \subset S$ corresponds to a single page (bucket) in the physical storage. The Grid File guarantees single point retrieval with at most two disk accesses, one to retrieve the appropriate part of the *Grid array* and one to acquire the proper data bucket. According to its inventors, the Grid File exhibits some advantageous features. Namely, i) it adapts to the shape of data guaranteeing uniform access even for non-uniform data, ii) it is symmetric which means that all keys can be exploited in queries with the same efficiency, and iii) it is dynamic, i.e. grows and shrinks by the size of data [91].

However, Grid File exhibits some disadvantages that concern space utilization, overhead of reorganizing Grid upon introduction of new hyperplanes and poor performance when the

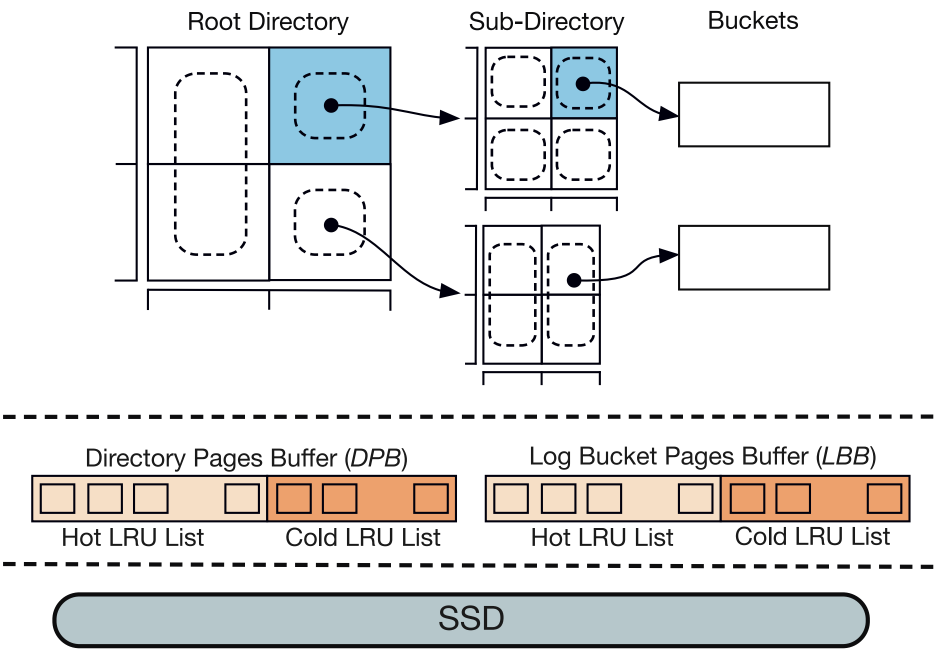


FIGURE 3.3: The structure of GFFM buffer

search attributes are related [93]. Continuous splitting may lead to super-linear growth of the directory, thus several variants of the Grid File have been introduced aiming to improve its efficiency. A two level implementation of the Grid File (Fig. 3.2) is proposed in [47]. This variant takes advantage of a scaled down *Root directory* which provides a coarse representation of the Grid and is always memory resident. The Root directory cells contain pointers to directory pages. A directory page stores a disk-persistent part of the Grid File, which has its own grid array (*sub-directory*) and scales (*sub-scales*). Sub-directories, in turn, point to data buckets. Thus, the two-disk access principle for accessing a single point is not violated. Query operations apply as in the original Grid File, however splits and merges are now performed locally in a more efficient way.

3.3 The GFFM

The GFFM is our first effort for a flash-aware Grid File. It is based on the two-level Grid File [47]. It was initially built around a buffering strategy that evicts the coldest pages first [34]. The dirty evicted pages are gathered into a write buffer. The buffer is persisted to the SSD at once, reducing the interleaving between read and write operations, and exploiting the internal parallelism of contemporary SSDs. Increasing the size of write buffer, and consequently the number of outstanding I/O operations we achieved remarkable performance gains.

In the current version of GFFM [35], we use separate buffers for directory and bucket pages (Fig 3.3). Each buffer is a pair of Least Recently Used (LRU) lists. The Hot LRU lists (*HLRU*) hold the recently accessed pages, while the Cold LRU lists (*CLRU*) accumulate the dirty pages that are evicted from *HLRUs*. When a *CLRU* overflows, all pages in it are

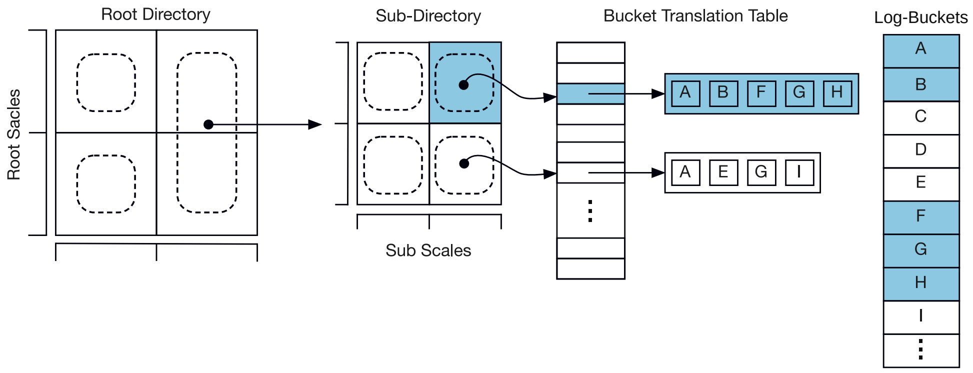


FIGURE 3.4: The logical structure of LB-Grid

persisted to the SSD as a batch. After that, all persisted pages are set clean; and are purged from main memory whenever is necessary. We exploit the same buffering strategy in the LB-Grid as well. We further discuss the processes of fetching and evicting directory pages and data buckets from the main memory in Section 3.4.3 (Alg. 4 and 5).

3.3.1 Query Processing in GFFM

We developed flash aware algorithms for range, kNN and group point queries for the GFFM. These algorithms seek to exploit the advantages of modest SSD devices (i.e. throughput, internal parallelism, NVMe protocol). To that end, we partition the query space into segments searching multiple sub-directories or data buckets each time. The size of the search space, each time, depends on the available buffer space. The query algorithms for the GFFM resemble the respective ones of LB-Grid. The main difference between LB-Grid's and GFFM's versions is in the management of data buckets. Thus, we omit their detailed description here, and we suggest the reader to refer at Section 3.4.4.

3.4 The LB-Grid

3.4.1 Overview of LB-Grid

With the GFFM, we studied the performance of Grid File in flash SSDs and we introduced a buffering policy that evicts the coldest pages as batches. The acquired results motivated us to propose a new implementation that further exploits the superior performance of the contemporary NVMe SSDs. Our objective is twofold: i) to improve the performance of update operations by reducing the number of small random writes, and ii) to enhance search efficiency, alleviating the increased reading of logging. Thus, we introduce the *Log Bucket Grid File (LB-Grid)*, a variant of the Grid File for the flash memory storage. LB-Grid utilizes batch read and write operations, taking advantage of the high internal parallelism of SSDs as well as the great efficiency of the NVMe protocol.

The logical structure of LB-Grid is depicted in Figure 3.4. We follow a two-level design, utilizing both a Root Directory and sub-directories. The Root Directory is memory resident, whereas the sub-directories are stored in the secondary store, one per page, exactly as suggested in [47]. In opposition to the Grid File, we follow a logging-based approach for the data buckets, which store the actual data. Each log-bucket may contain records that belong to different logical buckets. Therefore, we utilize a *Bucket Translation Table (BTT)* to associate each logical bucket with the log-buckets that accommodate its log records. Regarding the sub-directories we do not log them, instead we handle them as is described in the original Grid File. We make this choice, because directory pages are a small fraction of the index and, therefore, can be efficiently buffered in the main memory. Moreover, in this way, we reduce the complexity of the data structure and thus we can easily identify and study its performance parameters. The Root Directory and the sub-directories are maintained (grow or shrink) in the same way as in the original Grid File. However, the sub-directory cells do not point to SSD pages, but to the *BTT* (please refer to Fig. 3.4). BTT is memory resident and implemented as a hash table to enable fast lookups.

When an insert, delete, or update operation is issued to LB-Grid, a log record is composed that represents the requested operation. This record includes the coordinates of the points $x, y, \ldots$, an operation identifier op_id, bucket's identifier *bucket_id* and a $timestamp$. The op_id determines the requested operation (i.e. insert/delete/update) on the data. The *bucket_id* is an identifier for the corresponding logical bucket. Each log-bucket occupies only one page in the SSD. When a log-bucket is created, it is stored in a buffer pool in the main memory. Whenever this buffer gets full, its contents are saved to the SSD, in an append-only fashion, according to a flushing policy. Motivated by recent works [135, 20] that prioritize buffering of internal nodes, we maintain separate buffers for directory and log-bucket pages, respectively. Buffering is further discussed in Section 3.4.3.

In the following subsections we present the basic operations in LB-Grid and we examine several query algorithms. We consider a 2-dimensional grid for simplicity reasons, however all algorithms can also be applied to higher dimensional grids.

3.4.2 Basic operations

This section presents some basic operations of LB-Grid. We focus on the parts of algorithms that differentiate from the ones of the original Grid File.

Read bucket

ReadBucket (Algorithm 1) describes the procedure of retrieving one or more log-buckets in order to construct the logical representation of a certain data bucket B. The algorithm returns all valid log-records belonging to B. First, a list of log-bucket identifiers is retrieved from BTT (line 1). A fetch operation follows, that gathers all required log-buckets to the main memory (line 2): it receives a set of log-bucket identifiers and examines if any of

```
Algorithm 1: readBucket($B$, $BTT$)
   Data: the bucket $B$ to be read, the bucket translation table $BTT$
   Result: a log-bucket $logBucket$ with the records of $B$
1  $logbucketlist \leftarrow BTT[B].list$;
2  $LogBuckets = batchReadLogBuckets(logbucketlists)$;
3  foreach $logbucket$ in $LogBuckets$ do
4      foreach $record$ in $logbucket.records$ do
5          if $record.Bucketid == B$ then
6              $recordsofB \leftarrow recordsofB + record$;
7          end
8      end
9  end
10 parse $recordsofB$ to remove redundant records;
11 eliminate deleted records;
12 compose a new log-bucket $logBucket$ which corresponds to $B$;
13 return $logBucket$
```

the requested log-buckets already exist in the main memory buffers; for the rest of them, it composes a batch read and submits it to the SSD. The retrieved log-buckets are examined one-by-one, picking up the records of B (lines 4-8). In the sequel, redundant or deleted records are removed and a new log-bucket containing only the valid records of B is composed.

The two-disk-access principle for single point retrieval of Grid File is also preserved in LB-Grid. The first SSD access concerns the corresponding sub-directory page, whereas the second one is a *Group Read Operation* for the necessary log-buckets. Therefore, the average cost of searching a single point in LB-Grid is actually determined by the average cost of retrieving a logical bucket. So, if L is the average list length of a BTT list, ϖ the average gain from SSDs' internal parallelism, and R_p the cost for retrieving a single page, then the cost for searching a point is $C_s = R_p * (1 + L/\varpi)$.

Insert and Delete Operations

Insert Algorithm 12 describes the Insert operation. Its input comprises a point p and the logical bucket B wherein p has to be inserted. B has been previously located by searching the Root Directory and the corresponding Sub-directory. Each logical bucket B accommodates a certain number of records, so that all records of B can fit in one log-bucket. If B is not full, a log-record is composed and appended to the current working log-bucket, then BTT is updated accordingly (lines 2-7). Otherwise, a split operation of B is initiated, resulting in the introduction of a new logical bucket B'. Therefore, all records of B are read from secondary storage into a new log-bucket and a split boundary BR is established (lines 9-11). Subsequently, the records in the new log-bucket are redistributed according to BR by updating their *bucket_id* field. Finally, BTT is updated as needed. Successive insertions of new records may trigger sub-directory splitting. This operation is executed as in the original Grid File.

Algorithm 2: Insert($B, p, BTT, wLogBucket$)

```
Data: the bucket $B$ to be updated, the new entry $p$ to be inserted, the bucket
      translation table $BTT$, the working log bucket $wLogBucket$
$op \leftarrow insert$;
if $B$ is not full then
    insert record $(p, B, op, timestamp)$ to $wLogBucket$;
    if $wLogBucket.id \notin BTT[B].list$ then
        append $wLogBucket.id$ to $BTT[B].list$;
    end
    exit
else
    $newLogBucket \leftarrow readBucket(B)$; //read all records of $B$
    insert record $(p, B, op, timestamp)$ to $newLogBucket$;
    find a split boundary $BR$ ;
    let $B'$ a new logical bucket;
    insert $B'$ to the grid;
    update grid if necessary;
    redistribute records in $newLogBucket$ among $B$ and $B'$ according to $BR$;
    update $BTT$;
    exit
end
```

Delete The removal of an existing point in LB-Grid is performed by inserting a new log-record that represents the operation. Algorithm 3 describes deletion. Given the corresponding logical bucket B, a *delete* record is inserted (line 2) into the current working log-bucket. The necessary updates of BTT are following (lines 3-5). A merge operation is conducted, if necessary. Particularly, the occupancy of the logical bucket is examined. If it is dropped bellow a predefined level (e.g. 30%) then a merge operation is initiated. At first, an attempt to detect a proper bucket for merging takes place (line 9); the prerequisites for merging are described in [47]. If such bucket (*mB*) is found, the logical buckets *mB* and *B* are restored (lines 13, 14) and merging is performed by copying all log records of *mB* to *B*. The log-bucket of *mB* is deleted and necessary updates to BTT are subsequently performed. The logical bucket *mB* is removed from the Grid Directory, and, finally, if the boundary along *mB* and *B* is no longer used, it is expunged from the grid and the corresponding scale. The merging procedure may be propagated up to the root directory.

Update Cost Insert or delete operations are performed in batches as soon as the write buffer gets full. Let W_b be the size of the write buffer in pages, B_s the page size in items, S_r, S_w the average page reads and writes during splits or merges, respectively, W_p cost of writing a page, and P_{sm} the probability an insert or delete operation to trigger a split or merge operation. Then, the amortized average cost per update operation is $C_u = [W_b * B_s * (1 * R_p + (W_p * S_w + R_p * S_r) * P_{sm} * \varpi^{-1})]/W_b * B_s = 1 * R_p + (W_p * S_w + R_p * S_r) * P_{sm} * \varpi^{-1}$.

```
Algorithm 3: Delete($B, p, BTT, wLogBucket$)
  Data: the bucket $B$ to be updated, the entry $p$ to be deleted, the bucket translation
        table $BTT$, the working log bucket $wLogBucket$
  $op \leftarrow delete$;
  insert record $(p, B, op, timestamp)$ to $wLogBucket$;
  if $wLogBucket.id \notin BTT[B].list$ then
      $BTT[B].list \leftarrow BTT[B].list + wLogBucket.id$;
  end
  if $B.size > BUCKET\_OCCUPANCY$ then
      exit
  else
      $mB \leftarrow findMergingBucket(B)$;
      if $mB$ is $NULL$ then
          exit;
      end
      $BLogBucket \leftarrow readBucket(B)$;
      $mLogBucket \leftarrow readBucket(mB)$;
      merge $BLogBucket$ and $mLogBucket$;
      delete $mLogBucket$;
      update $BTT$;
      delete logical bucket $mB$ from sub-directory;
      if the merging boundary is in $X_i$ hyperplane and it is not anymore needed then
          remove redundant boundary from grid sub-directory and $X_i$ sub-scale;
      end
      exit
  end
```

3.4.3 Buffering and replacement policy

Traditionally, databases use in-memory buffers that reduce access to secondary storage, aspiring to improve their performance. The efficiency and the distinct features of flash SSDs have motivated researchers to study caching algorithms for flash based storage devices. A common strategy in the vast majority of the presented works is batching writes. Thus they reduce the interference between read and write operations and exploit SSDs' internal parallelism. The first effort for a flash efficient buffer management algorithm is CFLRU (Cold First LRU), which promotes the eviction of clean pages first [95]. AD-LRU (Adaptive double LRU) [52] improves CFLRU by considering the frequency of page references as well. It divides pages into two classes, cold and hot, preferring the cold pages for eviction. A buffering algorithm designed for database indexes is presented in [135]. Specifically, applies different priorities to the buffered pages based on their corresponding node type (leaf or internal node). In the same direction, [130] takes into account concurrent access to the buffer; and proposes a policy that provides greater priority to higher level nodes, as well as, to the leaf nodes whose parents already exist in the buffer. The generic framework presented in [20] also integrates a buffering mechanism that prioritizes caching of internal tree nodes.

In our previous work [34], we presented a flash efficient buffering scheme for the Grid

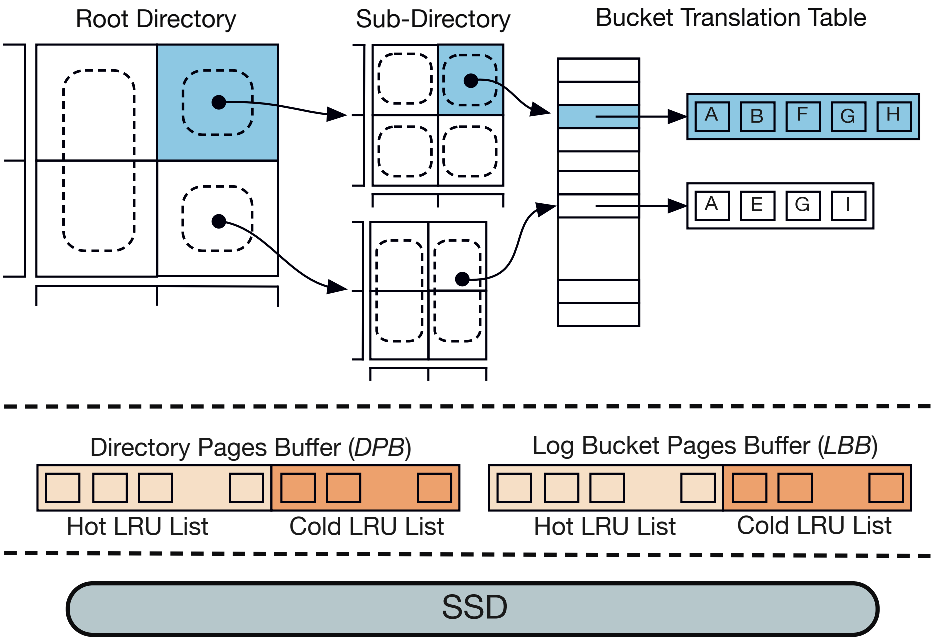


FIGURE 3.5: The LB-Grid buffer structure

File. Here, we employ separate buffers for directory pages and log-buckets, enabling even different policies for each page type. Figure 3.5 depicts the proposed buffer structure. Each buffer utilizes a pair of Least Recently Used (LRU) lists to manage cached pages. In both cases, the Hot LRU list (*HLRU*) holds recently accessed pages and the Cold LRU list (*CLRU*) accumulates dirty pages which are evicted from *HLRU*. When *CLRU* overflows, all pages in it are saved to the underling SSD at once.

Algorithm 4: FetchPage(p)

```
Data: the id $p$ of the requested page
Result: a reference to the requested page $p$
if $p \in HLRU$ then
    move $p$ to MRU position of $HLRU$;
    return a reference to the requested page $p$;
end
if $p \in CLRU$ then
    if $HLRU$ is full then
        Evict();
    end
    move $p$ to the MRU position of $HLRU$;
    return a reference to the requested page $p$;
end
if $HLRU$ is full then
    Evict();
end
fetch $p$ from SSD;
set $p$ to the MRU position of $HLRU$;
return a reference to the requested page $p$;
```

Algorithm 5: Evict()

```
select victim $v$ from the LRU position of $HLRU$;
if $v$ is dirty then
    if $CLRU$ is full then
        if page $cp$ in the LRU possition of $CLRU$ is not clean then
            flush $CLRU$ to SSD;
            set to clean all pages in $CLRU$;
        end
    end
    evict $cp$;
    evict $v$ from $HLRU$ to $MRU$ position of $CLRU$;
else
    evict $v$;
end
```

The processes of fetching and evicting log-bucket pages from main memory are detailed in Algorithms 4 and 5, respectively. In this work we use the same eviction policy for both directory and log-bucket pages, thus Algorithms 4 and 5 fit in both cases.

Page read requests are served through $HLRU$ (Alg. 4). If the requested page p is already in main memory ($HLRU$ or $CLRU$), then it is moved to the most recently used (MRU) position of $HLRU$. Otherwise, a read operation to SSD is issued. If p resides in $CLRU$, then $CLRU$ and $HLRU$ may require some adjustment after moving p to $HLRU$. This is performed by utilizing the eviction process. Eviction is also initiated to make free space in $HLRU$ prior a page read from SSD. During eviction, $CLRU$ is flushed to the SSD, exploiting a group write operation (Alg. 5). After a group write, all pages in $CLRU$ are set to "clean" and are evicted from it, one by one, whenever it is necessary. In Section 3.5 we discuss the performance parameters of the proposed buffering scheme.

3.4.4 Queries

This section details query processing in LB-Grid. Specifically, we describe range, kNN and group point queries. We try to highlight those parts of the algorithms that differentiate from the original Grid File's ones.

Range Queries

A range query returns all the points that are enclosed into a user defined region R_q. The proposed range query algorithm exploits the high efficiency of SSDs by composing groups of read requests that are issued at once. Figure 3.6 presents an execution example of a range query. Initially, all Root Directory cells (i.e. Sub-directories) intersecting R_q are located (Fig. 3.6a). Each retrieved Sub-directory is examined against the corresponding part of R_q to locate eligible buckets (Fig. 3.6b). The BTT is searched next, to get their log-buckets

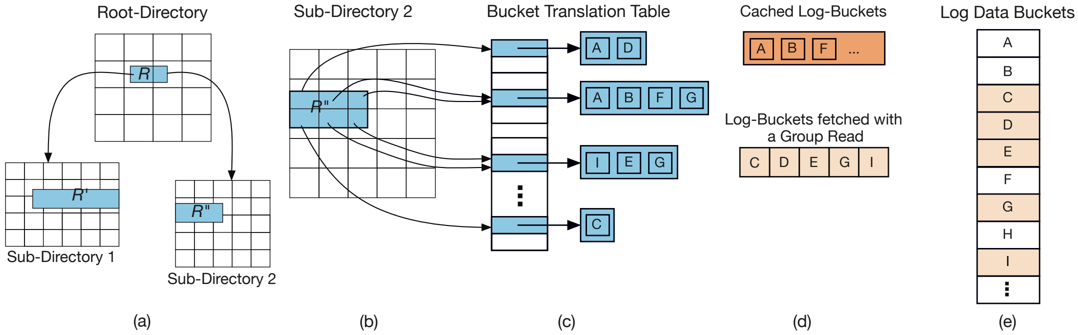


FIGURE 3.6: Execution example of a range query in LB-Grid

ids (Fig. 3.6c) and, after that, a group read operation is issued fetching the corresponding log-buckets from the SSD (Fig. 3.6d).

Algorithm 6 describes range query processing for a given sub-directory. During the first steps of the algorithm, the buckets that intersect with R_q are discovered (lines 1-3). These buckets are separated into groups that can fit into the log-buckets' buffer (LBB), utilizing next-fit policy. As soon as such a group has been formed, a batch read fetches it to the main memory (line 6). The gathered log-buckets are processed, picking up only the records that belong to the result (lines 7-14). This continues until all buckets have been fetched and processed.

This algorithm applies in a similar way to the Root Directory for retrieving the Sub-directories that overlap with R_q. The corresponding directory pages are fetched, using a group read operation for Sub-directories. This operation segments the set of directory pages into groups that fit into the Sub-directories' memory buffer (DPB). The fetched Sub-directories are processed and the algorithm continues by fetching and processing, until all groups of directory pages have been examined.

