Distributed and Parallel Databases (2022) 40:47–66
https://doi.org/10.1007/s10619-021-07327-0

Selective caching: a persistent memory approach for multi-dimensional index structures

Muhammad Attahir Jibril[1] · Philipp Götze[1] · David Broneske[2,3] · Kai-Uwe Sattler[1]

Accepted: 22 February 2021 / Published online: 14 March 2021

Abstract

After the introduction of Persistent Memory in the form of Intel's Optane DC Persistent Memory on the market in 2019, it has found its way into manifold applications and systems. As Google and other cloud infrastructure providers are starting to incorporate Persistent Memory into their portfolio, it is only logical that cloud applications have to exploit its inherent properties. Persistent Memory can serve as a DRAM substitute, but guarantees persistence at the cost of compromised read/write performance compared to standard DRAM. These properties particularly affect the performance of index structures, since they are subject to frequent updates and queries. However, adapting each and every index structure to exploit the properties of Persistent Memory is tedious. Hence, we require a general technique that hides this access gap, e.g., by using DRAM caching strategies. To exploit Persistent Memory properties for analytical index structures, we propose *selective caching*. It is based on a mixture of dynamic and static caching of tree nodes in DRAM to reach near-DRAM access speeds for index structures. In this paper, we evaluate selective caching on the OLAP-optimized main-memory index structure Elf, because its memory layout allows for an easy caching. Our experiments show that if configured well, selective caching with a suitable replacement strategy can keep pace with pure DRAM storage of Elf while guaranteeing persistence. These results are also reflected when selective caching is used for parallel workloads.

Keywords Persistent memory · Non-volatile memory · Index structures · Data management · Databases

✉ Muhammad Attahir Jibril
muhammad-attahir.jibril@tu-ilmenau.de

Extended author information available on the last page of the article

1 Introduction

In the competition for ever-increasing performance for cloud computing, cloud systems providers grant more and more access to specialized hardware. After offering dedicated GPUs and FPGAs, cloud providers like Google now offer Persistent Memory (PMem) as their new selling point. However, PMem is of no use for cloud applications that do not exploit its properties [27].

Depending on the underlying technology, the density and the performance of reading and writing on PMem differ. Although first promoted performance numbers suggested that reads on PMem are almost as fast as on DRAM, the real behavior is different. This paper is based on Optane DC Persistent Memory Modules (DCPMMs), which under heavy load cannot keep up with the latency of DRAM. This circumstance suggests that PMem is just filling up the gap between SSD and DRAM [9]. Hence, read and write-heavy applications such as main-memory database management systems have to cope with the new challenges by adapting their data structures to this new architecture.

Recent solutions for the architectural challenges of PMem—especially for index structures—consist in keeping the essential part of the data structure in PMem and, for faster access, the reconstructible part in DRAM. Especially PMem-tuned B$^+$-Tree structures and algorithms (e.g., NVTree [40], FPTree [28], FAST & FAIR [15], wB$^+$-Tree [6], CDDS-Tree [36]) can rely on the property that all data is redundantly stored in the leaf nodes. These nodes can be easily used to reconstruct parts of the upper tree that are kept in DRAM. Such selective persistence [28] can be used to balance between reconstruction effort (more levels of the tree stored in PMem) and query/maintenance effort (more levels of the tree stored in DRAM).

Despite their simplicity, the persistence approaches for B$^+$-Trees are not directly applicable to other index structures. Particularly in multi-dimensional index structures, the possibility to reconstruct upper tree nodes from the stored data in leaf nodes is lacking due to the stored data. Hence, new methods need to be designed to holistically support PMem for multi-dimensional index structures.

In this paper, we extend our proposed approach, selective caching [17], which was a first step to exploit PMem for multi-dimensional index structures. As a representative, we use Elf [4], a multi-dimensional index structure. Due to its explicit memory layout for main-memory-optimized database systems, it is a perfect fit for optimization towards PMem. The idea of selective caching is to persist the whole data structure in PMem and buffer nodes of the data structure in DRAM in order to improve query performance. Overall, our experiments show that when well configured, selective caching reaches query performance that is close to DRAM performance while keeping persistence guarantees untouched.

In summary, after an introduction of necessary background (Sect. 2) and related work (Sect. 3), we contribute the following:

- We present selective caching—an approach for caching tree nodes statically and dynamically in DRAM (Sect. 4).

Table 1 Main characteristics of different memory/storage technologies. (cf. [10, 22, 24, 29, 32, 35, 38])

	DRAM	Optane DC PM	NAND Flash
Idle seq. read latency	80 ns	175 ns	16 µm
Idle rand. read lat.	90 ns	325 ns	200 µm
Max. read bandwidth	85 GB/s	32 GB/s	3 GB/s
Max. write bandwidth	46 GB/s	13 GB/s	0.6 GB/s
Write endurance	$> 10^{15}$	*N*/A	$10^4 - 10^5$
Density	$1X$	$2X - 4X$	$4X - 8X$

- As an extension to our prior work [17], we implement several replacement strategies for dynamic caching (Sect. 4.2).
- As a baseline, we evaluate the impact of PMem storage compared to keeping the whole Elf in DRAM (Sect. 5.3). This evaluation shows clear deficiencies for PMem-only storage.
- We evaluate different configurations as well as eviction policies of our selective caching approach for different query types on uniform synthetic data and TPC-H data with correlated dimensions (Sect. 5.4). We show that selective caching effectively counters the PMem deficiencies, when well configured.
- We also analyze harder use cases for caching, which is running parallel queries on the Elf (Sect. 5.4). We conclude that apart from general benefits of caching for parallel workloads, there are use cases where a perfect caching strategy still needs to be found (Sect. 6).

2 Background

In this section, we first introduce the necessary background to understand PMem characteristics and workings. Afterwards, we introduce the selected multi-dimensional index structure Elf.

2.1 Persistent memory characteristics

The most common PMem technologies are PCM [37], STT-MRAM [14], and memristor [33]. However, up to now, only the 3D XPoint technology is available as Optane DCPMM [7]. All technologies provide byte-addressability, persistence, and DRAM-like performance. They can be directly accessed through the memory bus using the CPU's load and store instructions without the need for OS caches. Furthermore, they scale better in terms of capacity, while DRAM is hitting its limits.

