A Design Proposal for a Unified B-epsilon-tree: Embracing NVM in Memory Hierarchies

Sajad Karim[1], Johannes Wünsche[1], David Broneske[2], Michael Kuhn[1] and Gunter Saake[1]

[1]Otto von Guericke University Magdeburg, Germany
[2]DZHW, Germany

Abstract

Non-volatile memory (NVM) represents a new class in the traditional storage hierarchy. The technologies in this class share characteristics of both primary and secondary storage; they provide latency that approaches that of DRAM, albeit moderately higher[1], yet significantly lower than that of secondary storage devices, are addressable from cache lines, and, most importantly, offer persistence. NVM is often referred to as a disruptive memory technology because it has invalidated the traditional programming paradigms used in applications such as database management systems (DBMS) and file systems (FS). Substantial research have focused on integrating NVM into diverse DBMS and FS, specifically through optimizing data structures such as B-trees and LSM-trees. Despite these advancements, considerable opportunities remain to further exploit NVM to boost application performance, particularly by exploring variations of these data structures. In this work, we present a proposal to optimize the B^{ϵ}-tree for use with NVM. The proposed modifications aim to capitalize on the unique characteristics of NVM, enhancing the tree's efficiency in both read and write operations. Additionally, the suggested changes are designed to reduce computational overhead, particularly by minimizing the need for tree rebalancing operations. The development of the tree is in progress, and it is assumed that completion will take some time; therefore, this paper is presented as preliminary work.

Keywords

Non-Volatile Memory, Persistent Memory, B-tree, B-epsilon-tree, Key-Value Store

1. Introduction

One of the key challenges that applications used for data management and analysis face is keeping the data as close as possible to the CPU. This issue is exacerbated by the unbounded growth of data. The volume of data doubles approximately every two years [5]. This estimate was made before the onset of the recent pandemic, which has further fueled its growth as people increasingly relied on digital services for their day-to-day chores.

The extensibility of DRAM has resolved this issue to some extent, resulting in a new type of database system called the main memory database system [6, 7, 8], where all data resides in DRAM. Nevertheless, given the exponential growth of data, DRAM will never be sufficient to accommodate all the data, and more importantly, this solution is not viable for many businesses due to its cost impact. Moreover, its further expansion has become another challenging task. Consequently, applications devise optimized designs that logically arrange the memory and storage in the traditional storage landscape at different levels and incorporate different migration policies to move the data across the storage devices depending on their usage and access patterns. For example, bcache and dm-cache in the Linux kernel use solid-state drives as a cache for hard disk drives.

Non-volatile memory, or persistent memory, is a comparatively new storage class in the storage hierarchy that is considered a solution to the above-mentioned problem. It shares characteristics of primary and secondary storage. The modules in this class are byte-addressable, provide access latency close to DRAM[1], and offer much higher capacity than DRAM. Furthermore, they are capable of storing data persistently. NVM has not only added further heterogeneity to the storage landscape but has also invalidated the traditional programming paradigm because, contrary to the traditional model where data structures are generally categorized into memory and storage resident data structures [9], NVM bound data structures cover both aspects and linked intricacies [10].

Considerable research have been conducted to exploit the characteristics of NVM, and in particular to the design proposal presented in this paper, several designs for the index structures [11, 12, 13, 14, 15, 16] typical to key-value storage engines are presented. However, one key aspect not addressed in the cited literature (discussed briefly in Section 3) is the disregard for the heterogeneity of the modern storage landscape. They all present NVM-DRAM optimized B-trees and do not consider block

35th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), May 22-24, 2024, Herdecke, Germany.
sajad.karim@ovgu.de (S. Karim); johannes.wuensche@ovgu.de (J. Wünsche); broneske@dzhw.eu (D. Broneske); michael.kuhn@ovgu.de (M. Kuhn); saake@ovgu.de (G. Saake)
0009-0002-4910-8453 (S. Karim); 0000-0002-5304-7262 (J. Wünsche); 0000-0002-9580-740X (D. Broneske); 0000-0001-8167-8574 (M. Kuhn); 0000-0001-9576-8474 (G. Saake)

[1]There are various types of non-volatile memory, including Ferroelectric RAM [1], Magnetoresistive RAM [2], Phase Change Memory [3], and Resistive RAM [4], each offering distinct latencies.

devices, for instance. Moreover, to the best of our knowledge, no research has optimized the B^{ϵ}-tree for NVM despite the fact that it offers similar scan operations as other B-tree variants yet its inserts and deletes are an order of magnitude faster [17]. Therefore, in this work, several changes to the B^{ϵ}-tree are proposed that we argue will significantly benefit the index structure. These changes primarily utilize NVM to improve key tree workflows, including indexing and buffering, offering several notable benefits:

- The tree would become NVM-aware, allowing direct reads/writes to the nodes on NVM and further optimizing both read and write operations.
- In many instances, the design would reduce data fragmentation, I/O, and other computation costs.
- Last but not least, the tree would be able to perform low-latency point queries.

