# Data Structure Primitives on Persistent Memory: An Evaluation

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Arun Kumar Tharanatha
TU Ilmenau, Germany
arun-kumar.tharanatha@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

## ABSTRACT

Persistent Memory (PM) represents a very promising, next-generation memory solution with a significant impact on database architectures. Several data structures for this new technology have already been proposed. However, primarily only complete structures are presented and evaluated. Thus, the implications of the individual ideas and PM features are concealed. Therefore, in this paper, we disassemble the structures presented so far, identify their underlying design primitives, and assign them to appropriate design goals. As a result of our comprehensive experiments on real PM hardware, we can reveal the trade-offs of the primitives for various access patterns and pinpoint their best use cases.

## 1 INTRODUCTION

Data structures play a crucial role in all data management systems. Over the past decades, numerous structures have been designed for very different purposes and each design is always a compromise among the three performance trade-offs read, write, and memory amplification [2]. Furthermore, advances in hardware technology with changing characteristics make designing data structures an ever-lasting challenge. Persistent Memory (PM) is one of the most promising trends in hardware development which might have a huge impact on database system architectures in general, but also particularly on data structures. Intel® has recently commercialized Optane™ DC Persistent Memory Modules (DCPMMs) based on the 3D XPoint™ technology [9], on which we focus in this paper. Characteristics such as byte-addressability, read latency close to DRAM, and the inherent persistence open up new opportunities but require also new designs, e.g., to mitigate the read-write asymmetry or to guarantee consistent updates.

Idreos et al. [6] presented the idea of a periodic table of data structures to be able to argue about the design space of data structures. The work provides a great foundation for a systematic study of data structure designs. In this paper, we try to support this approach by identifying core primitives of tree-based data structures and evaluate different designs of these primitives on PM. To the best of our knowledge, this is the first evaluation considering various data structure types and designs on the micro level with real PM hardware. The goal of our work is to get deep insights into PM-optimized design patterns for data structures in databases. A more detailed breakdown is additionally given in [4].


*DAMON'20, June 15, 2020, Portland, OR, USA*

ACM ISBN 978-1-4503-8024-9/20/06…$15.00
https://doi.org/10.1145/3399666.3399900

## 2 DESIGN PRIMITIVES

Using the PM properties, we identify three design goals. The first design goal is the reduction of writes (**DG1**), which is based on the write endurance and read-write asymmetry. Due to the byte-addressability, much finer-grained accesses are possible that should be exploited (**DG2**). The direct load and store semantics further enable zero-copy memory mapping and, thus, new opportunities to ensure consistency and durability, e.g., by atomic primitives (**DG3**).

In the following, we extract the design primitives focusing on tree-based structures from the literature and connect them with our defined design goals. We have primarily worked with $B^+$-Trees, LSM-trees and single node structures since these already cover a vast part of the design space. The question we want to answer is what impact certain design primitives have in which scenarios in the presence of PM. The goal is to reveal their trade-offs for facilitating design decisions. This must be done in the form of white-box testing to avoid side effects in the measurements. As a result, we envisage a profile per design primitive, from which performance and memory impacts can be derived for each kind of access pattern.

**Definitions.** Similar to [6], we define a *design primitive* in this context as an indivisible layout or access concept. To achieve the goals mentioned above, it is necessary to reveal the possible primitives taking into account the properties of PM. For that, we study the approaches found in the literature and assign the ideas to the corresponding design goals. Furthermore, we consider existing micro-operations for trees/nodes and set these in relation to the derived primitives. In this context, a *micro-operation* describes a low-level access pattern whose result is independent of the chosen primitive(s). The typical macro-operation like get, insert, update, delete, and scan can be implemented by combining such micro-operations. Therefore, we classify them in read-, insert- and erase-based as well as recovery operations.