The cost C_{R_q} of retrieving the points in a region R_q is the sum of the costs for fetching the Sub-directories ($C^{\sigma}_{R_q}$) and the corresponding log-buckets ($C^{\beta}_{R_q}$). Let n_σ be the number of Sub-directories intersecting with R_q, S_i the i-th such Sub-directory, b_i the number of logical buckets of S_i that intersect R_q, and B_r the average utilization of the read buffer. Then $C_{R_q} = C^{\sigma}_{R_q} + C^{\beta}_{R_q} = R_p * (n_\sigma/(B_r * \varpi) + \sum_{i=1}^{n_\sigma}(b_i * L/(B_r * \varpi))) = R_p * (n_\sigma + \sum_{i=1}^{n_\sigma}(b_i * L))/(B_r * \varpi)$.

kNN Queries

Given a point p, a k-nearest neighbor query (kNN), retrieves the k nearest points to p. In the sequel, a new algorithm that exploits the advances in flash SSDs is described. To facilitate understanding, we describe how the algorithm is applied to a Sub-directory. However, the same procedure is used at the Root Directory level, as well, for locating the nearest Sub-directories and processing the corresponding directory pages.

The proposed algorithm traverses the grid one level after the other, as it is suggested in [88]. Figure 3.7 illustrates an execution example. The cell, wherein the point p resides, is

Algorithm 6: RangeQuery(R_q, BTT, LBB)

```
Data: the query region $R_q$, the bucket translation table $BTT$, the log-bucket buffer $LBB$
Result: a set of points $result$ enclosed by $R_q$
let $LogBucketGroup$ denote the set of log-buckets to be read as batch;
search the scales to convert the coordinates of $R_q$ into interval indexes;
use interval indexes to determine query region $R_q$ in the sub-directory;
walk through $R_q$ to obtain pertinent set $Buckets$;
foreach $B \in Buckets$ do
    if $LogBucketGroup + BTT[B].list > LBB$ then
        $batchReadLogBuckets(LogBucketGroup)$;
        foreach $B' \in Bsubset$ do
            $readBucket(B')$; // re-built bucket $B$ from prefetched log-buckets
            foreach $point \in B'$ do
                if $point \in R_q$ then
                    $result \leftarrow point$;
                end
            end
        end
        $LogBucketGroup \leftarrow \emptyset$;
        $Bsubset \leftarrow \emptyset$;
    end
    $LogBucketGroup \leftarrow LogBucketGroup + BTT[B].getlist$;
    $Bsubset \leftarrow Bsubset + B$;
end
fetch and process the residual part of $Bucket$ if necessary;
return
```

firstly located (the central cell in the figure), and searching continues with the cells in the perimeter. At each step, we examine another level, which is denoted by different colors in Fig. 3.7, towards the outer of the grid. The algorithm stops when k nearest neighbors of p have been discovered and no other can be found in a shorter distance or all cells have been traversed. The main idea behind our approach is to exploit the low latency and the high I/O throughput of contemporary SSDs, pre-fetching big portions of log-buckets from each level by employing group reads. However, this policy introduces a compromise between the number of data buckets we retrieve and the number that is really needed.

Before we proceed with the description of the algorithm, we have to introduce two new data structures. The first represents a neighboring point *npoint*, thus it stores the coordinates of the point and its distance from p. The discovered *npoints* are pushed into a priority queue (*kNNqueue*) which holds all neighbors of p in the grid, with the farthest one being at the top of the queue. We also use another data structure named *cell* to represent a cell c of the grid. This data structure encapsulates the id of the respective bucket (*bid*), and the minimum distance d of c from the point p; d designates the minimum possible distance of a point residing in bucket *bid* from p.

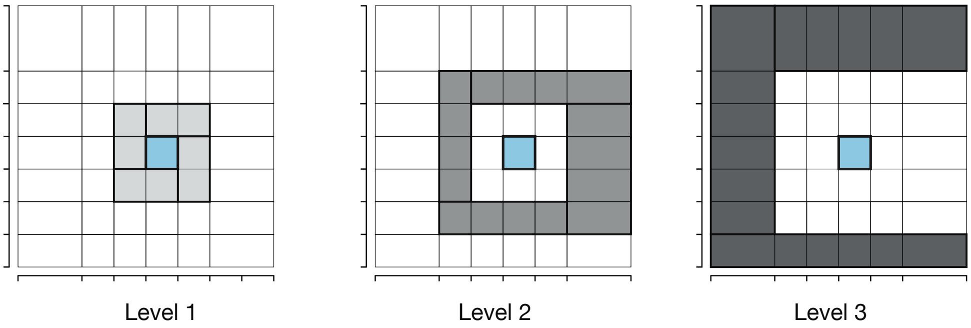


FIGURE 3.7: kNN query processing

Algorithm 7 describes the execution flow of a kNN query. Initially, the algorithm locates the cell c wherein the point p is lying in (lines 1-2). Afterwards, query processing continues traversing the grid, level by level. All cells of each level are examined (lines 4-13). A first check discards the cells corresponding to buckets that have already been seen in an inner level (line 6). A *cell* is eligible for processing if k neighbors have not been located yet, or *cell* is closer to p than the farthest neighbor until now (line 7). Hence, if one of these conditions stands, *cell* is pushed into two priority queues, $cellQueue$ and $cellQueue'$ (line 8-9). The closest cell to p is located at the top of the queues. $cellQueue$ is used for locating the nearest points, while $cellQueue'$ for composing the groups of log-buckets for batch retrieval. We use two priority queues because iterating over a priority queue actually empties the queue. In our implementation, the priority queues hold pointers to cell objects rather than the objects themselves.

The direction D controls the side of the perimeter (up, down, left or right) that is currently processed and the $step$ the distance (level) from c. All buckets of the current level are split into groups that fit into the memory buffer (LBB) and they are processed one group after the other (lines 15-38). The buckets corresponding to the closest cells are fetched first. Grouping and fetching is performed like in range queries, aiming to exploit the performance advantages of SSDs. In the sequel, the closer cells are dequeued from $cellQueue$ and examined one after the other (lines 19-37). The *visited list* is updated with the bucket which is under consideration (lines 21-22). The proposed algorithm, in opposition to the in-memory kNN query algorithms for moving objects, does not assume a uniform grid wherein each cell represents a different area. As such, many cells may point to the same bucket, thus the visited list is needed.

Initially, $kNNqueue$ is filled with the first k points that are shown up (lines 28-29). Subsequently, any following point q in sorter distance (from p) than the one at the top of the $kNNqueue$, is pushed to the result. If $kNNqueue$ has already k elements and one more is pushed into it, then the one at the top, which is the most distant, is popped (lines 30-33). Processing at a specific level ends if $kNNqueue$ already holds k neighbors and no

Algorithm 7: kNNQuery($p, k, kNNqueue$)

```
Data: a point $p$, the number $k$ of nearest neighbours, the queue $kNNqueue$ of
      discovered neighbors
Result: the $k$ nearest neighbours of $p$ in the subdirectory
search the scales to convert the coordinates of $p$ into interval indexes;
use interval indexes to locate the corresponding cell $c$ in the grid;
do
    foreach $direction\ D + step$ do
        foreach $cell \in D$ do
            if $cell \notin visitedList$ then
                if $(kNNqueue.size <= k)\ OR\ (mindist(p, cell) < kNNqueue.top.dist)$
                 then
                    $cellQueue \leftarrow cellQueue + < bid, mindist(p,c) >$;
                    $cellQueue' \leftarrow cellQueue' + < bid, mindist(p,c) >$;
                end
            end
        end
    end
    $++step$;
    do
        // prefetch as many log-buckets fit into the log-buckets buffer
        $prefetch\_logbuckets(cellQueue')$;
        $current\_dist \leftarrow cellQueue.top.dist$;
        while $cellQueue\ is\ not\ empty$ do
            $cCell \leftarrow cellQueue.top$;
            if $cCell \notin visitedList$ then
                $visitedList \leftarrow visitedList + cell$;
                if $kNNqueue.size > k\ AND\ cCell.dist > kNNqueue.top.dist$ then
                    $stop \leftarrow true$; $break$;
                end
                $cBucket \leftarrow readBucket(cCell.bid)$;
                foreach $point\ p \in cBucket$ do
                    if $kNNqueue.size <= k$ then
                        $kNNqueue \leftarrow kNNqueue + < p, d >$;
                    else if
                      $kNNqueue.size > k\ AND\ mindist(p, point) < kNNqueue.top.dist$
                      then
                        $pop\ kNNqueue$;
                        $kNNqueue \leftarrow kNNqueue + < p, d >$;
                    end
                end
            end
            $cellQueue.pop$; //remove cCell from priority queue
        end
    while $((cellQueue'\ is\ not\ empty)\ AND\ (stop\ is\ not\ true))$;
while $grid\ has\ not\ been\ explored\ AND$
  $((kNNqueue.size < k)\ OR\ (current\_dist < kNNqueue.top.dist))$;
return
```

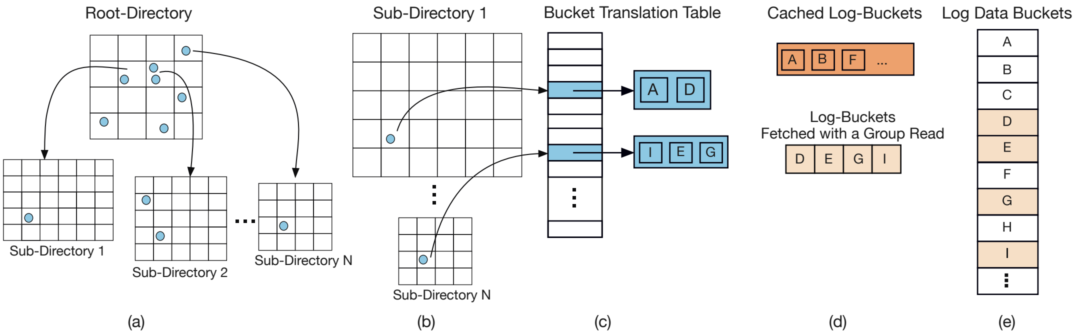


FIGURE 3.8: Execution example of a group point query in LB-Grid

other can be found in closer distance, or there are not any points to process at the specific level. The algorithm terminates when all points indexed by the grid have been examined or k nearest neighbors have been discovered and there is no other point q such that $mindist(p,q) < mindist(p,kNNqueue.top)$ (line 39).

The cost C_{kNN} of performing a kNN query is determined by the cost C^{σ}_{kNN} of reading the required Sub-directories and the cost C^{β}_{kNN} of fetching log-buckets. In the proposed kNN query algorithm, processing is performed level by level. For a certain level, prefetching takes place in several steps. The number of these steps is determined by the size of the read buffer. Let n_{ρ} the number of levels examined at the Root Grid, n^{σ} the number of Sub-directories examined in each level, and m_i the number of levels of the i-th Sub-directory. Then $C_{kNN} = C^{\sigma}_{kNN} + C^{\beta}_{kNN} = R_p * (\sum_{i=1}^{n_{\rho}} (n^{\sigma}_i - v_i)/(B_r * \varpi) + \sum_{i=1}^{n^{\sigma}} \sum_{j=1}^{m_i} (b_{ij} - v_{ij}) * L/(B_r * \varpi)) = R_p * (\sum_{i=1}^{n_{\rho}} (n^{\sigma}_i - v_i) + \sum_{i=1}^{n^{\sigma}} \sum_{j=1}^{m_i} (b_{ij} - v_{ij})) * L/(B_r * \varpi)$, where n^{σ}_i denotes the number of Sub-directories at level i, b_{ij} is the number of logical buckets at the j-th level of the i-th Sub-directory, and $v_i, v_{i,j}$ the number of buckets that have already been processed in a previous level (and contained in the visited list), during exploring the i-th Sub-directory and the buckets at j-th level of the i-th Sub-directory, respectively.

Group Point Queries

Search requests for single points do not exploit SSD's benefits since they involve only few page reads each time. Specifically, two page reads are needed in Grid File, one for the directory page and one for the data bucket. Similarly, in LB-Grid one page read is required to fetch the directory page and one group read for the log-buckets. It is clear that continuous single point requests underutilize the SSD and, thus, degrade performance. However, it is not unusual for search requests to arrive as bursts, hence more than one searches can be combined and processed together, increasing the utilization of contemporary flash based storages. Therefore, we introduce *Group Point Queries* for LB-Grid and Grid File, based on the assumption that a set S of search requests show up at the same time.

Figure 3.8 demonstrates a group point query in LB-Grid. Given a set of points S, the Root Grid is scanned, gathering all the references to directory pages (Fig. 3.8a). The participating

Algorithm 8: GroupQueryR(S)

```
Data: the set $S$ of query points
Result: the answer set $result$
foreach point $p \in S$ do
    foreach coordinate of $p$ do
        search the corresponding sub-scale to convert value into interval index;
    end
    use interval indexes to obtain the corresponding sub-directory $SD$;
    if $SD \notin subdir\_table$ then
        $subdir\_table \leftarrow subdir\_table + SD$;
    end
    $subdir\_table[SD].list \leftarrow subdir\_table[SD].list + p$;
end
do
    // prefetch as many sub-directories in $subdir\_table$ fit into sub-directories buffer
    $prf\_subdir \leftarrow prefetch\_subdirectories(subdir\_table)$ ;
    foreach $SubD \in prf\_subdir$ do
        $Subdir \leftarrow getSubdirectory(SubD)$;
        // bucket_table holds for each sub-directory its correspoding
        // log-buckets that participate to the search space
        $bucket\_table \leftarrow$
          $bucket\_table + Subdir.preprocessQuery(subdir\_table[SubD].list)$;
    end
    do
        // prefetch as many log-buckets fit into memory buffer
        // return the sub-directories whose corresponding
        // log-buckets have been prefetched
        $p\_subdirs \leftarrow prefetch\_logbuckets(bucket\_table)$;
        foreach $vSubD \in p\_subdirs$ do
            $Subdir \leftarrow getSubdirectory(vSubD)$;
            $result \leftarrow result + Subdir.GroupQueryS(subdir\_table[vSubD].list)$;
        end
    while no buckets are left in bucket_table for processing;
while no sub-directories are left in subdir_table for processing;
return
```

Sub-directories are fetched and processed in batches, like in range queries, in order to locate those logical buckets that contain points in S (Fig. 3.8b,c). Finally, all the necessary log-buckets are retrieved from the SSD (Fig. 3.8d,e) and scanned to detect the searched points, exploiting a group read operation as well.

Group point queries differ from range and kNN queries, since each one of the points in S may belong to a different Sub-directory. Therefore, they can not fully exploit spatial locality, as the two other query types do. For this reason, prefetching of log-buckets does not concern a Sub-directory only, but involves many of them.

Algorithm 8 details the read operation at Root Directory. Initially, the Root Directory is searched, gathering all Sub-directories that index the points of S (lines 1-5). After that, S is partitioned into subsets that correspond to each one of the discovered Sub-directories. These Sub-directories are retrieved in batches that fit into the in-memory buffer (line 13). At the next stage, all buckets that possibly contain points of S are located (lines 14-19), enabling group reads for the corresponding log-buckets (lines 20-29). Log-bucket reads also take place in several steps. As many log-buckets as possible to fit in the log-bucket buffer (LBB) are prefetched. A sequence of search operations is issued each time that processes the prefetched portions of the index (lines 25-28).

The cost of group point queries resembles the cost of the range queries. The difference between them lies in the policy of batching the read requests: the range query algorithm fetches and processes the log-buckets of each Sub-directory separately (that is, locally), exploiting the spatial locality of the request. This policy simplifies the implementation, but may induce under-utilization of the reading capacity. Thus, the cost of retrieving log-buckets is $C^{\beta}_{R_q} = R_p * \sum_{i=1}^{n_\sigma}(b_i * L/(B_r * \varpi))$, where n_σ is the number of Sub-directories that intersect the search region R_q. On the other hand, group point queries may not present spatial locality at all. For this reason, we compose batches of log-bucket requests that correspond to different Sub-directories. Let n^S_σ denote the number of Sub-directories that contain the points of the search space S. In this case, the respective cost is $C^{\beta}_{gpq} = R_p * \sum_{i=1}^{l}(b_i * L/(B_r * \varpi))$, where $l \simeq n^S_\sigma/B_r$. Thus, the total cost of group point queries is $C_{gpq} = C^{\sigma}_{gpq} + C^{\beta}_{gpq} = R_p * (n^S_\sigma/(B_r * \varpi) + \sum_{i=1}^{l}(b_i * L/(B_r * \varpi))) = R_p * (n^S_\sigma + \sum_{i=1}^{l}(b_i * L))/(B_r * \varpi)$.

3.5 Performance evaluation

3.5.1 Methodology and setup

In this section we evaluate the performance of the introduced flash efficient algorithms for LB-Grid and GFFM. We also study the performance of the R*-tree and FAST [113] in SSDs. Regarding the R*-tree, our objective is to provide the reader with the feeling of performance difference between flash efficient algorithms and a widely used spatial index such as R*-tree. We chose FAST (FAST-Rtree) as a flash efficient index since it is credited as one of the state-of-the-art spatial indexes for SSDs in the literature. During the experimentation, we disabled

TABLE 3.1: SSD Characteristics

	Intel DC P3700 (NVMe1)	Samsung SM951A (NVMe2)	Samsung 850 Pro (SATA)
Seq. Read	up to 2700MB/s	up to 2150MB/s	up to 550 MB/s
Seq. Write	up to 1100MB/s	up to 1550MB/s	up to 520 MB/s
Random Read	450K IOPS (100% span)	300K IOPS (4KB, QD32)	up to 100K IOPS (4KB, QD32) up to 10K IOPS (4KB, QD1)
Random Write	75K IOPS (100% span)	110K IOPS (4KB, QD32)	up to 90K IOPS (4KB, QD32) up to 36K IOPS (4KB, QD1)
DRAM	not available	512MB	512MB

the logging mechanism of FAST – it is used for recovery after system crash – in order to assure fair comparison.

All the experiments were performed on two Lenovo P310 workstations, running CentOS Linux 7 with Kernel 4.10.12. Each workstation was equipped with a quad-core Intel Xeon E3-1220 3.0GHz CPU, 8GB of RAM, and a SATA SSD for hosting the operating system. Two NVMe SSDs and one SATA SSD were employed for the experiments. The first workstation was equipped with an INTEL DC P3700 480GB PCI-e 3.0 NVMe SSD (NVMe1) and a Samsung 850 Pro 250GB SATA 3.0 SSD (SATA), and the second one with a 512GB SM951A Samsung NVMe PCI-e 3.0 SSD (NVMe2). Table 3.1 summarizes the performance characteristics of the three devices as provided by manufacturers' data sheets.

3.5.2 Results

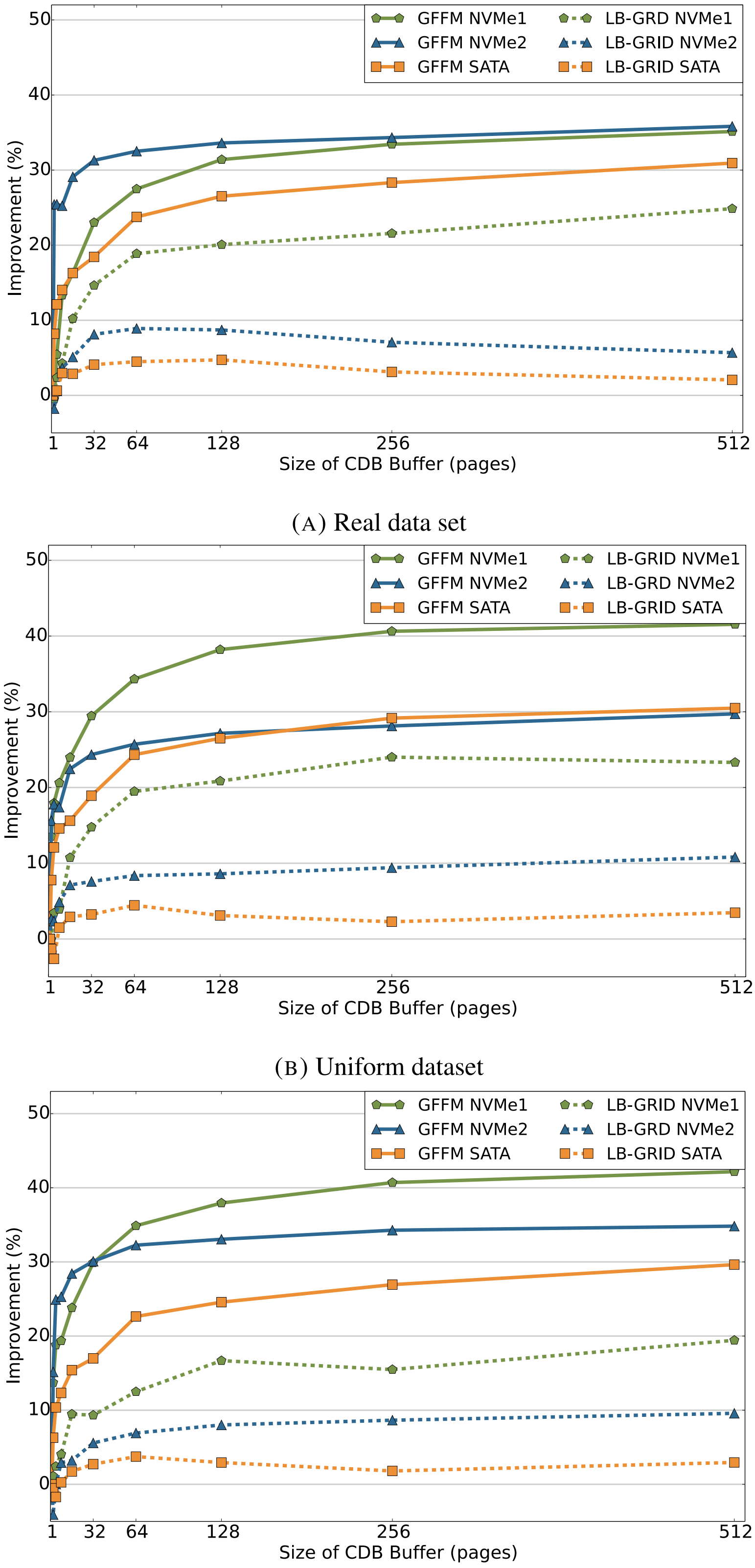
In the experiments, we used three different spatial datasets, one real and two synthetic. The real dataset was obtained from Spatial Hadoop website[1]. It contains geographical points on the planet and it has been extracted from Openstreetmap. The two synthetic datasets follow Gaussian and Uniform distributions, respectively. We utilize Direct I/O (*O_DIRECT*) to bypass the Linux caching system and measure the real improvement.

Performance parameters

We start the evaluation by studying two important performance parameters, the size of cold LRUs (*CLRU*) and the size of the in-memory buffers in total. We seek to identify how these two parameters influence the performance of LB-Grid and GFFM.

We first try to tune the size of write buffers (*CLRU*s). As we have described in Section 3.4.3, we use separate write buffers for directory and bucket/log-bucket pages. The size of the *CLRU*s determines the number of outstanding write operations that are send to the

[1] http://spatialhadoop.cs.umn.edu/datasets.html



(A) Real data set

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.9: Relative change in the performance of the construction of LB-Grid and GFFM vs write buffer (CLRU) size (number of 8KB pages) for various datasets. A number of 128 pages is suitable in the most cases.