The remainder of this paper is structured as follows. Section 2 provides background on the B-tree and its variants, and it also briefly describes non-volatile memory. Section 3 details some relevant work. The design proposal is discussed in Sections 4 and 5, which are then followed by some important use cases in Section 6. Section 7 details a preliminary experiment supporting the design, and Section 8 concludes with a summary.

2. Background

In this section, we discuss the B-tree and its commonly used variants, followed by a brief discussion on NVM and the Intel® Optane™ DC Persistent Memory.

2.1. B-tree family

B-trees are tree-based, one-dimensional, multi-level, and widely used index structures for secondary storage. They were first introduced by Bayer and McCreight in 1972 [18], and since then, many variants of B-trees have been introduced. B-trees are primarily dominant across different file systems [19, 20] and database management systems [21, 22], and no other index structure has been able to prove the same flexibility and generality [23].

A B-tree is a balanced m-ary search tree. It can also be considered as a generalization of a balanced search tree. It contains two types of nodes, internal and leaf nodes, with all the leaf nodes at the same level as the tree grows (logarithmically) upwards. Internal nodes, including the root node, contain an ordered sequence of keys and values, and they also contain pointers to the child nodes. Whereas the leaf nodes only contain the pair of keys and values in a sequenced order. B^{+}-tree is a widely known variant that was introduced to enhance the branching factor of the internal nodes, and contrary to the rudimental version, it does not store the values in the internal nodes, and the values are instead stored in the leaf nodes only [24].

B-trees perform read, insert, delete, and range query operations in logarithmic time. Insertion is efficient when made to a non-full node, or else it can trigger the node-split operation. Splitting propagates upwards, and in the worst case, it can result in an increase in the height of the tree. The delete operation can trigger a merge operation. B-trees have an optimal query performance, but their write operation is not optimal [25]. Therefore, in order to mitigate this limitation and make it write-optimized, buffers are added to the internal nodes in B^{ϵ}-tree [26].

B^{ϵ}-trees are basically B^{+}-trees with buffers in internal nodes that store messages for the child nodes. The key difference between a B^{+}-tree and the original B-tree is that in a B^{+}-tree, all data is stored in the leaves and the internal nodes only store keys for fast searching, whereas in a B-tree, both keys and data can be stored at any level [21]. Additionally, B^{+}-trees have leaves that are linked, facilitating efficient range queries, unlike B-trees where leaf nodes are not inherently linked [21]. On the other hand, the primary difference between a B^{+}-tree and a B^{ϵ}-tree lies in how they handle updates and buffering. A B^{ϵ}-tree extends the B^{+}-tree by introducing the concept of a "message" which is used to buffer insertions, deletions, and updates. These messages are stored in separate buffers at each node of the tree. This allows B^{ϵ}-trees to delay actual updates to the structure, potentially reducing disk I/O by batching operations, and efficiently merging these updates during searches or structural changes. This buffering mechanism is particularly advantageous for environments with high write loads, enhancing throughput and decreasing latency compared to B^{+}-trees, where updates directly modify nodes. Moreover, a message can be an insert, delete, and update operation, which is normally encoded in a single message called the upsert message. They propagate gradually from the root node to the target leaf node, and they are flushed down to the child node only when the buffer in the internal node is full. The allocation of space for data and buffer is controlled by a tuning parameter called epsilon (ϵ). The tree is a B^{+}-tree when it is set to '0', and the tree behaves like a buffered repository tree [17, 27] when the value is set to '1'. However, most configurations use ϵ=1/2, and with this value, B^{ϵ}-trees provide asymptotically better insert performance with the same asymptotic point query performance as B-trees [17].

2.2. Non-Volatile Memory

Non-volatile memory is an emerging storage class that effectively bridges the gap between main memory and secondary storage. This class offers significantly higher capacity than DRAM while maintaining reasonably com-

parable latency. Additionally, these technologies are non-volatile and directly accessible from cache lines. Examples include Phase Change Memory [28], Spin Transfer Torque RAM (STT-RAM) [29], Carbon NanoTube RAM (NRAM) [30], and Memristors [31].

