When Tree Meets Hash: Reducing Random Reads for Index Structures on Persistent Memories

KE WANG, Shanghai Qi Zhi Institute, China and Yale University, USA
GUANQUN YANG, Shanghai Qi Zhi Institute, China and New York University, USA
YIWEI LI, Tsinghua University, China
HUANCHEN ZHANG, Tsinghua University, China and Shanghai Qi Zhi Institute, China
MINGYU GAO, Tsinghua University, China and Shanghai Qi Zhi Institute, China

Indexing structures are widely used in modern data-processing applications to support high-performance queries, and there are a variety of recent designs specifically optimized for the newly available persistent memory (PM). The primary focus of previous PM indexes is on reducing the expensive PM writes for persisting data. However, we find that in tree-based PM indexes, because of the smaller performance gap between writes and random reads on real PM devices, the read-intensive tree traversal phase dominates the overall latency. This observation calls for further optimizations on existing indexing structures for PM.

In this paper, we propose Extendible Radix Tree (ERT), an efficient indexing structure for PM that significantly reduces tree heights to minimize random reads, while still maintaining fast in-node search speed. The key idea is to use extendible hashing for each node in a radix tree. This design allows us to have a relatively large fanout of the radix tree to keep the tree height small, and also to realize constant-time lookups within a node. Using extendible hashing also allows for incremental node modification without excessive writes during inserts and updates. Range queries are efficiently and robustly handled by enforcing partial ordering among the keys in the hash table of each node without introducing more hash collisions. Our experiments on both synthetic and real-world data sets demonstrate that ERT achieves up to 2.65×, 4.41×, and 2.43× speedups for search, insert, and range queries over the respectively state-of-the-art PM index.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Information systems → Data access methods**; *Storage class memory*.

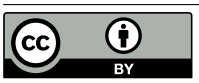
Additional Key Words and Phrases: persistent memory, index, radix tree, extendible hashing

ACM Reference Format:
Ke Wang, Guanqun Yang, Yiwei Li, Huanchen Zhang, and Mingyu Gao. 2023. When Tree Meets Hash: Reducing Random Reads for Index Structures on Persistent Memories. *Proc. ACM Manag. Data* 1, 1, Article 105 (May 2023), 26 pages. https://doi.org/10.1145/3588959

1 INTRODUCTION

Indexing structures provide a fundamental mechanism for quickly accessing records in a dynamic set. They are essential to achieve high performance in data-processing applications, such as relational databases [2, 26, 46, 56], key-value stores [11, 25, 27, 49, 57], and big data analytics [21, 31, 54]. Meanwhile, persistent memory (PM) technologies such as phase-change memory [40, 53] and 3D XPoint [16] are gaining more and more attention when building high-performance systems.

Authors' addresses: Ke Wang, wangke971111@gmail.com, Shanghai Qi Zhi Institute, Shanghai, China and Yale University, New Haven, USA; Guanqun Yang, gy2025@nyu.edu, Shanghai Qi Zhi Institute, Shanghai, China and New York University, New York City, USA; Yiwei Li, liyw19@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China; Mingyu Gao, gaomy@tsinghua.edu.cn, Tsinghua University, Beijing, China and Shanghai Qi Zhi Institute, Shanghai, China.



2836-6573/2023/5-ART105
https://doi.org/10.1145/3588959

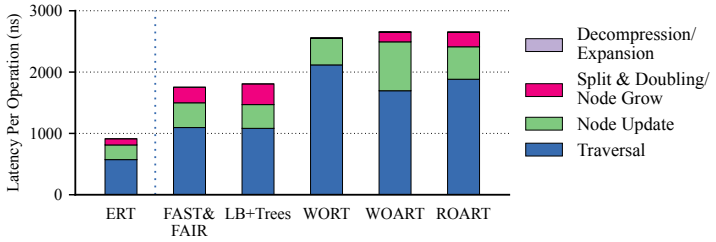


Fig. 1. Latency breakdown for insert operations on several state-of-the-art tree-based PM indexes [18, 22, 29, 32] and our design ERT. We use the Facebook user ID data set.

PM allows for byte-addressability with attractive features such as durability, high capacity, and relatively low latency. To leverage this technology, researchers have proposed a variety of indexing structures specifically optimized for PM [3, 8, 18, 22, 29, 30, 32, 35, 37, 52, 58, 59].