In the remainder of the paper, we focus on Optane DCPMMs due to their availability. Table 1 classifies this product in comparison to today's typical DRAM and NAND flash. The latency and bandwidth are measured on our own system (see Sect. 5). Due to a write-combining buffer within the PMem modules, it is difficult to measure actual write latencies. Therefore, we give bandwidth measures instead. Currently, there are two operating modes of the DCPMMs, namely *Memory* and *App*

Direct (or a mixture). The *Memory* mode extends the main memory capacity by utilizing DRAM as a cache above PMem. There are no persistence guarantees in *Memory* mode, but existing in-memory applications work out of the box with it. The *App Direct* mode provides persistence and allows full utilization of the device. However, developers still have to handle failure-atomicity, concurrency, and performance.

On the software part, we used the Persistent Memory Development Kit (PMDK) [16] to access and manage data on PMem. Several included libraries offer different abstraction levels and relieve the developer of some common steps. In this work, we used the C++ bindings of the *libpmemobj* library which provides general-purpose transactions and object management. In the following, the used terms and concepts of this library are briefly explained.

Persistent memory pools. PMem is managed by the operating system using a PMem-aware file system that grants applications direct access to PMem as memory-mapped files. These files are called *pools* in this context. *libpmemobj* provides interfaces to easily create, open, manage, and close those pools.

Persistent pointers. A persistent pointer to a persistent data object contains an 8-byte ID of the persistent memory pool and an 8-byte offset of the object within this pool. Since the actual address of a memory-mapped region can differ for each instance of the application, persistent pointers are used to map back objects in the virtual address space of the application.

Root object. A root object is an object to which all other data structures in the pool are attached. It is allocated from the pool, initially zeroed, has a user-defined size, and always exists. A persistent pointer to the root object is kept at a known offset, which enables the application to recover its data.

Persistent properties. Another class template within PMDK is called persistent property. By wrapping a variable with this property, all modifications are atomically registered without adding any extra storage overhead. Sect. 4.1 explains where these concepts are used in Elf.

2.2 The Elf storage layout

Elf is a multi-dimensional structure that clusters column values according to their prefix. Elf is well suited for analytical workloads due to its main-memory optimized storage layout. In the following, we outline Elf's key design choices, which is necessary for understanding our PMem adaptations to Elf.

Design principles and optimizations. Conceptually, Elf is a prefix tree similar to ART [20] which, however, works on the granularity of column values instead of digits. Hence, each level in the tree corresponds to the values of a specific column. In each node, the entries are sorted, ultimately introducing a total order into the data and allowing pruning within a node. In Fig. 1b, we visualize the conceptual Elf built for the example table in Fig. 1a consisting of four columns and six tuples. As data-sensitive optimizations, Elf features two different node types: `DimensionLists`, labeled (1), (2), (5), and (6) as inner nodes that hold *sorted* column values of several tuples and `MonoLists` as a special type of `DimensionList`, labeled (3), (4), (7), (8), (9), and (10), that represent values of a single tuple spanning across

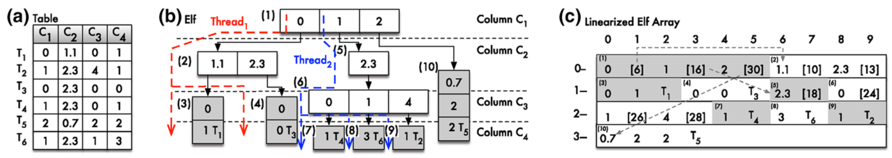


Fig. 1 Example table (**a**), conceptual Elf (**b**), and the OLAP and main-memory optimized storage layout of Elf (**c**)

several columns. The idea of `MonoLists` is that whenever there is no branch-out on deeper levels, the linked lists are merged to a single `MonoList`, thus eliminating pointers and distributed storage. To this end, on the upper level, Elf is similar to a column store. On deeper levels, it slowly converges to a row-store-like layout. This effectively compresses the data set [5].

Another optimization that is particularly designed for read-intensive analytical workloads is the linearization of Elf, which we show in Fig. 1c. Here, each `DimensionList` and `MonoList` is stored in a contiguous array. What used to be pointers are now offsets within the array itself. This optimization, which has also been applied for B-Trees [30], has proven to accelerate selections in Elf by a factor of 10 [5].

Parallel search algorithms. To evaluate the impact of parallel workloads on PMem, we also executed multi-threaded search algorithms on our Elf. For an efficient parallel search, there are different strategies to split the search space—especially for range queries. However, we found that a fine-granular splitting (on the granularity of single `DimensionLists`) adds too much synchronization overhead [3]. Hence, an interleaved per-subtree parallelization proved to be the best parallel search algorithm. This means that each thread is assigned an entry from the first `DimensionList` until all available threads are busy. Once a thread finishes, it is assigned the next entry until all entries of the first `DimensionList` are covered. For instance, given the Elf in Fig. 1 with three values in the first `DimensionList` and two threads, the first two entries (`0` and `1`) and their subtrees will be individually evaluated by these two threads. The first thread to finish will then work on the third entry. This strategy has shown to work well for the inherently imbalanced subtrees of Elf [3].

3 Related work

PMem-based data structures. Most prior work on PMem-based data structures focuses on B^{+}-Trees [1, 6, 23, 28, 36, 40] targeting OLTP systems. Their main consensus is to leave nodes unsorted and generally reduce writes. There is also some work on radix trees [19], LSM-Trees [18, 21] and hash maps [26, 31]. To the best of our knowledge, so far, only [8] considers a multi-dimensional layout and analytical queries. It bases on a clustering approach and unsorted blocks covering a three-tier architecture (DRAM, PMem, SSD). However, in contrast to Elf, this approach is only suitable as a storage layout and cannot serve as an index. Still, a combination of both seems promising. Since former experiments work on emulated PMem, in [22], the authors re-evaluated the B$^+$-Tree variants on real hardware. In [10, 11], the underlying primitives of these trees are also analyzed on real PMem.

Selective persistence. To keep up the performance of persistent data structures compared to volatile counterparts, only necessary fractions of data are stored in PMem. The remaining part is placed in DRAM and rebuilt upon recovery. In the FPTree [28] and a derivative of it optimized for 3D XPoint called LB$^+$-Tree [23], the leaf nodes are placed in PMem using a persistent linked-list, while the inner nodes are placed in DRAM. Consequently, only accessing the leaves is more expensive compared to the volatile tree, while minimizing the use of DRAM. HiKV [39] runs on hybrid memory, too: a hash index is placed in PMem and a B$^+$-Tree in DRAM. Thus, it allows for fast searching of the hash index for basic key-value operations (Put, Get, Update, Delete) which require locating the key-value item. However, for operations (Scan) that benefit from sorted indexing, the hybrid index employs a B$^+$ -Tree, whose updating involves many writes due to sorting, splitting, and merging of nodes and is, thus, placed in DRAM. The DPTree [41] also utilizes a volatile B$^+$ -Tree backed by a PMem log serving as a buffer. Once this buffer reaches the defined capacity, it is merged into the base tree. This is implemented as a volatile radix tree for navigating to persistent leaf nodes. A versioning scheme is used to support crash consistent merging. Instead of designing individual hybrid data structures, in [34] and [2], the authors investigate general-purpose multi-tier buffer management covering DRAM, PMem, and SSDs. This is a similar direction we strive for with our dynamic caching approach (Sect. 4.2).