Currently, Intel® is the only producer of commercially available persistent memory modules [32], developed through an advanced NVM technology known as 3D XPoint, often termed "cross-point." Developed in collaboration with Micron, 3D XPoint is thought to function similarly to PCM [33, 34, 35], recording data by changing the resistance of its material. However, further production of this technology has been discontinued [36]. These modules, designed for use with Intel® Xeon Scalable Processors, come in various generations, offering differing performance and capacity levels. They are available in the DIMM form factor, compatible with standard DDR4 sockets, and can coexist with conventional DDR4 DRAM DIMMs on the same memory channel. The modules feature an internal granularity of 256 bytes and can operate in three modes: memory, app direct, and dual modes [32].

3. Related Work

As already discussed, NVM is byte-addressable, and its capabilities cannot be leveraged using the traditional programming model. Therefore, numerous research studies have been conducted on optimizing different data structures to leverage NVM. CDDS B-Tree [11] is one of the earliest works that presents a single-level data store using a consistent and durable B-tree. It uses versioning instead of logging to achieve consistency and durability, and it uses copy-on-write to ensure the consistency of update operations. It also leaves the entries in the nodes unsorted to overcome the write amplification that is detrimental to the performance of NVM. In [12], a write-atomic B-tree (wBTree) is presented. It further reduces the overheads in CDDS B-Tree by using an indirect slot array/bitmap to prevent index items from moving during insertions and deletions. It ensures consistency using redo-logging or atomic operations. Moreover, in [13], a workload-adaptive and cache-optimized B^+-tree, NVTree, is presented where the leaf nodes are considered as the critical data and the consistency is only enforced on them. The internal nodes are built using the leaf nodes and maintained in a cache-optimized structure in the main memory, and also, the sizes of the nodes adapt over time to improve the performance of the tree.

Furthermore, in [14], FPTree, again an optimized B^+-tree is presented that follows NVTree, however, it incorporates a technique called "fingerprinting" to overcome search overhead for nodes in NVM. In [15], another DRAM-NVM B^+-tree is presented that segregates the data into cold and hot categories according to their access frequency. It maintains the data in different data structures across DRAM and NVM and ensures the consistency of data in DRAM using logging. LB-Tree [16] is another recently proposed optimized indexing solution. It also uses a tailored node layout for the nodes in NVM, and it uses three techniques to boost performance by moving entries within nodes and performing logless node splits to reduce write/update overheads.

Lastly, to our knowledge, the only storage engine that uses B^ϵ-tree and incorporates NVM in its storage stack is Haura[2] [37, 38]. It is a write-optimized key-value and object storage stack and follows the layered approach used in ZFS [39]. Its ObjectStore module facilitates basic operations such as create, read, write, and query, using a key-value system. It uniquely handles large objects by breaking them into chunks with distinct identifiers. The Database module oversees databases composed of B^ϵ-trees, managing data and metadata separately. The Tree module implements the B^ϵ-tree, while the DataManagement module maintains data integrity and interacts with other modules for functionalities like caching and compression. The StoragePool module queues and dispatches I/O operations, and the Vdev module offers various storage device interfaces, including single, mirror, and parity options for data redundancy. Although it uses DAX to access NVM, it does not fully leverage NVM capabilities due to the lack of support for direct reads and in-place updates. Nodes can be distributed among various storage devices – NVM, HDD, and SSD – according to user preferences. Nevertheless, the system treats nodes on NVM identically to those on other devices, replicating them to DRAM for processing during operations.

4. Design Goals

One key aspect not addressed in the literature mentioned above is the neglect of the heterogeneity of the modern storage landscape, except in [38]. They present NVM-DRAM optimized B-trees but do not consider block storage, such as solid-state drives. Moreover, while they optimize B^+-trees for NVM, the only implementation we found that uses B^ϵ-trees with NVM is [38].

Furthermore, the optimizations presented for B^+-trees cannot be fully applied to B^ϵ-trees due to the differences discussed in Section 2.1. Additionally, the design in [38] does not fully exploit NVM's characteristics. Therefore, by utilizing the optimizations discussed in the literature, we aim to present an NVM-optimized B^ϵ-tree that covers the following aspects:

- The design should consider the heterogeneity of the modern storage landscape and leverage the characteristics of the storage stack.

[2] https://github.com/julea-io/haura

- It should enable direct/granular reads and in-place updates on NVM.
- There should be a single tree, with nodes that can be stored or migrated to any storage media in the stack.

5. Design Proposal

The core functionalities of the B$^\epsilon$-tree include managing the index structure in the root and internal nodes, as well as maintaining the buffers to mitigate the cost of the writes and updates. The suggested design primarily targets enhancing these functionalities by incorporating NVM and adding other enhancements to address the goals mentioned in the previous section. A conceptual diagram of a well-grown tree spread across different storage media is presented in Figure 1.

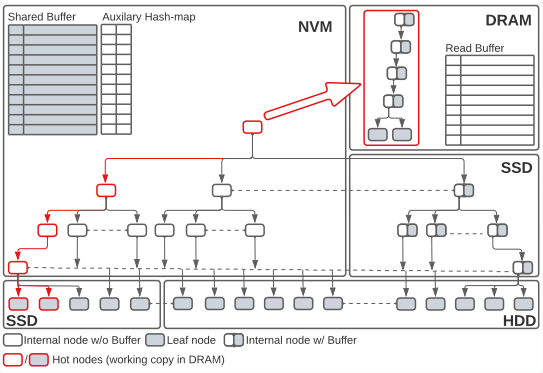

Figure 1: A schematic representation of B$^\epsilon$-tree optimized to leverage NVM.

In the figure, the tree is distributed across DRAM, NVM, SSD, and HDD. The internal nodes are persisted on NVM and SSD, and the leaf nodes are stored on SSD and HDD[3]. The nodes are depicted using various objects and colors to highlight differences in their layouts and structures. Nodes with a red border (in NVM and SSD) represent hot nodes whose working copies exist in DRAM (highlighted with a red box). When nodes are moved from NVM to DRAM, they are transformed into typical internal nodes with buffers. This process is discussed in Section 5.1. Next, nodes filled with white color and a gray border (in NVM) are also frequently accessed nodes, but they do not meet the criteria, for example [40][4], to be moved to DRAM. Unlike typical B$^\epsilon$-tree internal nodes, these nodes do not contain buffers; instead, their messages are maintained in a centralized buffer in NVM that is shared among all nodes in NVM. This centralized buffer and the auxiliary hash table are discussed in Section 5.2. The nodes split into white and gray parts in SSD represent typical B$^\epsilon$-tree internal nodes. The white part contains keys and pivots, and the gray part holds updates or messages. Furthermore, the nodes filled with gray color (in SSD and HDD) are typical leaf nodes. However, these leaf nodes might include a small buffer to mitigate minor updates, as detailed in Section 6.3. Lastly, the design includes a buffer (hash table) in DRAM to store recent reads, benefiting read-recent workloads.

The following sections describe the components of the diagram and the areas where NVM can be leveraged to improve different workflows of the tree in detail.

5.1. Indices in NVM

In the suggested design, the internal nodes are preferably[5] stored on NVM. As previously discussed, the internal nodes of a B$^\epsilon$-tree typically contain keys, pivots, and messages. In contrast, the proposed design stipulates that internal nodes in NVM contain only keys and pivots, while messages are maintained in a shared centralized buffer in NVM, as discussed in Section 5.2. The internal nodes on NVM are directly accessed for lookups and the centralized buffer for messages. This segregation offers the following benefits:

- Contrary to [38], which first copies the nodes to DRAM from storage, including NVM, before performing any processing on the nodes, direct access in the suggested approach – on the nodes that are located on NVM – would reduce transfer costs and related computations.
- [38] experiences high fragmentation as it moves nodes to and from memory for processing, and employs copy-on-write technique to ensure the consistency. In the suggested design, storing the messages in a different data structure (discussed in Section 5.2) reduces fragmentation. This is because only the specific parts that need changes would be updated, rather than the entire data set, making it more efficient for small changes.
- Read queries, especially point queries, will be faster because they will only access the needed parts of the data on NVM, not the whole data block.

Lastly, the design assumes the frequently accessed nodes (or hot nodes) would be in the main memory. However, as the internal nodes on NVM follow a different

[3] Node placement is determined by access frequency, with node layout (data structure) varying across different devices.

[4] The rule outlined in [40] serves as an illustrative example, yet it requires adaptation to more realistically reflect contemporary workloads and application environments.

[5] A cold or rarely accessed node can be moved to slow storage.

structure and its data is stored across different data structures, therefore, copying an internal node between NVM and DRAM requires transforming the nodes into the respective layouts. For example, moving a node from NVM to DRAM would require copying the keys and values from the node, and the respective messages from the shared buffer.

5.2. Shared Buffer for Internal Nodes in NVM

In [38], and generally in the B^ϵ-tree, updates involve first fetching the relevant nodes into main memory and deserializing them. Updates are then applied, and the nodes are serialized and written back to the persistent media. This process incurs significant I/O and computational costs, even for minor updates.

In the proposed design, these costs are reduced by using a centralized buffer, which is an NVM-optimized hash table [41, 42]. As mentioned previously, the internal nodes on NVM share this centralized buffer to store updates/messages. This buffer, a hash table, stores keys and messages. Updates to a key in an internal node on NVM are made directly in this hash table, where it retains the most recent messages and integrates new messages with existing ones, if necessary. Additionally, a second hash table is maintained to link the messages with their respective internal nodes on NVM, particularly useful during flush operations to identify messages associated with the donor node.