**Micro-Operations.** The great advantage of considering micro-operations is the disclosure of bottlenecks and optimization potential, which would be concealed at the macro level. For instance, inserts for hybrid DRAM/PM solutions are always faster than completely persistent ones almost independent of the node layout. Therefore, it is important to keep both access patterns and the design space concise. For the read category, the first micro-operation is the search for a key within a node (lookup). To get the target node, there are usually two types of traversing the tree, namely vertical (tree traverse from top) and horizontal (tree iterate from lowest left). The macro-operations get and scan can be built by combining lookups and traversals. Next, there are insert-based operations like placing records into a node. This can lead to splits, which require the allocation of new nodes. An insert or update macro-operation would need the micro-operations lookup, traversal, insert, and split. Erasing an entry from a node is a micro-operation downsizing the tree. This may cause an underflow which can be resolved by

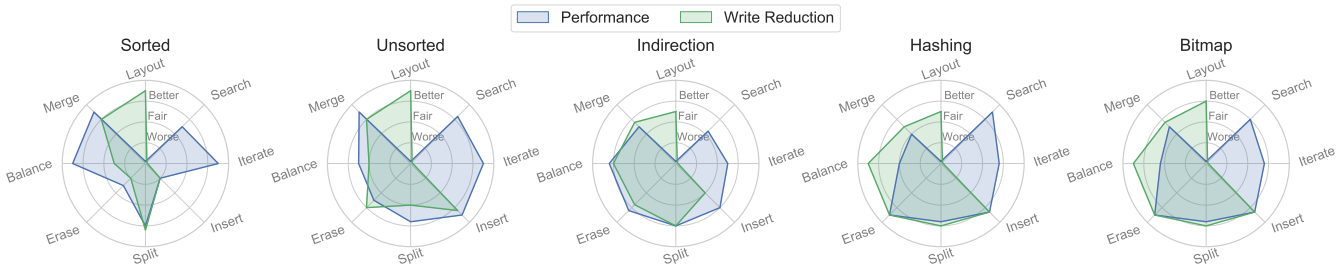


**Figure 1: Performance profiles of a set of design primitives for PM.**

balancing or merging with another node. The typical delete macro-operation consists of search, traversal, erase, balance and merge. The last class is recovery. It mainly consists of the operations of the read category combined with the recreation of volatile DRAM structures. Persisting operations depend on the primitives of DG3.

**Primitives.** We now briefly describe a set of found primitives. For the first design goal - reducing writes - the node layout was reconsidered and the main consensus was to leave data nodes unsorted [1, 3, 5, 8, 10, 14]. To keep the access fast indirection [3], hashing [10, 13], and bitmaps [3, 8, 10], as well as combinations of these, were used. For this, appropriate auxiliary structures are added at the start of each node. To further save writes, the node placement was adapted leading to selective persistence by placing some parts (e.g., inner nodes) in DRAM [5, 10, 13, 14]. Depending on the node layout there are different access primitives (DG2). A simple linear search over the keys is always possible. If the entries are sorted, a binary search is applicable. Both algorithms can also be modified with more fine-granular access using the cache-line-sized auxiliary structures. When splitting a node, as typical in B-Trees and Skip-Lists, we identified two approaches. The basic algorithms move all keys greater than the split key to the new node [1, 3, 5, 8, 11, 14]. An alternative when using a bitmap is to copy the full node, reset the greater keys in the bitmap, and finally store the inverted bitmap in the new node [10]. Regarding failure atomicity (DG3), the de facto standard Persistent Memory Development Kit (PMDK) [7] provides general-purpose transactions that can be placed around the algorithms and allocations. Alternatively, PMwCAS [12] could be used [1], which provides compare-and-swap operations for ranges bigger than eight bytes. Another method is to individually persist the data using flush and fence instructions [3, 8, 10, 11, 13, 14].