4 Selective caching for Elf

In this section, we first present our baseline implementations, which are naïve translations of the Elf to PMem. Afterwards, we discuss our improvement of selective caching as a strategy to exploit available DRAM as a cache.

4.1 Naïve PMem-based approaches

We propose two naïve PMem approaches: the Elf can be stored in PMem only, which means full persistence but also a possible performance degeneration under heavy load. The exact impact is an objective of the first experiment of our evaluation. The second possibility (*hybrid Elf*) is to create a redundant copy of Elf in DRAM, giving the best query performance, but at the cost of doubling the size.

Pure PMem-based Elf. To make Elf suitable for PMem, we rely on PMDK described in Sect. 2. The use of its features is illustrated in Fig. 2. The persistent tree is stored as a data object residing in a persistent memory pool. On opening, the position of the Elf object is determined by following the root and subsequent persistent pointer (see Sect. 2.1). We then always access the persistent Elf via the current

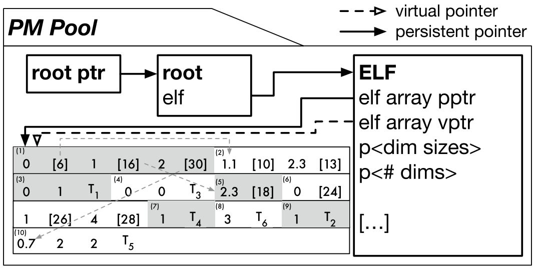


Fig. 2 Organization of persistent Elf in PMem pool

virtual object pointer.[1] The linearized Elf array is stored separately and reachable from the Elf object. Similarly, the virtual address of the array is stored to avoid costly persistent dereferencing. Another drawback of persistent pointers is that they are twice the size of virtual pointers. Actually, it is not necessary to store volatile pointers in the persistent pool, but it helps with the visualization of our utilization of them. To ensure atomicity, we used *libpmemobj* transactions in memory allocations for the persistent Elf object, the persistent Elf array, and the index build. We additionally wrap the member variables of the persistent Elf class, such as the sizes and the number of dimensions, with the persistent property class. Although in our experiments we do not modify the tree after initially building it, this is reasonable for later inserts or in-place updates. Due to its size, the data array is not wrapped as one persistent property. Instead, the modified ranges in the array need to be manually added to a transaction.

Hybrid Elf. In the hybrid Elf, we propose to create a volatile copy in DRAM upon the initial build of the persistent Elf (or reproduced upon subsequent restarts). All queries are run on the volatile Elf. We argue that the hybrid Elf results in query performance at DRAM speed. Moreover, we save the cost of rebuilding in case of a system failure, and we can alternatively execute queries on the persistent Elf if the available DRAM is insufficient to hold the copy.

However, the hybrid Elf has a performance and memory overhead. The performance overhead is the extra time required for constructing the volatile copy of Elf on DRAM, which we expect to be negligible compared with the initial build time of the persistent Elf. Moreover, since Elf mainly supports periodic insertions for its primary field of application like data warehousing, there would be recurrent copying of Elf to DRAM. The memory overhead of hybrid Elf is the DRAM space used to hold the volatile copy.

[1] A virtual object pointer is mapped to different addresses in the virtual memory address space at different instances of the application. We obtain the current memory address by dereferencing the persistent object identifier into a virtual object pointer.

4.2 Selective caching

Cached Elf. In case that keeping a full volatile copy of persistent Elf in DRAM (cf. hybrid Elf) needs too much space, we propose to cache crucial parts of Elf that are most frequently traversed in query execution in DRAM. There are two strategies to build up such a DRAM cache:

- *Dynamic caching* The first naïve way is to cache every traversed `DimensionList` in a hash map. This can be extended by a replacement/eviction strategy to limit the cache size. However, CPU caches are already dynamic caches and, hence, an additional dynamic cache has to be well designed to give an edge over CPU caches.
- *Static caching* Instead of dynamically caching, an alternative is to cache a fixed (static) part of Elf such as the first `x` dimension levels directly at build or recovery time. Hence, we do not have to probe the cache and the persistent Elf but know directly which part is in DRAM or in PMem.

Static caching creates an FPTree-like hybrid layout [28], keeping inner nodes in DRAM and leaf nodes in PMem. In addition, the dynamic approach can be used on the lower dimensions resulting in a split cache.

Replacement strategies for dynamic caching. Regarding the replacement strategy, Lersch et al. [21] already investigated the application of a dynamic cache with eviction policy in the context of LSM-Trees on PMem. As these have shown possible benefits for a read-only setup, we envisage to re-use the tested LRU (least recently used) and the more optimized 2Q policies in our analytical context. The latter basically divides the cache into two separate queues each implementing, e.g., LRU. One is holding the actual cached data items (AM) and the other only stores IDs—offsets in case of Elf—of uncached items (A1). If an item is requested and is not in any of the queues, its ID is stored in A1 and the data is loaded directly from PMem. If it is present in A1, the data is first copied to AM and the item is accessed over the DRAM cache. Otherwise, we have a cache hit in AM. Apart from LRU and 2Q, we add another typical candidate, namely LFU (least frequently used), which we further try to optimize as described in the following.

Our idea to populate the cache is to use probabilities for traversing each `DimensionList`. The probability assigned to a `DimensionList` is the ratio of the number of tuples, for whose retrieval the `DimensionList` is traversed, to the total number of inserted tuples. We start with the `DimensionList` of the first dimension which has a probability of one because traversals always begin from it. Every `MonoList` has a probability of the inverse of the number of inserted tuples because each `MonoList` is traversed to retrieve only a single TID. All other `DimensionLists` have probabilities as per the data and its prefix redundancies. Consider Fig. 1 with six inserted tuples. Since the `DimensionList` (6) is traversed to retrieve three tuples, it has a probability of $\frac{3}{6}$. Similarly, `DimensionLists` (1), (2), and (5) have probabilities of one, $\frac{2}{6}$, and $\frac{3}{6}$, respectively. Each of the `MonoLists` (3), (4), (7), (8), (9), and (10) has a probability of $\frac{1}{6}$. Note that, although `DimensionList` (6) is at a lower dimension