Consequently, messages are added to both the buffer and the auxiliary hash table without incurring additional I/O or computational costs. Furthermore, during a flush operation among the nodes in NVM, only the pointers in the auxiliary hash table need updating, significantly reducing I/O and computational expenses. The centralized buffer offers the following benefits:

- Updates to the nodes on NVM do not require moving the entire node to DRAM; instead, they can be made directly onto the shared buffer. This reduces I/O and computations such as serialization and deserialization.
- This would help reduce the fragmentation mentioned in the previous section.
- The flush operation within the nodes on NVM only requires updating the pointers in the auxiliary hash table, which again saves on I/O and computation.

The suggested change makes the tree more write-optimized by saving I/O and computation costs, thus allowing more time for batch updates to nodes on slow storage. However, this might negatively affect range queries since messages are scattered across the buffer and must be accessed individually for each key—unlike typical settings where messages benefit from spatial locality. Nevertheless, these settings can support write-heavy workloads, and the tree can be gradually adapted for read-heavy workloads by storing linked messages in a separate node (e.g., buffer node) to achieve spatial locality.

5.3. Read Buffer for Values in DRAM

This improvement is induced by the shared buffer in NVM. A small centralized buffer in DRAM is maintained to store recent reads. This data structure would have a significant impact on the frequently accessed values. The recently queried values are temporarily stored in this buffer, and an eviction policy is used to replace them with the recent ones.

5.4. In-place Writes

The proposition mentioned in this section is not particularly related to leveraging NVM; it could be applied to nodes stored on NVM and other block devices.

As already mentioned in [38], and typically in the B^ϵ-tree, updates on nodes are performed in DRAM, incurring significant I/O and computation costs. This workflow can be simplified, and the costs can be avoided in some cases, especially for small updates.

The proposed change enables direct in-place updates in the buffer segment of a B^ϵ-tree node on block storage by writing update messages sequentially to available offsets until the buffer, which is half of the standard 4 MB node size, is full. This method avoids the need to transfer the entire node into DRAM for read-modify-write cycles since updates are performed in granular block sizes of 4 KB. The updates in the buffer area are not normalized, meaning there could be multiple update messages for a single key. Normalization of messages occurs in DRAM when reading the buffer's contents, streamlining the process and reducing I/O and computational costs associated with data serialization and deserialization. When the buffer reaches capacity, the node is fetched into main memory for updates and written back to storage. A conceptual diagram is shown in Figure 2.

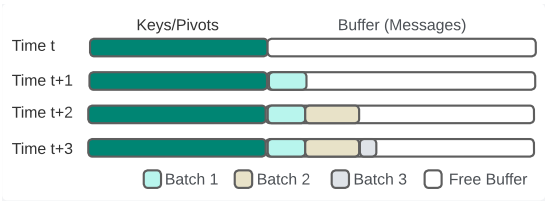

Figure 2: A conceptual diagram to illustrate the process of in-place update.

The image illustrates the state of a node at different timestamps. At time 't', the buffer contains keys and pivots, and its buffer part does not contain any messages or updates. At time 't+1', a batch of messages in cyan is moved from the parent node and written to the buffer. Next, at time 't+2', a batch of messages in brown is again flushed down from the parent node and written to the next available location in the buffer. Finally, at time 't+3', the batch of messages in blue is moved to the node.

On the other hand, in the typical design, each node undergoes three read-modify-write cycles for updates, which can lead to increased computation costs and fragmentation, given that the node is retrieved from persistent media or storage for each update.

6. Use cases

In this section, some important workflows, along with the impacts of the proposed design, are mentioned.

6.1. Tree building

As previously discussed, in the typical programming model, the B^{ϵ}-tree grows in DRAM using memory-resident data structures and is persisted using storage-resident data structures.

The suggested design begins with the same workflow. However, the flow changes when the nodes are flushed to the persistent media. In this design, the internal nodes are by default flushed onto NVM where keys and pivots are stored in an optimized node layout, and the messages are moved to the centralized buffer as discussed in Section 5.2. Later, when the tree needs to access the nodes on NVM, their data is directly accessed from the respective data structures, and only the nodes on other block storages are copied into DRAM. That said, frequently accessed nodes on NVM can be moved to DRAM to minimize latency.

6.2. Read and update on a grown tree

This use case discusses a scenario where the hot nodes are in DRAM, the internal nodes are on NVM, and the leaf nodes, or the least accessed nodes, are on slow storage.