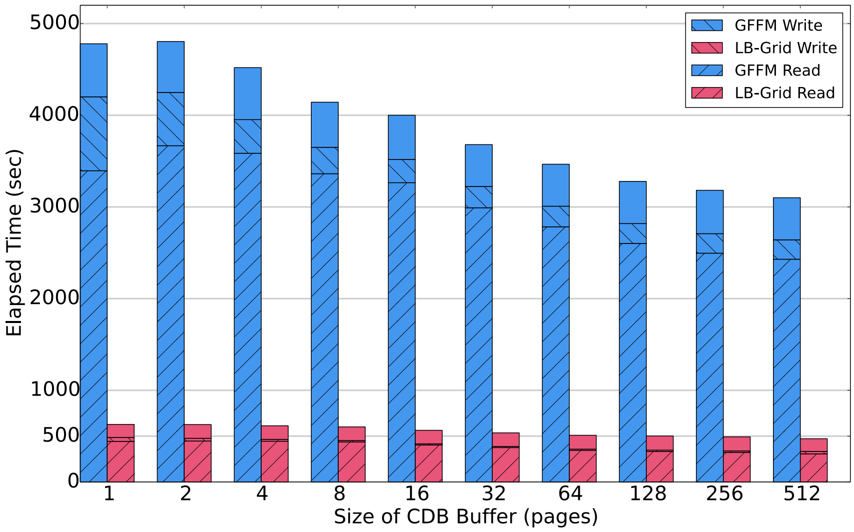
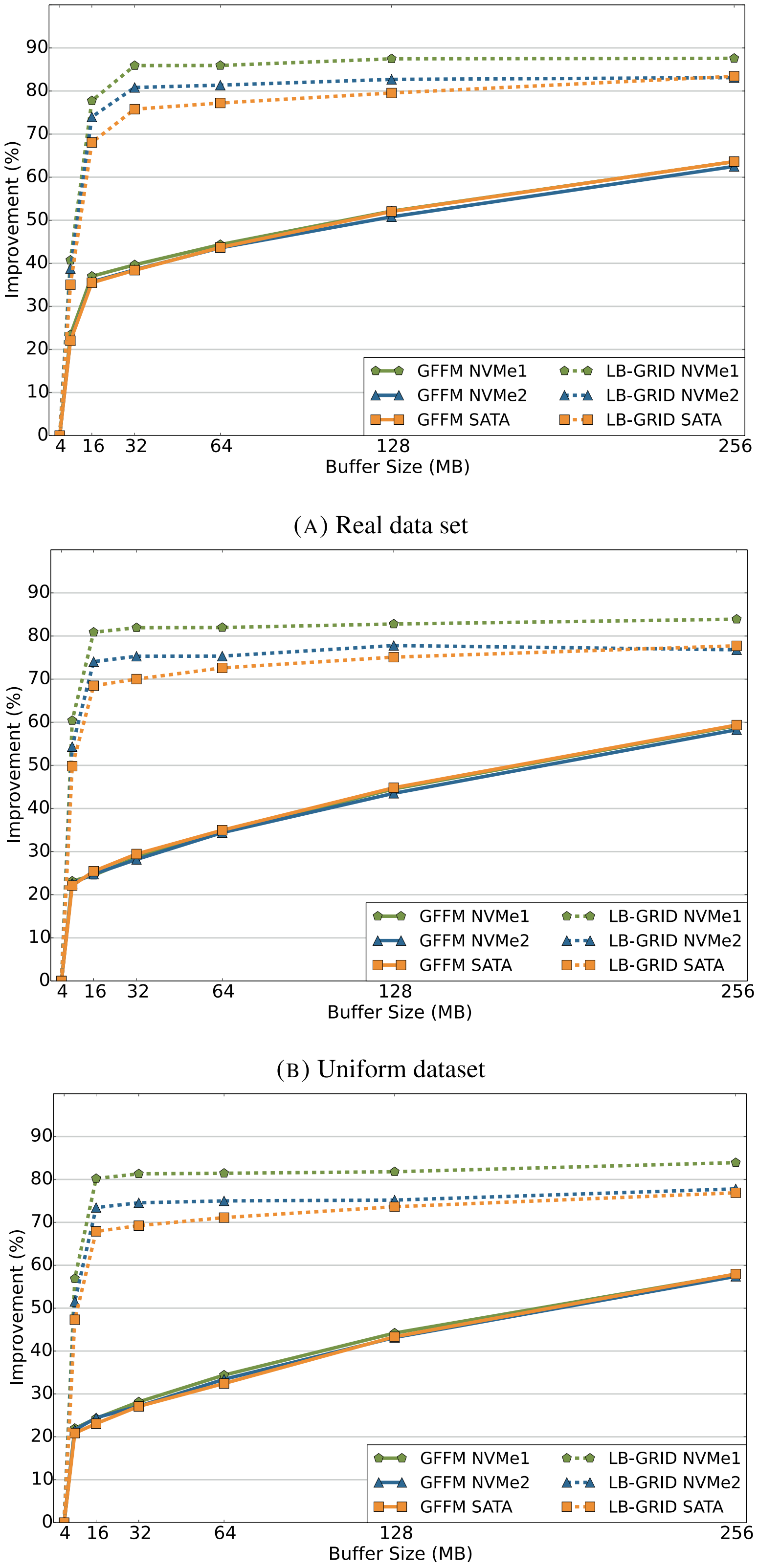


FIGURE 3.10: LB-Grid and GFFM construction times vs write buffer (CLRU) size (number of 8KB pages) of the Real dataset running in the NVMe1 device. The results for the rest of datasets and devices follow a similar pattern.

SSD at once. We measure the elapsed time for the construction of LB-Grid and GFFM, using 20M points in each run. We fix page size at 8KB in all tests. Figure 3.9 depicts the results. The size of each *CLRU* (in number of pages) is reported on the x-axis, while the relative change in performance is the dependent variable on the y-axis. The elapsed time for $CLRU_{size} = 1$ is used as reference value. The results show that a number of 128 pages (or 1MB) is the pertinent parameter in the most cases. This value is also confirmed by the findings of other works [76]. The results follow a similar pattern in all datasets and testing devices for both LB-Grid and GFFM. However, the performance gain for GFFM is higher. This has to be thought taking into account the total execution times, which are presented in Fig. 3.9a. According to them, the construction of LB-Grid is performed about 6 to 12 times faster than GFFM. Therefore, the room for improvement in LB-Grid is less. Figure 3.9a illustrates the attained run-times for the real dataset in the NVMe1 SSD. The results for the other SSDs and datasets are similar, thus they are omitted. Observing Fig. 3.9a we draw another important conclusion: as the size of *CDB* buffer increases, both read and write times decrease, thanks to the reduction of interleaving between read and write operations.

Moreover, we study the influence of the hot LRU buffers (*HLRU*s) in the performance of LB-Grid and GFFM. The total amount M of main memory that is reserved for all buffers in LB-Grid is $M = CLRU^{DP}_{size} + CLRU^{LB}_{size} + HLRU^{DP}_{size} + HLRU^{LB}_{size}$. This type applies correspondingly to the GFFM. We measure the execution times for the construction of the LB-Gird and the GFFM with 20M points. In line with the previous findings, we set the size of *CLRU*s to 1MB (128 8KB pages) each. The remaining available memory is shared evenly between the *HLRU*s. Thus, the size of each *HLRU* is $HLRU^{DP}_{size} = HLRU^{LB}_{size} = (M-2)/2$. We execute the experiment several times, doubling the total size M in each run, starting from 4MB up to



(A) Real data set

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.11: Relative change in performance as buffer size increases from 4MB up to 256MB. After 16MBs improvement accelerates slower.

256MB.

Figure 3.11 presents the relative change in the elapsed time as memory size M increases. The elapsed time for $M = 4$MB is the reference value. We observe that the improvement in LB-Grid is smaller than that of GFFM. This happens because LB-Grid continuously logs incoming points without retrieving any data buckets from the SSD. Thus, its memory requirements during insertions are low and mostly concern caching of directory pages. For this reason, increasing the size of the buffer over 16MB does not contribute any performance gain. On the other hand, in the GFFM, each insertion requires the retrieval of the corresponding data bucket first. Therefore, increasing the memory buffers from 4MB to 16MB provides a performance gain which varies from 23% to 37% and this increase continues up to 64% at 256MB.

Based on the above findings, we adjusted the size of write buffers ($CLRU$s) to 128 pages each, and the total size of the in-memory buffers to 16MB in all experiments that follow. We reserved the same amount of memory for buffering in R*-tree and FAST as well.

Insert/Search Queries

We evaluate the performance of the proposed indexes, using three datasets and six different workloads with varying insert/search ratios. The details of each workload are depicted in Table 3.2.

To further understand the influence of buffering to the proposed index, we executed the experiments twice. In the first run, we equally distribute the amount of 16MBs between the directory and the data pages buffers. Next, we reserved 75% of the buffer for directory pages and the rest 25% for data pages, respectively. In Table 3.3 we present a comparison between the two runs. Specifically, we show the percentage change in performance as the memory buffer for directory pages increases to 75% of the total reserved memory. The results confirm that promoting the caching of directory pages improves performance. This occurs because directory pages are referenced more frequently. In the following, we discuss only the second run since it provides better results.

Regarding the real dataset, all indexes (GFFM, LB-Grid, R*-tree and FAST) are initialized with 500M points. We report the elapsed time for 10M operations in Fig. 3.12a. The

TABLE 3.2: Experimental setup

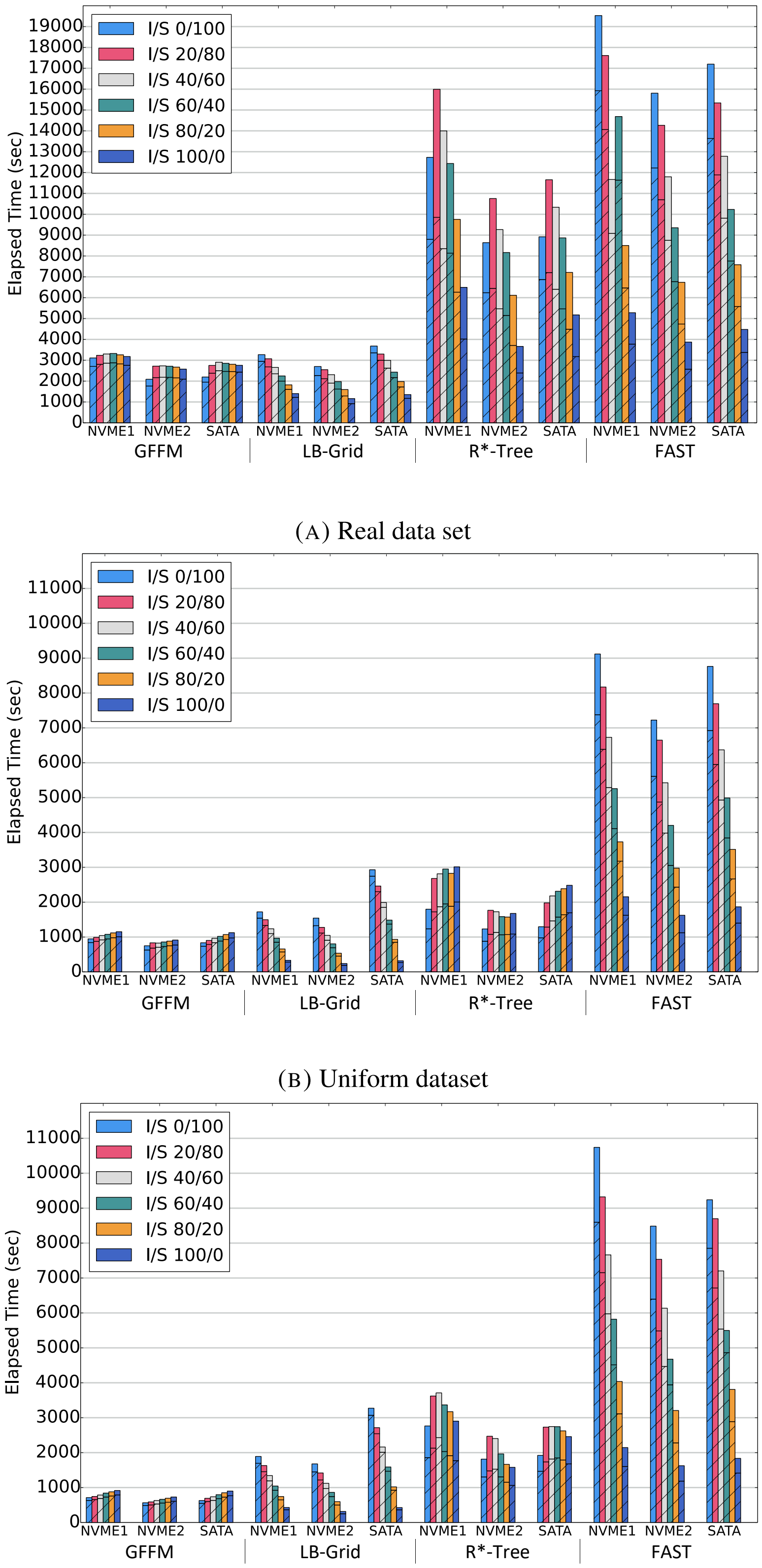
Dataset	Index Initialization	Workload operations
Real	500M	10M
Uniform	50M	5M
Gaussian	50M	5M

TABLE 3.3: Performance gains for various workloads

	LB-Grid								
	NVMe1			NVMe2			SATA		
I/S	Real	Gauss	Uniform	Real	Gauss	Uniform	Real	Gauss	Uniform
0/100	5.52	9.69	11.98	1.65	6.28	9.84	4.44	4.50	5.39
20/80	3.58	10.14	12.34	0.52	6.27	9.65	6.23	5.37	5.93
40/60	6.30	12.67	14.93	1.31	11.13	11.75	3.06	6.55	7.67
60/40	8.02	14.78	17.52	0.90	12.18	14.62	6.71	8.53	9.74
80/20	11.24	19.36	22.89	4.29	16.06	19.85	5.65	11.53	15.59
100/0	12.42	28.97	35.71	7.53	29.52	35.15	11.79	25.91	33.85
	GFFM								
	NVMe1			NVMe2			SATA		
I/S	Real	Gauss	Uniform	Real	Gauss	Uniform	Real	Gauss	Uniform
0/100	2.09	27.68	16.72	-1.58	28.27	16.76	4.13	24.41	14.32
20/80	3.00	27.88	16.96	1.67	28.41	11.21	6.41	26.56	16.29
40/60	3.20	27.84	16.84	1.01	27.59	16.97	3.81	27.12	16.38
60/40	2.43	26.71	17.35	3.73	27.31	16.54	7.47	25.83	16.45
80/20	5.46	26.37	17.02	7.32	26.84	17.42	8.35	24.76	15.74
100/0	5.90	26.02	17.23	5.15	26.33	17.29	6.67	24.32	15.30

bottom part of each bar in the graph corresponds to the I/O time. The LB-Grid is more efficient than the GFFM in the update-intensive test cases and in the NVMe SSDs. More specifically, for the NVMe devices, the GFFM is faster only if the workload does not include any insertions. The LB-Grid is considerably faster than the GFFM, starting from 1.24 times for the NVMe1 and 1.18 times for the NVMe2, when the insert ratio is 40% of the workload, and increases to 2.26 and 2.21 times, respectively, when the workload includes only writes. The R*-tree is 1.4 to 4.9 times slower than the GFFM and 2.4 to 5.5 times slower compared to LB-Grid. FAST does not perform well in the read sensitive workloads, since it has been developed as a write efficient algorithm. The absence of efficient algorithms for reading and of an adequate read buffering policy impair its read performance, comparing even with the R*-tree. With regard to the update heavy workload (I/S 100/0), FAST provides better results than R*-tree in most test cases, however it is slower than GFFM and LB-Grid in all experiments. For example, in the runs with the NVMe2 SSD, FAST is 1.5 and 3.3 times slower than GFFM and LB-Grid, respectively.

The same methodology is also followed in the case of synthetic datasets. Thus, 50M points are used for the initialization and then 5M I/O requests are issued to the indexes. The results for the Uniform and Gaussian datasets are depicted in the Figures 3.12b and 3.12c, respectively. GFFM performs better than LB-Grid in the workloads where reads are the majority. On the other hand, LB-Grid outperforms GFFM in the write-dominated workloads. LB-Grid presents worse performance in the SATA SSD, exhibiting large divergence between run times of read-dominated and write-dominated workloads. Indeed, it presents worse performance compared to the R*-tree in the heavy read workloads (I/S 0/100 & 20/80). This reveals the inherent weakness of SATA SSDs to support high volumes of simultaneous I/O



(A) Real data set

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.12: Execution times of different workloads in the various data structures. LB-Grid outperforms GFFM in the update dominated test cases. GFFM and LB-Grid overcome R*-tree and FAST.

operations as we detail in Section 1.3.2. However, LB-Grid is up to 6.7 and 9 times faster than the R*-tree in the write-oriented test cases of the Uniform and Gaussian datasets, respectively. FAST provides better results than R*-tree in the insertion-only workload, while it remains slower in the rest of them. Compared with GFFM and LB-Grid, it exhibits worse performance in all cases. The attained results advocate for the value of flash efficient spatial indexes.

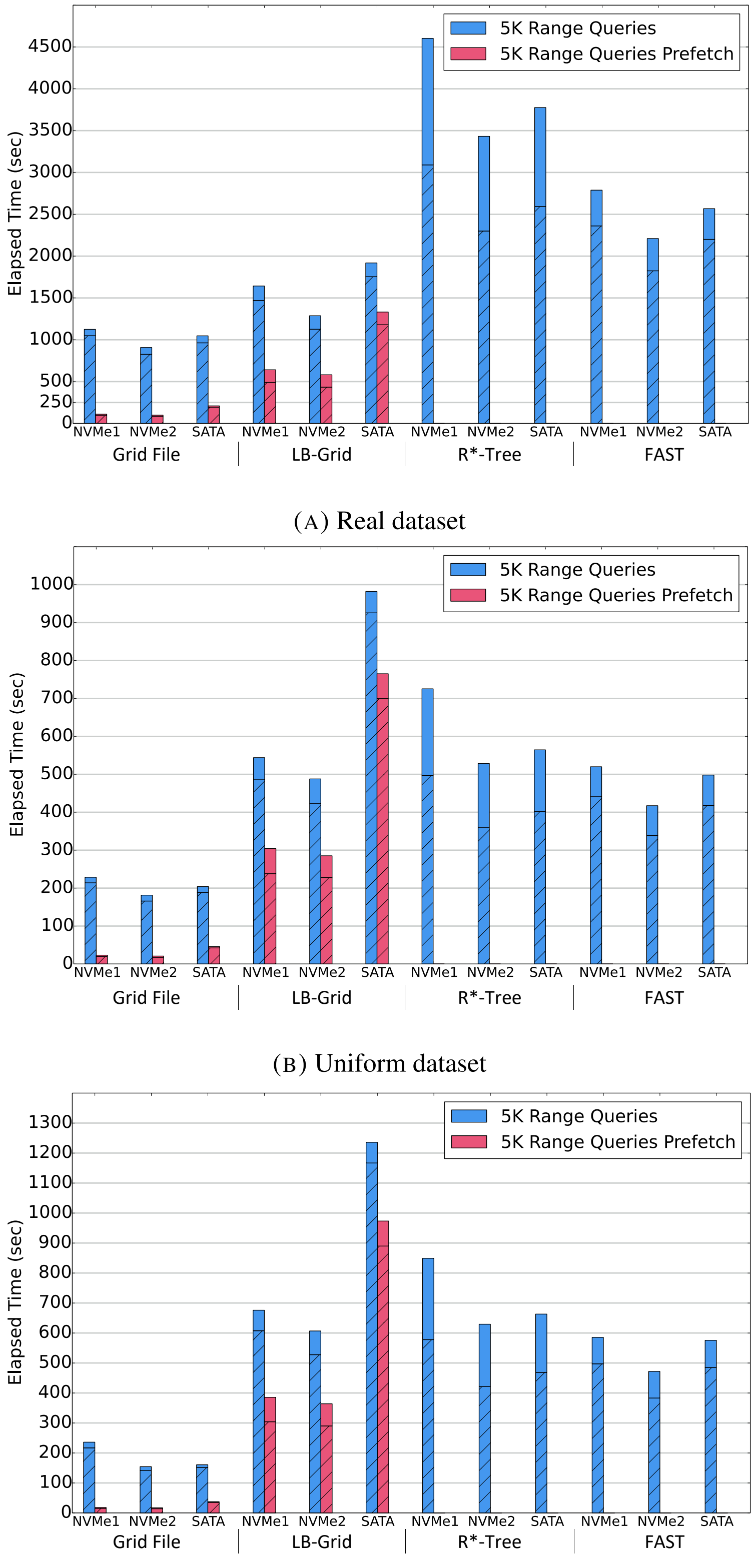
Range Queries

In this section we analyze the performance of range queries. We issued 5K requests to each one of the indexes, studying two different scenarios for each test case. The first scenario does not exploit the special features of SSDs, whereas the second one utilizes batch reads for retrieving directory and data pages. We initialized the indexes with the datasets from Table 3.2 prior to executing the queries. Figure 3.13a presents the elapsed times corresponding to the real dataset, while Figures 3.13b and 3.13c depict the respective times for the Uniform and Gaussian datasets. Table 3.4 summarizes read accesses to the secondary storage.

The results are remarkable, especially for the GFFM. Its optimized version is 10.2 (NVMe1) and 9.2 (NVMe2) times faster than the original one considering the real dataset in the NVMe SSDs, whereas it is 5 times faster for the SATA. Similar results acquired from the other two workloads. Higher speedups are presented in the Gaussian dataset test case, reaching 12.9x in NVMe1 and 9x in NVMe2 SSD. The LB-Grid achieves to improve its performance, however the gain is smaller in comparison to the GFFM. It presents better performance with the real dataset and the NVMe devices, improving its execution time by 2.6 times in the NVMe1 device and by 2.2 times in the NVMe2 one. R*-tree lacks significantly in performance compared to LB-Grid and Grid File in the experiments that utilize the large real dataset, but with the smaller synthetic datasets its performance gap with LB-Grid is reduced and in the case of the SATA SSD it provides better results. This happens due to the inherent weakness of SATA SSDs to support efficiently massive I/O operations which is fundamental for the performance of LB-Grid. FAST exhibits better results than R*-tree in all workloads. Comparing it with the Grid Files, it is slower in all the experiments that concern the large real dataset. However, it outperforms the un-optimized range query of LB-Grid when the synthetic datasets are evaluated. Additionally, it presents worse results than the proposed flash efficient range query

TABLE 3.4: Range Queries - Number of read operations issued to the SSD

	LB-Grid			GFFM		
	Real	Guassian	Uniform	Real	Guassian	Uniform
RQ	9.90E+06	2.13E+06	1.79E+06	8.86E+06	2.61E+06	1.76E+06
P-RQ	2,59E+05	1,30E+05	1,05E+05	1.16E+05	3.31E+04	4.22E+04
	FAST			R*-Tree		
RQ	2,04E+07	4,18E+06	3,71E+06	2.92E+07	5.29E+06	4.56E+06



(A) Real dataset

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.13: Range queries execution time of different workloads. The proposed algorithm speeds up the execution of range queries up to 12.9x and 2.6x for GFFM and LB-Grid respectively (real dataset).

algorithm for the GFFM and LB-Grid, with the exception of the runs concerning LB-Grid and the SATA SSD, similarly to R*-tree.

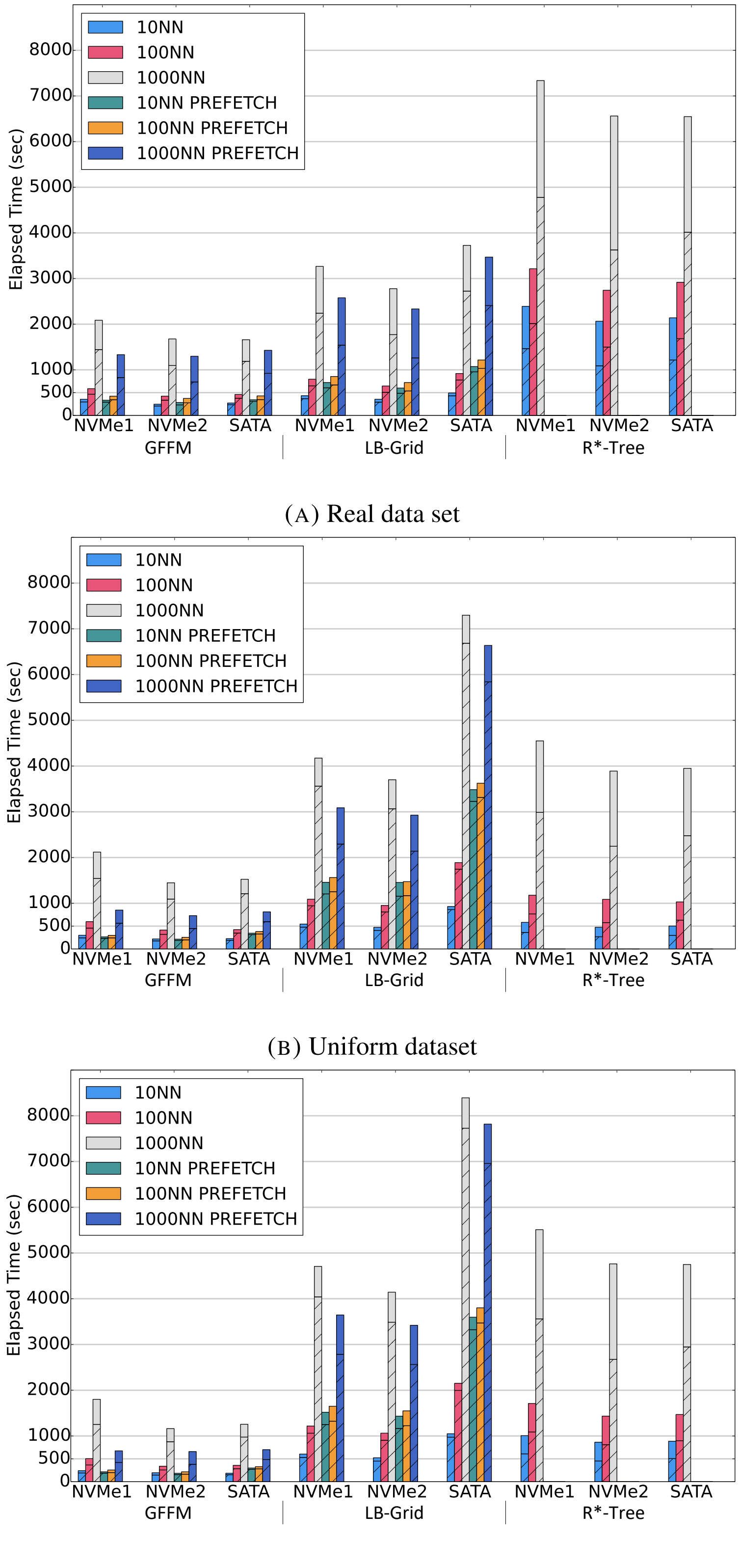
Regarding the number of I/O operations, the proposed range query algorithm achieves a reduction of up to 79 times to the number of the issued read operations in GFFM. In LB-Grid, the reduction varies between 16.3 times (Gaussian dataset) and 38.3 times (real dataset).

kNN Queries

We conducted a set of experiments to evaluate the performance of the kNN query algorithm for both LB-Grid and GFFM. The key idea behind the proposed kNN algorithm is to exploit the performance of SSDs prefetching pages before they are utilized. Although issuing multiple read requests at once accelerates I/O, it can degrade the overall performance, since many of the retrieved pages may be redundant. This occurs because of the pruning that the kNN algorithm performs. Therefore, we investigated three different experimental scenarios: the first one does not utilize prefetching (baseline), the second one exploits batch reads to prefetch only data pages (buckets/log-bucket), whereas the third one prefetches both directory and data pages. We measured the elapsed time for the execution of 1M kNN queries in each test case. Prefetching of directory pages (third scenario) imposes more I/O than actually is needed, degrading performance in the most runs. Thus, we omit to further discuss these results, focusing on the other two scenarios.