Table 2 Environment

Processor	2× Intel® Xeon® Gold 5215, 10 cores / 20 threads each, max. 3.4 GHz)
Caches	32 KB L1d, 32 KB L1i, 1024 KB L2, 13.75 MB LLC
Memory	2×6×32 GB DDR4, 2×6×128 GB Intel® Optane™DCPMM
OS & Compiler	CentOS 7.8, Linux 5.7.7 kernel, cmake 3.15.3, GCC 9.3.1 (-O3), PMDK 1.9.1

than (2), it still has a higher probability. Furthermore, `DimensionList` (5) has a probability of $\frac{3}{6}$ even though it has only one `DimensionList` element. After obtaining a 'tree' of probabilities, we choose which `DimensionLists` to cache in DRAM based on cut-off probabilities and the DRAM memory space allotted to caching the Elf. This approach is either implemented dynamically as part of an eviction policy or statically at the time the tree is built. The former is what we refer to as LLA (least likely to be accessed) in the experiments below. It behaves similarly to LFU but uses fixed frequency values (probabilities) for each `DimensionList` and it will only evict the `DimensionList` that is least likely to be accessed from the cache if the new `DimensionList` has a higher probability.

Overall, our two introduced approaches for selective caching (i.e., dynamic and static caching) have different advantages, which we investigate in the following section. However, selective caching in Elf comes at a price: query execution consists of switching between PMem and DRAM, which incurs penalties of more cache misses especially for the dynamic parts. Furthermore, the static parts cost extra build and recovery time.

5 Evaluation

In our experiments, we investigate the performance of Elf for the beforementioned persistent variants. We focus on the building time as well as three query types, namely exact-match, range, and partial-match. Obviously, the persistent Elf will be slower than the volatile counterpart. Hence, we want to quantify this overhead. Thereafter, we validate the optimization techniques proposed in Sect. 4.2, which should reduce the overhead. As a result, we show that with sophisticated optimizations, a DRAM-like performance is possible.

5.1 Environment

Our experiments were conducted on a dual-socket Intel Xeon Gold 5215 server as outlined in Table 2. Each socket is equipped with six DCPMMs interleaved to one region and namespace. The PMem modules are operating in *App Direct* mode via an *ext4* file system and *dax* mount option. All experiments allocate their resources always on the same socket to preclude NUMA effects.

5.2 Experimental setup

We carried out our experiments on two data sets. The primary set is a table of 100 million rows and 10 dimensions that follows a uniform distribution over all inserted tuples. Each dimension is of integer type[2]—with a cardinality of 100 if not stated otherwise—resulting in approximately 4 GiB. To show that our optimization techniques are also valid for more realistic data with characteristics such as correlation between dimensions, we also conducted experiments on the TPC-H Lineitem table (with 15 columns) at scale factor 10 (about 3.5 GiB). As Elf is built around prefixes of columns, the order of dimensions determine the structure of the index. Thus, we reordered the columns of the table in ascending order of cardinality to exploit prefix redundancies and maximize benefits of caching `DimensionLists`.

For the queries, we used a Zipfian distribution [13] with skewness parameter $\theta = 0.5$ to simulate a more realistic access pattern. The experiments were repeated at least ten times in a single-threaded environment (except for parallel measurements) to obtain reliable measurements. Besides the building, the three tested query types are briefly explained below. The throughput of these queries is expressed in queries per second (qps).

Exact-match query. The exact-match query takes one equality predicate for each dimension and returns the TID of the matching tuple. In each run, we first selected tuples from the table according to a Zipfian distribution, then used all dimension values of each tuple for constructing the exact-match selection predicates.

Range query. The range query returns a list of TIDs, as per the selection predicates. Two sets of `x` values define the lower and upper boundaries of the selection predicates for the respective `x` dimensions. In each run, we similarly selected tuples, based on a Zipfian distribution, whose dimension values served as lower boundaries and set the upper boundaries according to the defined range size (which we specify in due course).

Partial-match query. A partial-match is a special form of range query. The boundaries are used in the search only for pre-selected dimensions. All other dimensions are wildcarded and, hence, all dimension values are evaluated. We select the lower and upper boundaries in the same way as for the range queries and set the dimensions on which the boundaries are evaluated as well as those that are wildcarded.

5.3 DRAM vs. PMem

At first, we investigate the performance overhead of the pure PMem-based Elf against its DRAM counterpart on the uniform data set. In Fig. 3, the corresponding measurements are depicted. The reported runtimes are averages of 1M, 10K, and 1K executed queries for the three query types respectively. The range and

[2] Notably, wider data types increase the used DRAM-cache size, but also put more pressure on CPU caches. Hence, the DRAM cache becomes even more valuable as a layer between CPU caches and PMem.

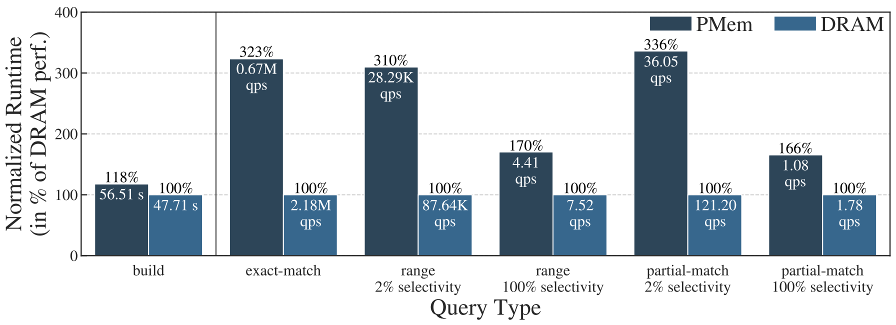


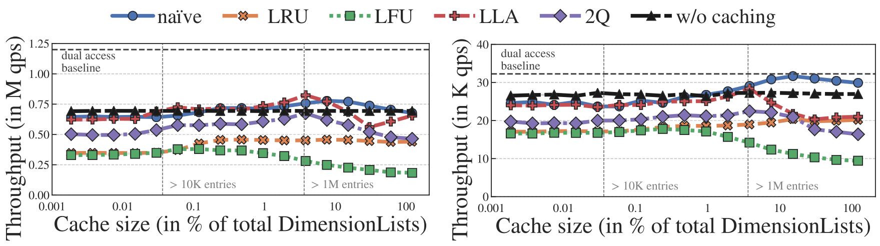
Fig. 3 Build and query performance of Elf

partial-match queries use a range size of 2% and 100% per set dimension to demonstrate the extremes. As expected, DRAM exhibits a better performance than PMem. The overhead of building the Elf and of executing the three query types yield to `18%`, `223%`, `210%/70%`, and `236%/66%`, respectively. For range and partial-match queries with a higher range size, the runtimes of the volatile and persistent versions are much closer. Greater ranges lead to more sequential access pattern and more commonly traversed `DimensionLists` which will end up in the CPU caches for both the volatile and the persistent version. This is not the case for most exact-match queries due to their tiny query windows. Altogether, our results show that the performance gap between DRAM and PMem is wider for queries—especially exact-match queries—than for building. Our explanation is that a sequential access pattern is better supported on PMem than a random one. Particularly during building, the write-combining buffer of PMem seems to be quite efficient if there is only a single sequentially writing thread. In the following experiments, we primarily focus on a low selection percentage since random access patterns offer the greatest potential for improvement by selective caching.