The scan operations on the nodes in DRAM follow the typical workflow. However, the flow changes when accessing nodes on NVM. First, they can be queried without being fetched into DRAM. Second, the centralized buffer can be queried simultaneously. Only the nodes on slow storage need to be moved into DRAM.

Moreover, the update operation on nodes on NVM does not require moving the nodes into DRAM. Contrary to [38], where all involved nodes must be in DRAM, in the suggested design, when the child node is on NVM, upon flush from the parent node, the respective messages of the node are pushed into the centralized buffer in NVM. This change in workflow allows the algorithm more time and space to perform batch updates to subsequent nodes and eventually to the leaf nodes. Furthermore, if the descent of the messages only involves the internal nodes on NVM, it involves merely updating the pointers in the auxiliary hash table in NVM.

6.3. Highly random workload

This section discusses a scenario involving a highly random workload. Consider a dense tree distributed across various storage devices, with internal nodes on NVM and leaf nodes on SSD and HDD. Here, small updates accumulate in a central buffer in NVM. Furthermore, most leaf nodes receive a single or few updates from their preceding internal node in NVM, necessitating occasional flushing of updates to the leaf nodes. The primary challenge with this flushing process is the significant write amplification it causes. For instance, flushing 1000 small updates from NVM to 1000 leaf nodes on SSD or HDD would result in 1000 writes of 4 MB each (the default node size). Additionally, this process also requires reading leaf nodes into DRAM first, which incurs additional cost.

There are two potential solutions to mitigate this issue. First, a small buffer could be integrated into each leaf node (as discussed in Section 5.4), allowing small updates to be directly written to this buffer area without needing to read the entire node into DRAM, deserialize, serialize, and then write it back to storage. This approach would still involve 1000 writes but would reduce the data size of each write, saving on read and computation costs. Second, instead of flushing small updates directly to leaf nodes, messages related to the internal node in NVM could be serialized and stored as a typical node (e.g., buffer node). This strategy would result in a single write. However, when accessing the internal node later, its corresponding buffer node would need to be retrieved from storage.

7. Preliminary Experiments

In this section, we present a small (single-threaded) experiment[6], in favor of the proposed design, particularly the granular reads on NVM. The experiments are performed in a sandbox where NVM is accessed using different approaches and access patterns. In it, multiple `Objects` are used to imitate the internal node structure (partially) where all the `Objects` contain 1024 `Strings`. However, the sizes of the `Strings` vary in each `Object`, from 4 KB to 2 bytes, which is done to cover primitive data types like `ints`.

[6] https://github.com/sajadkarim/libmem_sandbox.git (dimes_23.cpp)

The experiments are conducted on a dual-socket server featuring Intel® Xeon® Gold 5220R CPUs. Each CPU has 24 physical cores, with each socket being equipped with 4 PMem[7] and 6 DRAM DIMMs. The experiments are run on NUMA node '0' to minimize memory access overheads, and the machine runs Ubuntu 20.04.3 LTS (5.4.0-126-generic). The PMem module is accessed in app direct mode using fsdax[8].

The results are illustrated in Figure 3. The experiment involves writing 1024 objects to NVM and reading them back in different settings, either randomly or sequentially, for instance. This process is repeated for each `Object` type, as indicated on the horizontal axis, using their respective `String` sizes. The right vertical axis shows the data moved to DRAM from NVM, and the left one shows the time taken in milliseconds.

The direct approach reads (`pmem_memcpy`) the strings in the objects one by one from NVM. Conversely, the in-direct-II approach first copies (`pmem_memcpy`) the entire object into DRAM, then reads (`memcpy`) the strings in the objects one by one. It imitates the workflow in [38]. Next, the in-direct-I approach simply copies (`pmem_memcpy`) the entire object into DRAM without performing any other operation. Lastly, the DRAM case shows the time taken by the individual strings to be copied to a different location in DRAM in the aforementioned in-direct-II approach.

The case that took the least time, unsurprisingly, involves copying data within DRAM. The next expected pattern is the in-direct-I case, which consumes the same amount of time throughout the experiments as it copies the same volume of data each time from NVM. The in-direct-II case takes longer as it involves copying the data to DRAM, then copying the strings one by one to a different location in DRAM. The case that is of our interest is the direct case. It performs read operations in small blocks (the size of the `String` in the respective `Object` type), yet it takes less time than the in-direct cases, except for 4096. However, the cost to serialize/deserialize the node is not considered in the experiment. Therefore, this case supports the design decision of keeping the data in optimized data structures in NVM and performing direct/small reads instead of copying entire nodes to DRAM [38].

