The VLDB Journal
https://doi.org/10.1007/s00778-019-00559-8

SPECIAL ISSUE PAPER

Indexing in flash storage devices: a survey on challenges, current approaches, and future trends

Athanasios Fevgas[1] · Leonidas Akritidis[1] · Panayiotis Bozanis[1] · Yannis Manolopoulos[2]

Received: 31 December 2018 / Revised: 30 April 2019 / Accepted: 23 July 2019

Abstract
Indexes are special purpose data structures, designed to facilitate and speed up the access to the contents of a file. Indexing has been actively and extensively investigated in DBMSes equipped with hard disk drives (HDDs). In the recent years, solid-state drives (SSDs), based on NAND flash technology, started replacing magnetic disks due to their appealing characteristics: high throughput/low latency, shock resistance, absence of mechanical parts, low power consumption. However, treating SSDs as simply another category of block devices ignores their idiosyncrasies, like erase-before-write, wear-out and asymmetric read/write, and may lead to poor performance. These peculiarities of SSDs dictate the refactoring or even the reinvention of the indexing techniques that have been designed primarily for HDDs. In this work, we present a concise overview of the SSD technology and the challenges it poses. We broadly survey 62 flash-aware indexes for various data types, analyze the main techniques they employ, and comment on their main advantages and disadvantages, aiming to provide a systematic and valuable resource for researchers working on algorithm design and index development for SSDs. Additionally, we discuss future trends and new lines of research related to this field.

Keywords Indexing · Flash memory · Solid-state drives · Nonvolatile memory · New hardware technologies

1 Introduction

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 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 great importance, the study of indexing was no 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 [9], R-trees [56], linear [101], and extendible hashing [44]. The direct usage of these data structures on flash memory will lead to poor performance due to several remarkable properties that differentiate it from the magnetic disk. Namely:

1. Asymmetric read/write latencies: reads are faster than writes;
2. Erase-before-write: one can write to ("program") a page 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;

✉ Panayiotis Bozanis
pbozanis@e-ce.uth.gr

Athanasios Fevgas
fevgas@e-ce.uth.gr

Leonidas Akritidis
leoakr@e-ce.uth.gr

Yannis Manolopoulos
yannis.manolopoulos@ouc.ac.cy

[1] DaSE Lab, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece

[2] Faculty of Pure and Applied Sciences, Open University of Cyprus, Nicosia, Cyprus

Published online: 03 August 2019

Table 1 Flash-aware indexes

Index category	Index name
FTL-based B-trees	BFTL [149,151], IBSF [88], RBFTL [152], lazy-update B+tree [116], MB-tree [124], FD-tree [98,99], WOBF [51], FlashB-tree [73], FB-tree [75], TNC [59], uB+tree [141], FD+tree & FD+FC [139], PIO B-tree [125,126], Bw-tree [92,93], AB-tree [64], AS B-tree [123], BF-tree [7], bloom tree [66], BSMVBT [34], HybridB tree [68], WPCB-tree [61]
Raw flash B-trees	Wandering tree [13,43], μ-tree [90], μ*-tree [77], B+tree(ST) [114], F-tree [19], IPL B+tree [111], d-IPL B+tree [110], LA-tree [2], AD-tree [45], LSB+tree [71,72], LSB-tree [79]
Skip lists	FlashSkipList [142], write-optimized skip list [11]
FTL-based hash indexes	LS linear hash [97], HashTree [38], SAL-hashing [67,156], BBF [21], BloomFlash [41], FBF [102]
Raw flash hash indexes	MicroHash [100], DiskTrie [35], MLDH [157], SA Extendible Hash [145], h-hash [80], PBFilter [160]
FTL-based PAM indexes	F-KDB [94], GFFM [46], LB-Grid [47], xBR$^+$-tree [129], eFind xBR$^+$-tree [27]
Raw flash PAM indexes	MicroGF [100]
FTL-based SAM indexes	RFTL [150], LCR-tree [103], Flash-aware aR-tree [119], FOR-tree [65,146]
Generic frameworks	FAST [131,132], eFind [25,26]
FTL-based inverted indexes	Hybrid merge [96], MFIS [76]
Raw flash inverted indexes	Microsearch [137], Flashsearch [22]

3. Limited life cycle: every flash cell exhibits a high bit error rate (wear-out) after a certain number of program/erase cycles have been performed. 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 provide a sophisticated firmware, known as flash translation layer (FTL), which runs on the device controller. FTL employs an out-of-place policy, using logical 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 coded 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, 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. Table 1 summarizes the surveyed indexes, categorized as variants of their HDD-resident counterparts, when applied. The indexes either use FTL, and so they are characterized as FTL-based, or directly handle NAND flash memories and thus they are indicated as raw flash. Most of the works either attempt to postpone the individual write operations, by maintaining in-memory write buffers, or apply logging techniques to restrain small random writes, or strive to exploit the internal parallelization of SSDs.

In the sequel, we present a concise but broad overview of flash-aware indexes. The contributions of this work can be summarized as follows:

- We give an overview of the flash memory and NAND flash SSD technologies as well as the challenges they present to algorithm and data structure design;
- We broadly survey 62 flash-aware indexes for various data types;
- We summarize the main index design techniques employed;
- We discuss future trends and identify new and significant lines of research.

We aspire that this work will provide a systematic guide for researchers who work on or are interesting in algorithm

design and index development on SSDs. The remainder of this paper is organized as follows. Section 2 provides the necessary background information on flash memory and flash-based SSDs. Section 3 discusses the main design techniques employed in flash-aware indexes. One-dimensional indexes are presented in Sect. 4, Sect. 5 summarizes multi-dimensional indexes, and Sect. 6 generic frameworks. Section 7 covers inverted indexes, while future trends and interesting lines of research are briefed in Sect. 8. Finally, Sect. 9 concludes our work.

2 Background

In this section, we review nonvolatile 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.

2.1 Nonvolatile memories

Flash memory NAND flash has been introduced in 1999 by Toshiba as nonvolatile 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 [108]. 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 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 [30,62]. Blocks are the smallest erasable units in NAND

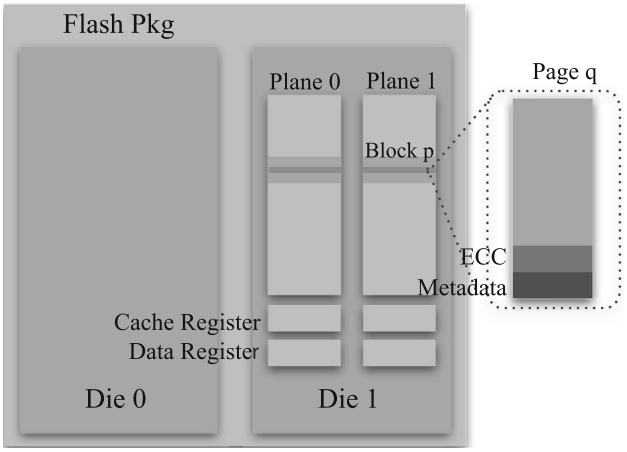

Fig. 1 The anatomy of a flash package

flash, whereas pages are the smallest programmable ones. Each page has a data region and an additional area for error correction data and metadata [107] and may comprise a few sectors. Page and block sizes are usually unknown. However, typical values are 2–16 KB and 128–512 KB, 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 [20,106,113,134]. The most important reasons causing increased error rates are wearing-out, aging, and exposure to high temperatures [106,134]. Latest storage systems utilized sophisticated error correction codes (ECC), such as BCH and LDPC, to decrease bit errors [108]. However, when a block presents an unrecoverable error [134], or its error rate exceeds the average [20], 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 nonvolatile memories The emergence of new NVM technologies is promising to change memory hierarchy in the future computer systems. 3DXPoint, Phase Change Mem-

ory (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 [109]. Several studies propose different approaches for integrating the upcoming NVMs into the memory hierarchy [109]. However, their integration into the memory system dictates changes to both hardware and software [85]. 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 [28,154].

3DXPoint in one of the most promising NVMs today. Intel and Micron announced 3DXPoint in 2015, and Intel started to ship block devices (Optane SSDs) based on 3DXPoint in 2017. 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 [107]. 3DXPoint SSDs provide 10 times lower latency compared to their NAND flash counterparts, while they can achieve high performance (IOPS) even at small queue depths [58,84]. On the contrary, the efficiency of flash-based SSDs is tightly coupled with high queue depths.

2.2 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 nonvolatile memories (usually NAND flash) to store data, instead of spinning magnetic-plates that HDDs use.

SSD architecture Uncovering an SSD device (Fig. 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.

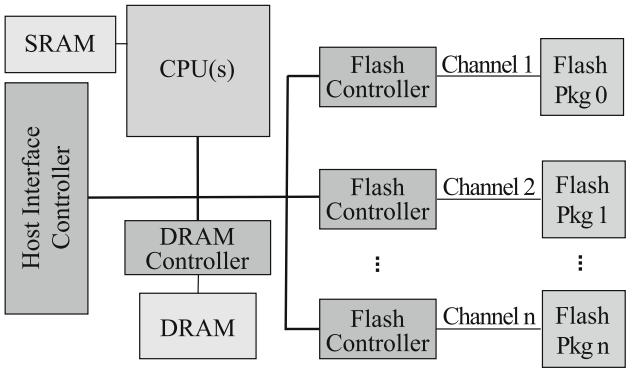

Fig. 2 Overview of the architecture of a solid-state drive

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 [108]. 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 [32]. The advancement of SSDs made SATA and SAS inadequate to support their high efficiency as they have been developed for magnetic disks [136]. 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 between 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 [148]. 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 [136]; it also reduces the overhead of system's software up to four times [155]. 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 leveling, 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 [87].

Out-of-place updates leave flash pages invalidated; thus, FTL initiates a garbage collection procedure to reclaim free space for incoming data [37]. 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 [87]. 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 leveling 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 preexisted as a result of the manufacturing process. FTL handles defective blocks through address mapping [32,118]. 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.

3 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 [4,5,127]. In particular, [5] presents a rather simplistic model which adopts different costs for reads and writes, whereas the approach in [127] requires bit level access to flash pages. [4] 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 [120], as well as the effects of internal processes, like garbage collection and wear leveling, 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 [30,31], whereas others employed simulation [3,18] 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 Sect. 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 [30]. Actually, access conflicts to the same resources between consecutive I/O operations may prevent the full utilization of parallelism [30,50].

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. [50] 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 [81].

Furthermore, the experimental evaluation of SSD devices has provided very interesting conclusions about their efficiency [30,31]. 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. [50] 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, [30] 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, [30] 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 [46,47,128,129], 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 [30].

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 leveling, 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 twofold: on the one hand, batch read operations are enabled, and 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.

4 One-dimensional indexes

4.1 B-tree indexes

B-trees are generalization of Binary Search Trees (BST) [9]. 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 (cf. Fig. 3). 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.

Due to its usability, a number of B-tree variations were introduced. Among them, one of particular interest for the area of SSDs is the *Log-Structured Merge-tree (LSM-tree)* [117,138], since it is used in log-structured file systems for SSDs. An LSM-tree consists of a number of levels, each one implemented as a B+tree, where internal nodes only store keys, whereas key-record pairs reside into leaves. Every B+tree leaf node maintains a pointer to its right sibling [36]. The top level of a LSM-tree is kept always in main memory. The sizes of B+trees are geometrically increasing. Update operations are always inserted ("logged") as (IU) entries in the top level. Whenever a level overflows, it is merged with the next level in one pass, during which insertion/deletion entries referring to the same keys are canceled out. If the next level also overflows, the procedure is repeated, until no overflow takes place. Search starts from the top level and visits each level, until either the item is found or all levels are exhaustively investigated.

In [140] three merge policies for LSM-trees located in SSD devices are presented. According to the first policy, termed round-robin, each time a level overflows the next δ percent of its entries are chosen for merging. ChooseBest, the second alternative, chooses for merging a continuous δ fraction of the entries, having the least number of overlaps with pages of next level. Finally, the mixed policy defines for each level a threshold. When the number of its entries drops below that threshold, a full merge is conducted, else ChooseBest is employed. Mixed has been proven to be the best policy against the full merge of LSM-trees.

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.

4.1.1 FTL-based indexes

BFTL Wu et al. [149,151] were the first researchers that introduced a B-tree variant especially designed for SSDs, named *BFTL*. BFTL employs log-structured nodes and a main mem-

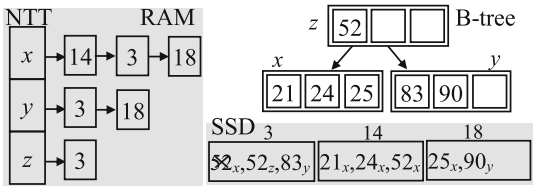

Fig. 3 BFTL: B-tree and its actual representation; subscripts denote page ids, × indicates delete IU

ory 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 (Fig. 3). 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 predefined 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 Lee and Lee [88] tried 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 [152] 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, employing repeatedly the IBSF policy. 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.

MB-tree Roh et al. [124] presented MB-trees. It employs a write buffer, termed *Batch Process Buffer (BPB)*, to delay updates. When the write buffer overflows, the leaf node with the most updates is chosen as victim. Since determining this may involve many reads, MB-tree employs the approximate method *Unique Path Search (UPS)*, which restricts the calculations along one path only: each time, it chooses the child node covering the greatest number of PBP entries.

Additionally, the scheme utilizes big leaf nodes to delay splitting, while upper levels are kept as in the standard B-tree. Each leaf (Fig. 4) comprises several pages and one head page (LNH) for facilitating the search among the pages. LNH contains entries consisting of page ids and key values, sorted on key values. In this way, it builds increasing ranges of values covered by page ids, likewise to the index nodes in a B-tree. Whenever a batch of updates are applied to a big leaf, deletions are performed first, whereas the best fit policy is employed for the insertions. In case a big leaf overflows, the keys are split evenly in two leaves, and the number of pages per leaf is adjusted/decided dynamically according to the LNH structure.

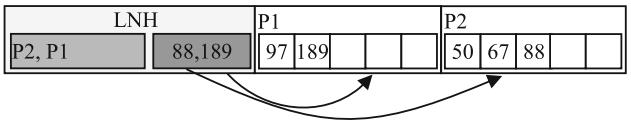

Fig. 4 MB-tree leaf configuration: there are two entries since all keys in P2, upper bounded by 88, are smaller that those in P1, bounded by 189

The performance of the index is evaluated both theoretically and experimentally. Using a rather simplistic analysis, it is found that, in worst case, a search operation needs $(2 + \lceil \log_M \frac{2*n}{M*n_l} \rceil)$ reads, n_l the number of leaves, and an update operation requires $3/n_f$ writes and $((n_l + \lceil \log_M \frac{2*n}{M*n_l} \rceil)/n_f)$ reads, n_f the number of PBP entries flushed due to an overflow. Experimental comparisons with BFTL, B+tree(ST) (described below), and B-tree show the good performance of the scheme in all cases except searching, where the B-tree prevails. Buffer flushes and LNH maintenance incur extra CPU and I/O costs, which are tried to be amortized to delayed (buffered) updates.

Lazy-update B+tree Lazy-Update (LU) B+tree [116] follows the approach of employing in memory buffering. Inside the write buffer, the updates are grouped together if they refer to the same leaf node, while opposite operations concerning the same key are canceled out. When the write buffer overflows, a group of nodes are committed according to two alternative polices: (i) *biggest group size*, which aims to increase the number of groups inside the buffer, and (ii) *minimum ratio of gain to group size,* where gain is defined as the number of leaves updated if the group is committed—this policy tries to minimize the cost of eviction. The former has the least cost in terms of computation time and page reads. The latter may increase the number of reads, since it involves accesses to leaf nodes for size computations. However, it achieves the best performance, while it may save page writes. Simulations conducted with respect to the B+tree proved update time gains, while the access performance is not degraded.

FD-tree, FD+tree/FD+FC, and BSMVBT FD-tree is a two-part index [98,99] (Fig. 5). 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 [12]. That is, their sizes are exponentially increasing from level to level. Between adjacent files, fences (i.e., pointers) are inserted to speed up searching, employing the algorithmic paradigm of fractional cascading [29,105]. 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.

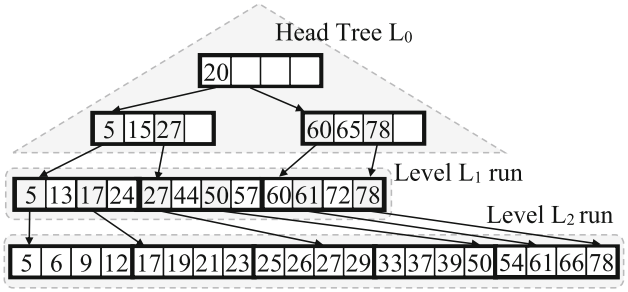

Fig. 5 A 2-level FD-tree: pointers between L_1, L_2 are fences

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, which, however, neither includes the cost of wear leveling nor discusses 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.