## 3 PERFORMANCE PROFILES

In our experiments, we focus on the micro-operations on tree-like data structures as introduced in the previous section. From the primitives described, we picked for the node layout: sorted, unsorted, indirection + bitmap ("indirection"), hash-probing + bitmap ("hashing") and bitmap only. For the access primitives, binary search with and without using indirection as well as linear search with and without using hashing and bitmaps are tested. We re-implemented the approaches from the literature – using PMDK – focusing on the corresponding primitive(s). The aim and contribution are to evaluate the design primitives independently of their original context and to compare their strengths and costs. Our implementations can be examined via our public repository.[1]

[1]https://github.com/dbis-ilm/PMem_DS

Using the results from our experiments, we created performance profiles summarizing the performance and write reduction of the main identified primitives in this paper (see Figure 1). This bases on the node organization and their corresponding access primitives.

**Insights.** The results of the experiments gave us some interesting general insights for designing data structures, choosing corresponding access primitives, and combining various ideas. The details of the evaluation can be found in [4].

(I1) There are still numerous untested possible primitives and combinations of them. Explicitly for $B^+$-Trees, we propose hash probing and bitmap for data nodes (1 KiB) and a sorted layout for inner nodes residing in DRAM. If inner nodes also need to be persistent and there are many writes, they should rely on indirection. This is basically a combination of the ideas from [3] and [10].

(I2) A hybrid DRAM/PM approach is highly recommended when aiming for the best performance and still requiring persistence. In particular, dereferencing and pointer chasing have an even greater impact on PM than on DRAM.

(I3) Jumping between non-sequential cache lines is quite expensive. This is evident, e.g., for binary *search* in a *sorted* node and using *indirection* to *iterate* (see Figure 1). Although PM allows byte-addressable random access, sequential access is still preferable.

(I4) The optimal size for nodes located in PM-based structures lies between 256 bytes and 1 KiB. The lower bound of 256 bytes results from the write-combining buffer of the DCPMMs. The upper bound is the size at which the performance typically drastically degrades and auxiliary structures grow beyond a cache line.

## 4 CONCLUSION

Ultimately every design decision depends on the specific application and there is no single best solution. This paper is intended to help determine the optimal PM-specific design parameters and primitives for a given use case. For instance, for a write-intensive workload with many structural adjustments, indirection is the primitive of choice. If there are many point queries, hashing is best suited. To accelerate deletes, a bitmap should be added. If mainly iteration through the data nodes is required, no auxiliary structure should be used. Our investigations still offer much potential for extension. For future work, further data structures, primitives, and access patterns shall be studied. The final result should be a far broader derived performance profile per design primitive as sketched in Figure 1.

## ACKNOWLEDGMENTS

This work was funded by the German Research Foundation (DFG) within the SPP2037 under grant no. SA 782/28.

# REFERENCES


[1] 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. http://www.vldb.org/pvldb/vol11/p553-arulraj.pdf

[2] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *EDBT*. OpenProceedings.org, 461–466.

[3] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797. https://doi.org/10.14778/2752939.2752947

[4] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data Structure Primitives on Persistent Memory: An Evaluation. *CoRR* abs/2001.02172 (2020). arXiv:2001.02172 http://arxiv.org/abs/2001.02172

[5] Weiwei Hu, Guoliang Li, Jiacai Ni, Dalie Sun, and Kian-Lee Tan. 2014. $B^P$-Tree : A Predictive $B^+$-Tree for Reducing Writes on Phase Change Memory. *IEEE Trans. Knowl. Data Eng.* 26, 10 (2014), 2368–2381. https://doi.org/10.1109/TKDE.2014.5

[6] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[7] Intel Corporation. 2019. Persistent Memory Development Kit. http://pmem.io/pmdk. Accessed: December 27, 2019.

[8] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *TOS* 14, 1 (2018), 5:1–5:17. https://doi.org/10.1145/3129263

[9] Micron Technology, Inc. 2019. 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. Accessed December 27, 2019.

[10] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 371–386. https://doi.org/10.1145/2882903.2915251

[11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*. 61–75. http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman

[12] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

[13] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[14] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang