Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures

Muhammad Attahir Jibril*, Philipp Götze*, David Broneske†, and Kai-Uwe Sattler*
*TU Ilmenau, Germany
Email: {muhammad-attahir.jibril, philipp.goetze, kus}@tu-ilmenau.de
†OvG University Magdeburg, Germany & Anhalt University of Applied Science
Email: david.broneske@ovgu.de
This work was partially funded by the DFG as part of SPP 2037 (SA 465/51-1 & SA 782/28).

***Abstract*—Since the proposal of Persistent Memory, research has focused on tuning a variety of data management problems to the inherent properties of Persistent Memory– namely persistence but also compromised read/write performance. These properties particularly affect the performance of index structures, since they are subject to frequent updates and queries. Nevertheless, the main research focuses on adapting B-Trees and its derivatives to Persistent Memory properties, aiming to reach DRAM processing speed exploiting the persistence property of Persistent Memory. However, most of the found techniques for B-Trees are not directly applicable to other tree-based index structures or even multi-dimensional index structures.**

To exploit Persistent Memory properties for arbitrary index structures, we propose *selective caching*. It bases 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 investigate the opportunities as well as limitations of selective caching on the OLAP-optimized main-memory index structure Elf. Our experiments show that selective caching is keeping up with pure DRAM storage of Elf while guaranteeing persistence.

I. Introduction

After the introduction of SSDs, *Persistent Memory (PM)* is the novel disruptive technology in the field of data storage [1]. PM does not only change the architecture of a database system (i.e., storage hierarchy and persistence methods) [2] but also influences the processing speed of systems. Depending on the memory technique (cf. Table I), the density and latency between reading and writing differ. However, their commonality is a notable difference between read and write latency.

Although first performance numbers suggested that reads on PM are almost as fast as on DRAM, the real behavior is different. This paper is based on Optane DCPMM, which under heavy load cannot keep up with the latency of DRAM. This circumstance suggests that PM is just filling up the gap between SSD and DRAM [2]. Hence, the question arises on how to exploit a system architecture where PM bridges the gap between SSD and DRAM in all data structures of the DBMS.

Recent solutions for the architectural challenges of PM– especially for index structures – is handled by keeping the essential part of the data structure in PM and, for faster access, a reconstructible part in DRAM. Especially PM-tuned B$^+$-Tree structures and algorithms (e.g., NVTree [3], FPTree [4], FAST&FAIR [5], wB$^+$-Tree [6], CDDS-Tree [7]) 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 [4] can be used to balance between reconstruction effort (more levels of the tree stored in PM) 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. Especially multi-dimensional index structures lack – due to the amount of stored data – the possibility to reconstruct upper tree nodes from the stored data in leaf nodes. Hence, new methods need to be designed to holistically support PM for multi-dimensional index structures.

In this paper, we present selective caching as a first step to exploit PM for multi-dimensional index structures. As a representative, we use Elf [8] as a multi-dimensional index structure. Due to its explicit memory layout for main-memory-optimized database systems, it is the perfect fit for optimization towards PM. The idea of selective caching is to persist the whole data structure in PM and buffer nodes of the data structure in DRAM in order to improve query performance. Overall, our experiments show that when being well configured, selective caching reaches query performance that is close to DRAM performance while keeping persistence guarantees untouched. In summary, we contribute the following:

- We present selective caching – an approach for caching tree nodes statically and dynamically in DRAM.
- As a baseline, we evaluate the impact of PM storage compared to keeping the whole Elf in DRAM. This evaluation shows clear deficiencies for PM-only storage.
- We evaluate different configurations of our selective caching approach for different query types. Our results show that the deficiencies by PM can be effectively countered by our selective caching approach.

II. Background

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

A. Persistent Memory Characteristics

The most common PM technologies are PCM [9], STT-MRAM [10], and memristor [11]. However, up to now only

TABLE I: Main characteristics of different memory/storage technologies [13] (cf. [14]–[17])

	DRAM	Optane DC PM	NAND Flash
Idle read latency	80 ns	175 ns	25 μs
Loaded rand. lat.	120 ns	400 ns	N/A
Write latency	80 ns	100 $ns - 2\ \mu s$	500 μs
Write endurance	$> 10^{15}$	N/A	$10^4 - 10^5$
Density	$1X$	$2X - 4X$	$4 - 8X$

the 3D XPoint technology has reached the market and is available as Optane DC Persistent Memory Modules (DCPMMs) [12]. What all technologies have in common is 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 soon hitting its limits. In the remainder of the paper, we focus on Optane DCPMMs due to their availability. Table I classifies this product in comparison to today's typical DRAM and NAND flash. The latencies for DRAM and PM are measured on our system (see Section V). Due to a write-combining buffer within the PM modules it is difficult to measure actual write latencies. Currently, there are two possible operating modes of the DCPMMs, namely *Memory* and *App Direct* (or a mixture). The former mode extends the main memory capacity by utilizing DRAM as a cache above PM. 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 the 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) [18] to access and manage data on PM. Several included libraries offer different abstraction levels and relieve some common steps from the developer. In this work, we used the C++ bindings of the *libpmemobj* library whereby we obtain general-purpose transactions and object management. In the following, the used terms and concepts of this library are briefly explained.

Persistent Memory Pools: PM is managed by the operating system using a PM-aware file system that grants applications direct access to PM 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 template class within PMDK is called persistent property. By wrapping a variable with this property, all modifications are atomically registered without adding any extra storage overhead.

Section IV-A explains where these concepts are used in Elf.

B. 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 PM adaptations to Elf.

Design Principles and Optimizations: Conceptually, Elf is a prefix tree similar to ART [19], which works however 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 allowing pruning within a node. In Figure 1.b, we visualize the conceptual Elf built for the example table in Figure 1.a consisting of four columns and three tuples. As data-sensitive optimizations, Elf features two different node types: `DimensionLists`, labeled (1) and (2), as inner nodes that hold *sorted* column values of several tuples and `MonoLists` as a special type of `DimensionList`, labeled (3), (4), and (5), that represent values of a single tuple spanning across 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 towards a row-store-like layout. This effectively compresses the data set [20].

Another optimization that is specially designed for read-intensive analytical workloads is the linearization of Elf, which we show in Figure 1.c. 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 [21], has proven to accelerate selections in Elf by a factor of 10 [20].

III. Related Work

PM-based Data Structures: Most prior work of PM-based data structures focuses on B^+-Trees ([3], [4], [6], [7], [22]) targeting OLTP systems. Their main consensus is to leave nodes unsorted and generally reduce writes. However, most of these approaches have chosen a PM-only placement. There is also some work on radix trees [23], LSM-Trees [24], [25] and hash maps [26], [27]. To the best of our knowledge, so far only [28] considers a multi-dimensional layout and analytical queries. It is based on a clustering approach and unsorted blocks covering a three-tier architecture (DRAM, PM, SSD). However, in contrast to Elf, this approach is only suitable as a storage layout and cannot serve as an index. Furthermore, since the index is not fixed in [28], a combination with Elf is also possible. Since former experiments work on emulated PM, in [14], the authors re-evaluated some of the B^+-Tree

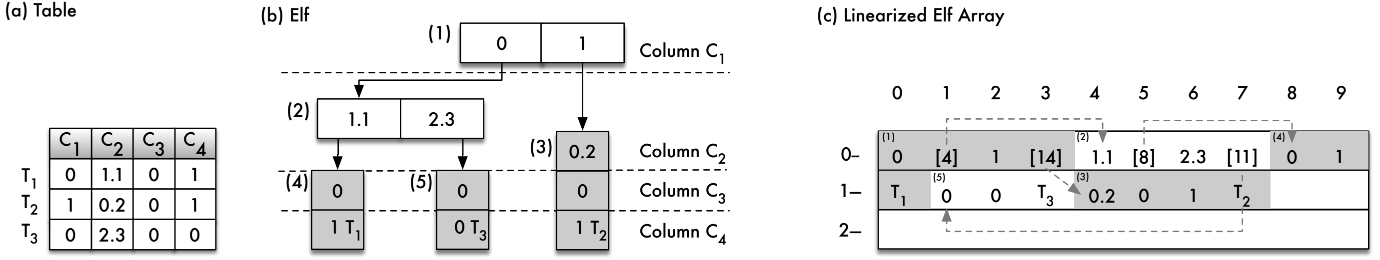

Fig. 1: Example table (a), conceptual Elf (b), and the OLAP and main-memory optimized storage layout of Elf (c).

variants on real hardware. In [13], the underlying primitives of these trees are analyzed on actual PM technology as well.

Selective Persistence: To keep up the performance of persistent data structures compared to volatile counterparts, only necessary fractions of data is stored in PM. The remaining part is placed in DRAM and rebuilt upon recovery. In the FPTree [4], the leaf nodes are placed in PM using a persistent linked-list, while the inner nodes are placed in DRAM and rebuilt upon recovery. Consequently, only accessing the leaves is more expensive compared to the volatile counterpart while using only a minimal portion of DRAM. The HiKV [29] runs on hybrid memory: a hash index is placed in PM 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 as well as splitting and merging of leaf nodes. The B+Tree is thus placed in DRAM. Furthermore, instead of designing individual hybrid data structures, in [30] and [31] the authors investigate general-purpose multi-tier buffer management covering DRAM, PM, and SSDs. This is a similar direction as we strive for with our dynamic caching approach (Section IV-B).

IV. Selective Caching for Elf

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

A. Naïve PM-based Approaches

As naïve PM approaches, there are two options: the Elf can be stored in PM 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, which would give the best query performance but is twice the size.

Pure PM-based Elf: To make Elf suitable for PM, we rely on PMDK and used its features described in the previous section. We visualize the usage of these features in Figure 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 object persistent pointer. We then always access the persistent Elf via the current virtual object pointer. 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.

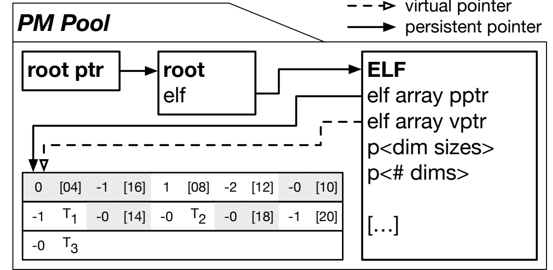

Fig. 2: Organization of persistent Elf in PM pool

Hybrid Elf: In the hybrid Elf, we propose a volatile copy of the Elf to be created in DRAM upon the initial build of the persistent Elf (or reproduced as necessary upon subsequent restarts). All queries are then run on the volatile Elf. We argue that with the hybrid Elf, we get better query performance at DRAM latency. Moreover, we save the cost of rebuilding Elf in events of system failure or restart, and we can alternatively execute queries on the persistent Elf in case available DRAM space is not sufficient to hold the volatile Elf.

However, the hybrid Elf has a considerable 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 is aimed to support initial builds and periodic insertions for its primary fields 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.

B. Selective Caching

Cached Elf: In case that keeping a full volatile copy of persistent Elf in DRAM (ref. hybrid Elf) needs too much space, we propose to cache crucial parts of Elf that are most frequently traversed in query executions 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 could be extended by a replacement strategy (e.g., LRU) 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. Hence, we do not have to probe the cache and the persistent Elf but know directly which part is in DRAM and which is in PM.