The primary focus of previous proposals for PM indexing structures has been to reduce write traffic while ensuring crash consistency. This is based on the assumption that the write bandwidth for PM is limited [30, 47, 51]. However, recent experiments on real PM products, such as Intel Optane DC PMM [19], show that the bandwidth for writes with `clwb` (i.e., cache line write back) is only $2-3\times$ lower than that for reads [47]. Moreover, architectural enhancements such as on-device write buffering and request scheduling have enabled writes with `clwb` to exhibit even lower latency, up to $5\times$ *lower* than random reads on PM [51]. These observations on real PM devices, together with the substantial algorithmic optimizations in state-of-the-art PM indexing structures, lead us to question whether write accesses continue to be the only dominating factor that limits the performance of tree-based indexes on PM.

We, thus, conducted a benchmark on a group of state-of-the-art tree-based PM indexes using real-world data sets. Figure 1 illustrates the breakdown of latency for insert operations. We notice that existing PM indexes spend most of their execution time (60.0% to 71.3%) performing tree traversals instead of node updates. Tree traversals involve multiple costly random accesses per query, and the number of random PM accesses during an index operation depends heavily on the tree height. One can increase the node fanout to reduce the tree height, but such an adjustment typically slows down the in-node search and in turn hurts the overall index performance.

This paper presents Extendible Radix Tree (ERT), an efficient indexing structure designed for PM that minimizes random accesses to PM and thus outperforms previous solutions. The key idea is to adopt the *extendible hashing* scheme in each node of a *radix tree*. This design allows us to increase the fanout of the radix tree (and therefore reduce the tree height) while keeping the in-node search fast. Additionally, the use of extendible hashing allows incremental index size growth without the need for a full hash table rebuild during resizing. Such a property avoids excessive data copying when updating the node.

To support efficient range queries, we extract the bit slices directly from the original keys to index the extendible hashing structure, instead of using a traditional hash function. This ensures that the keys in each ERT node are partially sorted. The probability of a hash collision when using the real key bit slices is still minimized because of the backbone radix tree's ability to compress the common prefixes of the keys. This ensures robust performance even against pathological key distributions. ERT further applies a multi-granularity structure in the node hash table [35] to allow for better tradeoffs between performance and memory efficiency, and also to match the read/write granularities in real-world PM devices. Finally, we carefully implement ERT so that it guarantees

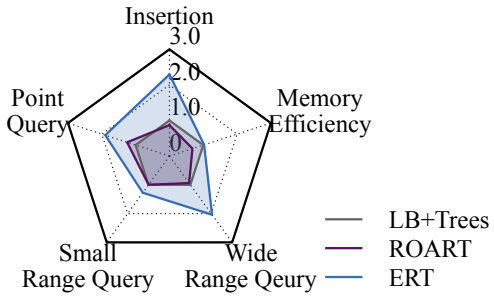


Fig. 2. Comparison of operation performance and memory efficiency among different indexing structures. Results are averaged over all benchmarks we evaluate, and normalized over LB+Trees. See Section 4 for more detailed evaluation.

crash consistency and scales well in a multi-threaded environment. The implementation of ERT is available at https://github.com/tsinghua-ideal/ExtendibleRadixTree.

We evaluated ERT using both synthetic and real-world data sets. Compared to state-of-the-art designs including FAST&FAIR [18], LB+Trees [29], WORT [22], WOART [22], and ROART [32], ERT exhibits superior performance for inserts (up to 4.41× and on average 2.11× against the second best), point queries (up to 2.65× and on average 2.14×), and range queries (up to 2.43× and on average 1.65×). Meanwhile, ERT also achieves comparable memory efficiency to the competitors above. Figure 2 summarizes the strengths and weaknesses of the different PM index categories based on our evaluation results, with LB+Trees and ROART as the representatives for B+Tree and radix tree variants, respectively.

The contributions of this paper are as follows.

- We identify that random reads (rather than writes) are the dominating factor that prevents current PM indexes from achieving higher performance.
- We propose a novel indexing structure ERT that integrates extendible hashing into a radix tree to reduce the number of random PM accesses during tree traversals while keeping range queries and node updates efficient.
- We demonstrate that ERT outperforms state-of-the-art solutions for common index operations under a variety of workloads.