Thonangi et al. [139] introduced an improvement of FD-tree, the FD+tree and its concurrent version FD+FC. Additionally to FD-tree invariants, the FD+tree dictates that: (i) the bottom level has enough items so that it cannot be accommodated at a higher level, (ii) some levels containing only fences can be skipped, and (iii) the number of stored entries can be at most twice the number of live (not deleted) entries; the opposite case is considered as an index underflow. A merge operation can be triggered by either a level overflow or an index underflow. After calculating which levels must be replaced, the merge procedure scans the levels in parallel and produces the new runs in one pass. Since the upper levels may contain only fences, some of them are skipped to reduce

look-up time. It is proven that an FD+tree with n records supports searching in $O(\log_\gamma \frac{n}{\kappa_0 \beta})$ I/Os (worst case), where γ is the logarithmic size ratio between adjacent levels, κ_0 is the size of level 0, β is the block size, whereas insertions or deletions are supported in $O(\frac{\gamma}{\beta-\gamma} \log_\gamma \frac{n}{\kappa_0 \beta})$ I/Os (amortized case).

The concurrent version FD+FC, during merging, reads one block from each to-be-replaced old level in main memory and transfers keys to the bottom-most new level. Old level transferred blocks are reclaimed. The new modifications are accommodated to the new top level and, thus, search is conducted in either the old structure (if the key sought is greater than the first key of the old top level), or the new structure (since the keys have already been transferred there). FD+FC is experimentally evaluated against 2 simpler versions of concurrent FD+trees, FD+XM and FB+DS.

Bulk Split Multiversion tree (BSMVBT) [34] constitutes the multiversion (partial persistent [42]) variant of FD-tree. In particular, the B+tree root is replaced by a transactional multiversion B+tree (TMVBT) [57]. The bottom level runs are sorted in (key, timestamp) order. During the merging of adjacent levels, when entries with the same key are met, the higher level entry is removed and the other is copied with its "dead flag" set to the dead flag value of the purged entry. When the number of dead entries exceeds a certain threshold, a new root tree is build containing all "live" entries. BSMVBT was experimentally evaluated against TMVBT and was found better in case of insert-only and insert/delete workloads.

WOBF Write Optimized B-tree layer (WOBF) [51] uses an in-memory buffer to accommodate updates and a NTT to map nodes to pages containing IUs, like BFTL. When IUs are inserted into the buffer, redundant entries referring to the same key are canceled out or deletions are grouped by node id as bit maps. Thus, WOBF adopts the IBSF policies. In case of buffer overflow, all IUs are sorted by node id, packed in a number of pages and flushed to flash memory. This way, WOBF tries to reduce the scattering of node contents into as few pages as possible. The performance of WOBF was experimentally evaluated against BFTL and IBSF, and was found to be more efficient. However, WOBF also inherits the disadvantages of these two schemes, like noteworthy usage of main memory and increased number of reads.

FlashB-tree [73] introduced Flash B-tree, a scheme in which the nodes are switching between *non-clustered* and *clustered* mode of operation, using an algorithm which counts the number of reads and writes and modifies properly a counter. Each read operation decreases the counter by the read cost, while a write operation increases it by the write cost. Whenever the counter reaches the predefined thresholds, the respective switch is made. In the non-clustered mode, NTT is activated to describe the logical nodes. This way, some nodes are similar to the ones in BFTL, i.e., scattered into a number of physical pages, whereas some are clustered in one page, as IBSF dictates. This dual mode of operation is also reflected to the buffer replacement policy. So, in the clustered mode, the LRU IU is chosen and committed along with all IUs belonging to the same node in one page. On the contrary, in the clustered mode all IUs are packed and flushed to the SSD. Additionally, to save one write during spitting, the *modify-two-nodes* policy is followed: the left sibling node is not altered. The *lazy-coalesce policy* is also employed during deletion and, thus, the underflow operations of merging and borrowing are not executed. A search needs h to $h * c$ reads, and an update costs $\frac{1}{n_{\text{iu}}}(\frac{2}{3} * \tilde{N}_{\text{split}})$ amortized writes, n_{iu} the number of buffered IUs. Simulations showed that Flash B-tree combines the best of BFTL and IBSF.

FB-tree Jørgensen et al. [75] introduced FB-tree, which follows the paradigm of write-optimized B-trees [53], employing direct storage management and buffering, aiming at large flushes to SSD. In particular, every time a node is updated, it is written out-of-place. For this reason, the nodes do not contain extra space for new keys, as the node can grow or shrink when it is rewritten. Thus, space is allocated in multiples of the sector size. The explicit flash storage manager stores the tree nodes according to the best fit policy and their current size, employing a bit map structure and a list of max holes. The buffer manager uses the clock version of the LRU policy, and, upon overflow, it collects dirty and clean pages. The number of dirty pages gathered are above a threshold to maximize the write size. When a node resides in the buffer, it holds a pointer to its father. Also, a node cannot be evicted unless it is not referenced by any child and its reference counter is below a threshold. Notice that reads and writes add different amounts to the counter. The experimental evaluation studied the effect of various scheme parameters. On the other hand, only favorable comparison with B+tree is provided.

TNC TNC (Tree Node Cache) [59] is a flash-aware B+tree designed for the Unsorted Block Image File System (UBIFS) [133], as an improvement of Wandering tree. It uses an in-memory buffer to cache parts of the structure according to the following two rules: (i) if a node is in flash, then so do its children, (ii) if a node is in main memory, its children may or may not be in main memory. Thus, a node may reside in main memory only if its father is in the buffer. During tree operations, the whole path is read into main memory and changes are executed therein. When the buffer is full, TCN searches for and evicts, as needed, clean nodes with all of their children in flash memory. In case of dirty nodes,

it chooses the ones with clean children. For recovery, the technique of journaling is employed. The author presented experimental results solely for TCN.

uB+tree Based on the assumption that writes are k times slower than reads, [141] introduces the unbalanced B+tree (uB+tree). As its name suggests, this index saves writes by not performing balancing operations unless it is absolutely necessary. Namely, when a leaf node overflows, instead of splitting, it acquires an overflow node. Each overflow node is maintained for $k =$ (cost of write/cost of read) reads; due to the asymmetry between writes and reads, each newly allocated overflow node has an "unexpended" gain of k. Then it is inserted as a normal independent node. Thus, insertions can occur even when search or delete operations are executed. During the update operations, redistributions between a node and its overflow counterpart are executed whenever possible to postpone splitting and, thus, an additional write to their father is spared. The author also presents the cases where overflow nodes are also allocated to internal nodes, and when unexpended gain is examined not locally but globally. Assuming that writes are five times slower than reads, through simulation it is shown that the uB+tree outperforms the B+tree, whereas the various uB+tree versions are more or less equivalent in terms of performance. uB+tree saves writes w.r.t. standard B+tree; however, it does not capitalize on the internal parallelization of modern SSDs.

PIO B-tree PIO B-tree [125,126] 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 canceled 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, to capitalize on striping.

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, while the insert operations demand $\sum_{l=\lfloor \eta \rfloor}^{h-2} \frac{1}{G(l)} - \frac{1/M'^{(\eta \% 1)}}{G(\log_{M'}(\mu - B) - 1)} + \frac{1}{G(h-1)}$ reads and $\frac{1}{G(h-1)}$ writes, where $\eta = \log_{M'} \frac{n}{L(\mu - b)} - 1$, M' the average number of entries in a node, μ the available main memory, and $G(i)$ the average number of update operations reading the same node at level i. Experiments conducted on SSDs show the superiority of PIO B-tree over BFTL, FD-tree, and B+tree. It should be noticed that Psync is a proposal for requesting outstanding random I/Os all at once and, to the best of our knowledge, it is not directly supported.

Bw-tree Bw-tree [92,93] is designed for multi-core systems equipped with large DRAM and SSD. It is a latch free B+tree, which employs the *Compare and Swap (CAS)* operation. It uses the techniques of delta logging—to bypass the FTL—and in memory buffering. Bw-tree is built with logical ids in main memory. To this end, it uses an in-memory mapping table for mapping node ids to pages either in memory or in flash. All operations are executed as delta entries, which always are the head of the respective formed chain of records in the mapping table. Periodically, the lists are consolidated for performance reasons. Also, they are regularly flushed to flash memory. So, one can say that Bw-tree follows an approach that resembles BFTL, enabling it to handle multi-threading and concurrency control. However, it does not exploit the internal parallelization to enhance query performance. Bw-tree is experimentally compared with BerkeleyDB and skip list.

AB-tree In [64], the B-tree variant Adaptive Batched tree (AB-tree) was introduced. Each node consists of a number of buckets, separated by key values as depicted in Fig. 6. Every bucket has a data part and a buffer part for storing item operations. A new item or a delete/update entry is always inserted in the proper bucket of the root node, in its respective part. When a bucket overflows, a pushed-down operation migrates entries in the associated lower level (child) node, in the proper part. During push-downs, deletion entries eliminate the index entries with the same key. A search operation is conducted top-down, starting from the root, checking in every visited bucket both the data and the buffer part. The data/buffer ratio is independently and dynamically adjusted for each bucket after push-downs, according to the workload. A search costs, on average, $\sum_{l=1}^{h} \frac{M'^{l-1}}{N_l} l$ reads, N_l the number of entries in the l-th level, while the worst-case average cost of an insertion is h/s_n, s_n the average size of a node. AB-tree was evaluated against B+tree, BFTL, and FD-tree and found to have the best overall performance, with search being its weakest operation. The latter is due to big node sizes: while a big node can be efficiently read with massive I/O operations, it incurs increased search cost.

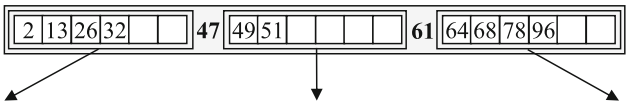

Fig. 6 An AB-tree node

AS B-tree Always Sequential B-tree (AS B-tree) [123] 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.

BF-tree Bloom Filter tree (BF-tree) [7] is an approximate tree index, which aims to minimize size 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 [16] (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 $p_{\text{fp}} * n_{pl}$ 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

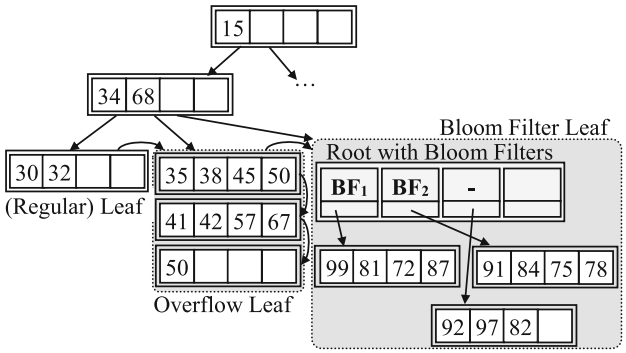

Fig. 7 A Bloom tree instance: the three types of leaves are depicted

significant space savings, especially when it indexes non-unique attributes, competing them in several cases.

Bloom tree Bloom tree [66] is also a probabilistic index, introduced to optimize both reads and writes. Specifically, as depicted in Fig. 7, 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 $p_{\text{fp}} * d + 2$ extra reads, respectively. The performance of the Bloom Tree was compared against B+tree, B+tree(ST) (presented below), FD-tree, and MB-tree, and experiments exhibited the behavior of the index in various test cases.

HybridB tree HybridB tree [68] 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

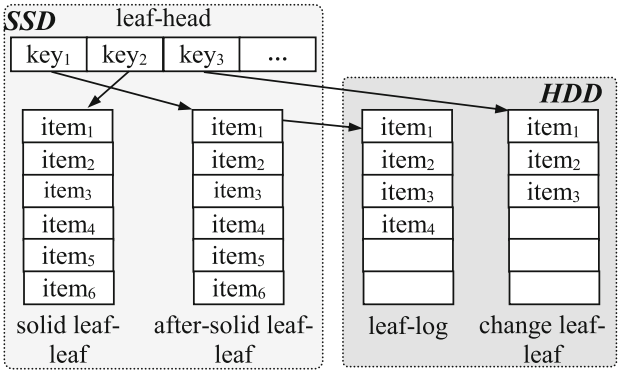

Fig. 8 HybridB tree leaf types

several pages, distributed between SSD and HDD (Fig. 8): 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. 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.

WPCB-tree The Write Pattern Converter B-tree [61] (WPCB-tree) follows the approach of converting random write patterns into sequential ones with little main memory. Specifically, it employs a flash-resident buffer, termed *Transit Buffer Manager (TBM)*, for accommodating updated pages—each page stores only one node. All page nodes with the same logical block number (LBN) are written to the same buffer block. A small in-memory summary table associates the LBNs of the buffer blocks with the logical page numbers (LPNs) of pages they contain. When TBM overflows, a victim block with the smallest number of node pages is greedily selected. Then, its pages are sorted according to LPNs, taking care that only the latest version of each page is included, and, then, are sequentially written to the data part of the SSD. The same procedure is followed whenever a TBM block overflows. To save writes, leaves with contiguous order of insertion are not split, while node underflows do not trigger merging operations. The WPCB-tree restricts the danger of data loss in case of a power failure, since the usage of main memory is very limited. However, TBM blocks may wear-off rapidly. The authors provide a rather simplistic and not so useful cost analysis of the various operations. Specifically, a search operation needs h reads, an insertion costs additionally 1 sequential write, $3*n_{sp}$ writes, n_{sp} the average number of split operations per insertion, and n_b block merge operations, while a deletion requires 1 sequential write and n_b block merge operations. Using simulation, it is found that the WPCB-tree performs better than the B-tree, the d-IPL, and the $\mu*$-tree (both described in next subsection).

Discussion BFTL trades reads for writes by scattering nodes to a number of pages. This is no longer adequate with contemporary SSDs, where the performance gap between reads and writes is narrowing. It also incurs increased main and secondary memory overhead due to the auxiliary information maintained. IBSF tries to delay buffer overflow and spares reads by dispensing with NTT. However, its performance still depends heavily on main memory. RBFTL expands IBSF approach, but at the expense of employing two extra NORs for crash recovery, additionally to main memory. WOBF combines the NTT of BFTL with the improved write buffer management of IBSF. While it partly lessens the scattering of tree nodes into several pages, nonetheless it suffers from increased RAM and read operations usage. Flash B-tree switches between BFTL and IBSF, depending on the current workload; therefore, it also inherits their common disadvantages, i.e., main memory dependence and increased read operations. Bw-tree is a latch-free B+tree based on the CAS operation. To deal with small writes, it uses the techniques of delta logging and buffering. It successfully supports multi-threading and concurrency control, howbeit it ignores the parallelization capabilities of SSDs by-passing the FTL.

Lazy-Update B+tree aims to control both writes and reads by grouping buffered updates w.r.t. the leaf involved. This however may be costly both in CPU and I/O time. MB-tree delays tree reconstruction with fat leaves and buffering. Fat leaves require an additional auxiliary page node, and thus, extra CPU and I/O operations for searching and maintenance. uB+tree spares writes by not performing balancing operations. Instead, overflow nodes are used, which are turned into normal ones when they are read k times, k the cost ratio between a write and a read operation. However, read and write costs differ from device to device. So, uB+tree does not exploit the internal parallelization of SSDs. PIO B+tree adopts the modern approach of utilizing the internal parallelization of SSDs, by buffering the issued operations and executing them in batches with mass I/O requests. Addition-

ally, it employs the technique of fat leaves where the various IUs are attached. One issue of this solution is that depends on Psync I/O which is not directly supported by OS kernels.

FD-tree transforms small random writes into sequential reads and writes, issued in batches, by employing the logarithmic method and the fractional cascading technique. Due to increased height, its search time may be worst than that of B-trees. Also, the excess number of writes harm the lifespan of SSDs. FD+FC, based on FD+tree—a modified version of FD-tree which employs single-pass level merging—, supports concurrency at the expense of increased space allocation during reorganizations. BSMVBT changes FD-tree to a multiversion index. Despite being experimentally evaluated against the HDD-oriented TMVBT, it is not clear whether it can be utilized on real systems. AS B-tree also turns random writes into sequential ones by buffering new or updated nodes and then committing them into contiguous LBAs at the end of the file. It follows that AS B-Tree is highly dependent on main memory, while it needs constant reorganization to reclaim invalid pages and achieve acceptable space overheads, which may damage the device endurance.

BF-tree is a probabilistic B-tree variant, using bloom filters for key membership inside the pages of each leaf node. In this manner, it spares space, while demonstrates good searching behavior. Since insertion is very costly, it must be used as a bulk-loading index, tolerating a moderate number of updates. BF-tree also assumes that data are organized on keys. As SSDs cost decreases while their capacity increases, index size may no longer be an issue. Bloom tree is another probabilistic index which uses three types of leaves (one fat) to delay splits and thus spares writes. To save reads, it uses BFs to guide the search inside the fat leaves. Nevertheless, it does not fully capitalize on the internal parallelization of SSDs.