5.4 Optimizations

Hybrid Elf. As a first optimization step, we evaluate the build and recovery performance of the hybrid Elf. In this case, the query performance will be the same as for DRAM. The actual measurement we did here is simply the cost of copying the Elf data array to DRAM. For our setup with 100M tuples ($\sim$ 4 GB), this took `1770 ms`. The total building time, thus, increases to `58.3 s` which is a mere `3%` overhead. Furthermore, recovery improves by a factor of around `30`. This solution is therefore highly recommended if there is enough DRAM.

Dynamic caching—eviction policies. Since the last condition is not always given especially for analytical tasks, we next evaluate the different caching approaches described above. We start with the analysis of the eviction policies applied to the dynamic caching and choose the best-performing ones for the subsequent experiments. The experiment includes the naïve, LRU, LFU, LLA, and 2Q strategies as explained in the previous section. During the experiments, we varied the cache size



(a) Exact-match queries. (b) Range queries.

Fig. 4 Throughput of dynamic caching variants on the uniform data set

to identify their optimal setting. On top of that, we added two baselines. First, the pure PMem-based variant (labeled as *w/o caching*) and second, a hybrid variant, which this time, however, only accesses the DRAM copy for `DimensionLists` and always obtains `MonoLists` via the PMem copy of the Elf (labeled as *dual access*).[3] The results are shown in Fig. 4a for exact-match queries and Fig. 4b for range queries.

The caches were warmed up with 100M exact-match queries and the throughput was measured as mean over another 1M. For the range queries, the warm-up includes 100K queries and the measurement is on another 1K. We omit results for partial-match queries as they behave similarly to the range queries.

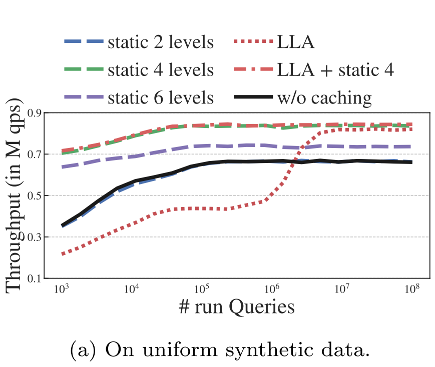
In the case of the uniform data set, the best setting for exact-match queries using the dynamic cache is LLA with a capacity of 1M entries (up to 3.8% of `DimensionLists`), whereas for range queries, naïve with 4M entries (up to 15.2% of `DimensionLists`) performs best. For the TPC-H data, the results for range queries are similar to the uniform data set and, thus, we also opted for the naïve approach with 4M entries. Exact-match queries, on the other hand, had their peak with LLA having 64K entries. Nevertheless, our initial assumption that an additional dynamic DRAM cache must be very sophisticated to keep up with the strengths of PMem and dynamic CPU caches has proven true. Only some of our tested settings could outperform the pure PMem-based Elf. Profiling revealed that the number of instructions between dynamic and without caching is about the same. However, the often worse performance is mainly due to more LLC misses. We identified three hot spots responsible for this behaviour (in warmed-up state):

(HS1) The lookup performance of the hashmap[4] used as part of the caches worsens drastically with increasing size (≈ 25% performance impact).

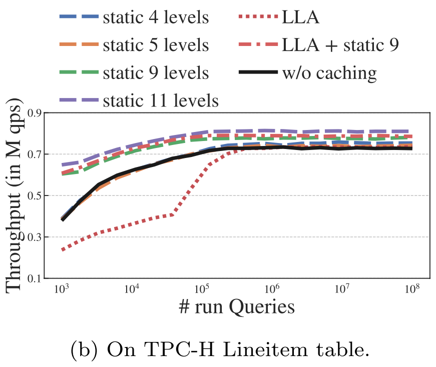
(HS2) The traversal of the `DimensionLists` which could be either in PMem or DRAM (≈ 25% performance impact).

[3] Since the PMem access is actually unnecessary for the second baseline as we already have a copy in DRAM, it is rather a theoretical baseline to show the upper limit.

[4] robin_hood unordered map: https://github.com/martinus/robin-hood-hashing



(a) On uniform synthetic data.



(b) On TPC-H Lineitem table.

Fig. 5 Continuous throughput of cached Elf variants for exact-match queries

(HS3) The access to the always persistent `MonoList` for each query ($\approx$ 50% performance impact).

The first hot spot (HS1) leads to the conflict that an increasing cache size provides higher chances of a DRAM cache hit but simultaneously worsens the lookup performance. We have taken countermeasures by partitioning all cache variants into several smaller hash tables. This has already led to a significant performance gain for larger caches. However, even this could not always surpass the pure PMem case and it probably needs further fine-tuning. The two latter hot spots (HS2 & HS3) cause DRAM and PMem parts to evict each other from the CPU caches. In case of no caching, the CPU caches are mainly used for PMem content. However, we cannot improve much for these two bottlenecks (HS2 & HS3) as they are inevitable steps that are algorithmically the same as without caching. The hybrid dual access baseline shows the theoretical limit if HS1 would be eliminated. The competition of PMem and DRAM for free space in the CPU caches is still included here.

In our initial results [17], we have shown that dynamic caching definitely gives benefits when executing the same set of queries twice (selective warm-up). Now, we query different points or ranges, but still keep a certain skewness and find that dynamic caching is no longer that significantly better. This means that the benefit of dynamic caching highly depends on the similarity or skewness of the queries. Nonetheless, we choose the best performing LLA and naïve strategies as mentioned above for the following experiments.

Throughput over time. Next, we compare the throughput of the three query types for the approaches over a time course expressed in queries run up to that point. In contrast to our prior work [17], the system and caches are not warmed up before to show how the setup and its performance evolves over time. We show the results for the persistent Elf without DRAM caching, the corresponding best dynamic caching, static caching of the first x dimension levels as well as the combination of dynamic and static caches. In Fig. 5, the results of the exact-match queries are shown.

