The Past, Present and Future of Indexing on Persistent Memory

Kaisong Huang
Simon Fraser University
kha85@sfu.ca

Yuliang He
Simon Fraser University
georgeh@sfu.ca

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Persistent memory (PM) based indexing techniques have been proposed to build fast yet persistent indexes that sit on the memory bus. Over the past decade, numerous techniques have been proposed with various assumptions and different properties (e.g., some of them were proposed before real PM became available), making it hard for researchers and practitioners to gain a comprehensive understanding of the area.

In this tutorial, we give a comprehensive overview of PM indexing techniques, covering both range and hash indexes. We contrast the designs proposed before and after real PM became available, summarize the common and useful design techniques, and discuss potential future challenges and opportunities in this area.

PVLDB Reference Format:
Kaisong Huang, Yuliang He, and Tianzheng Wang. The Past, Present and Future of Indexing on Persistent Memory. PVLDB, 15(12): 3774 - 3777, 2022.
doi:10.14778/3554821.3554897

1 INTRODUCTION

Persistent indexes (such as B+-trees and hash tables) are fundamental data structures in database systems. Traditionally, they have been built to cope with the characteristics of block devices such as SSDs and HDDs assuming a two-level storage hierarchy where data access is done through a buffer manager. Naturally, these traditional indexes suffer from the low bandwidth and high latency available of traditional storage devices, limiting overall system performance.

The recent advances in scalable persistent memory (PM) technologies such as PCM [35], STT-RAM [12] and Intel 3D XPoint [9] give a promising solution to this problem. PM is a range of devices that combines the best of both DRAM and SSDs, by offering nanosecond-level latency, high bandwidth and persistence with byte-addressability, all on the memory bus. Such features potentially allow us to build fast, *single-level* persistent indexes that directly operate and persist data on PM without having to use a DRAM-SSD hierarchy managed by a buffer pool.

Based on this PM vision, devising single-level PM indexes has become an active line of research and received a lot of attention from researchers from both data management and systems communities. Researchers started to propose PM indexes even before real devices were available [2, 4, 5, 13, 22, 29, 31, 34, 36, 37]. Since the release of Intel Optane Persistent Memory (PMem)—the only commercially available scalable PM—newer indexes have been proposed [6, 18, 24, 26, 27, 40] to cope with the actual characteristics of the device. A main challenge is that PMem, with ~300ns latency and up to tens of GB/s of bandwidth, is still slower than DRAM. This led to the development of various techniques to reduce unnecessary PM accesses, e.g., by using DRAM and relaxing sortedness of index nodes. At the same time, programming on PM is a challenging task by itself. Devising a recoverable PM index requires careful consideration of such issues as guaranteeing persistence, concurrency and preventing permanent PM leaks. These are new problems that were not commonly considered by traditional in-memory indexes and often require different techniques from traditional storage-centric persistent indexes for performance reasons.

This tutorial provides a comprehensive overview of PM indexing techniques. We cover both range indexes (such as B+-tree and trie variants) and hash tables proposed before and after the real PM devices are available to give a complete snapshot of the state of play in this area. In addition, we discuss important background knowledge of the PM hardware and software ecosystems for practitioners and researchers to devise practical solutions. Finally, we discuss future challenges and opportunities in this area.

Tutorial Overview. We plan for a 1.5-hour session that is split into five sections. The first section will give an introduction to PM ecosystems, including PM hardware features and PM's potential for data management systems, with a focus on indexing structures. Section 2 then discusses the necessary background for PM programming in general. Sections 3 and 4 make up the bulk of the tutorial to give an in-depth look at range indexes and hash indexes for PM, respectively. Finally, we conclude the tutorial in Section 5 with a discussion on future challenges and opportunities.

Target Audience. We target a broad range of audience that includes researchers and practitioners who are interested in exploring the use of PM-tailored indexing techniques in data management and storage systems. Given the versatility of indexing structures, the usefulness of this tutorial can go beyond database systems, to closely-related areas such as distributed systems and file systems.

We only expect basic background about commonly-used indexes, in particular B+-trees, tries and hash tables. Other than that, this tutorial is self-contained to include additional prerequisites such as PM hardware and PM programming issues for non-experts and new researchers to get started in this area.