AB-tree enhances standard B-tree nodes with buckets serving as storage structures for items and buffers for operations. In this way, insert, delete and update operations can be processed in batches, turning small random reads and writes into sequential ones. Additionally, each bucket can adapt to workload by adjusting the size of its two parts. While AB-tree uses the internal parallelization of SSDs, the big size of nodes cause increased CPU time which may be comparable to I/O time. To use memory frugally, WPCB-tree utilizes a designated part of SSD as buffer area, where the updated nodes are temporarily accommodated. In case of overflow, involved nodes are sequentially written to the data part. WPCB-tree has low risk of data loss, at the expense of increased number of writes. Additionally, it may wear off the buffer area.

FB-tree employs the approach of write-optimized trees in conjunction with direct storage management and in-memory buffering for grouping writes. Since it applies excessively the out-of-place policy for accommodating updated nodes, it may shorten the lifetime of the underlying SSD. TNC relies on the UBIFS file system to maintain a B+tree on SSD, keeping sub-paths involved in update operations in main memory buffer. TNC is not evaluated against other indexes. HybridB tree is a B+tree for hybrid systems, storing the internal nodes on SSD and the pages of fat leaves either on SSD, when there are full, or on HDD, otherwise. Deletions on SSD pages are accommodated by logging them on an associated HDD page. So, the tree nodes are moving between HDD and SSD. Essentially, HDD acts as a buffer mechanism for limiting the random writes to SSD, prolonging, thus, its lifespan. Nonetheless, this is based on the inferior write performance of HDDs.

Table 2 summarizes the design methods employed, the cost analysis and the experimental evaluation of the FTL-based B-trees against other indexes.

4.1.2 Raw flash indexes

Wandering tree Wandering tree [13,43] was introduced as a B+tree variant suitable for log-structured file systems, like JFFS3 and logFS. Its design is quite simple; instead of using in-place updates, every updated node is stored in a new location. Since the respective pointer in the father node changes, the father node is also modified and stored in a new free page, and so on, until the procedure finally modifies the root node. Thus, every update operation generates $O(h)$ writes, including those the balancing operations may cause, producing $O(h)$ dirty pages. So, Wandering tree may frequently trigger space reclamation and wear leveling.

μ-tree and $\mu*$-tree In [90], 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 (Fig. 9). 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. Due to its design, the μ-tree height is upper bounded by $-\log \frac{\sqrt{2}}{M} - (\sqrt{\log^2 \frac{\sqrt{2}}{M} - 2 \log \frac{n}{2}})$, n the number of items, M the fanout, and so do the number of entries indexed. Also, μ-tree undergoes a remarkable number of splits. Since this index pertains to raw flash, recycling must be explicitly implemented. Notably, it occupies more space

Table 2 FTL-Based B-trees design techniques and performance; Big O notation is omitted,—denotes lack of cost analysis, $[\ ]_*$ designates different costs, subscripts r, w, s, bm stand for reads, writes, sequential and block merges, respectively

Index	Search	Insertion‖deletion	Space	Experimental evaluation
SCATTERED LOGGING				
BFTL [149,151]	$h * c$	$2(\frac{1}{M-1} + \tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$	$n * c + B$	B-tree
WOBF [51]	–	–	–	BFTL, IBSF
SCATTERED LOGGING & NODE MODIFICATION				
Flash B-tree [73]	$(h, h * c)$	$\frac{1}{n_{\text{iu}}}(\frac{2}{3} * \tilde{N}_{\text{split}})$	$n + B$	BFTL, IBSF
NODE MODIFICATION				
uB+tree [141]	–	–	–	B+tree
BF-tree [7]	$h + [p_{\text{fp}} * n_{pl}]_{sr}$	–	$n * n_{pl}$	B+tree, FD-tree, hashing
IN MEMORY BUFFERING				
IBSF [88]	h	$\frac{1}{n_{\text{iu}}}(\tilde{N}_{\text{split}} + \tilde{N}_{\text{merge/rotate}})$	$n + B$	BFTL
RBFTL [152]	–	–	–	B-tree
LU B+tree [116]	–	–	–	B+tree
TNC [59]	–	–	–	–
AS B-tree [123]	–	–	–	B+tree, BFTL, LA-tree
FLASH BUFFERING				
FD-tree [98,99]	$\log_k n$	$[\frac{k}{f-k} \log_k n]_{srw}$	$c * n$	B+tree, BFTL, LSM-tree
FD+tree, FD+FC [139]	$\log_\gamma \frac{n}{\kappa_0 \beta}$	$[\frac{\gamma}{\beta-\gamma} \log_\gamma \frac{n}{\kappa_0 \beta}]_{srw}$	$c * n$	FD+XM, FB+DS [139]
BSMVBT [34]	–	–	–	TMVBT [57]
FLASH BUFFERING & NODE MODIFICATION				
AB-tree [64]	$\sum_{l=1}^{h} \frac{M'^{l-1}}{N_l} l$	h/s_n	n	B+tree, BFTL, FD-tree
WPCB-tree [61]	h	$[1]_{sw} + 3 * n_{sp} + [n_b]_{bm} \| [1]_{sw} + [n_b]_{bm}$	$n + B$	B-tree, the d-IPL, μ*-tree
IN MEMORY BUFFERING & NODE MODIFICATION				
MB-tree [124]	$2 + \lceil \log_M \frac{2*n}{M*n_l} \rceil$	$[3/n_{\text{f}}]_{\text{w}} + [(n_l + \lceil \log_M \frac{2*n}{M*n_l} \rceil)/n_{\text{f}}]_{\text{r}}$	$n + B$	BFTL, B+tree(ST), B-tree
FB-tree [75]	–	–	–	B+tree
Bw-tree [92,93]	–	–	–	BerkeleyDB, Skip List
Bloom tree [66]	$h + p_{\text{fp}} * d + 2$	–	$n + B$	B+tree, B+tree(ST), FD-tree, MB-tree
IN MEMORY BUFFERING & IN MEMORY BATCH READ BUFFERING & NODE MODIFICATION				
PIO B-tree [125,126]	$h - 1 + t_L$	$[\sum_{l=\lfloor \eta \rfloor}^{h-2} \frac{1}{G(l)} - \frac{1/M'^{(\eta \% 1)}}{G(\log_{M'}(\mu - B) - 1)} + \frac{1}{G(h-1)}]_{\text{r}} + [\frac{1}{G(h-1)}]_{\text{w}}$	$n + \mu$	BFTL, FD-tree, B+tree
HYBRID & NODE MODIFICATION				
HybridB tree [68]	$h + [1]_{\text{HDD}}$	$+3 + [2]_{\text{HDD}} \| + [1]_{\text{HDD}}$	n	B+tree, Hybrid B+tree

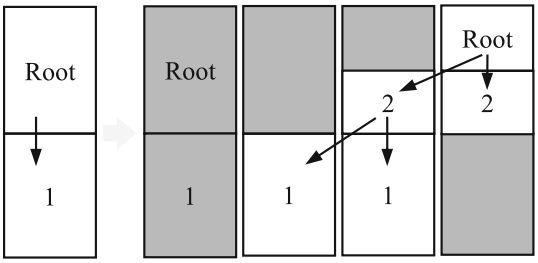

Fig. 9 μ-tree height increase due to splits, after an insertion. All node sizes are halved. The grey rectangles depict invalid nodes, the numbers indicate the level

than a B+tree due to the strict page structure. All experimental results against the original B+tree are based on simulation.

μ*-tree [77] constitutes a dynamically adaptive version of μ-tree, which tries to minimize the number of pages, postponing the height growth. Except leaf nodes, which occupy p_1 of a page, nodes at every level i have the same size $p_i = (1-p_1)/h-1$. After an update, if the root is full or the number of node splits is greater than a threshold, the leaf size is updated—either is decreased (during insertion), or increased (during deletion), or restored otherwise to max (insertion) or min (deletion) value, whereas the height increases or decreases by 1, respectively—and used afterward for future operations. As a result, nodes can split into k parts to adapt to the new size. The maximum height of a μ*-tree storing n records is $\lceil e^{W_{-1}(\frac{\log p_1+\log M-\log n}{e^{\log(1-p_1)+\log M}})} * e^{\log(1-p_1)+\log M} \rceil + 1$, $W_{-1}()$ the Lambert W Function, while the average cost of an update is $\frac{n_u+\frac{(n_u-e)n}{eMp_1}}{n_u}$, n_u the number of update operations and e the number of pages allocated. Using simulation for MLC flash, μ*-tree is found to perform better than μ-tree, B+tree and BFTL, while its size and height are bigger than those of a B+tree.

B+tree(ST) Self-tuning B+tree (B+tree(ST)) [114] aims to locally employ scattered logging, adapting thus to workload, as its name suggests. Namely, a node is either a "normal" B-tree node, stored in one page, or a "logged" one, following the approach of BFTL. The first category corresponds to read-intensive nodes, whereas the second to write-intensive ones. The mode of each node is decided constantly and independently with a 3-competitive online algorithm, by counting the number of reads and writes and comparing the accumulated difference between serving costs in either mode to the total cost for transition between the two modes. Also, its size is determined by the ratio utility (log #entries) to cost (amortized read/write), which is maximized when a node fits in exactly one page. Its performance was found experimentally to be better than B+tree and BFTL. The switching is conducted with some delay and, thus, the query patterns are not fully exploited. Also, frequent query pattern changes lead to frequent node switches, and so to increased maintenance cost, as well.

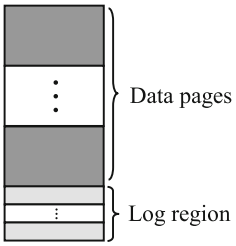

Fig. 10 IPL block organization

F-tree F-tree [19] is a B+tree with LZO compressed nodes, root excepted. Namely, when compression generates a node that occupies half page, the second half is left with 1s, so that its next update can be written there without erase. This operation is called *semi-write* and assumes that the underlying flash device supports turning 1s into 0s for "free." The main disadvantage of this method is its dependence on semi-write; even if supported, it involves complicated programming, as the authors noted. For this reason, the comparison with B+tree was conducted with simulation.

IPL B+tree and d-IPL B+tree According to In-Page Logging (IPL) scheme [89], every page is associated with a smaller logging area, and both are co-located in the same block, contrastingly to the usual logging policy of sequential appending. Thus, a block is divided into a sequence of pages, followed by the sequence of respective log areas (Fig. 10), each occupying a sector. So, IPL scheme depends on the capability of partial writing or partial programming a page with a number of independent sector writes. The total space of log areas is fixed. Every change related to a page is stored as a log entry in its log area. When a log area of the in memory buffer overflows, it is flushed to the corresponding block area in the flash memory. In case a block runs out of free log areas, all pages are read, merged with their log entries and then they are written to a new "fresh" block. Thus, IPL scheme follows the technique of replacing expensive writes with increased number of reads. Jin [69] and Jin et al. [70], aiming to exploit the multiple channels of modern SSDs, proposed extending IPL in the following way: m blocks are grouped together, each having k log pages. Writes are executed in a round-robin fashion within a group. Data are stored from top to bottom and logs from bottom to top. The parameters m and k are adjusted according to workloads, taking larger values when the writes and the file sizes are increasing.

Based on the IPL scheme, Na et al. [111] introduced IPL B+ tree, which is just a B+tree where every node is associated with a log area, both located in the same block. So, as it was noted above, the number of writes decreases at the expense of increased number of read operations. The authors employed an LRU policy to buffer the nodes and their respective log areas. Compared with B+tree, BFTL, and μ-tree, IPL

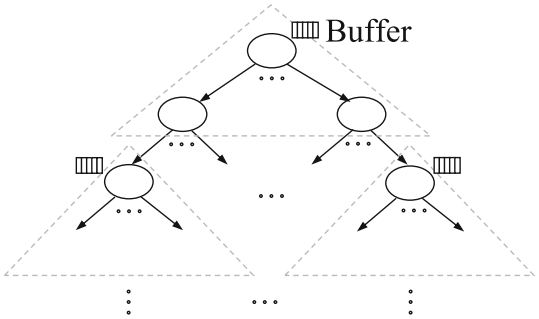

Fig. 11 LA-tree configuration

B+tree clearly prevails only against B+tree. Further, node splits cause frequent log overflows, since they must be represented by many log entries as there is no guarantee that sibling nodes reside in the same block. Also, in case the flash memory demands the pages of a block to be written consecutively, all free data pages are invalided as soon as the first log area is flushed, leading to underutilized blocks.

Dynamic In-Page Logging B+tree (d-IPL B+tree) [110] aims to tackle the weaknesses of IPL B+tree. In particular, the log area has variable size and is dynamically placed, based on the contiguous page write rule. Also, sibling nodes are stored in the same block: after a split, the new node is stored as *ghost*, i.e., as a series of log entries in the same block log area. When a split causes a block overflow, firstly the block is preemptively split, after applying all changes to the involved nodes, and then node splitting is carried out. In each resulting block, the log area occupies half the available space. Thus, the layout can be adjusted since new nodes are stored as ghosts. In case a log area overflows, the block is rewritten, after computing the new versions of the stored nodes. Additionally, node merging is not executed when underflow takes place. By simulation against IPL B+tree and BFTL, d-IPL B+tree found to outperform its opponents, while improving block utilization. However, the scheme still suffers from frequent flash memory reorganization and increased number of reads and erases.

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 (Fig. 11). 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, 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. The storage management procedure employs two equally sized partitions. Whenever the current partition is filled, the index is moved to the other partition and the former is erased. Overall, the scheme trades-off reduced update time for increased look-up time. It also assumes byte addressable raw flash. The experiments show performance gains over B+tree, BFTL, IPL B+tree and B+tree(ST).

AD-tree Adaptive Durable B+tree (AD-tree) [45] is a B+tree variant, designed for raw MLC flashes with partial programming constraints—block pages should be written sequentially. The nodes of the AD-tree are stored in areas of the flash, characterized as *cold-node* and *hot-node* buffers. Specifically, one buffer is reserved for cold pages. Initially, all nodes are considered cold and are placed according to level-order traversal. The hot-node buffers are dynamic allocated for pages with updates. Sibling nodes are placed in the same page, so they can absorb split or merge operations. On the other hand, nodes with fathers in the same page are placed in the same buffer (Fig. 12). Pages inside buffers are written in an append fashion, while (live) links are maintained between valid pages to facilitate node traversals. The authors propose two kind of reorganizations, local and global. The local one compacts a hot buffer when it becomes full, to speed up search among live pages. Periodically, when in at least 50% of the hot buffers the ratio of live pages linked by live links to fanout (termed *block filled factor—BFF*) equals or exceeds 100%, global reorganization takes place: cold pages, in the beginning of hot buffers, are moved to the cold buffer and hot pages are compacted to new (hot) blocks. Additionally, when reorganization takes place, the fanout is adjusted based on the block erase count (update intensive adjustment) or the response time (query intensive adjustment) since last reorganization. Experiments conducted on a emulator showed performance improvements against the LA-tree and the B+tree. However, the various reorganizations cause write overheads.

LSB+tree Jin et al. [71,72] used the In-Page Logging (IPL) scheme to design the Lazy Split B+tree (LSB+tree). The index utilizes a write buffer. The nodes of the LSB+tree are fat. They consist of a base node and at most k overflow ones. The latter are stored in the buffer and contain the changes the

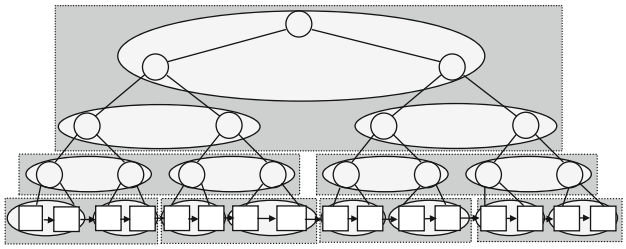

Fig. 12 AD-tree layout. Ellipses indicate a page. All nodes inside rectangles are placed in the same hot buffer

node undergoes (this feature attributes the qualification *lazy*). During a split, the modify-two-nodes policy is employed. Additionally, no merging or borrowing is executed in case of an underflow (lazy coalesce policy). A buffer entry (termed *buffer unit*) is categorized as *clean*, when it is a copy of the original node, as *semi-clean*, when it includes the original node plus the IUs, and *dirty*, if it comprises IUs only. A clean buffer entry can only become semi-clean. A semi-clean can turn into clean (full commit) or dirty (clean-part only commit). Also, the buffer is divided into two parts: the hot area, where hits occur, and the cold one, which consists of two lists, a clean/semi-clean and a dirty. During evictions, clean and semi-clean parts have the highest priority to get flushed; otherwise, a dirty entry according to minimum write criterion is chosen. When the buffer overflows, a number of entries are chosen to be flushed with minimum write criterion. A search for a key costs $2h$ reads, while the amortized cost of an insertion is $\frac{1}{n}(\frac{n_{lg}+n_{sp}}{n_{lgp}}+n_{sp})$, n the number of keys, n_{lg} the number of log entries, n_{lgp} the number of log entries per log page, and n_{sp} the number of split operations. The LSB+tree is compared with simulation against the B+tree, the BFTL, and the IPL B+tree, and was found to be both time and space efficient.