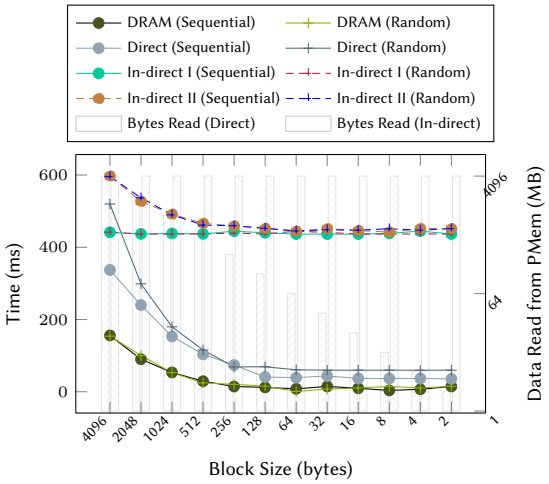

Figure 3: Accessing NVM in different settings and access patterns, and imitating the node reading behavior in [38].

8. Conclusion

In this work, a design proposal for an NVM-optimized B$^\epsilon$-tree is presented. While various B-tree variants have been developed to utilize the characteristics of NVM, none have specifically optimized a B$^\epsilon$-tree for this purpose. Additionally, the suggested designs do not account for the heterogeneity of the modern storage landscape, and those that do, fail to fully leverage NVM's potential. The observed gap likely results from uncertainty about future NVM product availability, particularly following Intel's discontinuation of its Optane Persistent Memory modules.

The insights from this study will deepen our understanding of storage solutions that balance the speed of DRAM with the capacity of NVMe SSDs. Particularly relevant is Compute Express Link (CXL), a leading interconnect technology that enables rapid data transfers and manages large, variable-speed memory pools. CXL's coherent memory protocol, which maintains data consistency across local memory caches, positions it as a transformative technology for future enterprise data storage [43]. Therefore, the findings from this work remain relevant as they contribute to the development of storage heterogeneity-aware index structures like B-trees, which are expected to benefit significantly from CXL's capabilities. The suggested design proposal primarily targets the incorporation of NVM to enhance the indexing and buffering of the tree, and it significantly impacts various workflows of the tree. Nevertheless, the actual impact can only be gauged after the tree is developed[9]. Lastly, this paper is presented as preliminary work, and we consider it a first step towards a heterogeneity-aware storage engine.

[7] Intel® Optane™ DC Persistent Memory modules are configured in interleaved settings, and the block size is 4 KB.
[8] https://docs.pmem.io/ndctl-user-guide/managing-namespaces

[9] The tree is currently in the development phase, and the status can be found at: https://github.com/sajadKarim/haldendb/

Acknowledgments

This work is part of SPP 2377 and funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 502268500.

References

[1] T. Mikolajick, et al., FeRAM technology for high density applications, Microelectronics Reliability 41 (2001) 947–950.

[2] J. Slaughter, et al., Fundamentals of MRAM technology, Journal of superconductivity 15 (2002) 19–25.

[3] S. W. Fong, C. M. Neumann, H.-S. P. Wong, Phase-change memory - Towards a storage-class memory, IEEE Transactions on Electron Devices 64 (2017) 4374–4385.

[4] Y. Chen, ReRAM: History, status, and future, IEEE Transactions on Electron Devices 67 (2020) 1420–1433.

[5] U. Kazemi, A Survey of Big Data: Challenges and Specifications, CiiT International Journal of Software Engineering and Technology 10 (2018).

[6] F. Faerber, et al., Main memory database systems, Foundations and Trends® in Databases 8 (2017) 1–130.

[7] P.-Å. Larson, et al., Modern main-memory database systems, Proceedings of the VLDB Endowment 9 (2016) 1609–1610.

[8] J. DeBrabant, et al., Anti-caching: A new approach to database management system architecture, Proceedings of the VLDB Endowment 6 (2013) 1942–1953.

[9] S. Scargall, Programming persistent memory: A comprehensive guide for developers, Springer Nature, 2020.

[10] A. Rudoff, Persistent memory programming, Login: The Usenix Magazine 42 (2017) 34–40.

[11] S. Venkataraman, et al., Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory, in: FAST, volume 11, 2011, pp. 61–75.

[12] S. Chen, Q. Jin, Persistent B^+-trees in non-volatile main memory, Proceedings of the VLDB Endowment 8 (2015) 786–797.

[13] J. Yang, et al., NV-Tree: A consistent and workload-adaptive tree structure for non-volatile memory, IEEE Transactions on Computers 65 (2015) 2169–2183.

[14] I. Oukid, et al., 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 '16, Association for Computing Machinery, New York, NY, USA, 2016. doi:10.1145/2882903.2915251.