Related Tutorials. To the best of our knowledge, this will be the first tutorial focused on PM indexing techniques. Several related tutorials appeared in the past. The most recent one was presented at VLDB 2021 and focused on extending the lifetime of PM hardware [17]. Another two tutorials were presented at SIGMOD 2015 [32] and SIGMOD 2019 [3] to discuss PM's impact on database systems. But both were based on emulation before real PM became commercially available. Earlier in VLDB 2016 a tutorial about main-memory database systems [20] discussed PM-based logging but did not cover PM indexes. Later, many new techniques based on real PM have been proposed, which are our main focus.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554897

Related Work from Authors. The authors have worked on the performance evaluation of representative PM range indexes [11, 23] on Intel Optane Persistent Memory (PMem), covering some of the indexes surveyed in this tutorial; part of the tutorial contents will be based on these prior evaluation results. The authors have also proposed Dash [26] which adapts extendible/linear hashing for PM, as well as APEX [25], a high performance learned index for PM.

Different from prior surveys which focused on a few representative indexes, this tutorial (1) covers a broader range of PM indexes (including hash tables and learned indexes, rather than only traditional B+-tree/trie variants in previous work), and (2) aims to provide the audience a systematic, "panoramic" view of this area that is infeasible in a regular paper and research talk.

2 TUTORIAL OUTLINE

We begin by introducing the vision of PM: bringing high capacity and persistence to the memory bus, overcoming DRAM's scaling limits. The combination of byte-addressability and persistence brings a set of challenges for software, which is discussed as background information about PM indexes. We then structure the remaining tutorial to cover PM indexing techniques based on their timing: before ("past") and after ("present") the real PM devices became available. We contrast these techniques and discuss their relevance in today's PM systems, and finally identify challenges and opportunities ("future").

2.1 Persistent Memory Systems

The idea of modern scalable PM appeared over a decade ago in search for alternatives to DRAM which is hitting scalability limitations [21, 39]. Various materials for manufacturing PM devices have been discovered, such as STT-RAM [12], memristor [30], phase change memory (PCM) [35] and Intel 3D XPoint [9]. Regardless of the materials under the hood, the common goal is to offer near-DRAM performance and byte-addressability. Coincidentally, these PM technologies all turned out to be non-volatile, offering also persistence on the memory bus and therefore potentially enabling single-level systems that directly operate and persist data on PM, without the need for additional secondary storage. For database systems, this indicates that major components, indexes in particular, can be directly placed in PM to achieve potentially instant recovery and high performance.

It was not until 2019 did the PM vision come true when Intel released Optane DC Persistent Memory Module (Optane DCPMM; currently referred to as Optane PMem) based on 3D XPoint. Today, Optane PMem remains the only commercially available and de facto standard PM device. Although DRAM/flash-based NVDIMM [1] also provides persistence on the memory bus, it can be limited by DRAM capacity or flash's low performance. Therefore, almost all research in this area has been targeting Optane PMem; our tutorial will therefore focus on PMem when discussing real PM features.

Optane PMem offers much larger capacity than DRAM, with up to 512 GB per DIMM. It exhibits low latency (~300ns read latency) compared to flash memory, but is still 4× higher than that of DRAM; it also exhibits asymmetric read/write performance with write being slower. For example, the first generation PMem (DCPMM 100 series) can deliver up to 40GB/s for reads and 10GB/s for writes under sequential workloads, which could scale down to 7.4GB/s and 5.3GB/s under random workloads, respectively. The recently released 200 series offers roughly 30% higher bandwidth thanks to the availability of more memory channels on the new platform [16].

Optane PMem can operate in the Memory, App Direct or Dual mode. The Memory mode provides bigger but slower volatile memory with DRAM as a transparent cache controlled solely by the hardware. Under the App Direct mode, software can judiciously use PM and DRAM to store data with persistence. The Dual mode combines Memory and App Direct modes. Since App Direct provides persistence, virtually all PM indexes are based on it, which is also our focus in this tutorial.

2.2 PM Programming

Building software for PM presents several major challenges: guaranteeing data persistence, managing PM space, and handling concurrency and recovery issues.