LSB-tree Log-structured B-tree (LSB-tree) [79] stores each leaf in one page and associates it with a log node, stored in a log area page. The log node holds the updates of its leaf. LSB-tree does not employ an in-memory write buffer, and thus, each modified node is written directly to flash. The mapping of leaves to log pages is carried out with an one-page-sized in-memory table. When a log node overflows, either it is merged with its leaf node, and the proper adjustments (like splitting) are made to the tree, or it is switched to a normal leaf, if its values are superset (in this case, the old leaf is invalidated) or smaller/bigger than those in the leaf (then, the respective entry in the association table is deleted). In case of mapping table overflow, the LRU entry is merged with its leaf. The OpenSSD platform was used to experimentally compare the approach with μ-tree, and it was found that LSB-tree is a winner, with the exception of writes where both indexes behave similarly.

Discussion Wandering tree adopts the very simple policy of rewriting all nodes along the insertion/deletion paths. Thus, a lot of writes and erasures are caused, leading to an increased need for space reclamation and diminished flash memory endurance. μ-tree resolves the problem of path rewriting by storing all updated nodes in a single page, using nodes of varying fan-out. On the other hand, it suffers from an increased number of splits (and so page writes), while small leaf sizes cause space overheads compared to B+tree. $\mu*$-tree tries to alleviate the shortcomings of μ-tree by adopting a dynamic layout scheme for pages that adjusts to workloads. Yet, this scheme still shows considerable number of writes due to node splits, space overheads, while the maximum number of entries that can be indexed, though improved compared to μ-tree, is upper bounded due to the restricted maximum height of the proposal.

IPL B+tree follows the approach of in-page logging to reduce the number of writes by increasing the number of reads. However, it addresses node splits poorly, while it shows space utilization problems. d-IPL B+tree attempts to deal with these drawbacks by employing a dynamic allocation scheme for log areas and applying extra procedures for block splitting and cleansing. Nonetheless, extra writing operations still occur due to reorganizations, while, as IPL B+tree does, it depends its operation on the capacity of partial programming.

LSB+tree also uses the in-page logging scheme and an elaborate buffer organization, jointly with (in memory) fat nodes, modify-two-nodes, and lazy coalesce policies. Due to the IPL scheme, the index, although improved, exhibits similar weaknesses to IPL B+tree.

LA-tree exploits the idea of attaching flash-resident buffers to tree nodes to minimize flash accesses. Since insert/delete operations find their way to the leaf level through repeated buffer emptyings and writings, write amplification and increased number of reads are not avoided. Additionally, LA-tree is designed mainly for byte-addressable raw flash memories.

LSB-tree also minimizes the need for main memory by storing all updates concerning a leaf to a dedicated log page. In this way, changes to leaves are postponed until the log page overflows, while the index can easily cope with power failures. However, the log page rewriting contributes to an increased number of writes.

The nodes of B+tree(ST) are in either read- or write-intensive mode. In the first case, they are normal B+tree nodes, while in the second one they use the technique of scattered logging. B+tree(ST) adapts to workloads with some delay and a switching cost. Its demands on main memory may be prohibitive for embedded systems.

AD-tree is designed for MLC flash memories. It uses two kind of flash-resident buffers, cold and hot, to aid wear leveling while contains the adverse effects of node splits and

Table 3 Raw flash B-trees design techniques and performance: big O notation is omitted,—denotes lack of cost analysis, $[\,]_*$ designates different costs, subscripts r, w stand for reads and writes, respectively

Index	Search	Insertion/Deletion	Space	Experimental Evaluation
SCATTERED LOGGING				
B+tree(ST) [114]	–	–	–	B+tree, BFTL
NODE MODIFICATION				
Wandering tree [13,43]	h	h	n	–
F-tree [19]	–	–	–	B+tree
NODE MODIFICATION & IN MEMORY BUFFERING				
μ-tree [90]	$-\log\frac{\sqrt{2}}{M}-\left(\sqrt{\log^2\frac{\sqrt{2}}{M}-2\log\frac{n}{2}}\right)$	–	$\lceil\frac{n}{M/2}\rceil$	B+tree
μ*-tree [77]	$\lceil e^{W_{-1}\left(\frac{\log p_1+\log M-\log n}{e^{\log(1-p_1)+\log M}}\right)}*e^{\log(1-p_1)+\log M}\rceil+1$	$(n_{\mathrm{u}}+\frac{(n_{\mathrm{u}}-e)n}{eMp_1})*n_{\mathrm{u}}^{-1}$	–	B+tree, BFTL
FLASH BUFFERING				
IPL B+tree [111]	–	–	–	B+tree, BFTL, μ-tree
LA-tree [2]	h	$[n_{sl}+h+\frac{8B}{M}]_{\mathrm{w}}+[n_{sn}+B]_{\mathrm{r}}$	–	B+tree, BFTL, IPL B+tree, B+tree(ST)
FLASH BUFFERING & NODE MODIFICATION				
d-IPL B+tree [110]	–	–	–	IPL B+tree, BFTL
LSB+tree [89]	$2h$	$\frac{1}{n}(\frac{n_{lg}+n_{sp}}{n_{lgp}}+n_{sp})$	$n+B$	B+tree, BFTL, IPL B+tree
AD-tree [45]	–	–	–	LA-tree, B+tree
LSB-tree [79]	$h+1$	–	$n+n_{\mathrm{lp}}$	μ-tree

merges by placing sibling nodes in the same page. The design achieves to prolong the life of the MLC flashes, and it is adaptable to workloads, however at the cost of write amplification. F-tree uses compression to create free space inside nodes to spare one erase before write. Howbeit, it is highly dependent on the *semi-write* operation which is not generally supported by flash memories. The design methods employed, the cost analysis and the experimental evaluation of the raw flash B-trees are presented in Table 3.

4.2 Skip lists

Skip list [121] is a probabilistic structure consisting of a logarithmic number of sorted linked lists, with high probability, called *levels*. The base level is just a sorted list of all keys. Each key in the i-th level is included in the level-(i+1) list with probability p. So, the keys of a list are samples (pivots) of those in the immediate lower level, partitioning it into a number of disjoint ranges. Each node of a list (termed *level node*) has two pointers, one to the next node at its level and one to the node containing its instance at the immediate lower level. A search operation starts from the highest level, following at each level pointers to the right while it finds smaller keys, and then descends to the next lower level. Alternatively, a skip list can be implemented using one node per key, termed *skip list node*, which has multiple right pointers, one per level it participates. This way, keys are stored only once.

Similarly to B-trees, one has to deal with small random writes when s/he accommodates skip lists in flash memory. There exist two SSD proposals for skip lists: the FlashSkipList and the Write-Optimized skip list. Both are using flash-resident write buffers whose contents are batch distributed to lower level. In FlashSkipList, a write buffer is dynamically associated with a skip list node at a certain level, based on its key distribution. Quite the opposite, the Write-Optimized skip list allocates buffer space inside level nodes. FlashSkipList is mainly experimentally evaluated, whereas Write-Optimized skip list is only (thoroughly) theoretically investigated, thus its practicality is left untested.

FlashSkipList Wang and Feng [142] introduced FlashSkipList (Fig. 13). It comprises three components: an in-memory write buffer, several flash-resident write buffers, termed write components, and a flash-resident read component. The in-memory write buffer is organized as a linked list of page-sized nodes. When it receives an incoming operation, it appends the respective IU at the end of the last node. When the write-buffer overflows, its contents are flushed and appended at the write component of the upper level, if any; otherwise, they are used to create its write component. Each write component is associated with a level and the range interval of a node of the read component. It is implemented as a linked list of pages, with upper bounded length, containing IU entries. The contents of a write component are distributed (pushed-down and appended) to the next level write components, using the read component infrastructure, when either its length reaches the upper bound or, upon serving a search operation, a two-competitive online algorithm decides that the distribution is more beneficial for the index performance. The read component is a modified skip-list that guides the various operations. In particular, it indexes chunks of items. Each chunk is essentially a fat leaf which consists of a number of consecutive (logical) pages, stores sorted items within a continuous range of values and is associated with a read component (skip list) node. Due to chunks, the skeleton skip list is kept small so that its maintenance cost is low. A skip list node may be associated with a number of write buffers, each at a different level, while its key value is set to the upper bound of the range of its associated chunk.

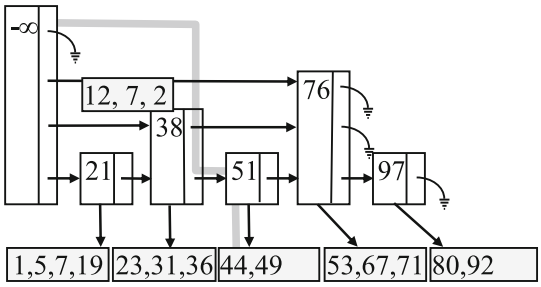

Fig. 13 A FlashSkipList; the write component (buffer) is associated with skip list node of 38 at second level. The search path of 45 is indicated

Searching is conducted top-down, starting from the highest level, as in the standard skip list. Additionally, at each level, the pertinent write components, if any, are queried. In the end, the procedure will end up in a chunk, which is also searched, using binary search. As pointed out, search operations may cause a number of write components to be reorganized. When a bottom level buffer is pushed-down, then its elements are merged into the respective chunks. In this work, only search time is theoretically analyzed, found to be $\log \frac{n}{s_{ch}} * s_{rc} + \log s_{ch}$, s_{ch} the maximum size of a chunk and s_{rc} the maximum size of a write component. Parameters that affect the read component, like s_{ch} and p, are only experimentally evaluated. Under various workloads, simulations showed that the FlashSkipList outperforms BFTL, FlashDB, LA-Tree, and FD-Tree. In short, FlashSkipList reduces the number of random writes, trading them with batch reads and writes. However, random writes are still used for the maintenance of the read component.

Write-optimized skip list Bender et al. [11] proposed write-optimized skip list for block devices, i.e., it is suitable for both HDDs and SSDs. Thus, the level nodes (Fig. 14) have size which is a multiple of page size B. Level-0 nodes constitute

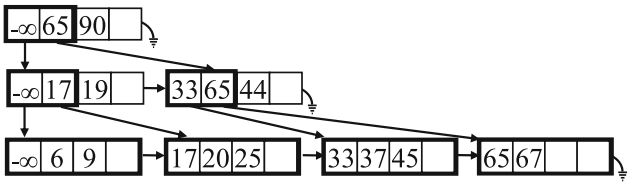

Fig. 14 A write-optimized skip list; each level node contains a write buffer (drawn with thinner contour)

the structure leaves. Each level node is partially filled with pivots, while it contains a write buffer of $\Theta(B)$ size. The smallest pivot in a level node is called *leader*. Every pivot has a pointer to the child level node that contains its copy. All keys in a write buffer are bigger than the leader of their level node and smaller than the leader of the next level node. The height (that is, the number of levels) of a key is determined by a hash function (meaning that even if the key is deleted and re-inserted, it will have the same height), by flipping a biased coin, the first time with probability $1/B^{1-\epsilon}$, $0 < \epsilon < 1$, and then with probability $1/B^{\epsilon}$. This guarantees that free space will be left for buffers in each level node. The highest level node is called *root* and stores $-\infty$. It is proven that the height of the structure is $O(\log_{B^\epsilon} n)$, in expectation and with high probability, the space complexity is $O(n/B)$ pages, both in expectation and with high probability, while a level node occupies $O(1)$ pages with high probability.

Searching is conducted by descending the appropriate path, from root level to level-0. Since the buffers contain elements that have not yet reached the bottom level, all nodes along the search path must be cached in-memory for proper processing, in $O(\log_{B^\epsilon} n)$ I/Os in expectation and with high probability. That calls for main memory with size greater than $B \log_{B^\epsilon} n$. Insertions and deletions are served by inserting a proper entry in the root buffer. When a buffer overflows, its elements are distributed among the children buffers, while themselves become pivots at the level node in question, pointing to the child node where their copy is forwarded. Additionally, if the height of the node is lower than the element's height, then the node is split and the element becomes the leader of the new node. If the flushed entry refers to a deletion, then the corresponding pivot is deleted and if it is also the leader, then the node is merged with its left sibling node. When elements are flushed to the leaf level, then the leaf elements are reorganized in $O(B)$ groups, each of them starting with a father pivot element and stored in a page. An insert or delete operation needs $O(\log_{B^\epsilon} n/B^{1-\epsilon})$ amortized accesses in expectation and with high probability, given that $\Omega(1)$ levels are constantly kept in main memory. The authors did not provide any experimental evaluation of their proposal. Write-optimized skip list poses certain demands on the size of the main memory, and the multiple copies of the pivot keys and their downward pointers contribute to increased space overhead. Also, the complicated insertion/deletion algorithm incurs increased batch read/write costs and may deteriorate the SSDs' lifespan.

4.3 Hash-based indexes

Hash-based indexes manipulate a set of data buckets. The buckets are organized with a table, usually termed as *directory*. The mapping of keys to buckets (i.e., the computation of the pertinent index) is done by a hash function, which ideally should distribute the keys with equal probability. For this reason, searching is quite fast. Due to its utilization in DBMSes, hash-based indexing was also considered when flash devices are used as secondary storage. Most proposals are adaptations of linear hashing, extendible hashing and Bloom Filters.

In linear hashing [101] the table is dynamic. It is expanded one bucket at a time, whenever the load factor exceeds a threshold. Additionally, the bucket pointed by a split pointer is split; split pointer advances in a linear order. When a bucket gets full, it acquires an overflow area. A search operation costs $O(1)$ accesses on the average case. On the other hand, extendible hashing does not use overflow buckets [44]; it always splits the overflowed bucket and, if necessary, doubles the directory. Due to the last action, several directory entries may refer to the same bucket. For this reason, each bucket has a *local depth* (minimum number of common prefix bits of stored keys), whereas the directory has a *global depth* (minimum number of common prefix bits necessary to differentiate the buckets). When a bucket with local depth equal to global depth overflows, the directory is doubled. Extendible hashing guarantees $O(1)$ worst-case access time; however, its directory size is super-linear.

Bloom Filter [16] is a probabilistic data structure that answers set membership queries with a false positive probability p_{fp}. Specifically, it is a bit array of m bits. When an item must be inserted or searched, k different hash functions are employed, each mapping the key to one array position, which, in case of an insertion is set to value 1. Parameters p_{fp}, m and k are related through a formula. Bloom Filters are very space and time efficient indexes.

Challenges in flash-aware hash indexes design Flash-aware hash-based schemes have to deal with in-place updates and small random writes. Buffering log records and massively committing them, techniques extensively used for tree indexes, usually have poor results, since the hash functions randomize the key distribution. Therefore, some approaches capitalize on in-memory buffering to pre-build or store index parts. Others serve updates by adding appropriate IUs to log areas inside or associated with buckets. Also, most schemes try to defer the expensive rebuilding operations, like, e.g.,

bucket splitting, using chains of overflow buckets, especially in write-intensive workloads.

4.3.1 FTL-based indexes

LS linear hash Lazy-Split (LS) Linear Hash [97] constitutes a flash-aware variation of Linear Hashing. Its main objective is to trade reads for writes. For this reason, firstly, deletions are executed by inserting an appropriate deletion record, and filtering is conducted during search; thus, index shrinking is avoided. Secondly, bucket splitting, after adding a new bucket due to the increased load factor, is deferred until the searching performance falls below a threshold. Then, the first $\max((b-1)-2^{\lfloor \log(b-1) \rfloor}, 2^{\lfloor \log(b-1) \rfloor}-1)$ of the b buckets are batched split; this happens to at most half of them. LS Linear Hash does not buffer insertions; thus, every such operation leads to a page update. Due to delayed splittings, a search operation may involve several buckets and their overflow areas in case the hash function directs it to an unsplit bucket. The authors do not provide any details about the parameter that triggers batched splits and, thus, adapts the index to workloads. Also, there is no cost analysis for neither update operations nor space allocation. The scheme is only compared with simulation against standard linear hashing (termed Eager-Splitting in the paper).

HashTree In [38] a hybrid scheme, called HashTree, was presented. It consists of two parts, one stored in main memory and one in flash memory. Namely, a linear hashing table and write buffers for each table bucket are maintained into RAM. The items of each table bucket are flash-resident, accommodated either in an overflow page and/or an FTree. FTree is an FD-tree variant with variable but upper bounded logarithmic size ratio f, and a critical depth d, which is the maximum depth value with accepted performance. FTree increases its logarithmic size ratio when the critical depth is reached and f did not reach its maximum value; otherwise, a new level is generated.

When the write buffer overflows, the items of the biggest bucket buffer are chosen for flushing. The commit procedure first tries to insert them into the overflow page. If this is impossible, then the overflow entries along with the committed ones are used to build an FTree from scratch, if such structure does not exist, so that all entries are accommodated to only one level of minimum depth. Otherwise, the items are recursively merged-inserted into the corresponding FTree, starting from the second level. If during this procedure a new level is about to be created, the depth of the FTree is checked. If it is already greater than or reaches d, and the FTree belongs to the next-to-be-split bucket, then the procedure stops and the bucket is split. Otherwise, in case f can be increased, the FTree is rebuild. HashTree was experimentally compared with BFTL and FD-tree, and found to prevail. Although hashing distributes the items in a number of smaller FD-trees, still the scheme suffers from a large number of sequential reads and writes.