Static caching creates an FPTree-like hybrid layout [4], keeping inner nodes in DRAM and leaf nodes in PM. Based on this, the dynamic approach could be additionally applied to the lower dimensions, which would result in a split cache.

Another idea to populate the cache is to use probabilities for traversing each `DimensionList`. The `DimensionList` of the first dimension (and only it) has a probability of 1 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. After obtaining a 'tree' of probabilities, we choose which `DimensionLists` to cache in DRAM in addition to the first `DimensionList`, which depends on cutoff probabilities and the available DRAM memory space for caching the Elf. This approach is either implemented dynamically as part of an eviction policy or statically at the time the tree is built.

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 PM and DRAM, which incurs penalties of more cache misses especially for the dynamic parts.

V. 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 Section IV-B, which should reduce the overhead. As a result, we show that with sophisticated optimizations, a DRAM-like performance is possible.

TABLE II: Experimental setup.

Processor	2x 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	384 GB DDR4, 1.5 TB Intel® Optane™ DCPMM
OS & Compiler	CentOS 7.7, Linux 5.4.8 kernel, cmake 3.15.3, GCC 8.3.1 (-O3), PMDK 1.7

A. Query Workloads

All our workloads were run on a table of 100 million rows and 10 dimensions using a uniform access distribution over all inserted tuples. This represents a worst-case scenario for our caching since a cached node of the Elf is less likely reused for subsequent queries. Each dimension is of integer type[1] (in the range of 0 and 100) resulting in approximately 4 GB.

The experiments were repeated at least ten times in a single-threaded environment to obtain reliable measurements. Besides the building, the three tested query types are briefly explained below. The throughput of these queries is depicted as queries per second (qps).

Exact-Match Query: The exact-match query takes one equality predicate for each dimension and returns the TID of the tuple whose values exactly match. In each run, we first selected random tuples from the table within a specified range, then used the 10 dimension values of each tuple for constructing a single exact-match selection predicate each.

Range Query: The range query returns a list of TIDs, as per the selection predicates, unlike the exact-match query that returns at most a single TID. Two sets of 10 values define the lower and upper boundaries of the selection predicates for the respective 10 dimensions. In each run, we selected random pairs of tuples whose dimension values served as lower and upper boundaries and then ran the range query on the constructed Elf.

Partial-Match Query: A partial-match is a special form of range query. The lower and upper boundaries are used in the search only for preselected dimensions. All other dimensions are evaluated for all values, by setting the boundaries for evaluation to 0 and maximum values of the dimensions.

B. Experimental Setup

Our experiments were conducted on a dual-socket Intel Xeon Gold 5215 server as outlined in Table II. Each socket is equipped with 6 DCPMMs interleaved to one region and namespace. The PM 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.

C. DRAM vs. PM

At first, we want to investigate the performance overhead of the pure PM-based Elf against its DRAM counterpart.

[1]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 PM.

In Figure 3, the corresponding measurements are depicted. As expected, DRAM exhibits a better performance than PM. The overhead for building the Elf and for executing the three query types yield to 15%, 230%, 95%, and 115%, respectively. Our results show that the performance gap between DRAM and PM is wider for queries – especially exact-match queries – than for building. Our explanation is that the sequential access pattern (when building the Elf) is still better supported on PM than a random one (when querying the Elf). Particularly during a build, the write-combining buffer of PM seems to be quite efficient if there is only a single sequentially writing thread. Notably, range and partial-match queries lead to better exploitation of the caches due to common DimensionLists being traversed. This is not the case for most exact-match queries due to their tiny query windows.

D. Optimizations

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 the cost of copying the Elf data array to DRAM. For our setup with 100M tuples ($\sim$ 4 GB), this took 1825 ms. The total build time, thus, increases to 63,4 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.