Persistence. Since PM is attached to the memory bus, it is behind multiple levels of volatile CPU caches. For safe persistence, the first generation of PMem mandates software to proactively issue cacheline flush instructions, including `CLWB`, `CLFLUSH`, and `CLFLUSHOPT` [14]. Among them, `CLWB` is preferred as it does not invalidate the cacheline after flushing. Upon flush, the data will reach the CPU's write buffer, which is included in the asynchronous DRAM refresh (ADR) domain. The ADR domain is power failure protected such that upon power failure the data can be safely drained to PM media. In other words, once data reaches the ADR domain, it is considered persistent. The latest of PMem platforms (e.g., the 3rd generation "Ice Lake" Xeon Scalable processors with PMem 200 series) feature extended ADR (eADR) which also covers CPU caches, virtually removing the need for software to proactively issue cacheline flushes. To guarantee specific ordering of writes, applications must also issue `SFENCE` to prevent stores from being re-ordered by the CPU, regardless of whether eADR is in place. Moreover, modern CPUs only guarantee 8-byte atomic writes. Therefore, atomic writes involving >8 bytes need special care, e.g., by using durable transactions based on logging [15] or other multi-word primitives [34].

PM Space Management. To leverage PM's memory nature, the application typically maps PM directly to its address space via the `mmap` interface to directly access data using load and store instructions without going through file system interface like POSIX `read` and `write`. However, the virtual address returned by `mmap` may not remain the same across reboots, invalidating any virtual memory pointer values stored in PM (e.g., child pointers stored in B+-tree inner nodes). A common solution is to keep track of offsets in PM and generate pointers by adding the offsets to a base address at runtime. Further, different from DRAM leaks, PM leaks are permanent, so the allocator must guarantee that a PM block is atomically allocated to the application, thus no dangling PM block exists in the system. The allocator should employ a safe PM ownership transfer protocol to resolve this issue [34].

Concurrency and Recovery. DRAM indexes can be recovered from storage without worrying about inconsistencies, while PM indexes need to recover data and program states (e.g., critical sections), because they are all persisted on PM. For example, a B+-tree

split may require changing multiple pointers atomically. This could be done by using locks, which are also persisted on PM. So a crash then can let the tree hold the lock forever. The software (B+-tree in this case) then needs additional recovery logic to restore to a previously consistent state, e.g., by releasing all locks upon recovery. This is often implemented with the help of some PM programming libraries, which we describe in more detail later.

Tutorial Coverage. Multiple research PM libraries have been proposed to resolve the aforementioned issues [8, 33, 34]. In practice, the Persistent Memory Development Kit (PMDK) [15] is the de facto standard. Although our focus is PM indexing, we provide an overview of these PM programming tools for non-experts and new researchers to get familiar with PM programming.

2.3 PM Range Indexes

PM range indexes adapt their in-memory counterparts (B+-trees, tries and hybrids) with additional designs to reduce PM accesses (thus better performance) and guarantee correct recovery. Depending on the time they appeared, in the tutorial we categorize them as "pre-Optane era" and "Optane era" proposals. In addition, we will also summarize the key techniques used in terms of index architecture, their node structure, concurrency and other feature support (variable-length keys, PM management and NUMA-awareness).

Pre-Optane Era. Before real PM appeared on the market, researchers had been using DRAM to emulate PM, by injecting additional delays. Among the various assumptions, some turned out to be not true (e.g., bandwidth profile). But several predictions and designs were validated later on real PM, especially the fundamental principle of reducing PM accesses and avoiding unnecessary cacheline flushes and fences to improve performance. Almost all the indexes were proposed with reduced PM accesses as a central goal [2, 4, 5, 13, 22, 29, 31, 37]. For B+-tree variants, since their inner nodes are just for guiding search traffic, and are reconstructible, proposals like FPTree and NV-Tree relax the consistency requirements for inner nodes by placing them in DRAM or dropping flushes and fences, at the cost of instant recovery because the DRAM content has to be rebuilt upon recovery. Excessive PM accesses can also be introduced by sorting nodes, thus unsorted nodes become popular (e.g., FPTree, NV-Tree and BzTree). FPTree also uses fingerprinting to avoid unnecessary existence checks, reducing PM accesses. In terms of concurrency, PM indexes tend to adopt optimistic over pessimistic approaches. FPTree uses hardware transactional memory (HTM) on its inner nodes to reduce traversal costs, and uses locking on leaf nodes to avoid HTM aborts caused by cacheline flushes in the leaf nodes. BzTree uses lock-free multi-word compare-and-swap [34] that can atomically modify multiple 8-byte words.