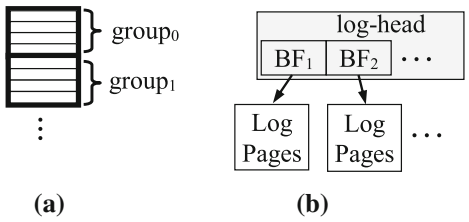

Fig. 15 SAL-hashing: **a** groups of four consecutive pages; **b** log area organization

SAL-hashing SAL-hashing [67,156] is a flash-aware linear hashing scheme, which adapts to access patterns, exploiting additionally the internal parallelization of modern SSDs to batch commit a number of small writes. The buckets of linear hashing are organized into groups (i.e., fixed sized collections of g consecutive buckets) to group small writes (Fig. 15a) into course-grained ones. Each group is either an empty-group (newly added), or a split-group (it was split), or a lazy-split-group (split was deferred). The groups are classified into sets. This permits to apply different split policies to each set and thus adapting it to access patterns. To this end, a set is categorized as being either a lazy-split-set (consisting of one lazy-split-group and one or more empty-groups; no keys are distributed to the newly added groups) or a split-set (consisting of one split-group). When a lazy-split-set splits, the keys of the lazy-split-group are distributed among the group members and, thus, it is split into a number of split-sets (i.e., split-groups). With each set, a flash-resident log area is associated. The log area consists of a log-head page, holding a number of Bloom Filters indexing the log pages (Fig. 15b).

Insertions, deletions, and updates are handled by inserting a pertinent log entry to an in-memory log buffer. The log buffer is organized as a collection of sub-buffers, each accommodating logs belonging to a specific set. Each sub-buffer comprises a number of hash structures to speed up look-ups. Every time the log-buffer overflows, the log entries of the maximum sized sub-buffer are batch flushed and appended to the corresponding log area, and the associated BFs are appropriately updated. The log area is merged with the main buckets, along with the entries (if any) of the respective sub-buffer, when it is full or whenever a ski-rental based online algorithm decides it is time to be merged for performance reasons. When the load factor of the scheme exceeds an upper bound, then a new group is added, becoming a member of a lazy-split-set. A search operation involves looking, first the

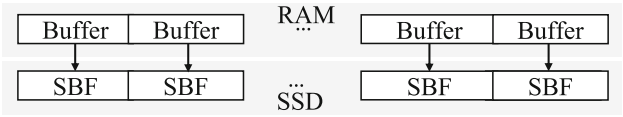

Fig. 16 A BBF instance

log buffer, then the pages in the pertinent set's log area—in reverse order since pages at the back contain the most recent log entries —, and last, if necessary, the bucket itself.

The authors analyzed both theoretically and experimentally the effect of the various SAL-hashing parameters. In the worst case, the update cost is $\frac{(r+4)*g_n}{B*s_p-2*g_n} * (2 + \frac{g}{n} + p_u * g)$, r the record size, g_n the number of groups, B the buffer size, s_p the page size, and p_u a probability parameter related with whether a record resides in a log area. A search operation involves, in the average worst case, $2 + n_{lp} * p_{fp}$ reads, p_{fp} the false positive probability and n_{lp} the number of pages in the log area. Experiments were conducted against B^w_Δ-tree (a Bw-tree implementation), LS Linear Hash, and standard linear and extendible hashing, showing that SAL-hashing is update efficient and achieves compatible search performance with extendible hashing.

BBF and BloomFlash Buffered Bloom Filter (BBF) [21] and BloomFlash [41] are two similar approaches for accommodating Bloom Filters in SSDs (Fig. 16). Given the maximum number of keys that will ever be inserted and the length of the standard BF array BFA, they partition BFA into n_{sbf} independent sub Bloom Filters (SBFs), each fitting in exactly one page. All SBFs are stored in consecutive (logical) pages. In this setup, the keys are mapped to the SBF they belong by a hash function h. Assuming that h assigns keys to SBFs with equal probability, it is proven that the false positive probability of each SBF is the same as if the keys were accommodated in a BF of n_{sbf} times bigger size. So, a search operation requires one page read, whereas an insertion causes only one page write, instead of k page accesses in the case of a single BF. To reduce the number of writes, an in-memory sub-buffer is associated with each SBF [21]. Thus, an insertion is firstly accommodated in the respective sub-buffer. In case of sub-buffer overflow, buffered bit changes are committed to the respective page. The same approach is followed to batch search requests; to limit the delay, buffers are flushed after a predefined time period whether they are full or not. [41] reserves one in-memory buffer for all SBFs and considers two alternative flushing policies : (i) the SBF containing the largest number of updates is selected as victim, and (ii) SBFs are updated in sequential order, one at a time. Their performance depends on the workloads and the underlying SSD. In both approaches, I/O cost is amortized to the maximum number of buffered operations n_{bf}. BBF and BloomFlash are experimentally evaluated against standard BF.

FBF In [102] Forest-structured Bloom Filter (FBF) is proposed, based on BBF and BloomFlash. Specifically, FBF is a dynamic tree-structured collection of independent sub Bloom Filters (SBFs). FBF's initial set up is the same as the one of a BBF/BloomFlash, that is a series of page-sized SBF, consisting the level one layer. Every group of δ consecutive SBFs are stored in one block, taking up λ blocks. When the level one layer reaches its limits with respect to false positive probability p_{fp}, b new child structures are introduced per block, adding thus a new second level. As soon as the capacity of the second level layer with respect to p_{fp} is reached, then a new third layer is added, in the same way (i.e., each 2-level layer block acquires b children), and so on. Buffering is employed only to the last level and in case of an overflow, the dirtiest block is flushed. Consequently, searching, after has failed in buffer area, proceeds level by level. To this end, two extra hash functions are employed; the first function decides which block is pertinent, and the second one returns the relevant SBF inside the block. Insertions are allowed only at the last level. FBF was experimentally investigated against the BBF/BloomFlash approach and linear BF, a dynamic version of standard BF, which, every time BF reaches its capacity, it is flushed and a new BF is initialized, producing thus a chain of flash-resident BFs. Despite the fact that FBF demands a logarithmic number of page accesses in the worst case, it was found to outperform the other approaches.

4.3.2 Raw flash indexes

MicroHash MicroHash [100] is a specialized hash-based index, designed for time-stamped sensor data. The data are circularly stored in a very small sized raw flash. Therefore, no deletions are executed. Since the flash pages are circularly reclaimed, this also acts as a free provision for wear leveling. Thus, the indexed space is divided into a number of buckets, each representing a continuous range of values, initially of equal size. The number of ranges (buckets) is determined by the fact that the directory (hash table) must reside in one page. Each bucket is connected to a list of index pages, in newest-to-oldest order. An index page comprises index records which point to the data pages where the corresponding measurements (data records) reside.

The data are maintained in an one-page write buffer. When the buffer overflows, before flushing it, the scheme creates index records which are associated with the proper buckets and stored into index pages. When the length of a bucket list exceeds an upper bound, then the least recently used bucket is written to flash memory, and the bucket is divided into two buckets of half range. Since the write operation is costly, the entries of the initial list are not distributed between the two buckets. On the contrary, they stay linked to the old bucket, which from now on will be associated with values of finer

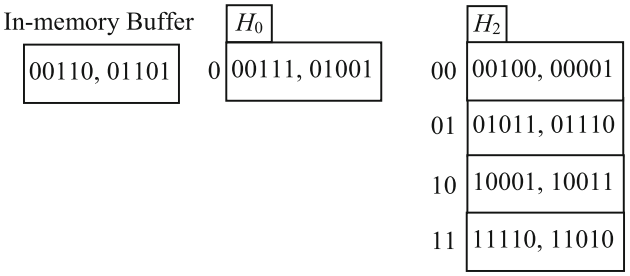

Fig. 17 MLDH: H_1 is next to be built

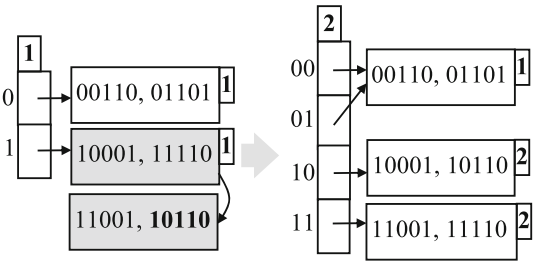

Fig. 18 h-Hash overflow: merge-split and directory doubling

granularity. MicroHash can support two kinds of queries: equality by value and timestamp-based equality, and range search by querying the directory and properly scanning the linked lists. The authors provided a number of experimental evaluations as well as design alternatives, like data or index compression, elf-like chaining [39].

DiskTrie DiskTrie [35] discusses how to store static LPC-Tries [6] in NOR and NAND flash devices. LPC-trie is a static trie where both path and level compression are employed: the former removes all nodes with only one child, whereas the latter replaces complete subtrees of level i by a single node with fanout 2^i. Firstly, the index is built in main memory, and then flushed to flash memory. Internal nodes are stored in NOR flash memory to exploit byte level addressability. The placement is conducted by a linearization procedure, which puts sibling nodes in consecutive locations. The data leaves are stored sequentially in the NAND flash memory, after a lexicographical sorting. The authors treated the time and space index performance only theoretically, while they discussed the necessary alterations when only NAND flash is available. Specifically, the storage is linear in the number n of strings, while a search operation needs $\log^* n$ disk accesses.

Multilevel dynamic hash Yang et al. [157] apply the logarithmic method to dynamize a static hash table. The approach is called Multilevel Dynamic Hash (MLDH) and consists of an in-memory bucket-sized index and a series of static hash tables H_i, where i is the level number (Fig. 17). Each H_i consists of 2^i buckets. When the load factor of the in-memory bucket reaches the predefined upper limit, it is flushed. Specifically, it is merged with the first k hash tables $H_0, H_1, \ldots, H_{k-1}, H_k$ with consecutive level numbers, such that $H_i \neq \emptyset$ and $H_{k+1} = \emptyset$, to produce H_{k+1}. During the merge operation, the buckets of each H_i are increasingly read in k dedicated in-memory read buffers M_i. The buckets of H_{k+1} are constructed in 2^{k+1} rounds, in increasing order. Before starting the construction of the l-th bucket, from each M_i all keys that do not share the first i bits with l are removed, and the next bucket from H_i is fetched into M_i. So, the main memory demands are not negligible. When items with same key are met during merge, the one from the lowest level is maintained. Deletions are handled by inserting proper deletions records. When bucket utilization falls below a predefined threshold, then reorganization is employed. MLDH exhibits $O(\log n)$ search time and $O(n \log n)$ construction time, n the number of keys. It was experimentally compared with the static hashing scheme on a NAND flash simulator, suggesting caching top level hash tables in main memory for sparing flash writes.

SA extendible hash Self-adaptive (SA) Extendible Hash was proposed in [145] as a flash-aware variation of the standard scheme. Each bucket, located in a block, is divided into a data and log area, without strict size limitations, contrary to the IPL proposal [89]. An update operation is executed by writing proper log entries in the respective block log area. When a bucket overflows, it is either merged or split, if the ratio of the number of log entries to the number of data entries exceeds or not a threshold SM. A merge operation, after canceling out entries concerning the same keys, writes the (cleaned) data back to the erased block. The split operation is executed as in the standard extendible hashing, after merging the data with the log entries. Thus, merges produce bigger data regions, whereas splits extend the log area size. The value of the SM parameter is dynamically changed after each merge or split, independently for each bucket. It was found by simulation, that SA Extendible Hash saves some erase operations in comparison with the standard version.

h-Hash index Kim et al. [80] introduced the structure of Hybrid Hash (h-Hash), an extendible hashing variant, where each bucket is chained with an upper-bounded number of overflow pages. An insertion or a deletion operation is served by writing proper IU entries. A bucket overflows when the maximum number of overflow pages are allocated and all are full, the main bucket included. In that case, based on whether the ratio number of updates to the number of deletes exceeds or not a threshold, the bucket is merged (i.e., IUs are applied to the data) or it is split after applying merging (Fig. 18). As a result, an insertion costs $\frac{n_{pb}}{2} + 2$ reads, 3 writes and $\frac{2}{n_{pb}}$ block erasures in the worst case, n_{pb} the size of bucket, and a

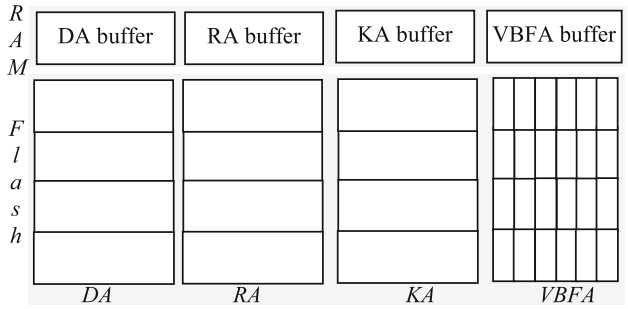

Fig. 19 The PBFilter organization

search needs $\frac{n_{pb}}{2}$ reads on average. The authors implemented h-Hash as a block mapping algorithm of FTL in OpenSSD platform, and investigated parameters like the merge ratio and the maximum number of overflow pages. They also compared h-hash with the standard extendible hashing, MLDH and LS linear hash, and found that it achieves the best overall performance. Only its searching performance is worse than that of standard extendible hashing due to the overflow page chains.

PBFilter PBFilter [160] aims to transform updates into append-only operations, so that writes are performed sequentially. In particular, it employs a sequential organization (Fig. 19): each inserted record is appended to the end of the record area (RA), whereas an insertion index is added to the end of the key area (KA). A deletion appends a proper deletion record to the end of the deletion area (DA). Thus, after searching (scanning) the key and the deletion area, one can find where the record with a certain key resides, if it exists. To speed up searching, a vertically Bloom Filter (VBFA) is build, in an incremental way, to find out the best number of partitions with minimum memory demands. As an improvement, k mini Bloom Filters are also suggested. The authors provided only theoretical analysis of their proposal against BFTL, B-tree and hash index as implemented in SkimpyStash [40]. So, a search operation, on average, needs $R_1+R_2+R_3+R_4$ reads, where $R_1 = \lfloor (N_{FB} - P_f)/N_{FB} \rfloor * \lceil (N_{FB}/L1) \bmod s)/2 \rceil$, R_2 the average number of partitions involved, $R_3 = \lceil N/(S_p * 8 * M_2)/2 \rceil *$ average number of BF activations, $R_4 \lceil p_{fp} * N_{KA} * \lceil M_1/M \rceil /2 \rceil$, N_{KA} the number of index key pages, M_1 the number of keys related with one BF, M the number of IUs in one page, N_{FB} the number of pages for storing BFs, P_f the number of partitions, S_p the page size, p_{fp} the false positive probability. Additionally, inserting N keys causes $N_{KA} + N_{FB} + N_E$ writes, N_E the total number of page erasures.

4.3.3 Discussion

LS Linear Hash delays bucket splitting until the search efficiency falls below a predefined level. Then, it executes splits in batches. So, writes are traded for reads. However, the discussion lacks details about tuning its parameters and the analysis is incomplete. HashTree uses linear hashing to distribute items into a number of FD-trees of bounded height. Thus, it transforms random writes into sequential ones. Although the involved trees are of smaller size, still the scheme exhibits considerable number of reads and writes. SAL-hashing groups buckets and uses log areas and fine-grained deferred splitting to modify linear hashing so that it can capitalize on the excellent performance of massive I/O writes. Although it introduces an online algorithm to merge log areas back to buckets, the scheme still has increased search times, while the extra write operations may affect the device lifespan.

BBF and BloomFlash are BF variants that localize reads and writes by employing a number of SBFs and buffering updates so that they involve one page read and write, amortized over a number of operations. The main disadvantage in both designs is the assumed knowledge of the maximum number of keys that may be stored. FBF proposes the dynamization of BBF and BloomFlash by organizing them in a number of hierarchical levels of consecutive blocks of several one-page-sized SBFs. When a level reaches its capacity w.r.t performance, then another one is generated; each block acquires b children blocks.