Figure 3.14a illustrates the results of the real dataset, Figures 3.14b and 3.14c refer to the synthetic ones, while Table 3.5 summarizes the accesses to the SSDs. As expected, the GFFM presents better performance than LB-Grid, since it requires less I/O to get the nearest neighbors of a given point. With regard to the execution of 10NN queries in GFFM, there is some improvement (up to 1.13x) to the results concerning the NVMe1 device. However, for the other two devices, either the improvement is very small or prefetching imposes even greater execution times. The proposed method performs better in 100NN queries, where an improvement starting from 1.38x (real dataset) up to 2x (Uniform dataset) is achieved for the NVMe1. Similarly, for NVMe2, the gain is from 1.13x (real dataset) to 1.62x (Uniform dataset). As expected, the gain is less for the run in the SATA SSD. The improvement is higher in 1000NN queries, reaching up to 1.56x for the real dataset in NVME1 device, whereas for the smaller synthetic datasets it is even greater.

Prefetching is shown not to be an advantage for LB-Grid in 10NN and 100NN queries, because it causes the retrieval of a large number of redundant log-buckets. In contrast, in the 1000NN test cases, prefetching from the NVMe SSDs provides a performance gain which ranges from 1.19x to 1.26x in the real dataset. This gain is further increased up to 1.35x in the experiments that employ the Uniform dataset. However, on the SATA SSD, the gain does not exceed 1.1x (uniform dataset). This leads us to the conclusion that the proposed method performs well in the cases where larger numbers of neighbors are sought.



(A) Real data set

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.14: kNN queries execution time. The proposed algorithms are more efficient when a large number of neighbors is searched.

The proposed algorithms for LB-Grid and the Grid File outperform the R*-tree in all

TABLE 3.5: kNN Queries - Number of read operations issued to the SSD

	LB-Grid			GFFM		
	Real	Gaussian	Uniform	Real	Gaussian	Uniform
10NN	2.38E+06	1.98E+06	1.90E+06	2.31E+06	1.54E+06	1.90E+06
100NN	4.15E+06	3.82E+06	3.62E+06	3.85E+06	2.98E+06	3.60E+06
1000NN	1.45E+07	1.41E+07	1.32E+07	1.27E+07	1.07E+07	1.31E+07
PR 10NN	2.01E+06	1.40E+06	1.35E+06	1.98E+06	1.06E+06	1.36E+06
PR 100NN	2.32E+06	1.59E+06	1.48E+06	2.22E+06	1.17E+06	1.49E+06
PR 1000NN	5.50E+06	3.64E+06	3.16E+06	4.78E+06	2.57E+06	3.18E+06
				R*-Tree		
10NN				1.37E+07	5.71E+06	3.40E+06
100NN				1.89E+07	1.02E+07	7.12E+06
1000NN				4.48E+07	3.29E+07	2.76E+07

experiments. Specifically, for 1000NN queries, in the real dataset, the Grid File is up to 5.5 times faster than the R*-tree, whereas LB-Grid is up to 2.8 times faster. This performance difference is further improved for the Grid Files in 10NN and 100NN queries. We omit to evaluate FAST for kNN queries, since no algorithm is described for this type of query.

Table 3.5 summarizes the disk access counters (number of operations). We can see that for the 10NN query the number of the issued I/O operations does not significantly change when prefetching is applied. The highest reduction in the number of operations is noticed in the 1000NN query where the best performance is achieved. Moreover, the Grid Files require less I/O operations to reach the result (even the baseline algorithm) compared to the R*-tree.

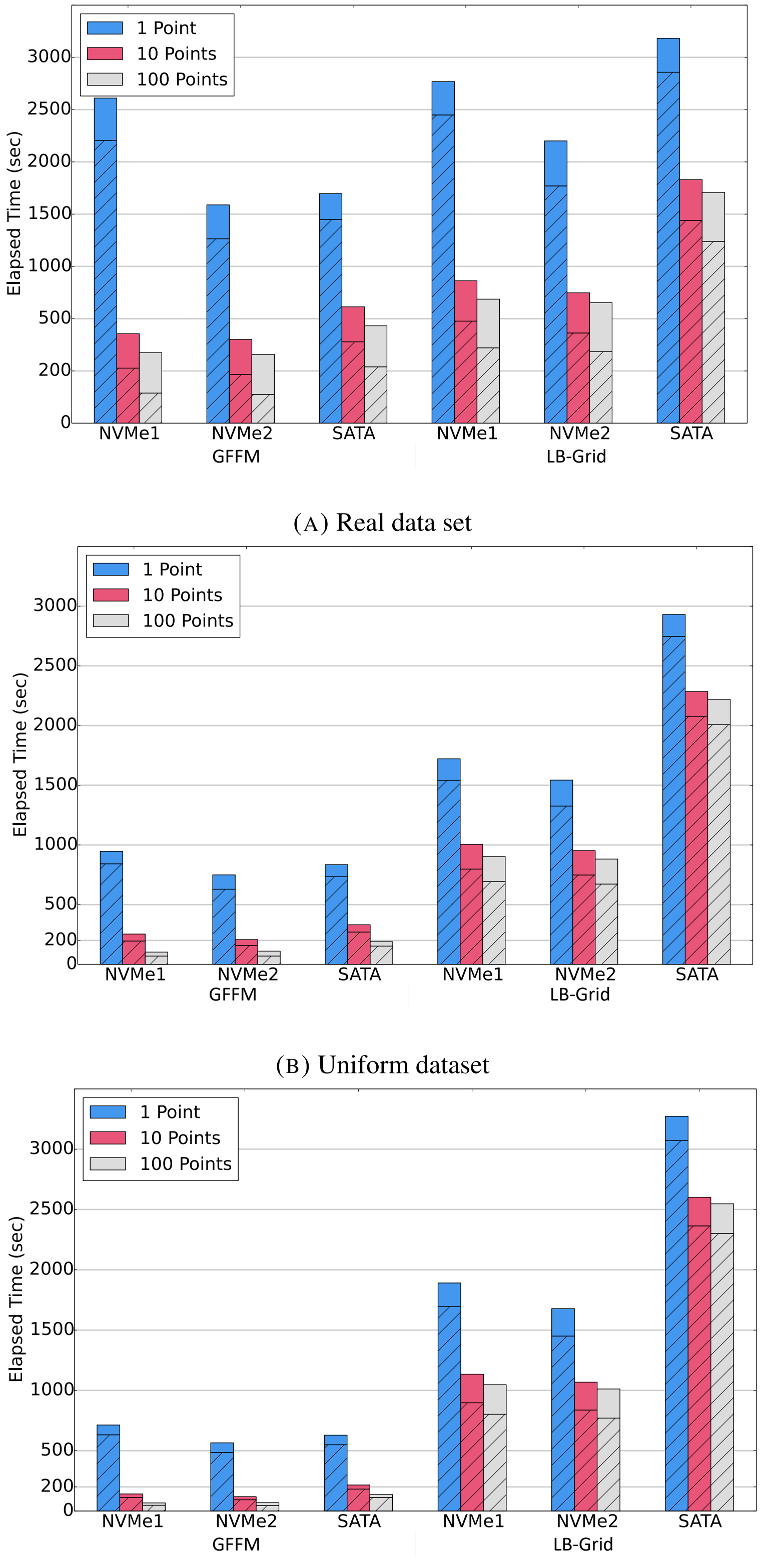
Regarding to the different SSD devices, the experiments unveil that the SATA SSD can not support batch reading of directory and data pages as efficiently as the NVMe devices do.

Group Point Queries

The evaluation of Group Point Queries was performed by using the datasets of Subsection 3.5.2. Specifically, we utilized the ones with insert/search ratio 0/100. Please recall that the real dataset contains 10M points and the synthetic ones 5M each. We study three different configurations; the first is actually a single point query which is used for reference, whereas the other two are considered with regard to groups of 10 and 100 points, respectively.

Figure 3.15a illustrates the running times for the real dataset, while Figures 3.15b and 3.15c refer to the running times for the Uniform and the Gaussian datasets, respectively. Table 3.6 presents the number of read operations for each test case. Once again, the results reveal the efficiency of the contemporary SSD devices in the parallel processing of multiple I/O requests.

Specifically, for the GFFM, group reading of 10 points in the real dataset experiment (Fig. 3.15a) is from 2 (SATA) up to 3.6 times (NVMe1) faster than single point queries. The improvement further increases for the groups of 100 points, varying from 2.36 in the SATA SSD to 4.6 times in the NVMe1. The results are even better in the smaller synthetic



(A) Real data set

(B) Uniform dataset

(C) Gaussian dataset

FIGURE 3.15: Group point queries execution times. An improvement of 3.6x is achieved for a group of 10 points from the real dataset in the NVMe1 device

TABLE 3.6: Group Point Queries - Number of read operations issued to the SSD

	LB-Grid			GFFM		
	Real	Guassian	Uniform	Real	Guassian	Uniform
1	1.95E+07	6.68E+06	6.45E+06	1.93E+07	5.07E+06	6.48E+06
10	2.00E+06	9.92E+05	9.84E+05	2.00E+06	5.77E+05	9.85E+05
100	3,12E+05	3,68E+05	3,30E+05	2.00E+05	9.08E+04	1.00E+05

workloads (Fig. 3.15b and 3.15c), wherein the improvement is up to 5x for the groups of 10 points and 10.6x for the groups of 100 points in the Gaussian dataset, and up to 3.7x for the 10 points and up to 9.2x for 100 points in the Uniform dataset, respectively. Regarding the read operations to the secondary storage, their number decreases in reverse proportion to the number of points in the group by an order of magnitude in each case, as Table 3.6 depicts.

Examining LB-Grid, the performance gain is also important, especially for the NVMe devices. The run time improvement in the real dataset varies from 1.6x (SATA) to 2.4x (NVMe1) for the 10 point groups, whereas for the 100 point ones the gain is increased to 1.7x and 2.7x, respectively. The results are similar for the synthetic datasets as well.

Concluding, the experimental results validate the efficiency of the proposed Group Point Query algorithms for both the Grid File and LB-Grid and affirm, once more, that grouping of several search requests together enhances the utilization of high performing flash storage devices.

3.6 The xBR$^+$-tree

The xBR$^+$-tree [110] is a hierarchical index for multidimensional points based on the Quadtree. It is an extension of the xBR-tree introduced in [121]. In the 2 dimensions it represents index space with a square (like Quadtree), which recursively decomposed into four equal quadrants. Each tree node occupies one page on the secondary storage. The *leaves* of the tree store the actual data, while the upper level nodes (*internal nodes*) contain pointers downwards to the data. A leave can store up to a predefined number of data elements. As soon as this number is reached, a split operation occurs partitioning the region of the leaf into two new regions using a Quadtree like decomposition. Thus, a new subregion is inserted, which corresponds to the quadrant that hosts the larger number of elements. The other region is actually what is left from the original leaf region, subtracting the new inserted one. The internal nodes are also split upon become full. The split operation aims at balancing space usage between the two nodes.

We have contributed to turn pre-existing algorithms for the XBR$^+$-tree into flash efficient ones. Specifically, our work regards bulk-loading, bulk-insertions and batch spatial queries (i.e. point-location, window and distance-range) [107, 109]. In the following we briefly describe the interventions we made to enhance the XBR$^+$-tree's performance in flash storage.

Bulk Loading and Bulk Insertions. There exist three methods to add new items into a data index, namely : bulk-loading, bulk-insertions and one-by-one insertions. Bulk loading algorithms build an index from scratch, using a pre-existing data set. Bulk insertions regard a batch update on an existing index with a bundle of new elements, while one-by-one insertions refer to the insertion of one data item each time.

[105, 108] present an efficient method for building an XBR$^+$-tree through bulk-loading. The algorithm is executed in four phases. During phase 1, the input file is split in 2^n sub-files, where n is the number of dimensions. Each sub-file corresponds to an n-cube of the index space. If the sub-files fit into the available main memory the algorithms proceeds in phase three. Otherwise, phase 2 takes place, that recursively splits the dataset into blocks that fit into memory. Each block corresponds to a sub-quadrant of the XBR$^+$-tree. The points in each sub-file are sorted according to z-order. The 3rd phase fetches a block into memory and constructs a main memory XBR$^+$-tree, that is merged to the disk resident XBR$^+$-tree (created in the previous iteration) in phase 4. Obviously, bulk-loading involves a large amount of I/O that determines its performance. Thus, we adapted the bulk-loading algorithm to better exploit the advantages of SSDs. Specifically, we modified phases 1 and 2 in order to perform reads and writes in large batches, thus, exploiting the high throughput of SSDs. Additionally, in phases 3 and 4 we postpone to write newly created nodes to the SSD. Instead, we accumulate them into a write buffer, which is persisted at once, whenever gets full.

We follow a similar approach for the bulk-insertions as well. The original bulk-insertion algorithm for the HDD is presented in [106]. For a given data set, the algorithm attempts to insert the points into the leaves of the tree. In the case that a leaf overflows, an in-memory sub-tree is created having root the parent of the leaf. Consequently, the in-memory sub-tree is merged with the original (disk-resident) one. We modified the algorithm in order to exploit SSDs' efficiency by performing batch reads and writes of the leaf nodes. We utilize an in-memory buffer to enable these batch operations.

The introduced algorithms overcome the original ones in terms of disk accesses and execution time. Specifically, the batch loading algorithm improves execution time in a range from 65.5% up to 97%. Similarly, the bulk insertions algorithm improves from 92% up to 96%. The gain is higher in the runs that utilize the larger datasets.

Batch Query algorithms. A point location query determines the existence of a given point p into an index T, a window query returns all the points of T that are contained into a rectangle w and a distance range query returns all the points of T that are found in distance sorter than r from a given point p. The proposed batch query processing algorithms exploit the high I/O capacity of contemporary SSDs. Specifically, for each one of the aforementioned query types, we propose processing of several queries together, as a batch, following an approach similar to that in group point queries of GFFM and LB-Grid. Thus, we initially segment query sets into batches that fit into the available main memory. Next, each batch is processed accessing recursively the tree from the root to the leaves. The appropriate nodes are fetched at once each time, exploiting the internal parallelism of SSDs and the benefits of NVMe protocol. The proposed algorithms achieve remarkable performance gains, especially when large datasets are used (up to 97.6%). This stems from the reduced latency since less read requests are submitted to the SSD.

3.7 Conclusions

In this chapter we studied several flash aware point access methods. Specifically, we analyze the performance of a flash efficient implementation of Grid File (GFFM) and we introduced LB-Grid, a log-based variant of Grid File. LB-Grid accelerates update operations without scarifying search efficiency, exploiting the performance characteristics of flash SSDs. We presented algorithms for Range, kNN and Group Point Queries for both the GFFM and LB-Grid, and a buffer manager which can discern bucket and directory pages. We focused on optimizing query processing, exploiting SSDs' internal parallelism and the advantages of the NVMe interface to accelerate grouped I/O operations. We conducted extensive experiments to evaluate our design, employing two NVMe and one SATA SSDs. Differentiating from previous works, we used large datasets with up to 500M points in various test cases. The conclusions we draw from our study show that LB-Grid provides high performance gains during index construction and in update intensive workloads. Additionally, it shows adequate performance in read intensive workloads. Regarding range, kNN and group point queries, significant gains are exhibited, mostly for the (adapted) Grid File, whereas speedups[2] up to 10.2x, 1.56x and 4.6x were measured, respectively.

We also present our contribution in the development of flash efficient algorithms for the xBR$^+$-tree. We introduce new bulk-loading, bulk-insertion and batch query processing methods that exploit the advantages of modest SSDs. The gathered results show performance improvement in all test cases, with the gain to be higher in the large dataset runs.

Our plans for future work include a new Grid File variant that combines both GFFM and LB-Grid, and automatically switches between log and normal modes depending on the workload type. The line of our research includes also the study of other types of operations, like batch insertion and bulk loading, for Grid File and LB-Grid.

[2] The aforementioned speedups concern the 500M points dataset. The gain is higher in the smaller datasets.

4 | Hybrid Data Structures

4.1 Introduction

The emergence of non-volatile memories has enabled new storage devices with amazing features. Data intensive applications, like DBMSes, have drawn significant performance advantages by this evolution. As a result, index structures for flash SSDs have become a promising field of study for many researchers. A new class of SSDs was introduced by Intel, under brand name “Optane”, earlier in 2017. These storage devices are based on 3DXPoint non-volatile memory technology. Their performance characteristics create new lines of research in data indexing.

The efficiency of a storage device is described by three performance metrics: IOPS, bandwidth and latency. IOPS determine the number of I/O operations that the device is able to carry out over the unit of time. On the other hand, the bandwidth expresses the throughput that a drive can deliver, measured in MBs/sec. Finally, latency is the amount of time that an I/O request takes to complete, i.e., the response time of an operation. Latency is of paramount importance for the efficiency of a storage system, since low latency is tightly connected with better user experience. Little’s law [74] for storage systems mandates that $IOPS = \frac{Queue}{Latency}$, where Queue is the number of outstanding requests, i.e. the number of I/O requests sent to the device in parallel. It is clear that reducing latency retains IOPS efficiency even with less concurrent I/O. Lower latency values enable workloads to finish into a fraction of the initial time. According to [139] the latency of high-performance NVMe SSDs contributes over 19% of the overall response time on online applications. New SSD devices have been introduced lately providing ultra-low latency; such devices are Intel’s Optane series and Samsung’s Z-NAND. Intel Optane SSDs (3DXPoint) can provide a latency reduction of one order of magnitude compared to the conventional NAND flash SSDs. 3DXPoint SSDs can deliver high IOPS even when a small number of concurrent outstanding I/O is used (small queue depth), while their NAND counterparts are more efficient under large batched I/O [61, 46].

Previous works for flash efficient database indexes focus on exploiting the high internal parallelism of SSD devices by issuing multiple read or write operations at once. Several works utilize large queue depths, pursuing to distribute the workload among multiple NAND chips, accelerating query performance [102]. Although this technique has been proved very useful so far, the low latency of new NVM technologies can further improve the performance, especially where limited opportunity for grouping I/O requests exists. To the best of our knowledge, this is the first time that the low latency 3DXPoint NMV is exploited to accelerate the performance of a spatial index.

4.2 Hybrid storage systems

Hybrid storages are not rare in database systems; several algorithms have been proposed so far [92]. Most of the related works, until now, consider flash-based solid state drives as the performance tier and magnetic disks as the storage tier. In fact, hybrid storage systems employ SSDs either as a cache between main memory and HDD or as high performing devices storing permanently the hottest data.

In [18] a flash based SSD acts as an extension of the standard main memory bufferpool accommodating high priority data. The hot data regions are identified using frequency and recency statistics, while an aging mechanism ensures that the cached regions are in line with the I/O pattern, as it changes over the time. The authors in [77] study different buffer management policies in relational DBMSes (i.e. MySQL), when a hybrid SSD/HDD scheme is used as persistent storage. Their findings indicate that the performance of hybrid systems, which employ SSDs for caching, is highly dependent on the ratio between SSD and HDD bandwidth.

Hystor [27] is an extension to Linux operating system that identifies hot and performance critical data blocks by monitoring the I/O sequence. This data is stored on a fast SSD instead of a magnetic disk. Following a different roadmap, MOLAR [78] proposes the implementation of the hot page detection mechanism into the SSD's controller. Simulated experiments have shown that MOLAR can reduce the average write latency in SSDs by 3.5 times.

The efficiency of hybrid storage systems is connected with the accuracy of hot data identification. [79] uses a probabilistic algorithm to locate hot data. The algorithm maintains two probabilities. The first probability contributes to the decision of which pages should be evicted from RAM and the second one determines the persistent storage (SSD/HDD) an evicted page should be moved to.

A recent study [136] investigates the use of 3DXPoint technology to enhance the performance of database systems. Specifically, the authors recognize write amplification, careless

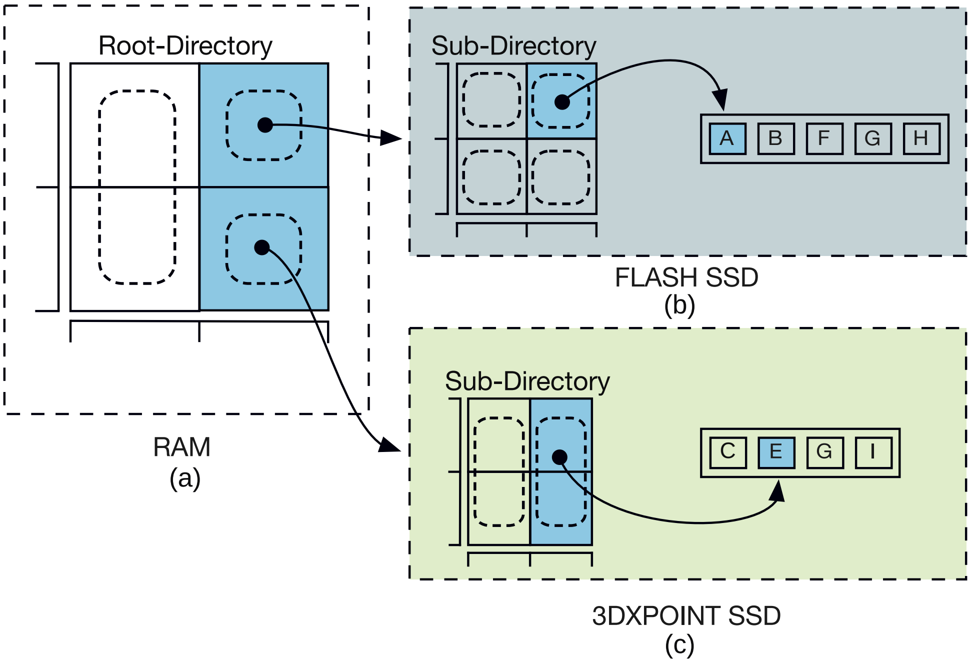


FIGURE 4.1: Overview of H-Grid. A part of the Grid-File is migrated to the 3DXpoint storage.

use of temporary tables and bufferpool cache misses as factors that degrade query performance. In the sequel, they experimentally show that an enterprise class 3DXPoint SSD can improve query performance by 1.1-6.5x compared to a flash counterpart.