The rest of the paper is organized as follows. Section 2 introduces the background of PM and indexing structures on PM, and also identifies the key performance bottlenecks which motivate our novel design philosophy. Section 3 proposes our design, Extendible Radix Tree (ERT), with detailed description of its structures, operations, parameter selection tradeoffs, and various feature support. Section 4 empirically evaluates ERT using both synthetic and real-world benchmarks and demonstrates its superior performance and efficiency. Section 5 compares ERT to related previous work, before we conclude the paper in Section 6.

2 BACKGROUND & MOTIVATION

Emerging persistent memory (PM) promises low latency, high capacity, and byte-addressability, which makes it attractive to store indexing structures for large-scale data-processing applications. Data stored in PM are durable and can survive power failures and system crashes. However, software is responsible to ensure *crash consistency*, meaning that data stored in PM must be in a consistent state after recovery from a crash. Because PM only guarantees an 8-byte failure atomicity,

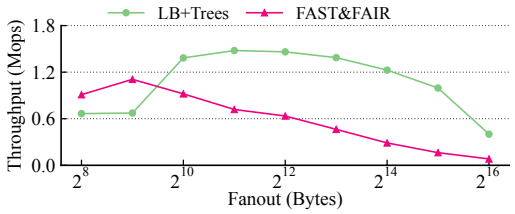


Fig. 3. Search performance of PM B+Trees [18, 29] when using different node fanout sizes, measured on a uniformly distributed dataset with 8-byte keys.

software must order PM writes using explicit memory fence (`mfence`, `lfence`, `sfence`) and cache line flush/writeback (`clflush`, `clflushopt`, `clwb`) instructions to avoid seeing partially-updated states after crash recovery. These extra instructions, however, often impose significant performance overheads due to the high cost of PM writes, making them a major optimization target when designing an indexing structure for PM [8, 20, 37]. Moreover, commercial PM devices organize their storage in blocks that are usually larger than a 64-byte cache line, e.g., 256 bytes [47, 51]. Therefore, while a read still accesses 64 bytes, every write would need to update 256 bytes, resulting in *different read/write granularities* compared to DRAM.

Most prior work tried to modify DRAM-based indexes to make them work efficiently on PM [3, 8, 18, 22, 29, 32, 37, 52, 58]. The focus is on reducing PM writes and persist cost (i.e., memory fences and cache line flushes). To ensure the aforementioned crash consistency when performing a structural modification operation (SMO), these solutions carefully order the atomic PM writes so that they can avoid write-ahead logging. For instance, FAST&FAIR [18] utilized dependencies between consecutive write operations to enforce the order of PM writes, which allowed it to only invoke cache line flushes when crossing cache line boundaries. It also proposed in-place update and rebalance operations to avoid logging during the node split process. LB+Trees [29] performed more word-granularity writes per cache line and reduced the number of cache line writes and flushes, based on the observation that the number of modified words in a cache line does not affect the performance of PM writes. ROART [32] minimized its persist overhead by adopting several optimizations like entry compression, selective metadata persistence, and minimally ordered split.

When the persist overhead has been minimized for PM indexes, new performance bottlenecks emerge. We benchmarked five state-of-the-art PM indexes, including two B+Trees (FAST&FAIR [18] and LB+Trees [29]) and three radix tree variants (WORT [22], WOART [22], and ROART [32]). This benchmark performs an insert-only workload on the Facebook user ID data set [41]. According to the latency breakdown shown in Figure 1, around 70% of the time for an insert operation is spent on the root-to-leaf tree traversal, which mainly involves random PM reads. The percentage would be even higher for lookup and range query operations that do not have the node update part. Minimizing the number of random PM reads, therefore, becomes the new primary optimization target for PM indexes to achieve further performance improvement.

A natural thought is to lower the index tree height so that each operation performs fewer pointer dereferences (random reads). However, simply increasing the node fanout would not suffice because it slows down the (binary) search within each node. Previous work proposed several optimizations to speed up in-node search. For example, FpTree [37] introduced a one-byte fingerprint for each key and formed a fingerprint array in every leaf node. Then, an in-node search could scan the fingerprint array first to avoid unnecessary PM reads. LB+Trees [29] improved this idea further by reorganizing the layout of the fingerprints to facilitate record moves and node splits. These

optimizations, however, are "relieves" rather than "cures" because scanning the fingerprints would quickly become the new performance bottleneck as the node size increases. We measured the point query throughput of LB+Trees [29] and FAST&FAIR [18] using uniformly distributed keys while varying their node sizes. As shown in Figure 3, neither index could perform well with large nodes. The throughput of LB+Trees peaks at 2048 bytes per node while that of FAST&FAIR drops consistently after a 512-byte node size.