MicroHash is a specialized hash-based index for storing and searching time-stamped data in wireless sensor nodes of limited hardware. Thus, it can not be used as a general-purpose data structure, since it lacks scalability. Disktrie considers storing compressed tries in NOR and NAND flash devices. It only applies to static collections of strings, it depends on main memory for building the index, while its applicability is not experimentally evaluated. MLDH employs the logarithmic technique, i.e., it maintains a series of increasing sized static hash tables to deliver worst case logarithmic search and amortized insertion/deletion times. Therefore, it uses an increased number of (sequential) reads and writes to deal with small random writes. SA Extendible Hash follows the approach of storing data and IUs in the same bucket. For this reason, it introduces, complementary to split, the merge operation. Which of split or merge will be executed in a bucket is decided dynamically and independently; however, both they cause block erasures. All IUs are immediately stored into the pertinent log area, a fact that increases the write amplification while triggering frequent block erasures. h-Hash also uses merging and splitting to reorganize locally the buckets of an extendible hashing scheme in conjunction with overflow buckets which delay the split operations but increase search time. Since it immediately writes a new entry to a flash page, h-hash incurs frequent writes and erasures. PBFilter uses BFs to build a purely sequential structure, mainly designed for indexing small data sets in embedded

Table 4 Hash indexes design methods and performance: Big O notation is omitted,—denotes lack of cost analysis, $[\,]_*$ designates different costs, subscripts r, w, e stand for reads, writes, and block erases, respectively

Index	Search	Insertion/Deletion	Space	Experimental Evaluation
FTL				
Bucket Modification				
LS Linear Hash [97]	–	–	–	Linear hashing
Flash Buffering & Bucket Modification				
HashTree [38]	–	–	–	BFTL, FD-tree
SAL-hashing [67,156]	$2 + n_{\text{lp}} * p_{\text{fp}}$	$\frac{(r+4)*g_n}{B*s_{\text{p}}-2*g_n} * (2 + \frac{g}{n} + p_{\text{u}} * g)$	n	B^w_Δ-tree, LS Linear Hash, linear & Extendible hashing
In Memory Buffering & Array Modification				
BBF [21], BloomFlash [41]	$1/n_{bf}$, 1	$1/n_{bf}$	$n + B$	Standard BF
FBF [102]	$\log_b n$	$1/B$	$n + B$	BBF, BloomFlash, linear BF
RAW FLASH				
Flash Buffering & Bucket Modification				
SA Extendible Hash [145]	–	–	–	Extendible hashing
Bucket Modification				
h-Hash [80]	$\frac{n_{pb}}{2}$	$[\frac{n_{pb}}{2} + 2]_{\text{r}} + [3]_{\text{w}} + [\frac{2}{n_{pb}}]_e$	–	Extendible hashing, MLDH, LS Linear Hash
In Memory Buffering & Bucket Modification				
MicroHash [100]	–	–	–	ELC [39]
In Memory Buffering				
DiskTrie [35]	$\log^* n$	–	$n + n$	–
MLDH [157]	$\log n$	$\log n$	$n + B$	Static hashing
PBFilter [160]	$R_1 + R_2 + R_3 + R_4$	$(N_{KA} + N_{FB} + N_E)/N$	$N + N_{FB}$	–

systems. PBFilter is only theoretically evaluated against other indexes.

Table 4 summarizes the design methods employed and the performance characteristics of the flash-aware hash indexes discussed.

5 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. [49].

Challenges in flash-aware spatial indexing design Consecutive insertions of spatial objects in leaf nodes (or data buckets) force reconstruction 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. More recent ones are targeting to the spatial queries' efficiency also, exploiting in-memory read buffers and batch read operations. Next, we present several works concerning flash-aware spatial indexes that fall in the aforementioned classes.

5.1 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 [122] is an extended B-tree to support high dimensional data, which retains its balance while adapting well to the shape of data. Grid File [115] is a multidimensional hashing structure that partitions space using an orthogonal grid. Finally, the XBR$^+$-tree [130] is a balanced index belonging to the Quadtree family [48].

5.1.1 FTL-based indexes

F-KDB F-KDB [94] 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 write 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 online 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 IU records in a flash page. F-KDB, contrary to almost all other studies [26,65,131,150], 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.

GFFM and LB-Grid Fevgas and Bozanis [46] presented GFFM, a first study on the performance of the two-level Grid File in flash-based storage. A buffering strategy that evicts the coldest pages first is adopted. 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 parallelization of contemporary SSDs. GFFM outperforms R*-tree in different experimental scenarios with varying insert/search ratios. Moreover, increasing the size of write buffer, and consequently the number of outstanding I/O operations, leads to remarkable performance gains. The authors do not discuss the cost of read and update operations. However, it is easily deduced that the cost of a read operation remains 2.

The LB-Grid [47] is another Grid File variant. It exploits scattered logging to reduce small random writes at the buckets' level. It exploits 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, which exploit the high IOPS, the internal parallelization of SSDs, and the efficiency of the NVMe protocol. The proposed query algorithms are also adapted for the GFFM. Extensive experiments on three different devices (2 NVMe, 1 SATA) are presented using one real dataset of 500M points and two synthetic ones of 50M points each, evaluating LB-Grid against GFFM, R*-Tree and FAST R-Tree (described below). 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 $1 + S_{\text{r}} * P_{sm} * \varpi^{-1}$ reads and $S_{\text{w}} * P_{sm} * \varpi^{-1}$ 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. The authors also describe the cost of range, kNN and group point queries. LB-Grid outperforms its competitors in all update intensive workloads, while it presents adequate performance in the read dominated ones. On the other hand, GFFM achieves significant performance gains utilizing the proposed flash-efficient query algorithms.

xBR$^+$-tree In [129], three flash-efficient XBR$^+$-tree algorithms for batch spatial query processing are presented. In particular, point location, window, and distance range queries are studied. The authors aim to accelerate query performance by exploiting the internal parallelization of SSDs. Thus, they partition query space into fragments that are submitted at once to the flash device. This way, they achieve a high number of outstanding I/O operations that maximizes SSD efficiency. The procedure of partitioning query space and composing batch queries is recursively repeated from the root to the tree leaves. The proposed algorithms achieve remarkable performance gains over the original ones, especially in experiments that utilize large datasets. Moreover, [128] presents algorithms for bulk-loading and bulk-insertions that outperform the previous HDD based ones by a significant margin. Nonetheless, a comparison with other spatial indexes, flash oriented or not, is not provided. An initial effort to integrate XBR$^+$-tree into the eFind generic framework (Sect. 6) is discussed in [27]. So, eFind XBR$^+$-tree outperforms an implementation that employs the FAST generic framework. However, it is not evaluated against the original XBR$^+$-tree.

5.1.2 Raw-flash indexes

MicroGF MicroGF [100] 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 [115] in their analysis and experimentation. However, a two-level approach [60] would be more appropriate.

5.2 Spatial access methods

R-tree [56] and its variants are the most popular SAMs, utilized in a wide range of data management applications such as spatial, temporal and multimedia databases [104]. 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 $\mathcal{T}$, MBR of $\mathcal{T}$). The MBR of a tree $\mathcal{T}$ 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 toward 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. Several variants of R-tree have been introduced aiming to enhance its performance; one of the most prominent and efficient is the R*-tree [10]. The R*-tree exploits several heuristics to enhance query performance, such as reinsertion of data, MBR adjusting and optimizations for node splits.

5.2.1 FTL-based indexes

RFTL RFTL [150] is a flash-efficient R-tree implementation, equivalent to BFTL [149,151]. 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 [103] 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 rewriting 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 Pawlik and Macyna [119] presented 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) [65,146] 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.

5.3 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. A recent work that also exploits scattered logging to maintain updates in data buckets, is LB-Grid. FOR-Tree and special purpose MicroGF use 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, LB-Grid, and GFFM are representative examples. The high IOPS of modern PCIe SSDs and the efficiency of NVMe protocol made the batch processing of spatial queries possible. XBR$^+$-Tree and LB-Grid are two such works that achieve significant performance gains during the execution of range, kNN (LB-Grid) and group point queries.

Table 5 recapitulates the design methods employed and the performance characteristics of the flash-aware spatial indexes presented.

6 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 FAST [131,132] is a generic framework that exploits the In-Memory Buffering technique. It has been successfully tested with both one- (B-tree) and multi-dimensional (R-tree) indexes. It uses deltas to log 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, they are audited into a sequential written log held in flash, to facilitate recovery after a system crash. The records 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 [23,24] 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 conducted experiments indicated that (i) FAST versions exhibit faster construction times, (ii) query performance depends on the

Table 5 Multidimensional indexes performance: Big O notation is omitted,—denotes lack of cost analysis, $[\,]_*$ designates different costs, subscripts r, w stand for reads and writes, respectively

Index	Search	Insertion/Deletion	Space	Experimental Evaluation
FTL PAM				
Scattered Logging				
F-KDB [94]	$h * c$	$\tilde{N}_{\text{split}} + 2/n_{\log}$	–	KDB-Tree
Scattered Logging & In-memory Batch Read Buffering				
LB-Grid [47]	$1 + L/\varpi$	$[1 + S_{\text{r}} * P_{sm} * \varpi^{-1}]_{\text{r}} + [S_{\text{w}} * P_{sm} * \varpi^{-1}]_{\text{w}}$	–	GFFM, R*Tree, FAST-RTree
In Memory Buffering & In Memory Batch Read Buffering				
GFFM [46]	2	–	–	R*Tree
XBR$^+$-tree [128,129]	–	–	–	–
RAW FLASH PAM				
In Memory Buffering & Bucket Modification				
MicroGF [100]	–	–	–	Qaudtree, Grid File
FTL SAM				
Scattered Logging				
RFTL [150]	$h * c$	$\frac{2*n}{M-1} + \tilde{N}_{\text{split}}$	$n * c + B$	R-Tree
Scattered Logging & Node Modification				
Flash aR-tree [119]	$2 * h * (c + 1)$	$\frac{2*h}{r}$	$n * c + B$	RFTL-aRtree
Flash Buffering				
LCR-tree [103]	–	–	–	RFTL, R-Tree
In Memory Buffering & Node Modification				
FOR-tree [65,146]	–	–	–	R-Tree, FAST-RTree

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 [25,26]. 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. eFind employs the In-Memory Buffering and In-Memory Batch Read Buffering techniques to turn any index into a flash-efficient one.

The eFind framework is composed of three components that control buffering, page flushing and logging—deltas are maintained in-memory. 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 (Fig. 20). 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.

7 Inverted indexes

The inverted index (alternatively inverted file, or postings file) is the most wide-accepted structure for indexing text documents [161]. Its principle is quite simple: for each term (word), a list of document identifiers containing it, termed *posting list*, is maintained. Inverted indexes have been thoroughly investigated in the context of HDD based systems. During the last years, few works dealt with the subject of building and online maintaining such structures in flash devices. Buffering in main memory, to contain small random writes, is the common technique one can discern

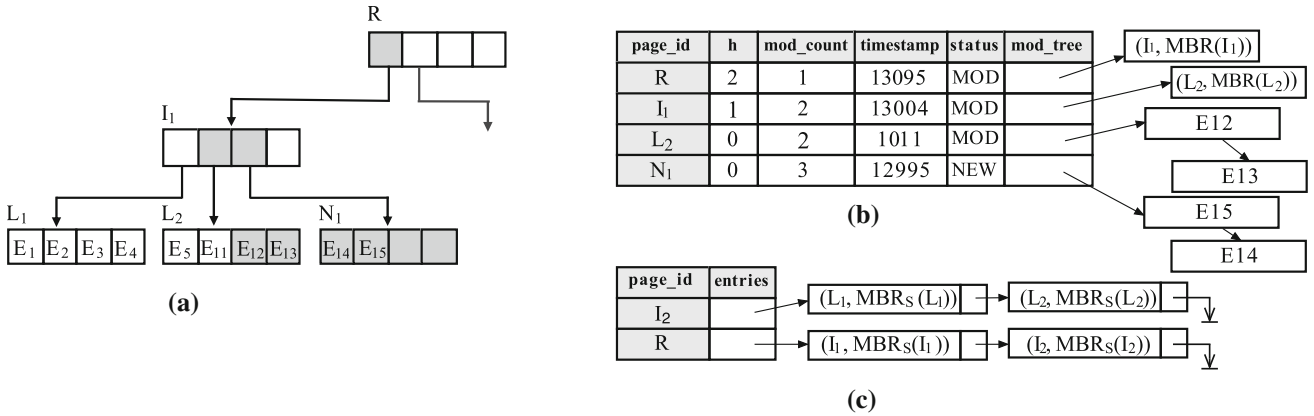

Fig. 20 eFind: **a** The underling R-tree; five new entries E_{12}–E_{15} are inserted to the tree, leading to the introduction of node N_1 and updates to nodes L_2 and I_1. **b** The applied updates are represented in the write buffer table. **c** The read buffer table holds recently fetched nodes

in these approaches, combined with well-established HDD based methods.

7.1 FTL-based approaches

Hybrid merge Li et al. [96] introduced a hybrid method for online inverted index maintenance, aiming to reduce writes and avoid random accesses. The main index resides in RAM and when it overflows, it is split into two parts. The first one includes the long posting lists, whereas the latter accommodates the short ones. The long lists are merged immediately (each posting list is stored as a separate file) with the respective lists accommodated on SSD. The short lists are stored separately without merging, since their scattering does not affect the performance. Moreover, the method provides the option of selective flushing, which flushes a certain amount P_f of data to SSD. If the aggregate size of short terms is at least P_f, then they are flushed in increasing sizes. Otherwise, long posting lists are flushed and merged to flash memory, in decreasing size, until reaching the P_f bound. The authors experimentally evaluated their approaches against no merge (NM)—a new inverted index is built for new terms and flushed—, logarithmic merge (LM) and geometric partitioning (GP) policies, all used for HDD based systems. Hybrid Merge has slightly worst maintenance cost than NM, while attains the same or better query performance compared to LM and GP policies.

MFIS Multi-path Flash In-place Strategy (MFIS) [76] uses Psync I/O to parallelize the read/write operations. Moreover, the posting lists are stored non-contiguously to exploit SSDs' internal parallelization. When the in-memory inverted index overflows, then (i) the new terms are copied to the output buffers, (ii) the last block (pages) of the old terms are read in parallel with Psync to the input buffers, until the input buffers are full, and merged with in memory postings into output buffers, and (iii) the old blocks are rewritten to same (logical) addresses, while the new blocks are appended to the end of the file with Psync. During query processing, all postings of a term are parallel read using Psync I/O. MFIS is thoroughly compared with various strategies and found to prevail.

7.2 Raw flash approaches

Microsearch Microsearch proposal [137] concerns building an inverted index for the constraint environment of a mote. This method maps multiple terms into the same inverted index slot. As a result, the generated index is less accurate but of smaller size. The flash memory is written cyclically. Microsearch employs an in-memory write-buffer, where the terms of each newly added document are inserted. When the buffer overflows, the terms are grouped together according to the slot they belong and the largest group is flushed. Specifically, they are either added to the newest (head) page of the posting list, or stored in a new page, which becomes the new head (Fig. 21). Thus, the posting lists are reverse pointed, forming essentially a stack of pages which may contain multiple terms. The queries are answered by employing a two-stage document-at-a-time style, reserving one-page size buffer for each term. First, data are loaded for the in-memory write buffer and then for the flash pages. During the first stage, document frequencies (DF) are accumulated. The second stage calculates term frequency/inverse document frequency (TF/IDF) scores. This approach capitalizes on the fact that the documents are stored in increasing addresses during insertions, so when posting pages are loaded, the addresses are decreasing. It is proven that inserting D documents requires $\frac{D*m}{x}$ reads and $\lfloor \frac{D*m}{E'} \rfloor$ writes, m the average number of terms per document, x the number of terms flushed per overflow,

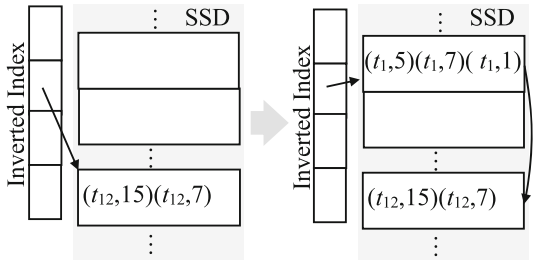

Fig. 21 Microsearch: new page head creation

and E' the number of terms stored per page. Also, a query involving t terms requires at most $\frac{2*D*m}{E'*H}$ reads, H the number of main directory slots. Experimental analysis for various system parameters is also provided.

Flashsearch In [22], the authors propose two solutions for maintaining inverted indexes in mobile devices. Both solutions operate by organizing the entries of the main directory into groups. The first one creates a stack (reverse list) of pages which contain documents belonging to the same group. In addition, each inverted list maintains its own in-memory buffer for receiving new postings. When the buffer becomes full, it is flushed to flash memory. In case the memory is limited, the method employs an LRU policy for accommodating buffers to groups. On the other hand, the second solution dictates the usage of a common buffer for all groups. When the buffer becomes full, the entries belonging to the same group are linked together and then flushed; in this way, a page can contain terms from different groups. The evaluation of these solutions with simulation indicated that the former is better for high query loads, whereas the latter is recommended for low query loads since involves reading more pages to locate the pertinent data.

7.3 Discussion

Both Hybrid Merge and MFIS deal with maintaining a growing inverted index on SSD. The first employs different policies for posting lists, depending on their length: the short ones are treated with NM, while the long ones with immediate merge—please notice that both of them are considered extreme cases for HDD storage. In this way, they restrict the expensive w.r.t. writes merging method to the subset of terms that they are frequently appeared in queries to the detriment of less occurring. MFIS also employs a not so popular technique, i.e., non-contiguous in-place updates. For this, it capitalizes on the internal parallelization of modern SSDs by issuing massive I/Os with large granularity. However, it is based on Psync I/O which, as discussed earlier, is not supported by OS kernels. In a nutshell, Hybrid Merge and MFIS do not avoid write amplification.