[15] Y. Zhan, et al., HBtree: A Heterogeneous B+tree with Multi-granularity for Hybrid NVM-SSD Storage, in: 2022 IEEE International Conference on Networking, Architecture and Storage (NAS), 2022, pp. 1–4. doi:10.1109/NAS55553.2022.9925539.

[16] J. Liu, et al., LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory, Proc. VLDB Endow. 13 (2020) 1078–1090. URL: https://doi.org/10.14778/3384345.3384355.

[17] M. A. Bender, et al., An Introduction to B^ϵ-trees and Write-Optimization, login; magazine 40 (2015).

[18] R. Bayer, E. McCreight, Organization and Maintenance of Large Ordered Indices, in: Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, Association for Computing Machinery, New York, NY, USA, 1970, p. 107–141. doi:10.1145/1734663.1734671.

[19] O. Rodeh, J. Bacik, C. Mason, BTRFS: The Linux B-tree filesystem, ACM Transactions on Storage (TOS) 9 (2013) 1–32.

[20] B. Djordjevic, V. Timcenko, Ext4 file system in Linux environment: Features and performance analysis, International Journal of Computers 6 (2012) 37–45.

[21] D. Comer, Ubiquitous B-tree, ACM Computing Surveys (CSUR) 11 (1979) 121–137.

[22] G. Graefe, et al., Modern B-tree techniques, Foundations and Trends® in Databases 3 (2011) 203–402.

[23] M. Jürgens, Data Storage and Index Structures, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 15–34. doi:10.1007/3-540-45935-9_3.

[24] Navathe, et al., Fundamentals of database systems (6th Edition), Pearson Education. ISBN 9780136086208, 2010.

[25] O.-Y. Khavrona, B^ϵ-tree and cache-oblivious lookahead array: a comparative study of two write-optimised data structures, Master's thesis, University of Twente, The Netherlands, 2021.

[26] G. S. Brodal, R. Fagerberg, Lower bounds for external memory dictionaries., in: SODA, volume 3, 2003, pp. 546–554.

[27] A. L. Buchsbaum, et al., On external memory graph traversal., in: Symposium on Discrete Algorithms (SODA), 2000.

[28] A. Faraclas, N. Williams, A. Gokirmak, H. Silva, Modeling of set and reset operations of phase-change memory cells, IEEE electron device letters 32 (2011) 1737–1739.

[29] A. K. Mishra, et al., Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs, in: Proceedings of the International Symposium on Computer Architecture (ISCA), IEEE, 2011, pp. 69–80.

[30] B. Gervasi, Will Carbon Nanotube Memory Replace DRAM?, IEEE Micro 39 (2019) 45–51.

[31] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, The missing memristor found, nature 453 (2008) 80–83.

[32] Intel, Intel® Optane™ DC Persistent Memory, Last accessed: April 27, 2024. URL: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[33] H.-Y. Cheng, et al., 3D cross-point phase-change memory for storage-class memory, Journal of Physics D: Applied Physics 52 (2019) 473002.

[34] M. Katsaragakis, et al., Energy Consumption Evaluation of Optane DC Persistent Memory for Indexing Data Structures, in: 2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC), IEEE, 2022, pp. 75–84.

[35] B. Kersting, M. Salinga, Exploiting nanoscale effects in phase change memories, Faraday Discussions 213 (2019) 357–370.

[36] Intel, Intel® Optane™ DC Persistent Memory, Last accessed: April 27, 2024. URL: https://www.intel.com/content/www/us/en/support/articles/000024320/memory-and-storage.html.

[37] F. Wiedemann, Modern Storage Stack with Key-Value Store Interface and Snapshots Based on Copy-On-Write B^ϵ-Trees, Master's thesis, Universität Hamburg, 2018.

[38] S. Karim, et al., Assessing non-volatile memory in modern heterogeneous storage landscape using a write-optimized storage stack, in: Grundlagen von Datenbanken, 2023.

[39] W. Fuzong, G. Helin, Z. Jian, Dynamic data compression algorithm selection for big data processing on local file system, in: Proceedings of the International Conference on Computer Science and Artificial Intelligence, 2018, pp. 110–114.

[40] J. Gray, F. Putzolu, The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time, in: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, 1987, pp. 395–398.

[41] L. Vogel, et al., Plush: A write-optimized persistent log-structured hash-table, Proceedings of the VLDB Endowment 15 (2022) 2895–2907.

[42] B. Lu, et al., Dash: Scalable Hashing on Persistent Memory, Proceedings of the VLDB Endowment 13 (2020) 1147–1161. doi:10.14778/3389133.3389134.

[43] S. Jain, et al., Memory Sharing with CXL: Hardware and Software Design Approaches, arXiv preprint 2404.03245 (2024).