The same dilemma applies to radix trees, too. A larger fanout means that every radix tree level "consumes" a longer key slice and, therefore, reduces the number of levels. However, a larger node leads to either performance degradation or memory space waste in a radix tree. If the node layout is an array of branching labels (e.g., `Node4` or `Node16` in ART [24]), enlarging the node size would incur more cache misses during the in-node search (similar to B+Trees). Alternatively, if the node is implemented using a fixed-length bit vector (e.g., `Node256` in ART), the space consumption would rise exponentially as the node fanout increases. HOT [6] proposed to combine a subtree (i.e., multiple BiNodes) into a compound node to enable a consistently high fanout, thus achieving a lower tree height. This approach, however, would become suboptimal when applied to PM, because it introduces complex bit operations (mainly involving updating the "discriminate bits") during insertion, leading to a high write amplification (must perform copy-on-write rather than in-place update) to guarantee crash consistency.

Our approach. We argue that a better way to achieve both *low tree height* and *fast in-node search* is *to integrate hash tables into radix trees*. Specifically, each radix tree node can be implemented as a hash table. In this way, we can significantly increase the node fanout without sacrificing the in-node search performance, because hash tables guarantee constant-time lookups regardless of the number of records stored.

Nevertheless, there are two additional challenges associated with such a design. First, hash table resizing caused by node expansion, merge, or split would need to reorganize all the existing records in the node, resulting in excessive data copying on PM, and slowing down inserts and updates to the index. As detailed in Section 3, we solve this challenge by leveraging extendible hashing [13] where the size of the hash table can grow/shrink incrementally without an expensive full rebuild.

The second challenge is to support efficient range queries despite that the key slices stored in a node are out of order. In a typical hash table, consecutive keys, e.g., 0000_2 and 0001_2, may be hashed to completely arbitrary places, and their orders are not preserved. This makes range query processing difficult to implement, as there is no easy way to locate the next key in the range directly from the position of the current key. The solution we propose to this challenge is to directly use the most significant bits obtained from the real keys (without invoking a hashing function) to index the extendible hash table, so that the records are partially sorted. This approach, however, may cause more hash collisions for skewed key distributions. Fortunately, this issue can be largely alleviated by the inherent prefix compression of the radix tree, which ensures that the key slices in each radix tree node do not share a common prefix. We also adopt a hierarchical node structure with a different ordering guarantee at each granularity to better balance hash collisions and range query efficiency. Our experiments in Section 4.2 show that the performance of our design is robust against worst-case densely-distributed keys.

We name the above design the Extendible Radix Tree (ERT). In the next section, we will elaborate on how we forge a radix tree and extendible hashing together to achieve the best of both worlds.

3 EXTENDIBLE RADIX TREE

Extendible Radix Tree (ERT) is a novel design that judiciously combines the advantages of existing indexing structures and meticulously addresses the challenges outlined in Section 2. To the best of our knowledge, ERT is the first data structure that integrates hash tables into a radix tree while

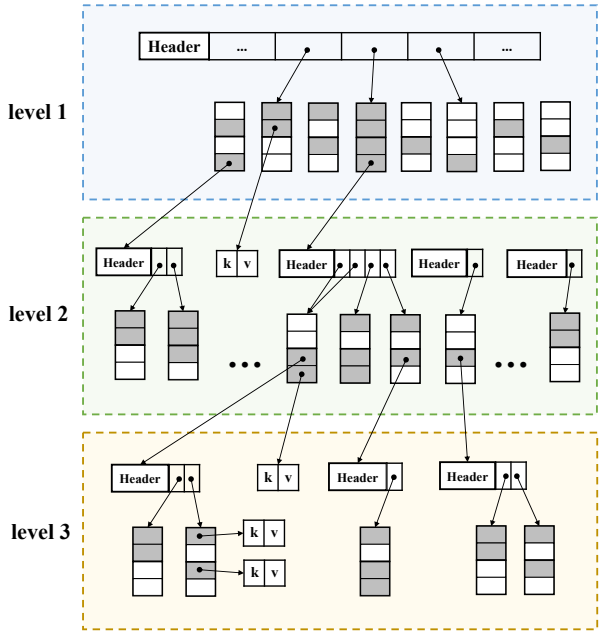


Fig. 4. Overall structure of Extendible Radix Tree (ERT).