Table 6 Flash-aware Inverted Indexes employed techniques: IM stands for In-Memory

Technique	Index
FTL	
IM Buffering & Merging	Hybrid Merge [96]
IM Buffering & In-Place Merging	MFIS [76]
RAW FLASH	
IM Buffering & Term Grouping	Microsearch [137], Flashsearch [22]

Microsearch and Flashsearch are designed for resource-constrained devices and follow the same approach of mapping more than one terms to each directory entry. This translates to increased number of reads for serving user queries. Table 6 sums up the works presented.

8 Indexing and new SSD technologies

Nowadays, there is an increasing demand for high performance storage services. Emerging technologies like NVMe, 3DXPoint and other nonvolatile 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 [112]. 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.

8.1 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; this fact does not longer hold.

Up to now, Linux AIO has been successfully utilized to accelerate the performance of one- and multi-dimensional indexes [47,125] exploiting the high bandwidth and the internal parallelization of SSDs. However, the advances in nonvolatile 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 (μs) for 3DXPoint, (μs) $for Z-NAND, >$ 70(μs) for commodity NAND SSDs.

The authors in [86] 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 [159]. 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 [159]. Recently, SPDK has been successfully used to enhance the performance of a key-value store [86]. Another similar framework for user-space I/O is NVMeDirect [82], 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, just as SSDs/HDDs have been previously used [68].

8.2 Open-channel architecture

The increasing adoption of SSDs in the enterprise data centers has introduced demands for high resource utilization and predictable latency [14,17,78,147]. 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 [135,147] have shown that flash-based key-values stores underutilize or even misuse standard NVMe SSDs. The complete isolation of SSDs' internal organization from the host applications leads to inefficiencies like redundant mapping, double garbage collection and superfluous over-provisioning [135]. 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, the largest Internet search engine in China, aiming to accelerate the performance of a modified Level-DB key-value store [147]. So, 3000 devices have been deployed, each one incorporating 44 channels accessible as independent block devices. DIDACache [135] 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 [14,52]. 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, while 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. So, they achieved limited results, since these devices exhibit diverse performance characteristics. 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 [95], providing a great opportunity for researchers that seek to exploit OC SSDs in data indexing, beyond accelerating key-value stores.

8.3 In-storage processing

In-storage processing [83], near-data processing [8,54], in-storage computing [74,143,144] and active SSDs [33,91] 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 [32]. 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 and to control raw flash operations. 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 [144].

An external sorting algorithm, implemented on the Openssd[1] platform, is demonstrated in [91]. 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 [143,144]. The authors sought to determine which search engine operations can be offloaded to SSD for execution. Namely, they studied list intersection, ranked intersection, ranked union, difference and ranked difference operations using Apache Lucene[2] as testbed.

A generic programming framework for developing near-data processing applications is presented in [54]. It is built around a commercial enterprise SSD by Samsung. The platform achieved significant performance gains during queries' evaluation in MariaDB[3]. In a bit different direction, the processing capabilities of flash memory controllers (FMC) are examined in [33,83]. 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.

8.4 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.

Xu et al. [153] proposed 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 direct 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 [154] have been introduced, or the old ones have been properly modified (e.g., ext4-DAX). Using NVMs as storage device does not capitalize on them fully. However, utilizing 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 [158]. The authors in [153] 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 [15,55,63] 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 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.

9 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, e.g., the very successful I/O model of HDDs, greatly complicates the design and analysis of efficient flash-aware indexes and algorithms.

In this survey, we concisely and critically reviewed a broad range of indexes on raw and block flash devices, describing various employed techniques, like buffering of updates or logging, either in main or in the flash memory, utilization of fat or overflow nodes. All these paradigms strive to achieve some goals, such as small random writes restriction, internal parallelization exploitation, and read/write mixing prevention. The plethora of proposals reveals that these are not easily attained objectives, often obstructed by the lack of complete knowledge about the device internals.

In our opinion, 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. In-memory buffering can be seen as a prerequisite for batched

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

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

[3] https://mariadb.org/.

I/O; it also accelerates the performance of indexes, since it reduces the number of accesses to the secondary storage. All proposed methods so far (e.g., logging, overflow nodes) exploit in-memory buffers and, more or less, use batch write or/and read operations. Early works (e.g., BFTL, RFTL, LCR-tree) used batch writes to alleviate the difference between reads and writes, whereas more recent ones target to the SSD internal parallelization.

The merits of batched I/O can be summarized as follows: The large batches of writes exploit the SSDs' internal parallelization and the potentials of the NVMe protocol in a more efficient way, thus better performance is achieved. They also eliminate small random write operations that may cause frequent garbage collection operations and fragmentation of the FTL mapping table; this is high dependent on the mapping algorithm of FTL which is proprietary. The large batches of random reads also exploit the SSDs' internal parallelization and the efficiency of the NVMe protocol. Additionally, they do not differ much, in terms of performance, from the sequential ones. Using large I/O batches, besides accelerating the performance of reads and writes, results also in the separation of reads and writes, restraining the interference between them.

Finally, we saw that new programming frameworks, recent architecture proposals like Open-Channel, arisen computing paradigms and upcoming NVMs bring new challenges in the design and deployment of efficient index structures.

References

1. Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Commun. ACM **31**(9), 1116–1127 (1988)
2. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S.: Lazy-adaptive tree: an optimized index structure for flash devices. Proc. VLDB Endow. **2**(1), 361–372 (2009)
3. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M.S., Panigrahy, R.: Design tradeoffs for SSD performance. In: Proceedings of the USENIX Annual Technical Conference (ATC), Boston, MA, pp. 57–70 (2008)
4. Ajwani, D., Beckmann, A., Jacob, R., Meyer, U., Moruz, G.: On computational models for flash memory devices. In: Proceedings of the 8th International Symposium on Experimental Algorithms (SEA), Dortmund, Germany, pp. 16–27 (2009)
5. Ajwani, D., Malinger, I., Meyer, U., Toledo, S.: 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, pp. 208–219 (2008)
6. Andersson, A., Nilsson, S.: Efficient implementation of suffix trees. Softw. Pract. Exp. **25**(2), 129–141 (1995)
7. Athanassoulis, M., Ailamaki, A.: BF-tree: approximate tree indexing. Proc. VLDB Endow. **7**(14), 1881–1892 (2014)
8. Barbalace, A., Iliopoulos, A., Rauchfuss, H., Brasche, G.: 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, pp. 56–61 (2017)
9. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. Acta Inform. **1**(3), 173–189 (1972)
10. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: 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, pp. 322–331 (1990)
11. Bender, M.A., Farach-Colton, M., Johnson, R., Mauras, S., Mayer, T., Phillips, C.A., Xu, H.: Write-optimized skip lists. In: Proceedings of the 36th ACM Symposium on Principles of Database Systems (PODS), Chicago, IL, pp. 69–78 (2017)
12. Bentley, J.L., Saxe, J.B.: Decomposable searching problems, I. Static-to-dynamic transformation. J. Algorithms **1**(4), 301–358 (1980)
13. Bityuckiy, A.B.: JFFS3 design issues. Technical report, Memory technology device (MTD) subsystem for Linux (2005)
14. Bjørling, M., González, J., Bonnet, P.: LightNVM: the Linux Open-Channel SSD subsystem. In: Proceedings of the 15th USENIX Conference on File & Storage Technologies (FAST), Santa Clara, CA, pp. 359–374 (2017)
15. Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Gu, Y., Shun, J.: Efficient algorithms with asymmetric read and write costs. In: Proceedings of the 24th Annual European Symposium on Algorithms (ESA), Schloss Dagstuhl, Germany (2016)
16. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
17. Bonnet, P.: What's up with the storage hierarchy? In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR), Chaminade, CA (2017)
18. Bouganim, L., Jónsson, B., Bonnet, P.: uFLIP: understanding flash IO patterns (2009). arXiv preprint arXiv:0909.1780
19. Byun, S., Huh, M., Hwang, H.: An index rewriting scheme using compression for flash memory database systems. J. Inf. Sci. **33**(4), 398–415 (2007)
20. Cai, Y., Ghose, S., Haratsch, E.F., Luo, Y., Mutlu, O.: Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. Proc. IEEE **105**(9), 1666–1704 (2017)
21. Canim, M., Lang, C.A., Mihaila, G.A., Ross, K.A.: Buffered Bloom filters on solid state storage. In: Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor & Storage Architectures (ADMS), Singapore, pp. 1–8 (2010)
22. Cao, Z., Zhou, S., Li, K., Liu, Y.: Flashsearch: document searching in small mobile device. In: Proceedings of the International Seminar on Business & Information Management, Wuhan, China, pp. 79–82 (2008)
23. Carniel, A.C., Ciferri, R.R., de Aguiar Ciferri, C.D.: The performance relation of spatial indexing on hard disk drives and solid state drives. In: Proceedings of the XVII Brazilian Symposium on Geoinformatics (GeoInfo), Campos do Jordão, SP, Brazil, pp. 263–274 (2016)
24. Carniel, A.C., Ciferri, R.R., de Aguiar Ciferri, C.D.: Analyzing the performance of spatial indices on hard disk drives and flash-based solid state drives. J. Inf. Data Manag. **8**(1), 34 (2017)
25. Carniel, A.C., Ciferri, R.R., de Aguiar Ciferri, C.D.: 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, pp. 229–243 (2017)
26. Carniel, A.C., Ciferri, R.R., Ciferri, C.D.: A generic and efficient framework for flash-aware spatial indexing. Inf. Syst. **82**, 102–120 (2019)
27. Carniel, A.C., Roumelis, G., Ciferri, R.R., Vassilakopoulos, M., Corral, A., Cifferi, C.D.d.A.: An efficient flash-aware spatial index for points. In: Proceedings of the XIX Brazilian Symposium on

Geoinformatics (GEOINFO), Campina Grande, Brazil, pp. 65–79 (2018)
28. Chakrabarti, D.R., Boehm, H.J., Bhandari, K.: Atlas: leveraging locks for non-volatile memory consistency. ACM SIGPLAN Not. **49**(10), 433–452 (2014)
29. Chazelle, B., Guibas, L.J.: Fractional cascading: a data structuring technique. Algorithmica **1**(1–4), 133–162 (1986)
30. Chen, F., Hou, B., Lee, R.: Internal parallelism of flash memory-based solid-state drives. ACM Trans. Storage **12**(3), 13 (2016)
31. Chen, F., Koufaty, D.A., Zhang, X.: 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, pp. 181–192 (2009)
32. Cho, S., Chang, S., Jo, I.: The solid-state drive technology, today and tomorrow. In: Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE), Seoul, Korea, pp. 1520–1522 (2015)
33. Cho, S., Park, C., Oh, H., Kim, S., Yi, Y., Ganger, G.R.: Active disk meets flash: a case for intelligent SSDs. In: Proceedings of the 27th ACM International Conference on Supercomputing (ICS), Eugene, OR, pp. 91–102 (2013)
34. Choi, W.G., Shin, M., Lee, D., Park, H., Park, S.: Optimization of a multiversion index on SSDs to improve system performance. In: Proceedings of the IEEE International Conference on Systems, Man & Cybernetics (SMC), Budapest, Hungary, pp. 1620–1625 (2016)
35. Chowdhury, N.M.M.K., Akbar, M.M., Kaykobad, M.: DiskTrie: an efficient data structure using flash memory for mobile devices. In: Proceedings of the 1st Workshop on Algorithms & Computation (WALCOM), Dhaka, Bangladesh, pp. 76–87 (2007)
36. Comer, D.: The ubiquitous B-tree. ACM Comput. Surv. **11**(2), 121–137 (1979)
37. Cornwell, M.: Anatomy of a solid-state drive. Commun. ACM **55**(12), 59–63 (2012)
38. Cui, K., Jin, P., Yue, L.: HashTree: a new hybrid index for flash disks. In: Proceedings of the 12th International Asia-Pacific Web Conference (APWeb), Busan, Korea, pp. 45–51 (2010)
39. Dai, H., Neufeld, M., Han, R.: Elf: an efficient log-structured flash file system for micro sensor nodes. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys), Baltimore, MD, pp. 176–187 (2004)
40. Debnath, B., Sengupta, S., Li, J.: SkimpyStash: RAM space skimpy key-value store on flash-based storage. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Athens, Greece, pp. 25–36 (2011)
41. Debnath, B., Sengupta, S., Li, J., Lilja, D.J., Du, D.H.: BloomFlash: bloom filter on flash-based storage. In: Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS), Minneapolis, MN, pp. 635–644 (2011)
42. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. **38**(1), 86–124 (1989)
43. Engel, J., Mertens, R.: LogFS-finally a scalable flash file system. In: Proceedings of the 12th International Linux System Technology Conference, Hamburg, Germany (2005)
44. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing: a fast access method for dynamic files. ACM Trans. Database Syst. **4**(3), 315–344 (1979)
45. Fang, H.W., Yeh, M.Y., Suei, P.L., Kuo, T.W.: An adaptive endurance-aware B^+-tree for flash memory storage systems. IEEE Trans. Comput. **63**(11), 2661–2673 (2014)
46. Fevgas, A., Bozanis, P.: Grid-file: towards to a flash efficient multi-dimensional index. In: Proceedings of the 29th International Conference on Database & Expert Systems Applications (DEXA), Regensburg, Germany, vol. II, pp. 285–294 (2015)
47. Fevgas, A., Bozanis, P.: LB-Grid: an SSD efficient grid file. Data Knowl. Eng. **121**, 18–41 (2019)
48. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta Inform. **4**(1), 1–9 (1974)
49. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2), 170–231 (1998)
50. Gao, C., Shi, L., Ji, C., Di, Y., Wu, K., Xue, C.J., Sha, E.H.M.: Exploiting parallelism for access conflict minimization in flash-based solid state drives. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(1), 168–181 (2018)
51. Gong, X., Chen, S., Lin, M., Liu, H.: A write-optimized B-tree layer for NAND flash memory. In: Proceedings of the 7th International Conference on Wireless Communications, Networking & Mobile Computing (WiCOM), Wuhan, China, pp. 1–4 (2011)
52. González, J., Bjørling, M.: Multi-tenant I/O isolation with open-channel SSDs. In: Proceedings of the 8th Annual Non-Volatile Memories Workshop (NVMW), San Diego, CA (2017)
53. Graefe, G.: Write-optimized B-trees. In: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), Toronto, Canada, pp. 672–683 (2004)
54. Gu, B., Yoon, A.S., Bae, D.H., Jo, I., Lee, J., Yoon, J., Kang, J.U., Kwon, M., Yoon, C., Cho, S., 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, pp. 153–165 (2016)
55. Gu, Y.: Survey: computational models for asymmetric read and write costs. In: Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), Vancouver, Canada, pp. 733–743 (2018)
56. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Boston, MA, pp. 47–57 (1984)
57. Haapasalo, T., Jaluta, I., Seeger, B., Sippu, S., Soisalon-Soininen, E.: Transactions on the multiversion B^+-tree. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), Saint-Petersburg, Russia, pp. 1064–1075 (2009)
58. Hady, F.T., Foong, A., Veal, B., Williams, D.: Platform storage performance with 3D XPoint technology. Proc. IEEE **105**(9), 1822–1833 (2017)
59. Havasi, F.: An improved B^+-tree for flash file systems. In: Proceedings of the 37th International Conference on Current Trends in Theory & Practice of Computer Science (SOFSEM), Nový Smokovec, Slovakia, pp. 297–307 (2011)
60. Hinrichs, K.: Implementation of the grid file: design concepts and experience. BIT Numer. Math. **25**(4), 569–592 (1985)
61. Ho, V.P., Park, D.J.: WPCB-tree: a novel flash-aware B-tree index using a write pattern converter. Symmetry **10**(1), 18 (2018)
62. Hu, Y., Jiang, H., Feng, D., Tian, L., Luo, H., Zhang, S.: 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, pp. 96–107 (2011)
63. Jacob, R., Sitchinava, N.: Lower bounds in the asymmetric external memory model. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms & Architectures (SPAA), Washington, DC, pp. 247–254 (2017)
64. Jiang, Z., Wu, Y., Zhang, Y., Li, C., Xing, C.: AB-tree: a write-optimized adaptive index structure on solid state disk. In: Proceedings of the 11th Web Information System & Application Conference (WISA), Tianjin, China, pp. 188–193 (2014)
65. Jin, P., Xie, X., Wang, N., Yue, L.: Optimizing R-tree for flash memory. Expert Syst. Appl. **42**(10), 4676–4686 (2015)
66. Jin, P., Yang, C., Jensen, C.S., Yang, P., Yue, L.: Read/write-optimized tree indexing for solid-state drives. VLDB J. **25**(5), 695–717 (2016)