Although many flash efficient database indexes have been proposed so far, there exist only a few hybrid ones. Recalling from Chapter 2, the *HybridB tree* [51] is a B+tree variant for hybrid SSD/HDD storage. It always keeps the internal nodes in the SSD, while it distributes the leaf node pages between HDD and SSD. Specifically, it organizes leaf nodes as huge-leaves aiming to reduce costly splits and merges. A huge-leaf occupies two or more pages in the secondary storage it includes a logging part and a part for metadata as well.

4.3 The H-Grid

Spatial data structures are of paramount importance for spatial query processing. They represent simple or complex spatial objects (e.g. points, lines polygons, etc) in a manner that simplifies execution of spatial queries [112]. In GFFM and LB-Grid we utilized flash SSDs to enhance the efficiency of Grid File [91]. Here, our objective is to take advantage of a new non-volatile memory technology, the 3DXPoint. Therefore, we introduce the H-Grid, a Grid File variant for hybrid storage.

4.3.1 H-Grid Design

A common method on past research for flash efficient database indexes is to group I/O operations, exploiting the high bandwidth, the internal parallelism of modern SSDs and the efficiency of NVMe protocol. This strategy provides sufficient results, especially in range and kNN queries as they usually involve access to multiple pages. Furthermore, grouping of incoming search requests into sets that are processed simultaneously have been also approved beneficial. However, in all aforementioned cases, accessing the upper level nodes of tree-indexes does not always exploit the full bandwidth of SSDs, even if multiple nodes are fetched with a single I/O request. The performance characteristics of 3DXPoint, i.e. its efficiency even at small size I/O, motivated us to introduce H-Grid. H-Grid is a Grid-File variant designed for hybrid, 3DXPoint/flash storage. It exploits a frequency based model for data placement. It detects performance critical regions placing them to the low latency 3DXPoint storage, while it leaves the rest of them to the flash SSD. To the best of our knowledge, H-Grid is the first attempt to introduce a spatial index that exploits hybrid 3DXPoint/flash I/O.

A running example of H-Grid is illustrated in Figure 4.1. The H-Grid implementation follows the two-level Grid File design as it is presented in [47]. Thus, H-Grid employs a small, memory resident Root Directory (Fig. 4.1a) and many sub-directories that reside in the physical storage. The sub-directories hold the addresses of data buckets that contain the actual data. The sub-directories and the data buckets can reside in either a flash (Fig. 4.1b), or a 3DXPoint SSD (Fig. 4.1c). A selection algorithm locates frequently accessed regions that are eligible for the 3DXPoint storage, considering weight values for each retrieved directory or data bucket page. These weights are calculated using the access frequencies of the pages. We use two hashing tables (one for directory and one for data pages) to associate each page in the 3DXPoint storage with its corresponding weight value.

H-Grid leverages in-memory buffers to accommodate pages that are either retrieved from the SSDs or temporary stored prior to a batch write operation [34, 35]. It employs separate buffers for sub-directories and data buckets, enabling different buffering polices that rely upon the page type (directory/data). At the moment, we utilize LRU as eviction policy in both buffers. The dirty evicted pages are not persisted immediately; they are accumulated into write buffers instead, enabling batch writes that accelerate performance.

We also examine a special case of H-Grid (Fig. 4.2), where all sub-directories are placed to the 3DXPoint storage, along with a number of selected data buckets. This approach can provide additional performance gain, since the sub-directories are referenced more frequently and their access pattern usually involves small-size I/O. The induced space overhead is not prohibitive since, as experimental results indicate, the size of directory pages in the physical storage is two orders of magnitude less than that of data buckets. The algorithms in the rest of the document were modified accordingly to comply with this special case.

In the sequel, we describe the Hybrid Bucket detection algorithm and the Search/Insert operations in H-Grid.

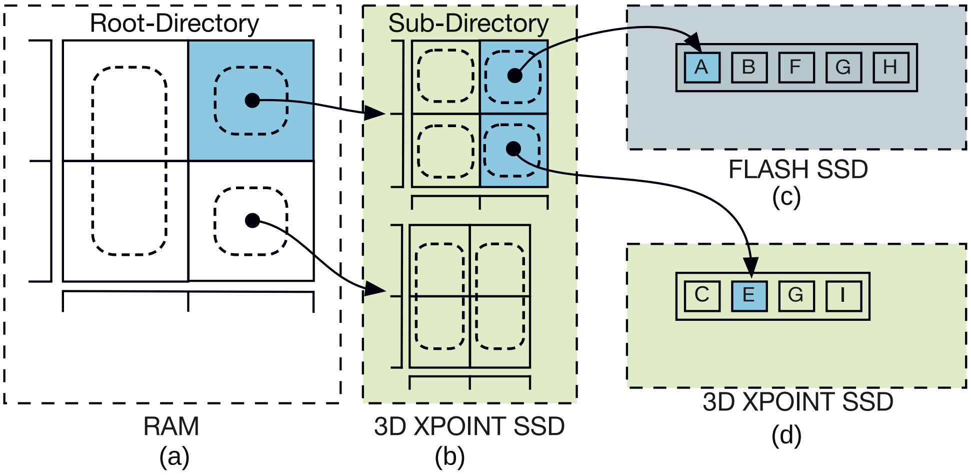


FIGURE 4.2: H-Grid special case. All sub-directories are hosted to the 3DXPoint SSD.

4.3.2 Hot Region Detection Algorithm

The role of the hybrid bucket detection algorithm is to reveal the most important, from performance viewpoint, sub-directories and data buckets. Only these will be migrated to the 3DXPoint storage. We use a temperature-based model to identify hot spatial regions that impose the highest I/O cost. These regions are represented by a number of sub-directories and data buckets. The weight of a sub-directory is highly correlated with the number of previous requests for it.

Equation 4.1 provides a metric for the weight W_i^σ of a certain sub-directory i.

$$W_i^\sigma = F_i^\sigma - \left(1 - \frac{t_i^\sigma}{T_i^\sigma}\right) \tag{4.1}$$

The first term expresses the frequency of accesses to the specific sub-directory, normalized into the range [0,1]. The second term refers to an aging policy, providing an advantage to sub-directories that were recently accessed. T_i^σ is the current timestamp, while t_i^σ is the timestamp of the previous access of sub-directory i. In other words, the second term reflects the changes occurring in the access patterns over time.

Regarding data buckets, we use a similar policy, as expressed by Eq. 4.2.

$$W_j^\beta = (W_i^\sigma + F_j^\beta) - \left(1 - \frac{t_j^\beta}{T_j^\beta}\right) \tag{4.2}$$

Specifically, we utilize the number of read requests F_j^β for the bucket j and the weight W_i^σ of its parent sub-directory i to determine its eligibility. The aging factor is also applied to decay the weight of buckets that are rarely used. An additional condition for the data buckets is the presence of their parent sub-directory in the 3DXPoint storage as well.

The selection Algorithm (Alg. 9) uses the weight values to identify the hottest buckets.

Algorithm 9: HybridBucketDetect(B,S,WS)

```
   Data: the bucket $B$, the parent sub-directory $S$, the weight of the parent sub-directory
         $WS$
   Result: Bucket is set hybrid or not
 1 $F \leftarrow getBucketStats(B.id)$;
 2 $D \leftarrow (T - B.t)/T$;
 3 $W \leftarrow WS + F - D$;
 4 $W_{SUM} \leftarrow W_{SUM} + W$;
 5 $++n$;
 6 if S not in 3DXPoint then
 7     return 0;
 8 end
 9 $CA \leftarrow W_{SUM}/n$;
10 if $W > s * CA$ then
11     $B.setHybrid \leftarrow 1$;
12     HBT[B.id]$\leftarrow W$;
13     set the 3DXPoint dirty flag of $B$;
14     return 1;
15 end
16 return 0;
```

Only these are migrated to the 3DXPoint storage. The algorithm initially calculates the weight of a bucket (lines 1-3). In the sequel, it uses the cumulative moving average (CMA) of the weights (line 7) to determine the bucket's eligibility for the 3DXPoint storage. The simple moving average (SMA) in sequential time windows can be used alternatively. Upon a hot bucket is detected, a dirty flag is set, forcing the bucket to be written on the 3DXPoint SSD during the next write-buffer flush (line 13). The parameter s is a tunable constant which controls the selectivity of the algorithm. *HBT* (Hybrid Bucket Table) is a hash table that maps all buckets in the 3DXPoint storage to their respective weights. The weight value of a bucket in the *HBT* is updated every time the bucket is retrieved. This algorithm is adapted for the sub-directories as well.

4.3.3 Queries

Single Point Search

In the two-level Grid-File, the search operation starts from the in-memory root directory by locating the sub-directory which contains a particular point. When the sub-directory is retrieved, the procedure continues at the sub-directory level, looking for the appropriate bucket. In this way, the Grid-File guaranties that a single point is reached in two disk accesses.

In H-Grid the search operation is adjusted to the hybrid storage configuration. Algorithm 10 describes the operation for a given point p at sub-directory level, while it adapts similarly at the root level. Initially, the linear scales and the grid are used to find out the address of bucket B that contains p. A fetch operation for B is issued either to flash or to the 3DXPoint

```
Algorithm 10: Search($p, S, WS$)
   Data: the search point $p$, the parent Sub-directory $S$, the weight $WS$ of the parent
         sub-directory
   Result: the bucket $B$ wherein $p$ is located
1  search the scales to convert the coordinates of $p$ into interval indexes;
2  use interval indexes to locate bucket $B$ in the sub-directory;
3  FetchBucket($B$);
4  if $HBT[B.id]$ is not $NULL$ then
5      update $HBT[B.id]$ with new weight value;
6  else
7      HybridBucketDetect($B$,$WS$);
8  end
9  update $B$ timestamp;
10 return $B$;
```

```
Algorithm 11: FetchBucket($B, HBT, MB$)
Data: the id of bucket $B$ to be read, the Hybrid Bucket Table $HBT$, in-memory buffer
      MB
Result: the bucket $B$
if $B$ is in main memory buffer $MB$ then
    move $B$ to the MRU position of main buffer;
else if $B$ is in flash SSD write buffer then
    move $B$ to the MRU position of main buffer;
else if $B$ is in 3DXPOINT SSD write buffer then
    move $B$ to the MRU position of main buffer;
else if $HBT[B.id]$ is not $NULL$ then
    read $B$ from 3DXPOINT SSD;
    move $B$ to the MRU position of main buffer;
else
    read $B$ from flash SSD;
    move $B$ to the MRU position of main buffer;
end
return $B$;
```

storage (line 3). If B already resides in the 3DXPoint, its weight value W_B^β is updated. Otherwise, Algorithm 9 is employed to decide if B is eligible for migration (lines 4-8). Finally, the last access timestamp of B is updated and B is returned.

Algorithm 11 details the bucket fetching operation in H-Grid. If the requested bucket B is already in the in-memory buffer (MB) or into one of the two write buffers (flash or 3DXPoint), then B is moved to the most recently used (MRU) position of MB. Otherwise, a fetch operation from the secondary storage is initiated. The HBT table is examined and a bucket read request is issued to the appropriate storage device. By the end of the operation, B is placed to the MRU position of the main buffer and a reference to it is returned.

From the above, it is obvious that the two disk access principle of Grid File is also preserved in H-Grid. The cost of searching a single point in H-Grid is determined by the cost of retrieving the directory and bucket pages from the physical storage.

Thus, for a given search query Q, let $x_s \in \{0,1\}$ represent whether sub-directory s is stored in the 3DXPoint storage or not, and $x_b \in \{0,1\}$ denote whether the bucket b is in 3DXPoint or not as well. The cost C_Q of Q is

$$\begin{aligned} C_Q &= x_s * R_x + R_f * (1 - x_s) + x_b * R_x + R_f * (1 - x_b) \\ &= 2 * R_f - (R_f - R_x) * (x_s + x_b) \end{aligned}$$

where R_f and R_x denote the cost of reading a page from the flash and 3DXPoint, respectively. The wider the difference in page read time between flash and 3DXPoint gets, the higher the performance gain of H-Grid becomes.

Insert Point

Algorithm 12 describes the insertion of a new point to the H-Grid. It receives as input a point p and exploits the *Search* operation to acquire the bucket B wherein p has to be inserted. If B is not full, a proper record is composed and is added to it (lines 2-14). In case B resides in the flash SSD, the hybrid bucket detect operation is invoked, testing its eligibility for migration to the 3DXPoint storage. A proper dirty flag is set denoting bucket's storage medium. This flag is exploited by the write operation. Each bucket B accommodates a certain number of records. In case B is full, a split operation of B is initiated, resulting in the introduction of a new bucket. Successive insertions of new records may cause a sub-directory split as well.

Algorithm 12: Insert(p, S, WS)

```
   Data: the new entry $p$ to be inserted, the parent sub-directory $S$, the weight of the
         parent sub-directory $WS$
 1 $B \leftarrow Search(p, S, WS)$
 2 if $B$ is not full then
 3     insert record $(p)$ to $B$;
 4     if $HBT[B.id]$ is not $NULL$ then
 5         set the 3DXPoint dirty flag of $B$;
 6     else
 7         if not HybridBucketDetect($B, S, WS$) then
 8             set the flash dirty flag of $B$;
 9         end
10     end
11     update $B$ timestamp;
12     return 1;
13 else
14     split bucket $B$;
15     Insert($p, S, WS$)
16 end
```

TABLE 4.1: SSD Characteristics

	Intel DC P3700	Optane Memory series
	(Flash)	(3DXPoint)
Seq. Read	up to 2700MB/s	up to 1350MB/s
Seq. Write	up to 1100MB/s	up to 290MB/s
Random Read	450K IOPS	240K IOPS
Random Write	75K IOPS	65K IOPS
Latency Read	120μs	7μs
Latency Write	30μs	18μs

4.3.4 Performance Evaluation

Methodology and setup

In this section we present the evaluation of H-Grid using both flash and 3DXPoint storage devices. We present the performance benefits of H-Grid against flash efficient (GFFM [34]) and traditional (R*-Tree [45]) spatial indexes that are unable to exploit diverse storages. We also test the special case of H-Grid, presented in Section 4.3.1, that persists all its sub-directories to the 3DXPoint storage.

All the experiments were performed on a workstation running CentOS Linux 7 (Kernel 4.14.12). The workstation is equipped with a quad-core Intel Xeon CPU E3-1245 v6 3.70GHz CPU, 16GB of RAM, and a SATA SSD for hosting the operating system. The experiments were conducted on an INTEL DC P3700 480GB PCI-e 3.0 SSD (FLASH) and an Intel Optane 32GB Memory Series device (3DXPoint). The latter belongs to the first generation of devices utilizing 3DXPoint memory. Table 4.1 summarizes the performance characteristics of the two devices as provided by manufacturers' data sheets.

We use two synthetic and one real dataset for the experiments. The synthetic datasets follow Gaussian and Uniform distributions, respectively, while the real one contains geographical points extracted from Openstreetmap[1]. All experiments were executed using the Direct I/O (O_DIRECT) option to bypass the Linux OS caching system. We varied the selectivity parameter s (Alg. 9) in the range 1.0 to 2.5 in the various workloads. In this way, a number of up to 40% of the sub-directories and up to 20% of data buckets migrated to 3DXPoint. We set the total size of the in-memory buffers for every examined index to 8MB. We did not manage to run R*-tree on the 3DXPoint using the real dataset due to lack of space. The single hatched part of the bars correspond to the I/O time spent in the flash SSD, while the double hatched to the I/O time in the 3DXPoint SSD.

Insert/Search Queries

We evaluated the performance of H-Grid using six different workloads for each dataset. Regarding the real dataset, the indexes were initialized with 500M points. Figure 4.3a presents

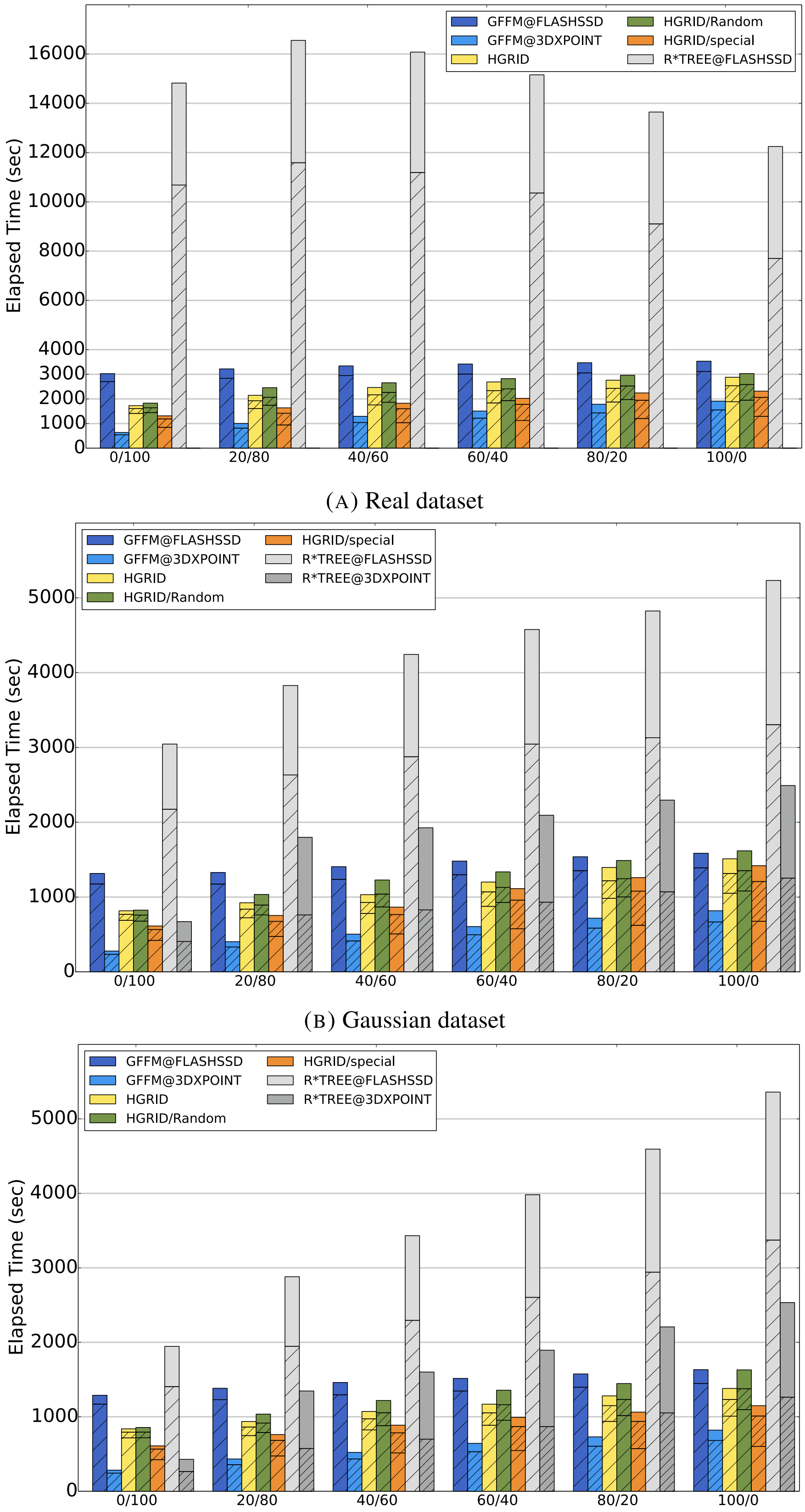
[1] http://spatialhadoop.cs.umn.edu/datasets.html

TABLE 4.2: I/S Queries - Number of I/O operations issued to the storage

	GFFM	HGRID		HGRID (Random)		HGRID (Special)	
		FLASH	3DXPOINT	FLASH	3DXPOINT	FLASH	3DXPOINT
Real Dataset							
0/100	1.98E+07	1.31E+07	6.78E+06	1.33E+07	6.66E+06	8.04E+06	1.18E+07
20/80	2.35E+07	1.58E+07	7.99E+06	1.65E+07	7.42E+06	1.03E+07	1.34E+07
40/60	2.70E+07	1.86E+07	9.18E+06	1.93E+07	8.63E+06	1.25E+07	1.47E+07
60/40	3.04E+07	2.11E+07	1.05E+07	2.19E+07	1.00E+07	1.47E+07	1.61E+07
80/20	3.37E+07	2.36E+07	1.17E+07	2.43E+07	1.16E+07	1.70E+07	1.72E+07
100/0	3.68E+07	2.56E+07	1.34E+07	2.63E+07	1.34E+07	1.94E+07	1.79E+07
Uniform Dataset							
0/100	9.04E+06	6.50E+06	2.87E+06	6.51E+06	2.85E+06	3.99E+06	5.37E+06
20/80	1.06E+07	7.59E+06	3.24E+06	7.67E+06	3.16E+06	4.99E+06	5.87E+06
40/60	1.25E+07	9.15E+06	3.72E+06	9.14E+06	3.74E+06	6.00E+06	6.95E+06
60/40	1.42E+07	1.07E+07	4.21E+06	1.06E+07	4.26E+06	7.00E+06	7.98E+06
80/20	1.59E+07	1.21E+07	4.78E+06	1.21E+07	4.70E+06	8.00E+06	8.97E+06
100/0	1.76E+07	1.35E+07	5.25E+06	1.35E+07	5.27E+06	9.00E+06	9.92E+06
Gaussian Dataset							
0/100	8.64E+06	6.21E+06	2.89E+06	6.15E+06	2.96E+06	3.95E+06	5.16E+06
20/80	1.01E+07	7.19E+06	3.12E+06	7.19E+06	3.13E+06	4.95E+06	5.43E+06
40/60	1.19E+07	8.64E+06	3.68E+06	8.76E+06	3.58E+06	5.96E+06	6.48E+06
60/40	1.36E+07	1.02E+07	4.06E+06	1.02E+07	4.10E+06	6.98E+06	7.47E+06
80/20	1.52E+07	1.17E+07	4.47E+06	1.16E+07	4.59E+06	7.99E+06	8.36E+06
100/0	1.68E+07	1.31E+07	4.90E+06	1.31E+07	4.92E+06	9.00E+06	9.16E+06

TABLE 4.3: kNN Queries - Number of I/O operations issued to the storage

	GFFM	HGRID		HGRID (Random)		HGRID (Special)	
		FLASH	3DXPOINT	FLASH	3DXPOINT	FLASH	3DXPOINT
Real Dataset							
10NN	2.36E+06	2.72E+06	6.69E+05	2.72E+06	6.63E+05	2.11E+06	1.29E+06
50NN	3.17E+06	3.41E+06	8.40E+05	3.42E+06	8.33E+05	2.77E+06	1.49E+06
100NN	3.92E+06	4.04E+06	9.94E+05	4.05E+06	9.84E+05	3.38E+06	1.67E+06
1000NN	1.28E+07	1.15E+07	2.75E+06	1.16E+07	2.72E+06	1.07E+07	3.61E+06
Uniform Dataset							
10NN	2.44E+06	2.80E+06	6.19E+05	2.80E+06	6.19E+05	2.18E+06	1.24E+06
50NN	3.37E+06	3.61E+06	7.98E+05	3.61E+06	7.98E+05	2.95E+06	1.46E+06
100NN	4.19E+06	4.32E+06	9.52E+05	4.32E+06	9.51E+05	3.61E+06	1.66E+06
1000NN	1.38E+07	1.25E+07	2.75E+06	1.25E+07	2.75E+06	1.14E+07	3.83E+06
Gaussian Dataset							
10NN	2.28E+06	2.57E+06	6.62E+05	2.62E+06	6.13E+05	2.06E+06	1.17E+06
50NN	3.09E+06	3.27E+06	8.27E+05	3.33E+06	7.73E+05	2.73E+06	1.37E+06
100NN	3.79E+06	3.87E+06	9.68E+05	3.93E+06	9.10E+05	3.30E+06	1.54E+06
1000NN	1.18E+07	1.06E+07	2.53E+06	1.07E+07	2.43E+06	9.78E+06	3.37E+06



(A) Real dataset

(B) Gaussian dataset

(C) Uniform dataset

FIGURE 4.3: Execution times of I/S queries for different workloads. H-Grid provides better results when searches are the majority.