Since the last condition is not always given especially for analytical tasks, we next evaluate the different caching approaches described above. We compare the throughput of the three query types for the approaches over a time course expressed in queries run up to that point. Before starting the measurements, we let the system warm up for 10,000 queries. We show the results for the persistent Elf without DRAM caching, the naïve dynamic caching, static caching of first x dimension levels as well as the combination of dynamic and static caches. In Figure 4a the results of the exact-match queries are shown. The access distribution in this experiment is uniform over all available tuples.

An unlimited dynamic cache initially performs quite well. However, as expected, at a specific DRAM cache size it is outperformed by the pure persistent Elf due to too many CPU cache misses. Caching the *first* level statically leads also already to initial better performance. After around 1M queries, the persistent Elf and the static approach behave nearly equally since the CPU caches contain the same data. Adding the

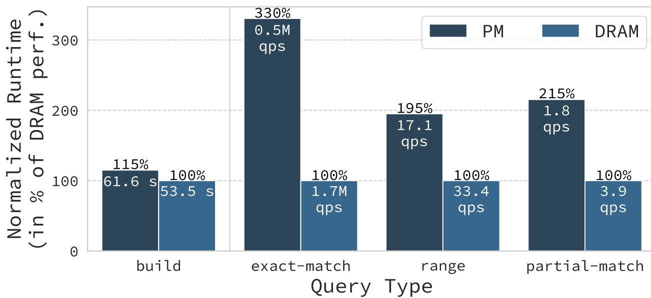

Fig. 3: Build and Query performance of Elf.

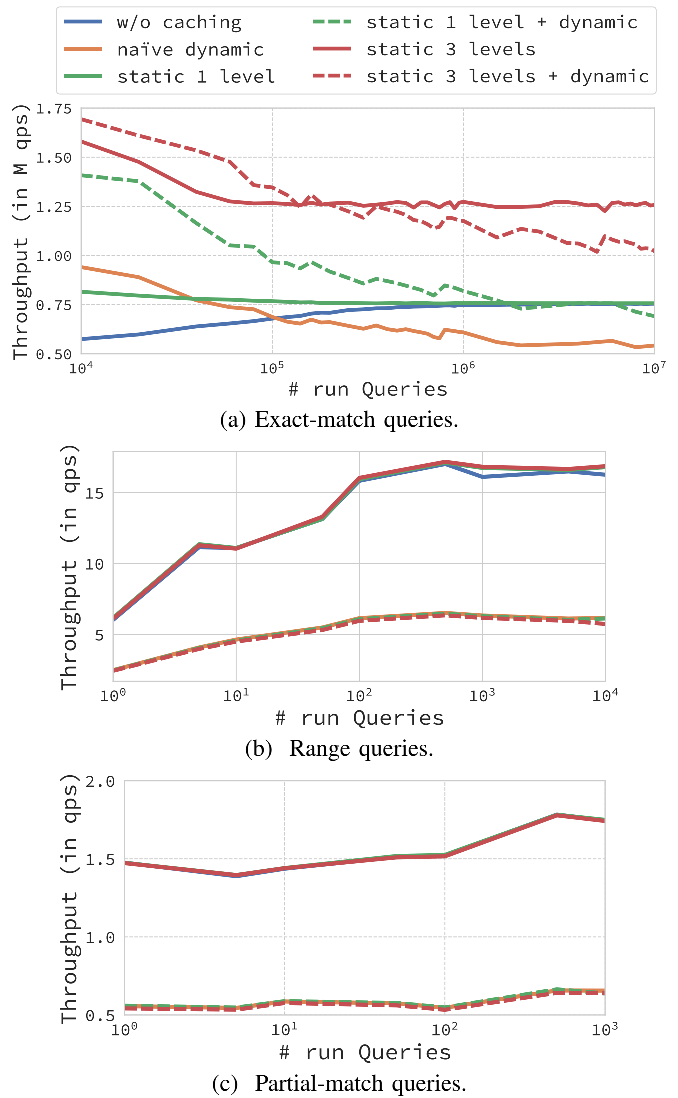

Fig. 4: Throughput progression of Cached Elf variants.