carefully balancing various tradeoffs, such as tree height vs. in-node search speed, and range query performance vs. memory utilization. This section presents the detailed design of ERT. We begin by giving an overview of ERT (Section 3.1). We then introduce its node structure (Section 3.2), and show the details of different operations on ERT, including search, insertion, and range queries (Section 3.3). We discuss the key parameter selection tradeoffs in ERT (Section 3.4), most importantly, the bit slice length of each radix tree level and the hierarchical node structure granularities, to match PM device access characteristics and to ensure robust range query performance. Finally, we talk about how ERT supports crash consistency, concurrency, and memory safety (Section 3.5).

3.1 ERT Overview

Figure 4 shows the overview structure of ERT. Essentially, ERT is a radix tree but with a much larger fanout (storing up to 2^{32} keys per node) compared with traditional radix trees (up to 256 keys per node). Each level of the tree corresponds to a bit slice from the key, referred to as the *subkey*, and the length of the subkey is called the *span*. Each node uses the subkey to index its internal node structure to determine the child node in the next level.

ERT supports variable-sized keys by adopting different numbers of tree levels for different key sizes. For example, if the span is 32 bits, a 64-bit key can be stored at level 2 and a 128-bit key can be stored at level 4. ERT further supports path compression and lazy expansion as in a typical radix tree, so that records could be stored at any level in the tree, as illustrated in Figure 4. For example, a 128-bit key can be also directly stored at level 1. A comprehensive summary of how to search a key in ERT is described in Section 3.3.

The main goal of ERT is to reduce the tree height while at the same time maintaining efficient in-node search. To achieve this, we utilize *extendible hashing* in each tree node to facilitate fast in-node search. This allows us to use a wide span at each tree level, e.g., 16 to 32 bits, and still

achieve constant search time within each node. With this much larger span, a typical ERT is very shallow, only requiring up to two (32-bit span) to four (16-bit span) levels for the common 64-bit keys. Another benefit of using extendible hashing is that compared to normal hash tables, it could eliminate the excessive PM writes of a full table rebuild when resizing is needed, and allows for dynamic and incremental increases in node size without significant memory space waste.

Directly adopting the vanilla extendible hashing scheme does not support efficient range query processing. Therefore, in ERT, instead of the hashed value, we *use the subkey itself*, and *take the most significant bits* (MSBs) from it rather than the least significant bits (LSBs), to index the internal structures of the extendible hash table. In other words, we only preserve the structure of extendible hashing without using a traditional hash function. This scheme is able to create some ordering between the subkeys stored in the node and, therefore, prevents range queries from scanning excessive amounts of unnecessary records (Section 3.3). For example, if we use the most significant two bits of the subkey as the index, all the subkeys with a prefix of 01_2 would be placed after those smaller subkeys with the prefix of 00_2. A range query can thus first obtain all the groups of records with their key prefix bits overlapping with the queried range, and then filter only within these small amounts of records.

A potential concern of not using a traditional hash function in the extendible hashing is that it may cause excessive collisions for skewed data sets. However, such a penalty is much alleviated by the following two design features. First, the backbone radix tree of ERT is actually quite friendly to skewed data sets due to the path compression capability. Essentially, the same prefix of the skewed keys, which is usually detrimental to hash tables, is compressed along the radix tree path, and only the remaining subkey bits are distributed in the extendible hash table of a tree node. Second, to further resolve hash collisions when the short subkey in each node is still skewed, ERT applies a hierarchical node structure similar to the state-of-the-art PM hash table CCEH [35]. It uses two granularities with different index bit selections from the subkey, i.e., adding an extra segment level, in order to achieve a better balance between the hash collision rate and the ordering requirement of range queries. This hierarchical structure also has another benefit that matches well with the different read and write granularities in commercial PM devices [35]. Section 3.4 discusses the subkey length selection and the node granularities in more detail, which would affect the aforementioned performance tradeoffs. Section 4.2 empirically shows that ERT performs even better on the skewed (dense) data set than on the sparse set.