In the pre-Optane era, almost all proposals focused on the easier integer keys without much consideration of variable-length keys (other than storing pointers to keys), which must be supported for real workloads. Due to the limitations of emulation based research, important issues such as PM space management/allocation and NUMA effect were generally not considered, either.

Optane Era. After Optane PMem became available in 2019, multiple new PM range indexes have been proposed. They inherited a lot of designs from pre-Optane proposals but also came with their own new designs to cope with real PM's properties. The hybrid PM-DRAM architecture is common with more aggressive use of DRAM. For example, DPTree [40] and μTree [6] place entire trees in DRAM to gain more performance at the cost of longer recovery time, more complex programming, and higher DRAM space use. Further, these new indexes started to adapt more than B+-trees. For example, PACTree [18] and ROART [27] are based on tries, but they stick with the PM-only design, potentially yielding suboptimal performance but keeping instant recovery. Updatable learned indexes [19] such as ALEX [10] are also adapted for PM. A major challenge is their use of bigger nodes (which is an advantage of learned indexes) turned out to be suboptimal on PM due to high-cost structural modification and insert operations. APEX [25] applies a series of PM indexing techniques to ALEX to mitigate such impact, among other issues. Fingerprinting, unsorted (leaf) nodes, and selective consistency for metadata are still popular and further optimized. For example, node alignment of 256 bytes can reduce unnecessary PMem accesses; LB$^+$-Tree uses SIMD instructions to compare up to 64 fingerprints in one instruction; ROART embeds a two-byte fingerprint inside pointers to key-value pairs, reducing pointer chasing at the leaf level; LB$^+$-Tree and DPTree both use extra metadata per leaf node to avoid logging (thus reducing PM writes).

Notably, variable-length keys, PM allocator and NUMA effect still lack enough attention. Only trie-based ROART and PACTree natively support variable-length keys; the others continue to use pointers to keys in the pre-Optane fashion. Most new PM indexes use PMDK to manage PM (e.g., to avoid permanent leaks), but many indexes need to tailor their allocators for better performance. Finally, only PACTree mitigates NUMA effect (with important tradeoffs).

2.4 PM Hash Tables

The exploration of PM hash tables also started before any real PM device was available.

Pre-Optane Era. Level hashing [41] is one of the early write-optimized static hash tables. It proposes a two-level scheme to bound the search cost while reducing PM writes. CCEH [28] adapts extendible hashing to reduce PM accesses by fixing bucket sizes to one cacheline. It also introduces a segment layer the directory and buckets, to reduce the size of the directory so that a record can be found by accessing at most two cachelines. The downside is splits can be triggered early (at the coarse-grained segment level), leading to high PM allocation cost and low space efficiency. SOFT [42] reduces PM flushes by avoiding persisting pointers, but it still has volatile nodes that are chained in DRAM to allow fast access. Because the size of each volatile node becomes irregular to accommodate the pointers, a single cacheline now loads one and a half volatile nodes, causing more cache misses. SOFT's DRAM+PM architecture can cause long recovery time, as structural information is not persisted and needs to be reconstructed in DRAM, which defeats one of the purposes of using PM indexes (i.e., instant recovery).

Similar to PM range indexes, most pre-Optane hash tables focused on reducing PM writes and simple 8-byte keys. Most solutions are also single-threaded without concurrency.

Optane Era. Based on PMem, Clevel [7] designs a lock-free scheme for level hashing [41]. However, its search and delete operations incur extra PM reads, which should also be reduced for performance. Dash [26] optimizes for both PM reads and writes by

adopting fingerprinting from PM range indexes, to avoid unnecessary bucket probing. It also proposes a load balancing strategy that can postpone segment splits to increase space efficiency.

3 CHALLENGES AND OPPORTUNITIES

This section partially bases on our recent work [11] which identifies new challenges and opportunities for PM indexes, especially on providing full functionality while maintaining high performance. We also highlight opportunities in learned PM indexes and hash tables. We will also discuss the challenges associated with adopting such new indexes in practice, due to cost and programming complexity reasons. We also show an interesting finding that PM indexes perform very similarly to DRAM-optimized indexes in a pure DRAM environment. This hints the performance and design of future PM and DRAM indexes may in fact converge, simplifying system design and integration. Finally, we will discuss the potential impact of eADR, which as we mentioned earlier potentially invalidates the need to proactively flush cache lines. Some recent work [38] has started to exploit it, but it remains to be seen how well the previous design will hold and how future PM-based systems should be designed to fully leverage it.