The dynamic caching (LLA) needs the most time to warm up as it has to fill an additional cache during runtime. However, after being warmed up (around 1M–10M

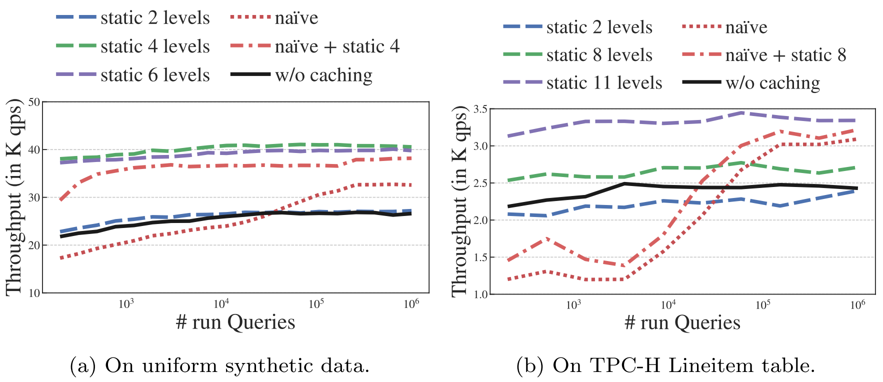


Fig. 6 Continuous throughput of cached Elf variants for range queries

queries), on the uniform data, it can outperform the pure PMem-based Elf by up to 25%. For the TPC-H data only 1–2% were possible. For static caching, we omitted some configurations to keep the figures clean. Caching the first one or two levels statically has similar performance and leads already to a little better initial performance. The peak for this setup could be achieved when statically caching the first four levels. Caching three or more than four levels behaves similarly to *static 6 levels* in Fig. 5a. The correlated Lineitem table reached its peak performance with eleven levels and higher. Again, the behaviour is not linear as, e.g., for four levels the performance goes up, with five levels down again, and from nine levels it gets continuously better. Contrary to what we assumed before [17], more caching levels do not necessarily result in better performance. Rather, the size compared to the CPU caches, successful branch predictions, the commonly accessed `DimensionLists`, and again the size of the underlying hash table[5] are more important. For Fig. 5a, dimension levels one and two completely fit in the L1 cache, level three is slightly larger than L2, and all others are greater than the LLC. For instance, four levels require 136 MiB of DRAM which is 10× the LLC. Compared to the total size of Elf , this is merely `3%` space overhead for about a 30% performance boost. Adding the dynamic cache on top of the static cache with four levels leads to the best currently achieved performance. It also increases the throughput for the TPC-H data. Here, we combined it with nine static levels since with eleven, the static and dynamic caches would contain almost the same `DimensionLists` and there would be no more `DimensionLists` left to be cached by the dynamic part. As mentioned earlier, we reordered the columns of the TPC-H Lineitem table to exploit prefix redundancies. Unique columns do not allow for `DimensionLists`. After reordering the columns, the dimensions twelve to fifteen were the primary and foreign keys, which have no `DimensionLists` except for a relatively few in dimension twelve. Thus the limitation of combining the dynamic cache with nine static levels instead of

[5] Partitioning the hash table, however, did not have a positive effect here.

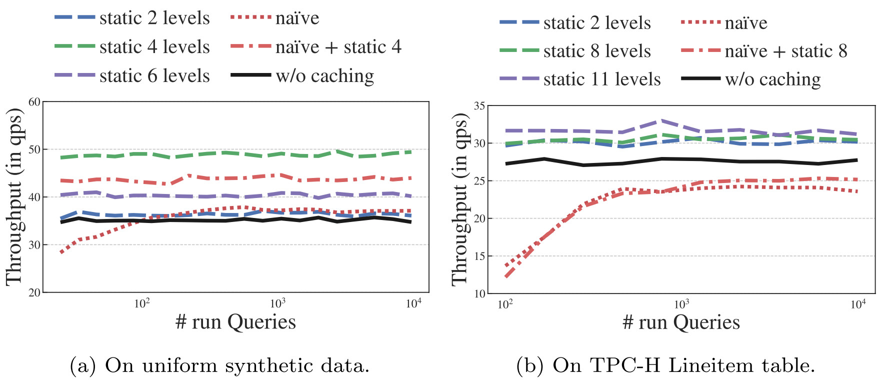


Fig. 7 Continuous throughput of cached Elf variants for partial-match queries

eleven. However, compared to the static-only caching it only led to a small increase, which may not justify the additional invested DRAM. Therefore, we conclude that the combination of dynamic and static caching may not be worthwhile in the current state. Both caching strategies separately, however, can be quite profitable.

Now, we consider range and partial-match queries with a range size of 2%. Since these are long-running queries, the number of queries and the throughput is much lower. Figures 6a, and 7a show the results for the uniform data set. The dynamic approach (this time *naïve*) performs a bit worse for these kinds of queries and is only faster with smaller ranges. The reason for this is that these queries probe much more `DimensionLists` per query. This causes more dynamic cache misses and also utilizes a more sequential access pattern which can be handled more efficiently by PMem (cf. Fig. 3). For partial-match queries, which probe even more `DimensionLists` than range queries due to the wildcarded dimensions, the profit of the dynamic strategy is only minimal (5–10%). For range queries, on the other hand, the performance improves by around 20%. Running the same experiments on the TPC-H data—as shown in Figs. 6b, and 7b—resulted in an improvement of over 30% for range queries. Dynamic caching for partial-match queries led to a deterioration of 10%. The static approach shows nearly the same behaviour as for exact-match queries, although fluctuating a little more. Interestingly, for range queries in Fig. 6a, increasing the number of statically cached levels beyond four does not significantly degrade performance. *Static 2 levels* in Fig. 6b is one of the rare cases where the static caching strategy performs worse than without caching. With the best setting, however, we were able to achieve a performance boost of 50% and 40% compared to the uncached version for the uniform and TPC-H data respectively. For partial-match queries, it resulted in a 40% improvement for the uniform data and 15% for the TPC-H data.