dynamic cache on top now drastically improves the initial performance. Once again, however, this throughput cannot be sustained and eventually also drops below the pure persistent variant. When putting the first three dimension levels statically into the cache, the potential of this approach becomes visible as we can achieve nearly DRAM performance (cf. Figure 3). Hence, we conclude that more statically cached levels lead to much better throughput. The limit here is determined by the DRAM buffer size. For a static cache of 1, 2 and 3 dimension levels the cache size results in approximately 1 KB, 120 KB, and 12 MB, respectively. Compared to the total size of Elf this is merely 0,3% space overhead. The dynamic variants should be limited in any case, although the question of the optimal eviction policy remains open.

The above experiments were executed on the complete tuple range. Since analytical queries may possess a certain selectivity, we varied it in the following experiment. In Figure 5 the average throughput when running 1 million queries is shown considering the uncached and the best performing caching

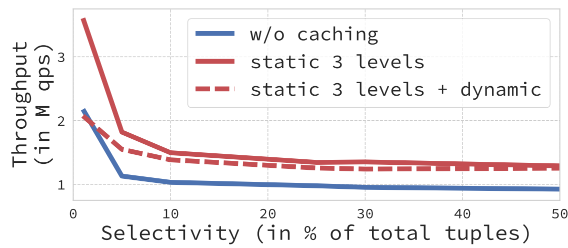

Fig. 5: Throughput of Elf variants for exact-match queries varying the selectivity.

approaches from before. Obviously, a higher selectivity leads to better performance. What is decisive, however, is the relative distance between the approaches. While at a selectivity of 50% the best caching approach is 1.4 times faster, at 1% it is even 1.7 times faster. This confirms that the previous experiment is already the worst case for the caching approaches.

In our last micro-benchmark, we considered range and partial-match queries. Since these are long-running queries, the number of queries run for warm up and the actual measurement is much lower. The results are visualized in Figures 4b and 4c. The dynamic approach and combinations with it perform much worse for these kinds of queries. The reason for this is that the range queries caused by the probing and warm-up make the dynamic cache too large. The static approach, on the other hand, shows nearly the same performance as no caching. Hence, there is at least no performance disadvantage for range-based scans using a static DRAM cache.

VI. Conclusion

In this paper, we investigated various approaches to accelerate OLAP queries on multi-dimensional index structures utilizing PM. Particularly, we proposed selective caching mixing static and dynamic caching approaches of tree nodes in DRAM. The results of our experiments revealed that, especially for exact-match queries, it is possible to achieve DRAM-like performance with our hybrid approaches.

Due to these promising results, we aim for an application of selective caching on further index structures, because it is a comprehensive method for arbitrary index structures or even data structures. However, the granularity of cached objects may differ and sweet spots have to be identified by a suitable cost model. Another idea for future work is to split the cache into a static and dynamic part with a fitting eviction policy that should be further analyzed. With higher selectivity, we are convinced that this will also increase the throughput of range-based queries. Another interesting direction to examine is the performance of the DRAM-based Elf when using PM in *Memory* mode. This would mean no more persistence, but significantly more memory capacity.

References

[1] I. Oukid and L. Lersch, "On the Diversity of Memory and Storage Technologies," *CoRR*, vol. abs/1908.07431, 2019.

[2] P. Götze, A. van Renen *et al.*, "Data Management on Non-Volatile Memory: A Perspective," *DB-Spektrum*, vol. 18, no. 3, pp. 171–182, 2018.

[3] J. Yang, Q. Wei *et al.*, "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems," in *USENIX FAST*, 2015, pp. 167–181.

[4] I. Oukid, J. Lasperas *et al.*, "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory," in *SIGMOD*, 2016, pp. 371–386.

[5] D. Hwang, W. Kim *et al.*, "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree," in *USENIX FAST*, 2018, pp. 187–200.