4 PRESENTERS

Kaisong Huang is a PhD student in the School of Computing Science at Simon Fraser University advised by Tianzheng Wang. His research is mainly focused on database engines, transaction processing and storage management, in the context of modern storage technologies like NVMe SSDs and persistent memory.
Yuliang He is a Thesis MSc student in the School of Computing Science at Simon Fraser University advised by Tianzheng Wang. His research interests lie in database indexes with persistent memory.
Tianzheng Wang is an assistant professor in the School of Computing Science at Simon Fraser University. He is interested in database systems on modern hardware and has been working on persistent memory research for over ten years. His work has been recognized by a 2021 ACM SIGMOD Research Highlight Award, a 2019 IEEE TCSC Award for Excellence in Scalable Computing (Early Career Researchers) and nominations for best/memorable paper awards.

ACKNOWLEDGMENTS

This work is partially supported by an NSERC Discovery Grant, Canada Foundation for Innovation John R. Evans Leaders Fund and the B.C. Knowledge Development Fund.

REFERENCES

[1] AgigaTech. 2022. Non-Volatile RAM. http://www.agigatech.com Last accessed: June 7, 2022.
[2] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.
[3] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *SIGMOD*. 1753–1758.
[4] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*.
[5] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.
[6] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a Persistent B+-Tree with Low Tail Latency. *PVLDB* 13, 11 (2020), 2634–2648.
[7] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *USENIX ATC*. 799–812.
[8] Joel Coburn et al. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *ASPLOS*. 105–118.
[9] Rob Crooke and Mark Durcan. 2015. A Revolutionary Breakthrough in Memory Technology. *3D XPoint Launch Keynote* (2015).
[10] Jialin Ding et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.
[11] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes, Part Two. *arXiv preprint* 2201.13047 (2022).
[12] M. Hosomi et al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. *IEEE IEDM* (2005), 459–462.
[13] Deukyeon Hwang et al. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *FAST*. 187–200.
[14] Intel. 2021. Intel Architectures Software Developer's Manual. (2021).
[15] Intel. 2021. Persistent Memory Development Kit. (2021). http://pmem.io/pmdk Last accessed: June 7, 2022.
[16] Intel. 2022. Optane PMem 200 Series Brief. https://www.intel.ca/content/www/ca/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html Last accessed: June 7, 2022.
[17] Saeed Kargar and Faisal Nawab. 2021. Extending the lifetime of NVM: challenges and opportunities. *PVLDB* 14, 12 (2021), 3194–3197.
[18] Wook-Hee Kim et al. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP*. 424–439.
[19] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.
[20] Per-Åke Larson and Justin Levandoski. 2016. Modern Main-Memory Database Systems. *PVLDB* 9, 13 (sep 2016), 1609–1610.
[21] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable Dram Alternative. In *ISCA*. 2–13.
[22] Se Kwon Lee et al. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *FAST*. 257–270.
[23] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.
[24] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB* 13, 7 (2020), 1078–1090.
[25] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *PVLDB* 15, 3 (2021), 597–610.
[26] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *PVLDB* 13, 8 (2020), 1147–1161.
[27] Shaonan Ma et al. 2021. ROART: Range-query Optimized Persistent ART. In *FAST*. 1–16.
[28] Moohyeon Nam et al. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST*. 31–44.
[29] Ismail Oukid et al. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.
[30] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.
[31] Shivaram Venkataraman et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*. 5.
[32] Stratis D. Viglas. 2015. Data Management in Non-Volatile Memory. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1707–1711.
[33] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS (ASPLOS XVI)*. 91–104.
[34] Tianzheng Wang, Justin Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE*. 461–472.
[35] H. S P Wong et al. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.
[36] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-value Store for DRAM-NVM Memory Systems. In *USENIX ATC*. 349–362.
[37] Jun Yang et al. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *FAST*. 167–181.
[38] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *PVLDB* 15, 6 (2022), 1187–1200.
[39] Ping Zhou et al. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *ISCA*. 14–23.
[40] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *PVLDB* 13, 4 (2019), 421–434.
[41] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *OSDI*. 461–476.
[42] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *OOPSLA* 3 (2019), 1–26.