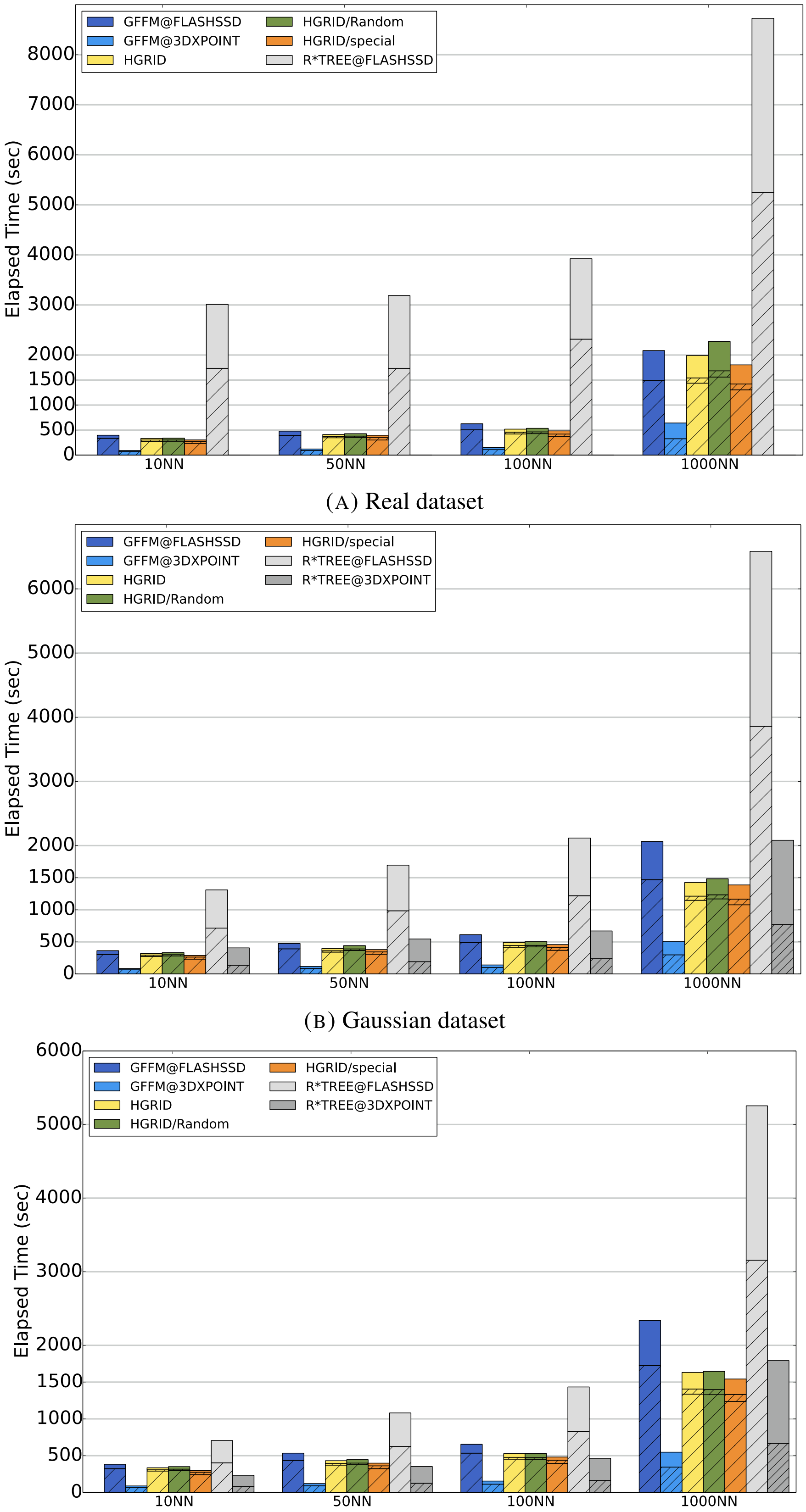
the elapsed time for 10M operations with the specified search and insert ratios. Specifically, H-Grid achieves a speedup which ranges from 18.4% to 43% in comparison to the execution of GFFM on the flash SSD (baseline). H-Grid does not provide adequate results when the buckets for the 3DXPoint are randomly selected. This fact reveals the efficiency of the proposed hot region detection algorithm. The special case of H-Grid, which considers placing all sub-directories in the 3DXPoint storage, achieves a significant performance gain that ranges from 34.4% to 56.6%. The acquired results are even better when GFFM exclusively utilizes the 3DXPoint SSD as persistent storage (best case), providing a speedup reaching 78.9% in comparison to the execution on the flash SSD.

For the synthetic dataset runs, we used 50M points for initialization and 5M I/S operations for testing. As depicted in Figures 4.3b and 4.3c, there is remarkable improvement in all experiments involving read sensitive workloads. Figure 4.3b presents the results for the Gaussian dataset. Specifically, using the 3DXPoint SSD as sole storage medium for GFFM, we achieve an improvement ranging from 49.7% to 77.9% comparing with its execution on the flash SSD. H-Grid achieves a performance gain up to 35% in comparison to the GFFM run on the flash SSD. The special case of H-Grid exhibits even better performance (29.6%-52.7%) as expected. The results are similar in the test cases that utilize the uniformly distributed dataset (Fig. 4.3c). The H-Grid is up to 35% faster than the GFFM execution on the flash SSD, while the special case improves further the result. The low-latency of 3DXPoint SSD also contributes significant performance gains for R*-Tree comparing with its flash-based execution.

In regard to the number of I/O operations issued in the secondary storage, Table 4.2 presents the respective counters for all test cases. The number of accesses in the flash SSD is almost double than the number of accesses in the 3DXPoint counterpart in all H-Grid runs; specifically the ratio $\frac{flash\ accesses}{3XPoint\ accesses}$ varies between 1.90 to 2.67. In opposition, in the special H-Grid case, the number of access in the two storage devices is almost the same (in the most cases), with the aforementioned ratio ranging from 0.68 up to 1.08.

kNN Queries

In this section we analyze the performance of kNN queries. We previously initialized the indexes, using the same datasets as in I/S queries (500M points for the real dataset, 50M points for each synthetic). We present the elapsed time for 1M queries posed to each one of the examined indexes. Figure 4.4a depicts the results of the real dataset, while Figures 4.4b and 4.4c correspond to the Gaussian and Uniform test cases. Regarding the real dataset, H-Grid provides a gain up to 17% in the 100NN case, while in the 1000NN case the gain is only 4.7%. This is due to the large number of bucket reads that imposes. The results are better in the smaller synthetic workloads. Particularly, for the Gaussian dataset, the improvement ranges from 12.3% for the 10NN query, and up to 31% for the 1000NN one. Similarly, in the experiments with the Uniform dataset, a speedup ranging from 12.4% up to 30% is achieved.



(A) Real dataset

(B) Gaussian dataset

(C) Uniform dataset

FIGURE 4.4: Execution times of kNN queries for different workloads.

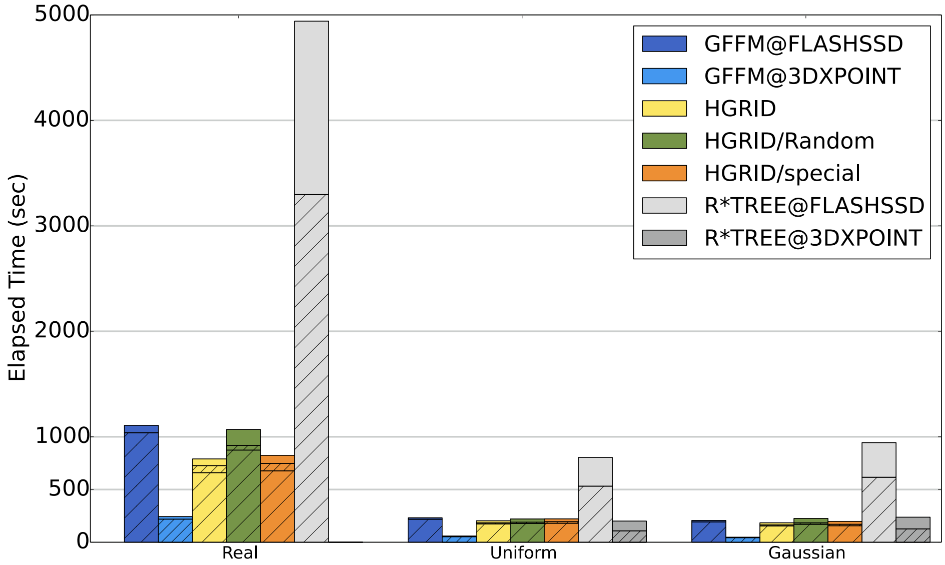


FIGURE 4.5: Execution times of range queries for three different datasets.

H-Grid achieves better results when all the sub-directories reside in the 3DXPoint (special case). Specifically, for the real dataset, it improves its execution time starting in a range from 13.6% up to 23.5%. The number of accesses in the secondary storage (Table 4.3) enlightens the acquired results. Specifically, H-Grid issues, in average, 4.21 times more I/O requests to the flash than to the 3DXPoint device. On the other hand, in the special H-Grid case this number lowers to 2.18, thus the better results are explained.

Range Queries

We discuss the performance of range queries next. Specifically, we present the elapsed times of 5K requests issued to the previously initialized indexes. Figure 4.5 summarizes the execution times for all test cases, while Table 4.4 depicts the access counters in secondary storage. H-Grid improves GFFM on flash SSD (baseline case) up to 28% in the real dataset run, while the gain for H-Grid is smaller in the runs that use the synthetic data. The efficiency of the proposed hot region detection algorithm is proven to be true once again, since the random

TABLE 4.4: Range Queries - Number of I/O operations issued to the storage

GFFM	**HGRID**		**HGRID (Random)**		**HGRID (Special)**	
	FLASH	3DXPOINT	FLASH	3DXPOINT	FLASH	3DXPOINT
Real Dataset						
8.87E+06	6.36E+06	2.56E+06	7.67E+06	1.25E+06	6.44E+06	2.50E+06
Uniform Dataset						
1.78E+06	1.46E+06	3.32E+05	1.46E+06	3.31E+05	1.40E+06	3.98E+05
Gaussian Dataset						
1.53E+06	1.25E+06	2.96E+05	1.25E+06	2.95E+05	1.21E+06	3.30E+05

selection of buckets for migration leads to worse results. The sole execution of GFFM in the 3DXPoint SSD provides significant performance improvements which range from 74.5% to 78%. Remarkable is also the speedup for the R*-tree (75%), when it utilizes the 3DXPoint SSD, for the Gaussian and Uniform workloads.

4.4 R-tree for Hybrid Storage

Examining the attained results from H-Grid, we can infer that tree indexes can also benefit significantly by storing hot nodes to the low-latency 3DXPoint. This motivated us to study the performance characteristics of R-tree in hybrid non-volatile storage.

4.4.1 R-tree

R-tree was introduced by Guttman [44] in 1984, with the aim to facilitate VLSI design. However, very soon it became a popular data access method in both industry and academia with a wide range of applications. Geographical information systems and multimedia databases are considered being among them [112]. A large number of R-tree variants have been proposed since its introduction, pursuing to improve its efficiency or to modify it for different applications [81].

R-tree is a dynamic hierarchical data structure akin to that of B+tree. It uses multidimensional minimum bounding rectangles (MBRs) to organize spatial objects. Thus, each leaf node entry stores the smallest MBR that encloses a single geometric object O, and a pointer to the address of the particular object rather than the object itself, i.e. $(MBR_O, \vec{O})$. Similarly, each internal node entry contains an MBR that encloses all MBRs of its descendants, and a pointer to their sub-tree T, namely $(MBR_T, \vec{T})$. If the R-tree resides in secondary storage, its nodes correspond to disk pages. Each R-tree node (not the root) can store at least m and at most M entries, with $m \leq M/2$. The root can store two records at minimum, unless it is a leaf; 0 or 1 entries are allowed in such a case.

MBRs from different nodes may overlap. Even if an object is enclosed by many MBRs, it is always associated with only one of them. Therefore, the search procedure for a spatial object O starts from the root and traverses the tree towards the leaves; however, it might follow several paths in order to ascertain the existence (or not) of O. This results in the worst case of retrieving a small number of objects to a cost that is linear to the size of the data.

The height of an R-tree determines the least number of pages that must be retrieved in order to touch a spatial object. Assuming that an R-tree accommodates N rectangles, then its maximum height is

$$H_{max} = \log_m N - 1 \tag{4.3}$$

Figure 4.6 illustrates a group of MBRs in the plane and one possible R-tree ($m = 2, M = 4$). Rectangles k, l, m, n, o, p, q, r, s, t, u and v enclose the spatial objects (not depicted

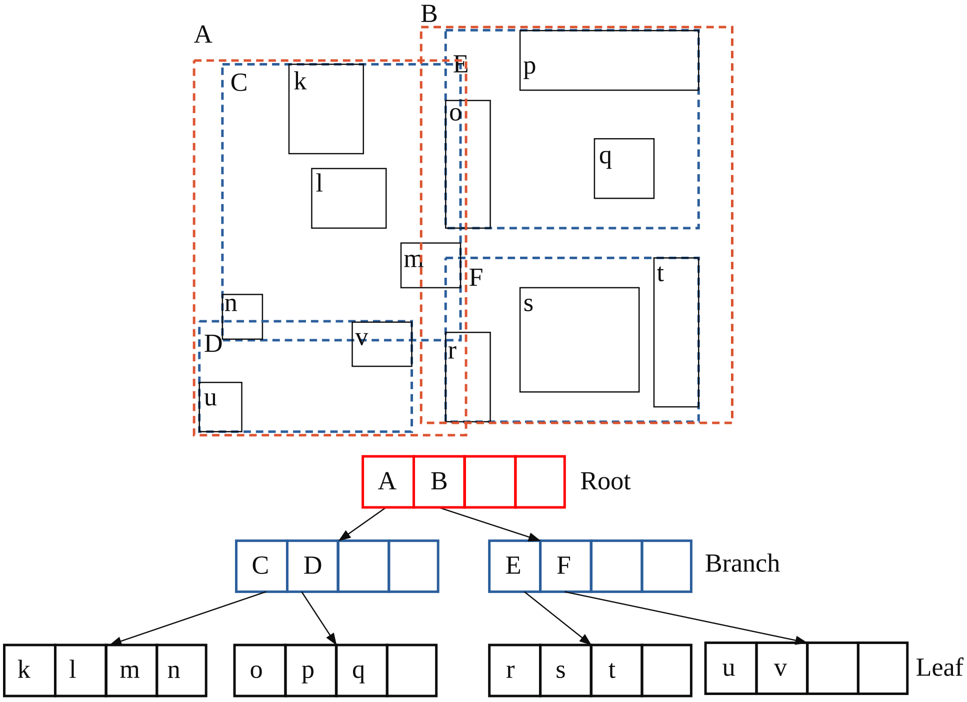


FIGURE 4.6: A set of rectangles $\{k,l,m,n,o,p,q,r,s,t,u,v\}$ and the corresponding R-tree (m=2, M=4)

here), forming the leaves of the tree. Similarly, C, D, E, F, organize the leaves forming R-tree's internal nodes, while A, B designate the root. It must be noted that alternative R-trees may be constructed indexing the same spatial objects. The structure of the resulting R-tree depends, to a large extent, on the order of the insert and/or delete operations issued to it.

Several variants of R-tree have been proposed in order to improve its efficiency; some representative examples are R$^+$-tree [116], R*-tree [9], Hilbert R-tree [56], Cubetree [111], Historical R-tree [120] and LR-tree [16].

4.4.2 Design an R-tree for Hybrid Storage

Previous works for flash efficient data access methods target to reduce the number of random writes and exploit the high throughput and internal parallelism of SSDs. The performance characteristics of 3DXPoint SSDs (i.e. low latency and high IOPS) and their cost, which remains significant higher than that of the flash-based ones, motivated us to study indexing methods under a different perspective. Therefore, we believe that a hybrid storage configuration that combines both NVM technologies (flash and 3DXPoint) can be a decent option. Next, we describe some key features that a hybrid tree index should have and we provide some implementation details.

A running example of a hybrid tree T is illustrated in Figure 4.7. A part of the tree is stored on the 3DXPoint SSD (Fig. 4.7(a)) while the rest of it is left on the flash based one (Fig. 4.7(b)). Such a design is based on two fundamental operations: i) a selection algorithm that detects hot data regions and ii) a replacement policy that retains only the hottest data to

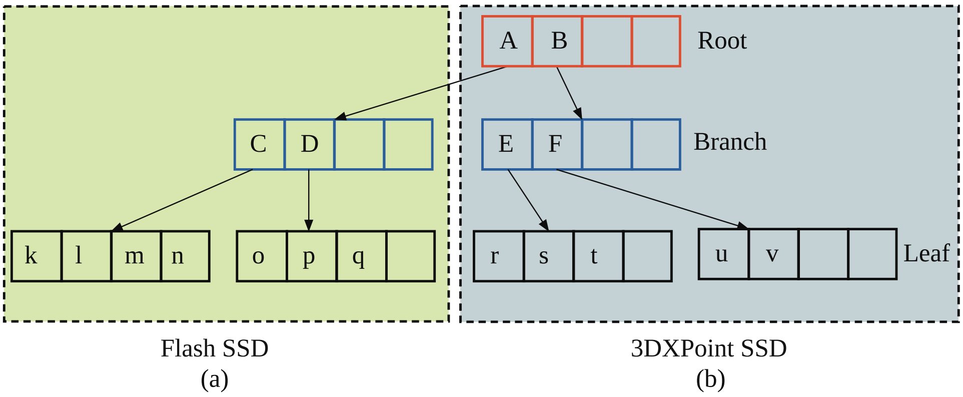


FIGURE 4.7: Indicative example of a hybrid R-tree; a part of the tree is stored to the 3DXPoint storage.

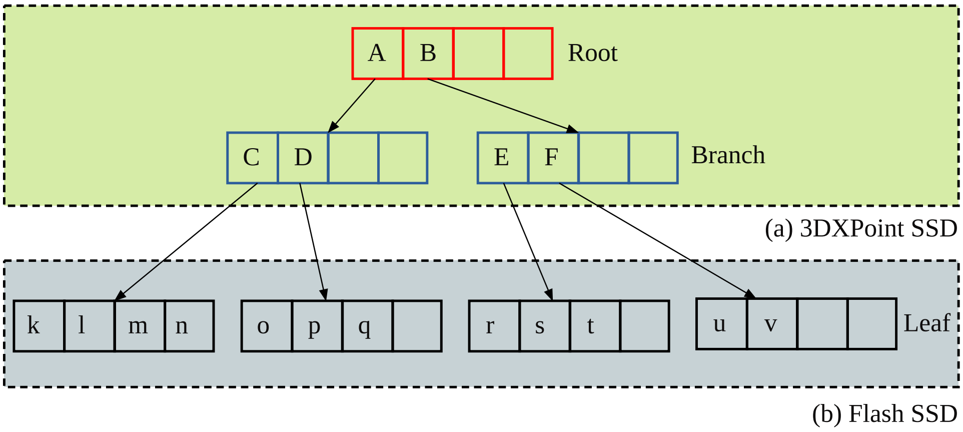


FIGURE 4.8: A simple yet illuminating case of a hybrid R-tree index (sHR-tree); all non-leaf nodes are stored to the 3DXPoint storage.

the fast storage over time. The hot data selection algorithm can be based on usage statistics, combined with other factors, such as the spatial properties of the indexed objects. In more detail, the frequency of accesses in a specific region R as well as the recency of these accesses can provide a strong indication of its popularity. Let R be a sub-tree of T. As the available space in the performance tier (3DXPoint) is reduced, an adequate reclaim policy is needed to migrate cold regions to the storage tier (flash).

We present some implementation guidelines bellow. As we mentioned above, the selection of the hottest regions is performed using various criteria. As a result, a weight value is calculated for each region. The weights are maintained in-memory, using hash tables that guarantee fast access. Also, an in-memory buffer is exploited to keep recently accessed node pages. Here we consider LRU as replacement policy, however alternative algorithms can also be applied. Each tree node of can reside in either the main memory or the flash, or the 3DXPoint storage. Algorithm 13 describes node retrieval. If the requested node N is already in the in-memory buffer (MB), it is moved to the most recently used (MRU) position of MB. Otherwise, a fetch operation from the secondary storage is initiated. By the end of the operation, N is placed to the MRU position of the main buffer and a reference to it is returned.

Algorithm 13: RetrieveNode(N,MB)

```
Data: $N$ the node to be retrieved, the in-memory buffer MB
Result: the node $N$
if $N$ is in main memory buffer $MB$ then
    move $N$ to the MRU position of main buffer;
else if $N$ is in 3DXPOINT then
    read $N$ from 3DXPOINT SSD;
    move $N$ to the MRU position of main buffer;
else
    read $N$ from Flash SSD;
    move $N$ to the MRU position of main buffer;
end
return $N$
```

Let $x_N \in \{0,1\}$ denote whether N is stored in the 3DXPoint storage or not. Assuming that N occupies a single page in the secondary storage, the cost of retrieving N is

$$C_N = x_N * R_x + R_f * (1 - x_N) = R_f - x_N * (R_f - R_x) \tag{4.4}$$

where R_f and R_x are the reading costs from the flash and 3DXPoint, respectively.

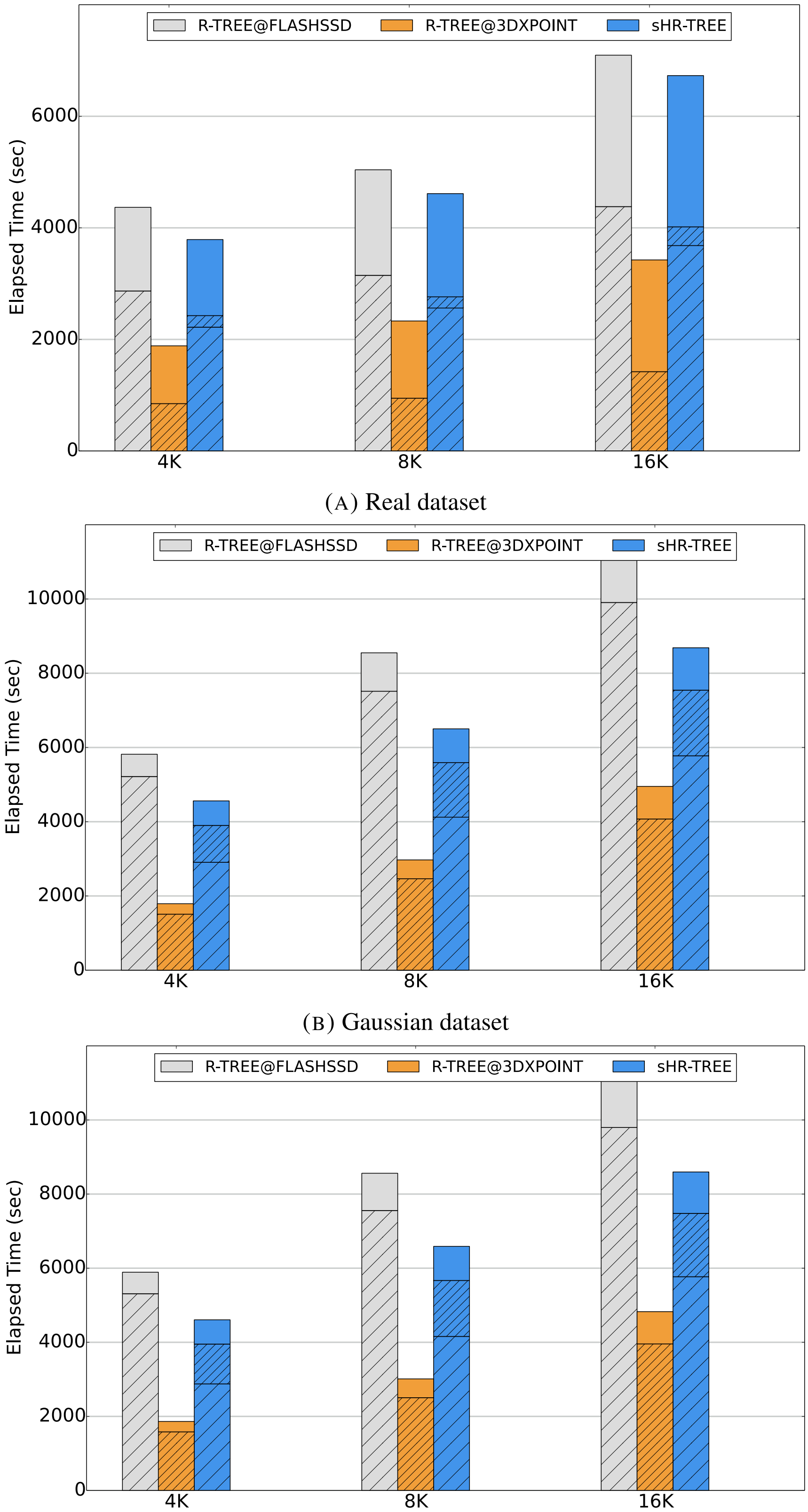
We also study a simple yet illuminating case of a hybrid R-tree (Fig. 4.8), where all non-leaf nodes are stored to the 3DXPoint SSD. We refer to it as sHR-tree from now on. The upper level nodes of an R-tree are referenced more often than the leaves and present a small-size I/O access pattern, as the tree is traversed from the root to the leaves. Therefore, the efficiency of 3DXPoint at small queue depths can lead to considerable performance improvement. Using sHR-Tree we can draw useful conclusions about R-tree performance, when a hybrid storage scheme is applied.