[6] S. Chen and Q. Jin, "Persistent B+-Trees in Non-Volatile Main Memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.

[7] S. Venkataraman, N. Tolia *et al.*, "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory," in *USENIX FAST*, 2011, pp. 61–75.

[8] D. Broneske, V. Köppen *et al.*, "Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach," in *IEEE ICDE*, 2017, pp. 647–658.

[9] H. P. Wong, S. Raoux *et al.*, "Phase Change Memory," *PIEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[10] M. Hosomi, H. Yamagishi *et al.*, "A Novel Nonvolatile Memory with Spin Torque Transfer Magnetization Switching: Spin-RAM," *IEEE IEDM*, pp. 459–462, 2005.

[11] D. B. Strukov, G. S. Snider *et al.*, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[12] I. Cutress and B. Tallis, "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, 2018, accessed January 15, 2020.

[13] P. Götze, A. K. Tharanatha, and K. Sattler, "Data Structure Primitives on Persistent Memory: An Evaluation," *CoRR*, vol. abs/2001.02172, 2020.

[14] L. Lersch, X. Hao *et al.*, "Evaluating Persistent Memory Range Indexes," *PVLDB*, vol. 13, no. 4, pp. 574–587, 2019.

[15] A. van Renen, L. Vogel *et al.*, "Persistent Memory I/O Primitives," in *DaMoN @ SIGMOD*, 2019, pp. 12:1–12:7.

[16] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE TPDS*, vol. 27, no. 5, pp. 1537–1550, 2016.

[17] D. S. Rao, S. Kumar *et al.*, "System software for persistent memory," in *EuroSys*, 2014, pp. 15:1–15:15.

[18] Intel Corporation, "Persistent Memory Development Kit," http://pmem.io/pmdk, 2019, accessed: January 15, 2020.

[19] V. Leis, A. Kemper, and T. Neumann, "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases," in *IEEE ICDE*, 2013, pp. 38–49.

[20] D. Broneske, V. Köppen *et al.*, "Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory," *IEEE TKDE*, vol. 31, no. 7, pp. 1296–1311, 2019.

[21] J. Rao and K. A. Ross, "Making B^+-Trees Cache Conscious in Main Memory," in *SIGMOD*, 2000, pp. 475–486.

[22] J. Arulraj, J. Levandoski *et al.*, "BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory," *PVLDB*, vol. 11, no. 5, pp. 553–565, 2018.

[23] S. K. Lee, K. H. Lim *et al.*, "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems," in *USENIX FAST*, 2017, pp. 257–270.

[24] L. Lersch, I. Oukid *et al.*, "An analysis of LSM caching in NVRAM," in *DaMoN @ SIGMOD*, 2017, pp. 9:1–9:5.

[25] S. Kannan, N. Bhat *et al.*, "Redesigning LSMs for Nonvolatile Memory with NoveLSM," in *USENIX ATC*, 2018, pp. 993–1005.

[26] D. Schwalb, M. Dreseler *et al.*, "NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories," in *IMDM @ VLDB*, 2015, pp. 4:1–4:8.

[27] M. Nam, H. Cha *et al.*, "Write-Optimized Dynamic Hashing for Persistent Memory," in *USENIX FAST*, 2019, pp. 31–44.

[28] P. Götze, S. Baumann, and K. Sattler, "An NVM-Aware Storage Layout for Analytical Workloads," in *HardBD & Active @ ICDE*, 2018, pp. 110–115.

[29] F. Xia, D. Jiang *et al.*, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," in *USENIX ATC*, 2017, pp. 349–362.

[30] A. van Renen, V. Leis *et al.*, "Managing Non-Volatile Memory in Database Systems," in *SIGMOD*, 2018, pp. 1541–1555.

[31] J. Arulraj, A. Pavlo, and K. T. Malladi, "Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory," *CoRR*, vol. abs/1901.10938, 2019.