3.2 ERT Node Structure

We first explain the layout of our node structure design, and later describe the operations with more complete details in Section 3.3. Figure 5 shows the detailed node structure of ERT. Each node consists of a 16-byte header and an extendible hash table.

The first 8 bytes in the header are used for crash-consistent path compression of the radix tree [22], containing a 1-byte `depth` field, a 1-byte `prefix length` field, and a 6-byte `prefix array` field. The `depth` field denotes the *true* level of this node in the tree, i.e., the subkey bit slice position in the full key, rather than the depth of its parent plus one. The `depth` of the root node is 0. The `prefix length` and `prefix array` fields are used for the hybrid path compression scheme as in ART [24]. Specifically, `prefix length` stores the common key prefix length and `prefix array` stores (part of) the common key prefix. If the key prefix fits in the header (less than 6 bytes), we compare the queried key to it before proceeding to the next level (pessimistic). Otherwise, we only use the length and delay the exact comparison to the end when finally reaching a stored record (optimistic). For the next 8 bytes in the header, a `pointer` field directly points to the record whose key fully and exactly matches the `prefix array`. Because current processor architectures only use 48-bit virtual addresses, we repurpose the remaining 2 bytes to store the state bits including

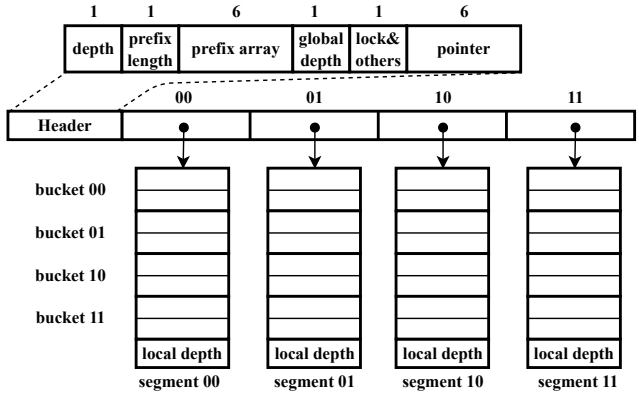


Fig. 5. ERT node structure.

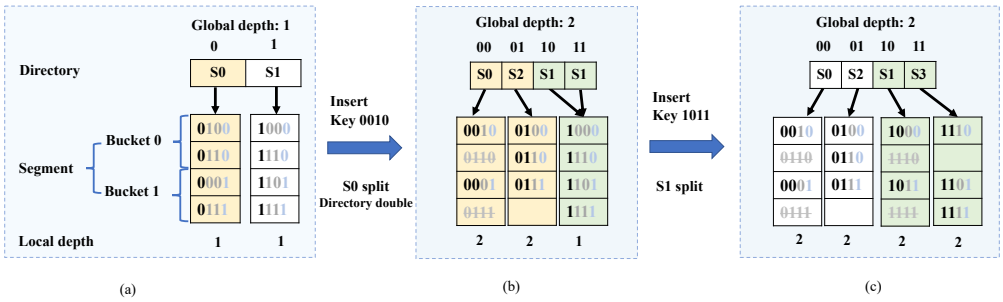


Fig. 6. Examples of the ERT node extendible hash table structure and its insert operations.

the lock and the extendible hash table `global depth` value (explained shortly). Note that `global depth` is a parameter of extendible hashing, and `depth` is a parameter for the radix tree. Overall the header can be densely compacted within 16 bytes.

The central part of an ERT node is the extendible hashing structure to index a subkey in the node. ERT makes several changes to the vanilla extendible hashing structure. First, similar to the state-of-the-art PM hash table design CCEH [35], in addition to the global directory and the buckets in a vanilla extendible hash table, we add an extra intermediate layer named *segments*, as shown in Figure 5. Each slot in the global directory points to a segment, and each segment consists of several buckets. This is different from a vanilla extendible hash table, where the global directory slot directly points to each bucket. The two levels of segments and buckets in ERT could better match the different read and write granularities of commercial PM devices [35], which we discuss further in Section 3.4. Second, as mentioned before, we use the plain subkey directly, without applying a hash function. And instead of using the LSBs of the key hash as in a typical extendible hash, we use the MSBs of the subkey to index into a segment. This allows us to better support range queries.