Looking back at Fig. 4b, the static approach is even better than the adapted hybrid dual access version. This additionally supports the idea of selective caching instead of caching all `DimensionLists`. However, for both query types, combining the static with the dynamic cache reduces the throughput, making this option rather

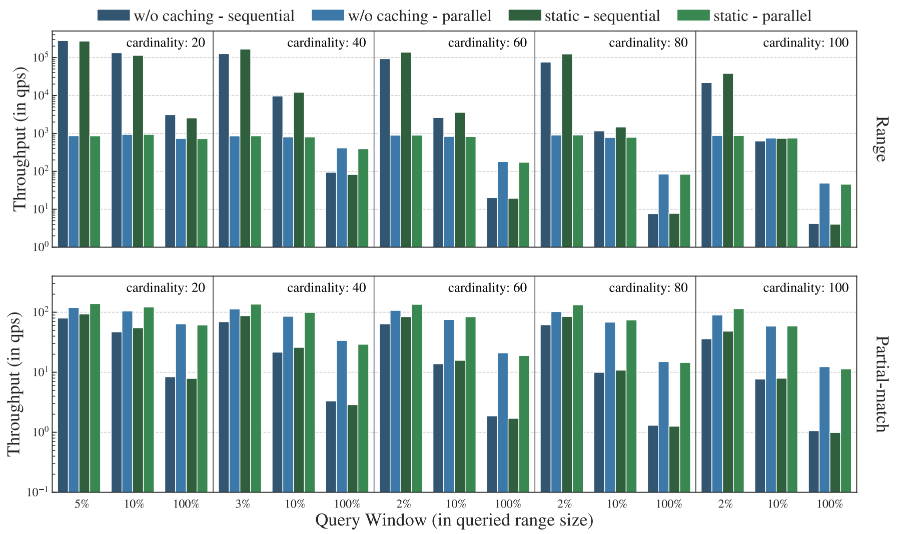


Fig. 8 Sequential vs. parallel range and partial-match queries

counterproductive. It was only with the TPC-H data that we achieved an improvement on range queries—which however was minimal. Hence, we come to the same conclusion as for exact-match queries that in the current state, only the stand-alone strategies are worth the extra DRAM spend.

5.5 Parallel range queries

The last experiment investigates the impact of parallel range and partial-match queries including the application of static caching. Dynamic caching was omitted here since the eviction process would require additional synchronization mechanisms, making it perform much worse. Furthermore, this would add more parameters, complicating the analysis. We compare the parallel and sequential performance both without caching and with static caching (four levels) on the uniform data set since it can be easily customized. In doing so, we vary the cardinalities and the query range size for all 10 dimensions. The experiments were bound to a single socket with the same number of threads as available logical cores (20). The results are shown in Fig. 8.

When looking at the results, it becomes clear that using both no caching and also static caching profit from a parallel execution if the cardinality and range size is large enough. The most extreme example here is shown at the right lower edge with a cardinality of 100 and a range size of 100%. This provides a speedup of 12× compared to the sequential execution. The cardinality and the number of total tuples are upwardly open, which means that the speedup in higher regions will most likely also increase up to a certain extent. With smaller query windows (e.g., the other extreme on the left upper edge with a cardinality of 20 and a range size of 5%), the overhead of creating threads and collecting their results is too high, causing the sequential execution to outperform the parallel one. This worsens the throughput by two orders of magnitude , which is quite drastic. Thus, the parallel implementation cannot be used as a pure replacement. Instead, the system should decide to choose either the sequential or parallel execution based on the passed range parameters. This could be implemented with the help of a cost model calculating a specific threshold for this decision.

The differences between static and w/o caching with the used logarithmic scale are barely visible. However, if we take a closer look, we find that a static cache only provides an additional gain with smaller range sizes. This is consistent with our first experiment, where we had already assumed that there is less potential for improvement with strongly sequential access patterns.

6 Discussion & conclusion

In this work, we investigated various caching approaches to accelerate OLAP queries on multi-dimensional index structures utilizing PMem. In particular, we proposed selective caching consisting of static and dynamic strategies to cache tree nodes in DRAM. In our experimental setup using a skewed distribution, we found that especially random access patterns can highly profit by investing extra DRAM to buffer commonly traversed nodes. For example, we were able to reduce the overhead for exact-match queries from about 223% to 150% compared to a pure DRAM solution with a sole space overhead of 3%. For range and partial-match queries with a 2% range, there is an improvement from about 210% to 110% and 236% to 140%, respectively. However, it has also turned out that a combination of static and dynamic strategies is not always worthwhile. It is necessary to adjust the caching very sophisticatedly to the access pattern to achieve bigger performance advantages. Particularly, the typical eviction policies like LRU, LFU, and 2Q never paid off in our setup. Only the naïve approach, which never replaces entries, and LLA, an approach we devised which works on access probabilities, achieved positive results. Especially the LLA strategy offers even more potential if the probabilities are adapted more precisely to the access patterns. In the parallel investigations, we have found that caching brings additional advantages as well. However, it is more suitable for range queries with a higher selectivity (i.e., smaller ranges). With increasing cardinality of dimensions, parallel execution becomes more lucrative.

Overall, we have shown that selective caching of parts of PMem-based data structures in DRAM is definitely beneficial. Although the research was done on the Elf data structure, we envisage that this approach is generic enough to be easily applicable to other index structures. However, as underpinned with our two different data sets, the granularity of cached objects may differ and sweet spots have to be identified manually or by a suitable cost model. Since we have determined the underlying hash table of the caches as one of the primary bottlenecks, an idea for further optimization is the application of pointer swizzling [12, 25]. Furthermore, instead of only caching `DimensionLists`, we could also add `MonoLists` as cache items. Thus, some queries would run without ever touching PMem. On the other hand, each `MonoList` is only used by a single data point, which may not pay off. Another idea for future work is to enable parallel queries with dynamic caching. This would require either a synchronisation protocol for the eviction policies but could also be realized by leveraging the partitioned hash tables where each partition is assigned to only one thread.