67. Jin, P., Yang, C., Wang, X., Yue, L., Zhang, D.: SAL-hashing: a self-adaptive linear hashing index for SSDs. IEEE Trans. Knowl. Data Eng. (2018). https://doi.org/10.1109/TKDE.2018.2884714
68. Jin, P., Yang, P., Yue, L.: Optimizing B+-tree for hybrid storage systems. Distrib. Parallel Databases **33**(3), 449–475 (2015)
69. Jin, R.: B-tree index layer for multi-channel flash memory. In: Proceedings of the 4th International Conference on Mobile & Wireless Technology (ICMWT), Kuala Lumpur, Malaysia, pp. 197–202 (2017)
70. Jin, R., Cho, H.J., Chung, T.S.: A group round robin based B-tree index storage scheme for flash memory devices. In: Proceedings of the 8th International Conference on Ubiquitous Information Management & Communication (ICUIMC), Siem Reap, Cambodia, p. 29 (2014)
71. Jin, R., Cho, H.J., Chung, T.S.: LS-LRU: a lazy-split LRU buffer replacement policy for flash-based B^+-tree index. J. Inf. Sci. Eng. **31**(3), 1113–1132 (2015)
72. Jin, R., Cho, H.J., Lee, S.W., Chung, T.S.: Lazy-split B^+-tree: a novel B^+-tree index scheme for flash-based database systems. Des. Autom. Embed. Syst. **17**(1), 167–191 (2013)
73. Jin, R., Kwon, S.J., Chung, T.S.: FlashB-tree: a novel B-tree index scheme for solid state drives. In: Proceedings of the ACM Symposium on Research in Applied Computation (RACS), Miami, FL, pp. 50–55 (2011)
74. Jo, I., Bae, D.H., Yoon, A.S., Kang, J.U., Cho, S., Lee, D.D., Jeong, J.: YourSQL: a high-performance database system leveraging in-storage computing. Proc. VLDB Endow. **9**(12), 924–935 (2016)
75. Jørgensen, M.V., Rasmussen, R.B., Šaltenis, S., Schjønning, C.: FB-tree: a B^+-tree for flash-based SSDs. In: Proceedings of the 15th Symposium on International Database Engineering & Applications (IDEAS), Lisbon, Portugal, pp. 34–42 (2011)
76. Jung, W., Roh, H., Shin, M., Park, S.: Inverted index maintenance strategy for flashSSDs: revitalization of in-place index update strategy. Inf. Syst. **49**, 25–39 (2015)
77. Kang, D., Jung, D., Kang, J.U., Kim, J.S.: $\mu*$-tree: an ordered index structure for NAND flash memory with adaptive page layout scheme. IEEE Trans. Comput. **62**(4), 784–797 (2007)
78. Kang, J.U., Hyun, J., Maeng, H., Cho, S.: The multi-streamed solid-state drive. In: Proceedings of the 6th USENIX Workshop on Hot Topics in Storage & File Systems (HotStorage), Philadelphia, PA (2014)
79. Kim, B., Lee, D.H.: LSB-tree: a log-structured B-tree index structure for NAND flash SSDs. Des. Autom. Embed. Syst. **19**(1–2), 77–100 (2015)
80. Kim, B.K., Lee, S.W., Lee, D.H.: h-Hash: a hash index structure for flash-based solid state drives. J. Circuits Syst. Comput. **24**(9), 1550128 (2015)
81. Kim, E.: SSD performance: a primer. Technical report, Solid State Storage Initiative (2013)
82. Kim, H.J., Lee, Y.S., Kim, J.S.: 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)
83. Kim, S., Oh, H., Park, C., Cho, S., Lee, S.W., Moon, B.: In-storage processing of database scans and joins. Inf. Sci. **327**, 183–200 (2016)
84. Koltsidas, I., Hsu, V.: IBM storage and NVM express revolution. Technical report, IBM (2017)
85. Koltsidas, I., Pletka, R., Mueller, P., Weigold, T., Eleftheriou, E., Varsamou, M., Ntalla, A., Bougioukou, E., Palli, A., Antanokopoulos, T.: PSS: a prototype storage subsystem based on PCM. In: Proceedings of the 5th Annual Non-Volatile Memories Workshop (NVMW), San Diego, CA (2014)
86. Kourtis, K., Ioannou, N., Koltsidas, I.: Reaping the performance of fast NVM storage with uDepot. In: Proceedings of the 17th USENIX Conference on File & Storage Technologies (FAST), Boston, MA, pp. 1–15 (2019)
87. Kwon, S.J., Ranjitkar, A., Ko, Y.B., Chung, T.S.: FTL algorithms for NAND-type flash memories. Des. Autom. Embed. Syst. **15**(3), 191–224 (2011)
88. Lee, H.S., Lee, D.H.: An efficient index buffer management scheme for implementing a B-tree on NAND flash memory. Data Knowl. Eng. **69**(9), 901–916 (2010)
89. Lee, S.W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Beijing, China, pp. 55–66 (2007)
90. Lee, Y.G., Jung, D., Kang, D., Kim, J.S.: 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, pp. 21–30 (2008)
91. Lee, Y.S., Quero, L.C., Lee, Y., Kim, J.S., Maeng, S.: 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)
92. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-tree: a B-tree for new hardware platforms. In: Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE), Washington, DC, pp. 302–313 (2013)
93. Levandoski, J.J., Sengupta, S., Redmond, W.: The BW-tree: a latch-free B-tree for log-structured flash storage. IEEE Data Eng. Bull. **36**(2), 56–62 (2013)
94. Li, G., Zhao, P., Yuan, L., Gao, S.: Efficient implementation of a multi-dimensional index structure over flash memory storage systems. J. Supercomput. **64**(3), 1055–1074 (2013)
95. Li, H., Hao, M., Tong, M.H., Sundararaman, S., Bjørling, M., Gunawi, H.S.: The CASE of FEMU: cheap, accurate, scalable and extensible flash emulator. In: Proceedings of the 16th USENIX Conference on File & Storage Technologies (FAST), Oakland, CA, pp. 83–90 (2018)
96. Li, R., Chen, X., Li, C., Gu, X., Wen, K.: Efficient online index maintenance for SSD-based information retrieval systems. In: Proceedings of the 14th IEEE International Conference on High Performance Computing & Communication (HPCC), Liverpool, UK, pp. 262–269 (2012)
97. Li, X., Da, Z., Meng, X.: A new dynamic hash index for flash-based storage. In: Proceedings of the 9th International Conference on Web-Age Information Management (WAIM), Zhangjiajie, China, pp. 93–98 (2008)
98. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on flash disks. In: Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE), Shanghai, China, pp. 1303–1306 (2009)
99. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. Proc. VLDB Endow. **3**(1–2), 1195–1206 (2010)
100. Lin, S., Zeinalipour-Yazti, D., Kalogeraki, V., Gunopulos, D., Najjar, W.A.: Efficient indexing data structures for flash-based sensor devices. ACM Trans. Storage **2**(4), 468–503 (2006)
101. Litwin, W.: 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, pp. 212–223 (1980)
102. Lu, G., Debnath, B., Du, D.H.: A forest-structured Bloom filter with flash memory. In: Proceedings of the 27th IEEE Symposium on Mass Storage Systems & Technologies (MSST), Denver, CO, pp. 1–6 (2011)
103. Lv, Y., Li, J., Cui, B., Chen, X.: Log-compact R-tree: an efficient spatial index for SSD. In: Proceedings of the 16th International Conference on Database Systems for Advanced Applications (DASFAA), Hong Kong, China, vol. III, pp. 202–213 (2011)

104. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer, Berlin (2010)
105. Mehlhorn, K., Näher, S.: Dynamic fractional cascading. Algorithmica **5**(1–4), 215–241 (1990)
106. Meza, J., Wu, Q., Kumar, S., Mutlu, O.: 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, pp. 177–190 (2015)
107. Micheloni, R.: 3D Flash Memories. Springer, Berlin (2016)
108. Micheloni, R.: Solid-State-Drives Modeling. Springer, Berlin (2017)
109. Mittal, S., Vetter, J.S.: A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1537–1550 (2016)
110. Na, G.J., Lee, S.W., Moon, B.: Dynamic in-page logging for b+-tree index. IEEE Trans. Knowl. Data Eng. **24**(7), 1231–1243 (2012)
111. Na, G.J., Moon, B., Lee, S.W.: IPLB$^+$-tree for flash memory database systems. J. Inf. Sci. Eng. **27**(1), 111–127 (2011)
112. Nanavati, M., Schwarzkopf, M., Wires, J., Warfield, A.: Non-volatile storage: implications of the datacenter's shifting center. ACM Queue **13**(9), 20 (2015)
113. Narayanan, I., Wang, D., Jeon, M., Sharma, B., Caulfield, L., Sivasubramaniam, A., Cutler, B., Liu, J., Khessib, B., Vaid, K.: SSD failures in datacenters: What? when? and why? In: Proceedings of the 9th ACM International on Systems & Storage Conference (SYSTOR), Haifa, Israel (2016)
114. Nath, S., Kansal, A.: FlashDB: dynamic self-tuning database for NAND flash. In: Proceedings of the 6th International Symposium on Information Processing in Sensor Networks (IPSN), Cambridge, MA, pp. 410–419 (2007)
115. Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The Grid file: an adaptable, symmetric multikey file structure. ACM Trans. Database Syst. **9**(1), 38–71 (1984)
116. On, S.T., Hu, H., Li, Y., Xu, J.: Lazy-update B$^+$-tree for flash devices. In: Proceedings of the 10th International Conference on Mobile Data Management (MDM), Taipei, Taiwan, pp. 323–328 (2009)
117. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Inform. **33**(4), 351–385 (1996)
118. Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., Kim, J.S.: A reconfigurable FTL architecture for NAND flash-based applications. ACM Trans. Embed. Comput. Syst. **7**(4), 38 (2008)
119. Pawlik, M., Macyna, W.: 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, pp. 65–72 (2012)
120. Pearce, R., Gokhale, M., Amato, N.M.: 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, pp. 1–11 (2010)
121. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–677 (1990)
122. Robinson, J.T.: 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, pp. 10–18 (1981)
123. Roh, H., Kim, S., Lee, D., Park, S.: AS B-tree: a study of an efficient B$^+$-tree for SSDs. J. Inf. Sci. Eng. **30**(1), 85–106 (2014)
124. Roh, H., Kim, W.C., Kim, S., Park, S.: A B-tree index extension to enhance response time and the life cycle of flash memory. Inf. Sci. **179**(18), 3136–3161 (2009)
125. Roh, H., Park, S., Kim, S., Shin, M., Lee, S.W.: B$^+$-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proc. VLDB Endow. **5**(4), 286–297 (2011)
126. Roh, H., Park, S., Shin, M., Lee, S.W.: MPSearch: multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs. IEEE Data Eng. Bull. **37**(2), 3–11 (2014)
127. Ross, K.A.: 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, pp. 11–16 (2008)
128. Roumelis, G., Fevgas, A., Vassilakopoulos, M., Corral, A., Bozanis, P., Manolopoulos, Y.: Bulk-loading and bulk-insertion algorithms for xBR-trees in solid state drives. Computing (2019). https://doi.org/10.1007/s00607-019-00709-4
129. Roumelis, G., Vassilakopoulos, M., Corral, A., Fevgas, A., Manolopoulos, Y.: 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, pp. 301–317 (2018)
130. Roumelis, G., Vassilakopoulos, M., Loukopoulos, T., Corral, A., Manolopoulos, Y.: The xBR$^+$-tree: an efficient access method for points. In: Proceedings of the 26th International Conference on Database & Expert Systems Applications (DEXA), Valencia, Spain, pp. 43–58 (2015)
131. Sarwat, M., Mokbel, M.F., Zhou, X., Nath, S.: 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, pp. 149–167 (2011)
132. Sarwat, M., Mokbel, M.F., Zhou, X., Nath, S.: Generic and efficient framework for search trees on flash memory storage systems. GeoInformatica **17**(3), 417–448 (2013)
133. Schierl, A., Schellhorn, G., Haneberg, D., Reif, W.: Abstract specification of the UBIFS file system for flash memory. In: Proceedings of the 16th International Symposium on Formal Methods (FM), Eindhoven, the Netherlands, pp. 190–206 (2009)
134. Schroeder, B., Lagisetty, R., Merchant, A.: Flash reliability in production: the expected and the unexpected. In: Proceedings of the 14th USENIX Conference on File & Storage Technologies (FAST), Santa Clara, CA, pp. 67–80 (2016)
135. Shen, Z., Chen, F., Jia, Y., Shao, Z.: Didacache: an integration of device and application for flash-based key-value caching. ACM Trans. Storage **14**(3), 26:1–26:32 (2018)
136. Son, Y., Kang, H., Han, H., Yeom, H.Y.: An empirical evaluation and analysis of the performance of nvm express solid state drive. Clust. Comput. **19**(3), 1541–1553 (2016)
137. Tan, C.C., Sheng, B., Wang, H., Li, Q.: Microsearch: a search engine for embedded devices used in pervasive computing. ACM Trans. Embed. Comput. Syst. **9**(4), 43 (2010)
138. Teng, D., Guo, L., Lee, R., Chen, F., Zhang, Y., Ma, S., Zhang, X.: A low-cost disk solution enabling lsm-tree to achieve high performance for mixed read/write workloads. ACM Trans. Storage **14**(2), 15 (2018)
139. Thonangi, R., Babu, S., Yang, J.: A practical concurrent index for solid-state drives. In: Proceedings of the 21st ACM International Conference on Information & Knowledge Management (CIKM), Maui, HI, pp. 1332–1341 (2012)
140. Thonangi, R., Yang, J.: On log-structured merge for solid-state drives. In: Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE), San Diego, CA, pp. 683–694 (2017)
141. Viglas, S.D.: Adapting the B$^+$-tree for asymmetric I/O. In: Proceedings of the 16th East European Conference on Advances in Databases & Information Systems (ADBIS), Poznan, Poland, pp. 399–412 (2012)
142. Wang, H., Feng, J.: FlashSkipList: indexing on flash devices. In: Proceedings of the ACM Turing 50th Celebration Conference (ACM TUR-C), Shanghai, China (2017)

143. Wang, J., Park, D., Kee, Y.S., Papakonstantinou, Y., Swanson, S.: 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)
144. Wang, J., Park, D., Papakonstantinou, Y., Swanson, S.: SSD in-storage computing for search engines. IEEE Trans. Comput. (2016). https://doi.org/10.1109/TC.2016.2608818
145. Wang, L., Wang, H.: A new self-adaptive extendible hash index for flash-based DBMS. In: Proceedings of the IEEE International Conference on Information & Automation (ICIA), Harbin, China, pp. 2519–2524 (2010)
146. Wang, N., Jin, P., Wan, S., Zhang, Y., Yue, L.: 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, pp. 1–14 (2012)
147. Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., Cong, J.: 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)
148. Workgroup, N.E.: NVME overview (Online) http://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf. Accessed 29 Apr 2019
149. Wu, C.H., Chang, L.P., Kuo, T.W.: 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, pp. 409–430 (2003)
150. Wu, C.H., Chang, L.P., Kuo, T.W.: 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, pp. 17–24 (2003)
151. Wu, C.H., Kuo, T.W., Chang, L.P.: An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embed. Comput. Syst. **6**(3), 19 (2007)
152. Xiang, X., Yue, L., Liu, Z., Wei, P.: A reliable B-tree implementation over flash memory. In: Proceedings of the 23rd ACM Symposium on Applied Computing (SAC), Fortaleza, Brazil, pp. 1487–1491 (2008)
153. Xu, J., Kim, J., Memaripour, A., Swanson, S.: Finding and fixing performance pathologies in persistent memory software stacks. In: Proceedings of the 24th International Conference on Architectural Support for Programming Languages & Operating Systems (ASPLOS), Providence, RI, pp. 427–439 (2019)
154. Xu, J., Swanson, S.: NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: Proceedings of the 14th USENIX Conference on File & Storage Technologies (FAST), Santa Clara, CA, pp. 323–338 (2016)
155. Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., Balakrishnan, V.: 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)
156. Yang, C., Jin, P., Yue, L., Zhang, D.: Self-adaptive Linear hashing for solid state drives. In: Proceedings of the 32nd IEEE International Conference on Data Engineering (ICDE), Helsinki, Finland, pp. 433–444 (2016)
157. Yang, C.W., Lee, K.Y., Kim, M.H., Lee, Y.J.: An efficient dynamic hash index structure for NAND flash memory. IEICE Trans. Fundam. Electron. Commun. **92–A**(7), 1716–1719 (2009)
158. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-tree: reducing consistency cost for NVM-based single level systems. In: Proceedings of the 13th USENIX Conference on File & Storage Technologies (FAST), Santa Clara, CA, pp. 167–181 (2015)
159. Yang, Z., Harris, J.R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V., Paul, L.E.: 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, pp. 154–161 (2017)
160. Yin, S., Pucheral, P.: PBFilter: a flash-based indexing scheme for embedded systems. Inf. Syst. **37**(7), 634–653 (2012)
161. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. **38**(2), 6 (2006)