4.4.3 Evaluation

Methodology and setup

In this section we discuss the performance evaluation of R-tree in the various storage configurations. We conducted a series of experiments using both flash and 3DXPoint storage devices. We aim at unfolding the benefits of hybrid index configurations, against an approach that considers single storage medium. For this reason, we evaluate two different workloads, concerning i) index construction, and ii) execution of 5000 range queries.

The experimental platform is described in paragraph 4.3.4. We utilized three datasets in the experiments, one real-world containing 300M points and two synthetic, having Gaussian and Uniform distributions, of 50M points each. The total size of the in-memory buffers was configured to 4MB. We did not employ a special write buffer in the experiments. The single hatched part of the bars correspond to the I/O time spent in the flash SSD, while the double hatched to the I/O time in the 3DXPoint SSD.

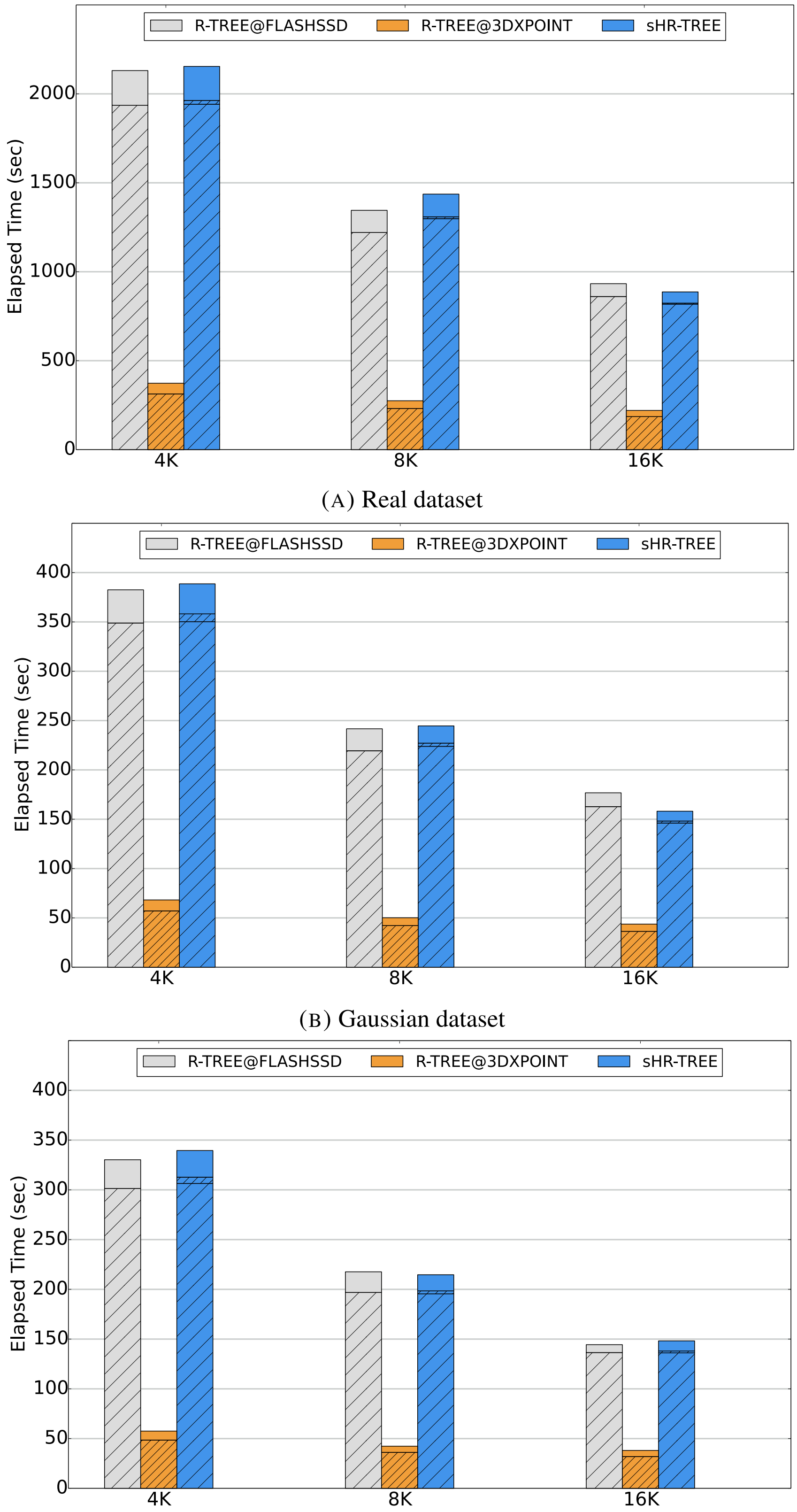


(A) Real dataset

(B) Gaussian dataset

(C) Uniform dataset

FIGURE 4.9: Execution times of index construction for different workloads. A gain up to of 13% for the real dataset and up to 24% for the synthetics is achieved by the sHR-tree.



(A) Real dataset

(B) Gaussian dataset

(C) Uniform dataset

FIGURE 4.10: Execution times of range queries for different workloads. R-tree@3DXPoint achieves an improvement up to 82% in comparison to the R-tree@Flash.

TABLE 4.5: Index Construction - Number of I/O operations

	R-tree	sHR-tree	
		FLASH	3DXPOINT
Real Dataset			
4K	3.53E+07	2.96E+07	5.71E+06
8K	2.76E+07	2.39E+07	3.74E+06
16K	2.56E+07	2.23E+07	3.21E+06
Uniform Dataset			
4K	6.38E+07	3.89E+07	2.48E+07
8K	6.02E+07	3.90E+07	2.12E+07
16K	5.47E+07	3.90E+07	1.58E+07
Gaussian Dataset			
4K	6.14E+07	3.86E+07	2.27E+07
8K	5.95E+07	3.89E+07	2.05E+07
16K	5.51E+07	3.90E+07	1.61E+07

TABLE 4.6: Range Queries - Number of I/O operations

	R-tree	sHR-tree	
		FLASH	3DXPOINT
Real Dataset			
4K	2.10E+07	2.04E+07	6.18E+05
8K	1.05E+07	1.03E+07	2.30E+05
16K	5.27E+06	5.18E+06	8.94E+04
Uniform Dataset			
4K	3.22E+06	3.01E+06	2.08E+05
8K	1.69E+06	1.61E+06	8.11E+04
16K	9.35E+05	8.96E+05	3.86E+04
Gaussian Dataset			
4K	3.71E+06	3.45E+06	2.59E+05
8K	1.89E+06	1.80E+06	8.93E+04
16K	1.02E+06	9.75E+05	4.28E+04

Index Construction

Regarding index construction, we examine three different test cases: i) construction of R-tree on flash, ii) construction of R-tree on 3DXPoint and iii) construction of the sHR-tree, described in Section 4.4.2, that uses both storages. We present the execution times for three different page sizes (4KB, 8KB and 16KB) in Figure 4.9. The results are quite impressive for the run on the 3DXPoint SSD. It improves the execution time, compared to the run on the flash SSD, up to 57% for the real dataset and up to 69% and 68% for the Gaussian and Uniform datasets, respectively. The improvement is significant for the sHR-tree as well; it achieves a gain up to 13% for the real dataset and up to 24% for the synthetic ones compared to the flash run. The index construction time is less in the real dataset case, despite the fact that its size is quite bigger than the size of the synthetic ones. This occurs because the objects in the real dataset exhibit spatial locality, which contributes to a high number of cache hits. We also observe that the page size influences performance. Specifically, when 4KB pages are used the results are better in all test cases. This is expected up to a certain point, since the page size determines the size of data written each time. The Table 4.5 depicts the numbers of I/O operations issued in the secondary storage for the two indexes and the three datasets. The ratio of the number of accesses in the flash SSD to the number of accesses in the 3DXPoint SSD ($\frac{flash\ accesses}{3XPoint\ accesses}$) ranges from 5.2 to 6.9 in the real dataset runs and from 1.6 to 2.5 in the synthetic ones.

Range Queries

In this section we discuss the performance of range queries. The obtained results for the R-tree residing in the 3DXPoint SSD are quite impressive; we get an improvement in comparison to the flash SSD execution up to 82%. On the other hand, sHR-tree improves only in two cases w.r.t. the real and Gaussian datasets (16KB page runs). This happens because the amount of data on the fast 3DXPoint SSD is not enough to contribute substantial performance gain. In fact, the ratio ($\frac{flash\ accesses}{3XPoint\ accesses}$) is an order of magnitude higher than in the previous experiment. Therefore, a hybrid approach that persists not only the upper-level nodes in 3DXPoint storage, but identifies and stores the hottest regions (including the leaves) on it, can significantly contribute to query performance. This argument is strengthened by results concerning the 3DXPoint execution. Moreover, the performance is improved in all test cases by increasing the page size. This happens since larger pages require less I/O operations to fetch the requested objects, which is in-line with the I/O counters that are presented in the Table 4.6.

4.5 Conclusions

In this chapter we highlighted the opportunities that new or upcoming non-volatile memory technologies create for data indexing. We studied the performance of spatial indexes (GFFM and R-tree) that utilize 3DXPoint NVM as secondary storage; we introduced the H-Grid, a variant of Grid File for hybrid storage; and we issued detailed guidelines towards developing hybrid tree indexes.

H-Grid detects hot regions and persists them in a low-latency 3DXPoint SSD, while it stores the rest of them to a flash counterpart. The experimental results show significant performance improvement for H-Grid in comparison to GFFM, a flash-based Grid File variant. Specifically, the gain ranges from 35% up to 43% in the single point retrieval, while the achieved speedup for range and kNN queries is up to 28% and 32%, respectively. The results unveil that even small amounts of 3DXPoint in the secondary storage layer can accelerate spatial queries performance at affordable cost (e.g. a 32GB Optane module costs under 100 USD).

Regarding R-tree we experimentally evaluated three different cases, namely: i) R-tree on flash, ii) R-tree on 3DXPoint, and iii) a simple hybrid R-tree implementation (sHR-tree). The experimental results support our design hypothesis. Specifically, the R-tree execution exclusively on the 3DXPoint device improves index construction up to 69%. Similarly, the hybrid R-tree improves up to 24%. Regarding range queries, a gain up to 82% is achieved when 3DXPoint is the sole storage. However, the gain is marginal for the hybrid approach, since only a small number of nodes reside in the fast storage.

Our plans for future work in H-Grid include a method for tuning the selectivity parameter s based on workload's characteristics and a cooling process for buckets that stay long time in the 3DXPoint storage without being accessed. Our plans w.r.t. R-tree, include a hot region detection algorithm that locates and migrates regions of high interest to a high performing 3DXPoint based device. Finally, we aim to further investigate spatial query processing in order to take the full advantage of 3DXPoint properties.

5 | Future Research Challenges

5.1 Indexing and New SSD Technologies

Nowadays, there is an increasing demand for high performance storage services. Emerging technologies like NVMe, 3DXPoint and other non-volatile memories are for storage such a big advent as many-cores were for CPUs. However, the difficulty of getting the maximum performance benefits out of these contemporary devices, results in wasting valuable resources [89]. The software I/O stack imposes significant overhead to data access, rendering new programming frameworks imperative. Moreover, host software is unaware of SSD internals, since they are well hidden behind a block device interface. This leads to unpredictable latencies and waste of resources. On the other hand, SSD controllers comprise CPUs and DRAM, which are closer to the data than the host CPU itself. These facts have recently created new lines of research in the area of data management. The first results are promising, disclosing new challenges to be dealt with, as well as the weaknesses of the current technology.

5.1.1 Fast NVMe Devices and Programming Frameworks

As SSDs are becoming increasingly faster, software turns into a bottleneck. The I/O stack was designed on the assumption that the CPU can smoothly process all data from many I/O devices, a fact that does not longer hold.

Up to now, Linux AIO has been successfully utilized to accelerate the performance of one- and multi- dimensional indexes [100, 35] exploiting the high bandwidth and the internal parallelization of SSDs. However, the advances in non-volatile memories (e.g. 3DXPoint, Z-NAND) enabled a new class of storage devices that provide high IOPS in small queue depths and ultra low latency – $7\mu s$ for 3DXPoint, $12\mu s$ for Z-NAND, $>70\mu s$ for commodity NAND SSDs.

The authors in [63] categorize this new device family as *Fast NVMe Devices* (FNDs). According to them, AIO is not adequate to exploit the full performance benefit of FNDs. Therefore, new programming frameworks are needed to enable user programs to directly access storage. The Storage Performance Development Kit (SPDK) is such a framework [138]. It provides a user-space driver that eliminates redundant data copies inside operating system's I/O stack and facilitates high parallel access to NVMe SSDs. Thus, it achieves 6 to 10 times better CPU utilization compared to the NVMe kernel space driver [138]. Recently, SPDK has been successfully used to enhance the performance of a key-value store [63]. Another similar framework for user-space I/O is NVMeDirect [59], aiming to avoid the overhead of kernel I/O by exploiting the standard NVMe interface. Its performance is comparable with that of SPDK.

Summarizing the above, we believe that FNDs and new programming models can be a point of departure for future research in data indexing. Specifically, most works so far focus on exploiting the high bandwidth and internal parallelization of SSDs, or to alleviate the difference between read and write speeds. To achieve these goals, they usually group I/O operations issuing them into batches. However, the performance characteristics of FNDs render, in some cases, these strategies obsolete, providing a stepping stone for new research. From a different point of view, FNDs can also be exploited in hybrid (FND/SSD) storage configurations.

5.1.2 Open-channel Architecture

The increasing adoption of SSDs in the enterprise data centers has introduced demands for high resource utilization and predictable latency [11, 14, 57, 125]. Although NAND flash solid state drives provide high performance surpassing their predecessors, the spinning disks, they exhibit unpredictable latencies. This shortcoming originates from the way raw NAND flash is managed by FTL. Internal operations like garbage collection may charge a certain workload with extra latency. Similar delays are also introduced by I/O collisions on flash chips, since writes are slower than reads. These issues are aggravated as the capacities of SSDs are becoming larger and many different applications submit I/O requests to the same device. Furthermore, today's SSDs have been designed as general purpose devices, which is sub-optimal for certain applications. Specifically, some recent works [118, 125] have shown that flash-based key-values stores under-utilize or even misuse standard NVMe SSDs. The complete isolation of SSDs' internal from the host applications leads to inefficiencies like redundant mapping, double garbage collection and superfluous over-provisioning [118]. Therefore, a new class of SSDs, referred to as open-channel (OC) SSDs, is anticipated to overcome these limitations. OC SSDs expose their resources directly to the host, enabling applications to control the placement of data.

A first considerable effort to develop OC SSDs has been made by Baidu[1] aiming to accelerate the performance of a modified Level-DB key-value store [125]. So, 3000 devices have been deployed, each one incorporating 44 channels accessible as independent block devices. DIDACache [118] is another OC SSD prototype for key-value stores. It is accompanied by a programming library which gives access to drive's data. The authors demonstrated a key-value caching mechanism based on Twitter's Fatcache. A more generic cross-vendor implementation for OC SSDs was proposed in [11, 41]. It comprises the minimal FTL firmware code, running on the SSD controller, and the LightNVM kernel subsystem in the host. Minimal FTL enables access to SSD resources, whilst the host subsystem controls data placement, I/O scheduling and garbage collection.

All studies until now handle SSDs as black boxes, relying on assumptions about their performance. OC technology enables the development of new, more efficient data structures that have full control of internal parallelization, data placement and garbage collection. Hence, OC architecture can also be the staring point for new, simple yet powerful computational models. However, the required hardware platforms are rare and of considerable cost. Fortunately, an OC SSD simulator has been introduced recently [70], providing a great opportunity for researchers that seek to exploit OC SSDs in data indexing.

5.1.3 In-Storage Processing

In-storage processing [60], near-data processing [7, 42], in-storage computing [55, 122, 123] and active SSDs [30, 68] are alternative terms used to describe recent research efforts to move computation closer to the data, inside the storage devices.

Accelerating query performance involves reducing the overhead of moving data from persistent storage to main memory [29]. An intuitive way to achieve this is to aggregate or filter the data locally, inside the SSD. As mentioned, modern SSDs incorporate embedded processors (e.g. ARM) to execute FTL. Moreover, their internal bandwidth is much higher than that of host interface. Thus, the SSD controller is located close to data and can access it really fast. Local processing inside the SSD improves performance and energy consumption since the transfer of high volumes of data is avoided [123].

An external sorting algorithm, implemented on the Openssd[2] platform, is demonstrated in [68]. The host CPU is used to perform partial sorts, which are stored on the SSD, whereas the embedded CPU assumes to merge the final result. The SSDs' computing capabilities are used to accelerate the performance of search engines in [122, 123]. The authors seek to determine which search engine operations can be offloaded to SSD for execution. Particularly, they study list intersection, ranked intersection, ranked union, difference and ranked difference operations using Apache Lucene[3] as testbed.

[1] Baidu Inc. is the largest Internet search engine in China.

[2] http://www.openssd.io

[3] https://lucene.apache.org/

A generic programming framework for developing near-data processing applications is presented in [42]. It is built around a commercial enterprise SSD by Samsung. The platform achieved significant performance gains during queries' evaluation in MariaDB[4]. In a bit different direction, the processing capabilities of flash memory controllers (FMC) are examined in [30, 60]. A new architecture, based on stream processors placed inside each flash memory controller, is studied. The proposed system succeeded to accelerate the performance of database scans and joins in simulated experiments.

In-storage processing is a quite interesting research field. It has been successfully utilized to enhance the performance of databases queries. Data indexing may gain significant benefits from offloading certain operations (e.g. scans, sorts) to the SSD, avoiding exhaustive data transfers. This requires access to special hardware prototype platforms. To the best of our knowledge, only one public available platform exists (Openssd). The rest of the examined prototypes in the literature come from SSD manufacturers and they are not widely accessible.

5.1.4 NVM as Main Memory

These days Optane DC memory, the first product based on NVM, is becoming widely available to the market. It was developed with 3DXPoint memory and is packed in DIMM modules like DRAM. It can be alternatively configured as either volatile memory, extending the capacity of DRAM, or persistent main memory. NVMs bring a new era in computing, providing high capacities and extremely low latency. However, the integration of NVMs into current computer systems introduces challenges that have to be addressed.

[133] proposes different methods to deploy the advantages of NVMs. Briefly, NVMs can be used either as secondary storage attached to DRAM bus, or as persistent main memory. A straightforward method to use NVMs as storage devices is through a file system. Traditional files systems, designed mainly for spinning disks, are unsuitable for NVMs. For this reason, new NVM-aware file systems [132] have been introduced, or the old ones have been properly modified (e.g. ext4-DAX). Using NVMs as a storage medium does not take full advantage of them. However, exploiting them as persistent main memory requires redesigning of all well-know data structures to keep data consistency in a system crash; recovering is not an easy procedure, since contemporary CPUs reorder commands to improve performance. NV-Tree is a representative example of a high efficient B+tree for NVMs [137]. The authors in [133] describe an interesting alternative to port legacy applications to NMVs, employing the Direct Access (DAX) mechanism.

Another aspect is the design of comprehensive NVM cost models. Recent research efforts [12, 49, 43] study the lower bounds of fundamental problems, such as sorting, graph traversal, sparse matrix operations, etc., taking into account the asymmetric reading and writing cost of NVMs. NVMs hosted in the memory bus are going to revolutionise computing, imposing

[4] https://mariadb.org/

new challenges, different from those the SSDs present. A considerable amount of studies already exists; however, much more needs to be done to exploit their full potential.

5.2 Conclusions

During the last years, the market share of flash-based secondary storage devices has increased at very high rates. This can be attributed to the appealing properties of the flash technology: high throughput, low access latencies, shock resistance, small and low power consumption to name a few. However, a number of medium peculiarities, like erase-before-write, asymmetric read/write latencies and wear-out, prevent its use as direct substitute (either blocked or “raw”) of the magnetic disk. Actually, the lack of a realistic flash model, like, e.g. the very successful I/O model of HDDs, greatly complicates the design and analysis of efficient flash-aware indexes and algorithms. Recent research has shown that large batches of parallel I/Os must be utilized to ensure that SSDs internal parallelization and NVMe protocol are fully exploited. With this type of I/O, there is adequate workload supply in all parallel levels of the device. Moreover, the small random write operations, that may cause frequent garbage collection operations, are eliminated; and the interference between reads and writes is restrained.

We saw that recent advances in programming frameworks, devices’ architecture (e.g. Open-Channel SSDs), arisen computing paradigms and upcoming NVMs, create new lines of research in the design and deployment of efficient index structures. In our point of view open-channel architecture is maybe the most challenging advancement, since it provides access to SSDs’ internals, ensuring predictable latency to data retrieval tasks. Additionally, the new programming frameworks can not be ignored in the implementation of efficient indexes for the next generation SSDs. On the other hand, in-storage processing is a interesting research topic, however, it requires access to specialized equipment. Finally, the introduction of NVMs in the memory hierarchy is going to revolutionize data access imposing new questions to research.

A | Publications

This dissertation presents contributions described in the following publications.

Journal Publications

1. Fevgas, A., and Bozanis, P. LB-Grid: An SSD Efficient Grid File. Data & Knowledge Engineering, vol. 121, pp. 18-41, 2019.
2. Fevgas, A., Akritidis, L., Bozanis, P., and Manolopoulos, Y. , Indexing in Flash Storage Devices: A Survey on Challenges, Current Approaches, and Future Trends, VLDB Journal (under review).

Conference Publications

1. Fevgas, A., Akritidis, L., Alamaniotis, M., Tsompanopoulou, P., AND Bozanis, P. A Study of R-Tree Performance in Hybrid Flash/3DXPoint Storage. 10th International Conference on Information, Intelligence, Systems and Applications (IISA2019), Rion-Patra, Greece, July 15-17, 2019.
2. Fevgas, A., and Bozanis, P. A Spatial Index for Hybrid Storage. In Proceedings of the 23rd International Database Engineering & Applications Symposium (IDEAS 2019), Athens, Greece, 10-12 June, 2019.
3. Fevgas, A., and Bozanis, P. Grid-File: Towards to a Flash Efficient Multi-Dimensional Index. In Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA 2015), Valencia, Spain, September 1-4, 2015, Proceedings, Part II, pp. 285-294, 2015.

The author has also contribution to the following publications that are closely related to the subject of this dissertation.

1. Roumelis, G., Fevgas, A., Vassilakopoulos, M., Corral, A., Bozanis, P., and Manolopoulos, Y. Bulk-loading and Bulk-Insertion Algorithms for xBR+-trees in Solid State Drives, Computing Journal, Springer, 2019.

2. Roumelis G, Vassilakopoulos M, Corral A, Fevgas A., and Manolopoulos Y. Spatial Batch-Queries Processing Using xBR$^{+}-$trees in Solid-State Drives. International Conference on Model and Data Engineering 2018 Oct 24 (pp. 301-317). Springer, 2018.

The author has also contribution to the following publications that study the impact of flash based storage to various scientific applications.

1. Akritidis, L., Fevgas, A., Tsompanopoulou, P., and Bozanis P. Investigating the Efficiency of Machine Learning Algorithms on MapReduce Clusters with SSDs. In Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2018) , 5-7 November, 2018, Volos, Greece, 2018.

2. Fevgas, A., Daloukas, K., Tsompanopoulou, P., and Bozanis, P. A Study of Sparse Matrix Methods on New Hardware: Advances and Challenges. IJMSTR 3(3): 1-19, 2015.