Acknowledgements This work was partially funded by the German Research Foundation (DFG) in the context of the project "Transactional Stream Processing on Non-Volatile Memory" (SA 782/28) and the ADAMANT project (SA 465/51-1) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037). We particularly thank Lucas Lersch for his suggestions regarding the implementation of dynamic caching.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Arulraj, J., Levandoski, J., et al.: BzTree: a high-performance latch-free range index for non-volatile memory. PVLDB **11**(5), 553–565 (2018)
2. Arulraj, J., Pavlo, A., Malladi, K.T.: Multi-tier buffer management and storage system design for non-volatile memory. CoRR abs/1901.10938 (2019)
3. Blockhaus, P.: Parallelizing the Elf: a task parallel approach. Bachelor thesis, University of Magdeburg (2019)
4. Broneske, D., Köppen, V., et al.: Accelerating multi-column selection predicates in main-memory—the Elf approach. In: IEEE ICDE, pp 647–658 (2017)
5. Broneske, D., Köppen, V., et al.: Efficient evaluation of multi-column selection predicates in main-memory. IEEE TKDE **31**(7), 1296–1311 (2019)
6. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. PVLDB **8**(7), 786–797 (2015)
7. Cutress, I., Tallis, B.: Intel launches optane DIMMs up to 512GB: apache pass is here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here. Accessed 14 Dec 2020
8. Götze, P., Baumann, S., Sattler, K.: An NVM-aware storage layout for analytical workloads. In: HardBD & Active @ ICDE, pp 110–115 (2018)
9. Götze, P., van Renen, A., et al.: Data management on non-volatile memory: a perspective. DB-Spektrum **18**(3), 171–182 (2018)
10. Götze, P., Tharanatha, A.K., Sattler, K.: Data structure primitives on persistent memory: an evaluation. CoRR abs/2001.02172 (2020)
11. Götze, P., Tharanatha, A.K., Sattler, K.: Data structure primitives on persistent memory: an evaluation. In: 16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, June 15, 2020, pp 15:1–15:3 (2020) https://doi.org/10.1145/3399666.3399900
12. Graefe, G., Volos, H., Kimura, H., Kuno, H.A., Tucek, J., Lillibridge, M., Veitch, A.C.: In-memory performance for big data. Proc VLDB Endow **8**(1), 37–48 (2014). https://doi.org/10.14778/2735461.2735465
13. Gray, J., Sundaresan, P., et al.: Quickly generating billion-record synthetic databases. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24–27, 1994., pp 243–252 (1994). https://doi.org/10.1145/191839.191886

14. Hosomi, M., Yamagishi, H., et al.: A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM. IEEE IEDM, pp 459–462 (2005)
15. Hwang, D., Kim, W., et al.: Endurable transient inconsistency in byte-addressable persistent B+-tree. In: USENIX FAST, pp 187–200 (2018)
16. Intel Corporation Persistent Memory Development Kit. http://pmem.io/pmdk. Accessed 14 Dec 2020
17. Jibril, M.A., Götze, P., Broneske, D., Sattler, K.: Selective caching: a persistent memory approach for multi-dimensional index structures. In: 36th IEEE international conference on data engineering workshops, ICDE workshops 2020, Dallas, TX, April 20–24, 2020, pp 115–120 (2020). https://doi.org/10.1109/ICDEW49219.2020.00010
18. Kannan, S., Bhat, N., et al.: Redesigning LSMs for nonvolatile memory with NoveLSM. In: USENIX ATC, pp 993–1005 (2018)
19. Lee, S.K., Lim, K.H., et al.: WORT: write optimal radix tree for persistent memory storage systems. In: USENIX FAST, pp 257–270 (2017)
20. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: IEEE ICDE, pp 38–49 (2013)
21. Lersch, L., Oukid, I., et al.: An analysis of LSM caching in NVRAM. In: DaMoN @ SIGMOD, pp 9:1–9:5 (2017)
22. Lersch, L., Hao, X., et al.: Evaluating persistent memory range indexes. PVLDB **13**(4), 574–587 (2019)
23. Liu, J., Chen, S., Wang, L.: LB+-trees: optimizing persistent index performance on 3DXPoint memory. PVLDB **13**(7), 1078–1090 (2020). https://doi.org/10.14778/3384345.3384355
24. Mittal, S., Vetter, J.S.: A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE TPDS **27**(5), 1537–1550 (2016)
25. Moss, J.E.B.: Working with persistent objects: to swizzle or not to swizzle. IEEE Trans. Softw. Eng. **18**(8), 657–673 (1992). https://doi.org/10.1109/32.153378
26. Nam, M., Cha, H., et al.: Write-optimized dynamic hashing for persistent memory. In: USENIX FAST, pp 31–44 (2019)
27. Oukid, I., Lersch, L.: On the diversity of memory and storage technologies. CoRR abs/1908.07431 (2019)
28. Oukid, I., Lasperas, J., et al.: FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: SIGMOD, pp 371–386 (2016)
29. Rao, D.S., Kumar, S., et al.: System software for persistent memory. In: EuroSys, pp 15:1–15:15 (2014)
30. Rao, J., Ross, K.A.: Making B$^+$-trees cache conscious in main memory. In: SIGMOD, pp 475–486 (2000)
31. Schwalb, D., Dreseler, M., et al.: NVC-hashmap: a persistent and concurrent hashmap for non-volatile memories. In: IMDM @ VLDB, pp 4:1–4:8 (2015)
32. Shanbhag, A., Tatbul, N., Cohen, D., Madden, S.: Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In: DaMoN @ SIGMOD, pp 4:1–4:8 (2020) https://doi.org/10.1145/3399666.3399933
33. Strukov, D.B., Snider, G.S., et al.: The missing memristor found. Nature **453**(7191), 80–83 (2008)
34. van Renen, A., Leis, V., et al.: Managing non-volatile memory in database systems. SIGMOD, 1541–1555 (2018). https://doi.org/10.1145/3183713.3196897
35. van Renen, A., Vogel, L., et al.: Persistent memory I/O primitives. In: DaMoN @ SIGMOD, pp 12:1–12:7 (2019)
36. Venkataraman, S., Tolia, N., et al.: Consistent and durable data structures for non-volatile byte-addressable memory. In: USENIX FAST, pp 61–75 (2011)
37. Wong, H.P., Raoux, S., et al.: Phase change memory. PIEEE **98**(12), 2201–2227 (2010)
38. Wu, Y., Park, K., Sen, R., Kroth, B., Do, J.: Lessons learned from the early performance evaluation of Intel Optane DC Persistent Memory in DBMS. In: DaMoN @ SIGMOD, pp 14:1–14:3 (2020). https://doi.org/10.1145/3399666.3399898
39. Xia, F., Jiang, D., et al.: HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: USENIX ATC, pp 349–362 (2017)
40. Yang, J., Wei, Q., et al.: NV-tree: reducing consistency cost for NVM-based single level systems. In: USENIX FAST, pp 167–181 (2015)
41. Zhou, X., Shou, L., Chen, K., Hu, W., Chen, G.: DPTree: differential indexing for persistent memory. PVLDB **13**(4), 421–434 (2019)

Authors and Affiliations

Muhammad Attahir Jibril[1] · **Philipp Götze**[1] · **David Broneske**[2,3] · **Kai-Uwe Sattler**[1]

Philipp Götze
philipp.goetze@tu-ilmenau.de

David Broneske
david.broneske@ovgu.de

Kai-Uwe Sattler
kus@tu-ilmenau.de

1 Technische Universität Ilmenau, Ilmenau, Germany

2 OvG University Magdeburg, Magdeburg, Germany

3 German Centre For Higher Education Research And Science Studies (DZHW), Hannover, Germany