3. Fevgas, A., Daloukas, K., Tsompanopoulou, P., and Bozanis, P. Efficient solution of large sparse linear systems in modern hardware. In Proceedings of 6th International Conference on Information, Intelligence, Systems and Applications, (IISA 2015), Corfu, Greece, July 6-8, 2015, pp. 1-6, 2015.

4. Fevgas A., Tsompanopoulou P., and Bozanis P., "Exploring the Performance of Out-of-Core Linear Algebra Algorithms in Flash based Storage", 6th International Conference on Numerical Analysis (NumAn 2014), 02-05 Sep 2014, Chania, Crete, Greece.

Bibliography

[1] Alok Aggarwal, Jeffrey Vitter, et al. "The input/output complexity of sorting and related problems". In: *Communications of the ACM* 31.9 (1988), pp. 1116–1127.

[2] Devesh Agrawal et al. "Lazy-adaptive tree: An optimized index structure for flash devices". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 361–372.

[3] Nitin Agrawal et al. "Design tradeoffs for SSD performance". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. Boston, MA, 2008, pp. 57–70.

[4] Deepak Ajwani et al. "Characterizing the performance of flash memory storage devices and its impact on algorithm design". In: *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA)*. Provincetown, MA, 2008, pp. 208–219.

[5] Deepak Ajwani et al. "On computational models for flash memory devices". In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA)*. Dortmund, Germany, 2009, pp. 16–27.

[6] Manos Athanassoulis and Anastasia Ailamaki. "BF-tree: Approximate tree indexing". In: *Proceedings of the VLDB Endowment* 7.14 (2014), pp. 1881–1892.

[7] Antonio Barbalace et al. "It's Time to Think About an Operating System for Near Data Processing Architectures". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*. Whistler, Canada, 2017, pp. 56–61.

[8] R. Bayer and E. M. McCreight. "Organization and maintenance of large ordered indexes". In: *Acta Informatica* 1.3 (1972), pp. 173–189.

[9] Norbert Beckmann et al. "The R*-tree: an efficient and robust access method for points and rectangles". In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. Atlantic City, NJ, 1990, pp. 322–331.

[10] Jon Louis Bentley and James B. Saxe. "Decomposable searching problems, I. Static-to-dynamic transformation". In: *Journal of Algorithms* 1.4 (1980), pp. 301–358.

[11] Matias Bjørling, Javier González, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD subsystem". In: *Proceedings of the 15th USENIX Conference on File & Storage Technologies (FAST)*. Santa Clara, CA, 2017, pp. 359–374.

[12] Guy E Blelloch et al. "Efficient algorithms with asymmetric read and write costs". In: *24h Annual European Symposium on Algorithms*. 2016.

[13] Burton Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[14] Philippe Bonnet. "What's Up with the storage hierarchy?" In: *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*. Chaminade, CA, 2017.

[15] Luc Bouganim, Björn Jónsson, and Philippe Bonnet. "uFLIP: Understanding Flash IO Patterns". In: *arXiv preprint arXiv:0909.1780* (2009).

[16] Panayiotis Bozanis, Alexandros Nanopoulos, and Yannis Manolopoulos. "LR-tree: a logarithmic decomposable spatial index method". In: *The computer journal* 46.3 (2003), pp. 319–331.

[17] Yu Cai et al. "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives". In: *Proceedings of the IEEE* 105.9 (2017), pp. 1666–1704.

[18] Mustafa Canim et al. "SSD bufferpool extensions for database systems". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1435–1446.

[19] Anderson C Carniel, Ricardo R Ciferri, and Cristina DA Ciferri. "A generic and efficient framework for flash-aware spatial indexing". In: *Information Systems* (2018).

[20] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. "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 & Information Systems (ADBIS)*. Nicosia, Cyprus, 2017, pp. 229–243.

[21] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. "Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives". In: *Journal of Information & Data Management* 8.1 (2017), p. 34.

[22] Anderson Chaves Carniel, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. “The performance relation of spatial indexing on hard disk drives and solid state drives”. In: *XVII Brazilian Symposium on Geoinformatics (GeoInfo)*. Campos do Jordão, SP, Brazil, 2016, pp. 263–274.

[23] Anderson Chaves Carniel et al. “An Efficient flash-aware spatial index for points”. In: *Proceedings of the XIX Brazilian Symposium on Geoinformatics (GEOINFO)*. Campina Grande, Brazil, 2018, pp. 65–79.

[24] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. “Atlas: Leveraging locks for non-volatile memory consistency”. In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 433–452.

[25] Bernard Chazelle and Leonidas J. Guibas. “Fractional cascading: A data structuring technique”. In: *Algorithmica* 1.1-4 (1986), pp. 133–162.

[26] Feng Chen, Binbing Hou, and Rubao Lee. “Internal parallelism of flash memory-based solid-state drives”. In: *ACM Transactions on Storage* 12.3 (2016), p. 13.

[27] Feng Chen, David A Koufaty, and Xiaodong Zhang. “Hystor: making the best use of solid state drives in high performance storage systems”. In: *Proceedings of the International Conference on Supercomputing*. 2011, pp. 22–32.

[28] Feng Chen, David A Koufaty, and Xiaodong Zhang. “Understanding intrinsic characteristics and system implications of flash memory based solid state drives”. In: *Proceedings of the 11th International Joint Conference on Measurement & Modeling of Computer Systems (SIGMETRICS/Performance)*. Seattle, WA, 2009, pp. 181–192.

[29] Sangyeun Cho, Sanghoan Chang, and Insoon Jo. “The solid-state drive technology, today and tomorrow”. In: *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*. Seoul, Korea, 2015, pp. 1520–1522.

[30] Sangyeun Cho et al. “Active disk meets flash: A case for intelligent SSDs”. In: *Proceedings of the 27th ACM International Conference on Supercomputing (ICS)*. Eugene, OR, 2013, pp. 91–102.

[31] Michael Cornwell. “Anatomy of a solid-state drive”. In: *Communications of the ACM* 55.12 (2012), pp. 59–63.

[32] Ronald Fagin et al. “Extendible hashing - A fast access method for dynamic files”. In: *ACM Transactions on Database Systems* 4.3 (1979), pp. 315–344.

[33] Athanasios Fevgas and Panayiotis Bozanis. “A Spatial Index for Hybrid Storage”. In: *23rd International Database Engineering & Applications Symposium. IDEAS 2019*. ACM. 2019.

[34] Athanasios Fevgas and Panayiotis Bozanis. “Grid-file: Towards to a flash efficient multi-dimensional index”. In: *Proceedings of the 29th International Conference on Database & Expert Systems Applications (DEXA)*. Vol. II. Regensburg, Germany, 2015, pp. 285–294.

[35] Athanasios Fevgas and Panayiotis Bozanis. “LB-Grid: An SSD Efficient Grid File”. In: *Data Knowledge Engineering* (2019).

[36] Athanasios Fevgas et al. “A Study of R-tree Performance in Hybrid Flash/3DXPoint Storage”. In: *10th International Conference on Information, Intelligence, Systems and Applications IISA 2019*. 2019.

[37] Athanasios Fevgas et al. “Indexing in Flash Storage Devices: A Survey on Challenges, Current Approaches, and Future Trends”. In: *Submitted to VLDB Journal, under review* (2019).

[38] Raphael A. Finkel and Jon Louis Bentley. “Quad trees a data structure for retrieval on composite keys”. In: *Acta Informatica* 4.1 (1974), pp. 1–9.

[39] Volker Gaede and Oliver Günther. “Multidimensional access methods”. In: *ACM Computing Surveys* 30.2 (1998), pp. 170–231.

[40] Congming Gao et al. “Exploiting parallelism for access conflict minimization in flash-based solid state drives”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems* 37.1 (2018), pp. 168–181.

[41] Javier González and Matias Bjørling. “Multi-tenant I/O isolation with open-channel SSDs”. In: *Proceedings of the 8th Annual Non-Volatile Memories Workshop (NVMW)*. San Diego, CA, 2017.

[42] Boncheol Gu et al. “Biscuit: A framework for near-data processing of big data workloads”. In: *Proceedings 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. Seoul, Korea, 2016, pp. 153–165.

[43] Y. Gu. “Survey: Computational Models for Asymmetric Read and Write Costs”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 733–743.

[44] Antonin Guttman. “R-trees: A dynamic index structure for spatial searching”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. Boston, MA, 1984, pp. 47–57.

[45] Marios Hadjieleftheriou. *libspatialindex 1.8.5*. `http://libspatialindex.github.io/`. [Online; accessed 20-Feb-2015]. 2015.

[46] Frank T Hady et al. “Platform storage performance With 3D XPoint technology”. In: *Proceedings of the IEEE* 105.9 (2017), pp. 1822–1833.

[47] Klaus Hinrichs. “Implementation of the grid file: Design concepts and experience”. In: *BIT Numerical Mathematics* 25.4 (1985), pp. 569–592.

[48] Yang Hu et al. “Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity”. In: *Proceedings of the 25th International Conference on Supercomputing (ICS)*. Tucson, AZ, 2011, pp. 96–107.

[49] Riko Jacob and Nodari Sitchinava. “Lower Bounds in the Asymmetric External Memory Model”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 2017, pp. 247–254.

[50] Peiquan Jin, Puyuan Yang, and Lihua Yue. “Optimizing B^+-tree for hybrid storage systems”. In: *Distributed & Parallel Databases* 33.3 (2015), pp. 449–475.

[51] Peiquan Jin, Puyuan Yang, and Lihua Yue. “Optimizing B^+-tree for hybrid storage systems”. In: *Distributed & Parallel Databases* 33.3 (2015), pp. 449–475.

[52] Peiquan Jin et al. “AD-LRU: An efficient buffer replacement algorithm for flash-based databases”. In: *Data & Knowledge Engineering* 72 (2012), pp. 83–102.

[53] Peiquan Jin et al. “Optimizing R-tree for flash memory”. In: *Expert Systems with Applications* 42.10 (2015), pp. 4676–4686.

[54] Peiquan Jin et al. “Read/write-optimized tree indexing for solid-state drives”. In: *The VLDB Journal* 25.5 (2016), pp. 695–717.

[55] Insoon Jo et al. “YourSQL: a high-performance database system leveraging in-storage computing”. In: *Proceedings of the VLDB Endowment* 9.12 (2016), pp. 924–935.

[56] Ibrahim Kamel and Christos Faloutsos. *Hilbert R-tree: An improved R-tree using fractals*. Tech. rep. 1993.

[57] Jeong-Uk Kang et al. “The multi-streamed solid-state drive”. In: *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage & File Systems (HotStorage)*. Philadelphia, PA, 2014.

[58] Eden Kim. *SSD performance - a primer*. Tech. rep. Solid State Storage Initiative, 2013.

[59] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. “NVMeDirect: A User-space I/O Framework for application-specific optimization on NVMe SSDs”. In: *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage & File Systems (HotStorage)*. Denver, CO, 2016.

[60] Sungchan Kim et al. “In-storage processing of database scans and joins”. In: *Information Sciences* 327 (2016), pp. 183–200.

[61] Ioannis Koltsidas and Vincent Hsu. *IBM storage and NVM express revolution*. Tech. rep. IBM, 2017.

[62] I Koltsidas et al. “PSS: A prototype storage subsystem based on PCM”. In: *Proceedings of the 5th Annual Non-Volatile Memories Workshop (NVMW)*. San Diego, CA, 2014.

[63] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. “Reaping the performance of fast NVM storage with uDepot”. In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA, 2019, pp. 1–15.

[64] Pradeep Kumar and H Howie Huang. “Falcon: Scaling io performance in multi-ssd volumes”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. 2017, pp. 41–53.

[65] Se Jin Kwon et al. “FTL algorithms for NAND-type flash memories”. In: *Design Automation for Embedded Systems* 15.3 (2011), pp. 191–224.

[66] Hyun-Seob Lee and Dong-Ho Lee. “An efficient index buffer management scheme for implementing a B-tree on NAND flash memory”. In: *Data & Knowledge Engineering* 69.9 (2010), pp. 901–916.

[67] Yong-Goo Lee et al. “μ-tree: A memory-efficient flash translation layer supporting multiple mapping granularities”. In: *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT)*. Atlanta, GA, 2008, pp. 21–30.

[68] Young-Sik Lee et al. “Accelerating external sorting via on-the-fly data merge in active SSDs”. In: *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage & File Systems (HotStorage)*. Philadelphia, PA, 2014.

[69] Guohui Li et al. “Efficient implementation of a multi-dimensional index structure over flash memory storage systems”. In: *The Journal of Supercomputing* 64.3 (2013), pp. 1055–1074.

[70] Huaicheng Li et al. “The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator”. In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA, 2018, pp. 83–90.

[71] Yinan Li et al. “Tree indexing on flash disks”. In: *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*. Shanghai, China, 2009, pp. 1303–1306.

[72] Yinan Li et al. “Tree indexing on solid state drives”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1195–1206.

[73] Song Lin et al. “Efficient indexing data structures for flash-based sensor devices”. In: *ACM Transactions on Storage* 2.4 (2006), pp. 468–503.

[74] John DC Little and Stephen C Graves. “Little’s law”. In: *Building intuition*. Springer, 2008, pp. 81–100.

[75] Witold Litwin. “Linear hashing: A new tool for file and table addressing”. In: *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB)*. Montreal, Canada, 1980, pp. 212–223.

[76] Hang Liu and H Howie Huang. “Graphene: Fine-Grained IO Management for Graph Computing”. In: *FAST*. 2017, pp. 285–300.

[77] Xin Liu and Kenneth Salem. “Hybrid storage management for database systems”. In: *Proceedings of the VLDB Endowment* 6.8 (2013), pp. 541–552.

[78] Yi Liu et al. “MOLAR: A cost-efficient, high-performance hybrid storage cache”. In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–5.

[79] Yanfei Lv et al. “A probabilistic data replacement strategy for flash-based hybrid storage system”. In: *Asia-Pacific Web Conference*. 2013, pp. 360–371.

[80] Yanfei Lv et al. “Log-compact R-tree: an efficient spatial index for SSD”. In: *Proceedings of the 16th International Conference on Database Systems for Advanced Applications (DASFAA)*. Vol. III. Hong Kong, China, 2011, pp. 202–213.

[81] Yannis Manolopoulos et al. *R-trees: theory and applications*. Springer, 2010.

[82] Kurt Mehlhorn and Stefan Näher. “Dynamic fractional cascading”. In: *Algorithmica* 5.1-4 (1990), pp. 215–241.

[83] Justin Meza et al. “A large-scale study of flash memory failures in the field”. In: *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*. Portland, OR, 2015, pp. 177–190.

[84] Rino Micheloni. *3D flash memories*. Springer, 2016.

[85] Rino Micheloni. *Solid-state-drives modeling*. Springer, 2017.

[86] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[87] Sparsh Mittal and Jeffrey S Vetter. “A survey of software techniques for using non-volatile memories for storage and main memory systems”. In: *IEEE Transactions on Parallel & Distributed Systems* 27.5 (2016), pp. 1537–1550.

[88] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. “Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring”. In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. Baltimore, MD, 2005, pp. 634–645.

[89] Mihir Nanavati et al. “Non-volatile storage: Implications of the datacenter’s shifting center”. In: *ACM Queue* 13.9 (2016).

[90] Iyswarya Narayanan et al. “SSD Failures in Datacenters: What? When? and Why?” In: *Proceedings of the 9th ACM International on Systems & Storage Conference (SYSTOR)*. Haifa, Israel, 2016.

[91] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. “The Grid file: AAn adaptable, symmetric multikey file structure”. In: *ACM Transactions on Database Systems* 9.1 (1984), pp. 38–71.

[92] Junpeng Niu, Jun Xu, and Lihua Xie. “Hybrid Storage Systems: A Survey of Architectures and Algorithms”. In: *IEEE ACCESS* 6 (2018), pp. 13385–13406.

[93] Apostolos N Papadopoulos et al. “Grid file (and family)”. In: *Encyclopedia of Database Systems*. Springer, 2009, pp. 1279–1282.

[94] Chanik Park et al. “A reconfigurable FTL architecture for NAND flash-based applications”. In: *ACM Transactions on Embedded Computing Systems* 7.4 (2008), p. 38.

[95] Seon-yeong Park et al. “CFLRU: a replacement algorithm for flash memory”. In: *Proceedings of the International Conference on Compilers, Architecture & Synthesis for Embedded Systems*. 2006, pp. 234–241.

[96] Maciej Pawlik and Wojciech Macyna. “Implementation of the Aggregated R-tree over flash memory”. In: *Proceedings of the 17th International Conference on Database Systems for Advanced Applications (DASFAA), International Workshops: FlashDB, ITEMS, SNSM, SIM3, DQDI*. Busan, Korea, 2012, pp. 65–72.

[97] Roger Pearce, Maya Gokhale, and Nancy M Amato. “Multithreaded asynchronous graph traversal for in-memory and semi-external memory”. In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage & Analysis (SC)*. New Orleans, LA, 2010, pp. 1–11.

[98] John T Robinson. “The KDB-tree: a search structure for large multidimensional dynamic indexes”. In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. Ann Arbor, MI, 1981, pp. 10–18.

[99] Hongchan Roh et al. “AS B-tree: A study of an efficient B^+-tree for SSDs”. In: *Journal of Information Science & Engineering* 30.1 (2014), pp. 85–106.

[100] Hongchan Roh et al. “B^+-tree index optimization by exploiting internal parallelism of flash-based solid state drives”. In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 286–297.

[101] Hongchan Roh et al. “B^+-tree index optimization by exploiting internal parallelism of flash-based solid state drives”. In: *Proceedings of the VLDB Endowment* 5.4 (2011), pp. 286–297.

[102] Hongchan Roh et al. “MPSearch: Multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs”. In: *IEEE Data Engineering Bulletin* 37.2 (2014), pp. 3–11.

[103] Hongchan Roh et al. “MPSearch: Multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs”. In: *IEEE Data Engineering Bulletin* 37.2 (2014), pp. 3–11.

[104] Kenneth A Ross. “Modeling the performance of algorithms on flash memory devices”. In: *Proceedings of the 4th International Workshop on Data management on New Hardware (DaMoN)*. Vancouver, Canada, 2008, pp. 11–16.

[105] George Roumelis et al. “An efficient algorithm for bulk-loading xBR+-trees”. In: *Computer Standards & Interfaces* 57 (2018), pp. 83–100.

[106] George Roumelis et al. “Bulk Insertions into xBR$^+$-trees”. In: *International Conference on Model and Data Engineering*. Springer. 2017, pp. 185–199.

[107] George Roumelis et al. “Bulk-loading and bulk-insertion algorithms for xBR-trees in Solid State Drives”. In: *Computing* (2019). available online. DOI: 10.1007/s00607-019-00709-4.

[108] George Roumelis et al. “Bulk-Loading xBR$^+$-trees”. In: *MEDI*. 2016.

[109] George Roumelis et al. “Spatial batch-queries processing using xBR$^+$-trees in solid-state drives”. In: *Proceedings of the 8th International Conference on Model & Data Engineering (MEDI)*. Marrakesh, Morocco, 2018, pp. 301–317.

[110] George Roumelis et al. “The xBR$^+$-tree: An efficient access method for points”. In: *Proceedings of the 26th International Conference on Database & Expert Systems Applications (DEXA)*. Valencia, Spain, 2015, pp. 43–58.

[111] Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. “Cubetree: organization of and bulk incremental updates on the data cube”. In: *ACM SIGMOD Record* 26.2 (1997), pp. 89–99.

[112] Hanan Samet. “Applications of Spatial Data Structures”. In: Addison-Wesley, 1990.

[113] Mohamed Sarwat et al. “FAST: a generic framework for flash-aware spatial trees”. In: *Proceedings of the 12th International Symposium in Advances in Spatial & Temporal Databases (SSTD)*. Minneapolis, MN, 2011, pp. 149–167.

[114] Mohamed Sarwat et al. “Generic and efficient framework for search trees on flash memory storage systems”. In: *GeoInformatica* 17.3 (2013), pp. 417–448.

[115] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. “Flash reliability in production: The expected and the unexpected”. In: *Proceedings of the 14th USENIX Conference on File & Storage Technologies (FAST)*. Santa Clara, CA, 2016, pp. 67–80.

[116] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. “The R+-Tree: A Dynamic Index For Multi-Dimensional Objects”. In: 1987, pp. 507–518.

[117] Eric Seppanen, Matthew T O'Keefe, and David J Lilja. "High performance solid state storage under Linux". In: *Proceedings of the 26th IEEE Symposium on Mass Storage Systems & Technologies (MSST)*. 2010, pp. 1–12.

[118] Zhaoyan Shen et al. "DIDACache: An Integration of Device and Application for Flash-based Key-value Caching". In: *ACM Transactions on Storage* 14.3 (2018), 26:1–26:32.

[119] Yongseok Son et al. "An empirical evaluation and analysis of the performance of NVM express solid state drive". In: *Cluster Computing* 19.3 (2016), pp. 1541–1553.

[120] Yufei Tao and Dimitris Papadias. "Efficient historical R-trees". In: *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*. IEEE. 2001, pp. 223–232.

[121] Michael Vassilakopoulos and Yannis Manolopoulos. "External balanced regular (x-BR) trees: new structures for very large spatial databases". In: *Advances In Informatics*. World Scientific, 2000, pp. 324–333.

[122] Jianguo Wang et al. "SSD in-storage computing for list intersection". In: *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN)*. San Francisco, CA, 2016.

[123] Jianguo Wang et al. "SSD in-storage computing for search engines". In: *IEEE Transactions on Computers* (2016).

[124] Na Wang et al. "OR-tree: An optimized spatial tree index for flash-memory storage systems". In: *Proceedings of the 3rd International Conference in Data & Knowledge Engineering (ICDKE)*. Wuyishan, China, 2012, pp. 1–14.

[125] Peng Wang et al. "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD". In: *Proceedings of the 9th Eurosys Conference*. Amsterdam, The Netherlands, 2014.

[126] NVM Express Workgroup. *NVME Overview*. Online: `http://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf`.

[127] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. "An efficient B-tree layer for flash-memory storage systems". In: *Revised Papers of the 9th International Conference on Real-Time & Embedded Computing Systems & Applications (RTCSA)*. Tainan, Taiwan, 2003, pp. 409–430.

[128] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. "An efficient R-tree implementation over flash-memory storage systems". In: *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems (GIS)*. New Orleans, LO, 2003, pp. 17–24.

[129] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. “An efficient B-tree layer implementation for flash-memory storage systems”. In: *ACM Transactions on Embedded Computing Systems* 6.3 (2007).

[130] Chin-Hsien Wu and Yu-Hsun Lin. “A concurrency buffer control in B-trees for flash-memory storage systems”. In: *IEEE Embedded Systems Letters* 4.1 (2012), pp. 9–12.

[131] Xiaoyan Xiang et al. “A reliable B-tree implementation over flash memory”. In: *Proceedings of the 23rd ACM Symposium on Applied Computing (SAC)*. Fortaleza, Brazil, 2008, pp. 1487–1491.

[132] Jian Xu and Steven Swanson. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA, 2016, pp. 323–338.

[133] Jian Xu et al. “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 427–439.

[134] Qiumin Xu et al. “Performance analysis of NVMe SSDs and their implication on real world databases”. In: *Proceedings of the 8th ACM International Systems & Storage Conference (SYSTOR)*. Haifa, Israel, 2015.

[135] Chengcheng Yang et al. “Efficient buffer management for tree indexes on solid state drives”. In: *International Journal of Parallel Programming* 44.1 (2016), pp. 5–25.

[136] Jinfeng Yang and David J Lilja. “Reducing Relational Database Performance Bottlenecks Using 3D XPoint Storage Technology”. In: *Proceedings of the 17th IEEE International Conference on Trust, Security & Privacy in Computing & Communications / 12th IEEE International Conference On Big Data Science & Engineering (TrustCom/BigDataSE)*. 2018, pp. 1804–1808.

[137] Jun Yang et al. “NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems”. In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA, 2015, pp. 167–181.

[138] Ziye Yang et al. “SPDK: A development kit to build high performance storage applications”. In: *Proceedings of the IEEE International Conference on Cloud Computing Technology & Science (CloudCom)*. Hong Kong, China, 2017, pp. 154–161.

[139] Jie Zhang et al. “FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. 2018, pp. 477–492.