Putting things together, in a node in ERT, we use the G MSBs of the plain subkey to index to one of the 2^G segments following the pointer stored in the global directory. Within the segment, we use the B LSBs of the subkey to index to a bucket. G is the *global depth* stored in the header. As

records are continuously inserted, the global depth can increase dynamically and incrementally by doubling the directory size (see below for details), in order to accommodate more segments and buckets. On the other hand, B keeps as a constant, so a segment contains a fixed number of buckets. Each segment also maintains a *local depth* as in typical extendible hashing, which indicates the common prefix bit length of the subkeys in this segment and must be equal to or less than the global depth. When the local depth of a segment is smaller than the global depth, this segment is shared by multiple global directory slots, and the subkeys in it could match multiple G-bit prefix bit slices that correspond to these global directory slots. For example, when $G = 3$ and a segment is shared by the global directory slots 100_2, 101_2, 110_2, and 111_2, its local depth is 1, meaning that only the first MSB is the common prefix of the subkeys in this segment.

Figure 6(a) shows an example of the ERT extendible hashing scheme, with two segments, two buckets per segment, and each bucket can store up to two records. In this example, $G = 1$ and $B = 1$, and both segments' local depths are equal to the global depth. Assume we want to search for a key 0111_2. We first locate segment $S0$ in the directory through the MSB 0_2. Then we access bucket 1 in $S0$ following the LSB 1_2. Finally we scan the bucket to find the record.

Insertion is more complicated and may result in node structure changes. We first find the target bucket as in search. If the target bucket is not full, we directly insert into it. Otherwise, as in Figure 6(b) when inserting key 0010_2, we find bucket 0 in $S0$ is full. We then need to split the segment into two to allow for more spaces to accommodate more keys (*segment splitting*). However, in this case the segment's local depth is equal to the global depth, meaning there is no empty space in the directory to store the additional segment pointers. We need to first increment the global depth to 2 and double the directory size (*directory doubling*). A new segment $S2$ is allocated for prefix 01_2, and the original $S0$ only keeps the keys with prefix 00_2. Both segments $S0$ and $S2$ now have their local depth values as 2. On the other hand, $S1$ is kept unchanged, shared by both the 10_2 and 11_2 slots in the directory, and has the local depth as 1 which is smaller than the global depth. If we further insert key 1011_2 in Figure 6(c), we can directly split segment $S1$ without directory doubling; the new segment $S3$ can be stored in the 11_2 slot in the directory. To summarize, there are three cases for an insert operation, in the order of increasing complexity: direct insertion, segment splitting, and directory doubling.

When migrating keys between segments for segment splitting (e.g., 0110_2 and 0111_2 from $S0$ to $S2$, and 1110_2 and 1111_2 from $S1$ to $S3$), ERT uses *lazy deletion*, and does not delete the migrated keys in the old segment to reduce extra writes to PM. There is no correctness issue because they will not match the expected prefix (because the prefix length is determined by the local depth) and can be easily filtered out during search.

For range queries, assume we want to search for keys between 0111_2 and 1100_2 in Figure 6(c). We first locate the leftmost segment $S2$ and the rightmost segment $S3$ which contain the begin and end keys. Scanning these two segments, we find 0111_2 in $S2$ is in the range and there is no key in $S3$ in the range. Besides these two edge segments, we return all the keys in the segments between $S2$ and $S3$, i.e., 1000_2 and 1011_2 in $S1$ in this example. Combing both parts, we return three keys as the final results, 0111_2, 1000_2, and 1011_2. We do not need to scan segment $S0$ because all the keys in it have the same prefix 00_2, which does not overlap with the queried range.

As another optimization, in Figure 5, we append the extendible hashing global directory to the ERT node header in one contiguous memory space. This saves one pointer dereference and improves cache line utilization. However, the downside is that when the directory gets doubled, the header must be copied together and result in extra writes. We argue that such overheads are negligible. When the directory is initially small, e.g., with only 2 to 4 slots, the total size of the header plus the directory is smaller than the minimum PM write granularity of 256 bytes, and thus causes no extra writes. When the directory has grown large, the fixed 16-byte header will then only

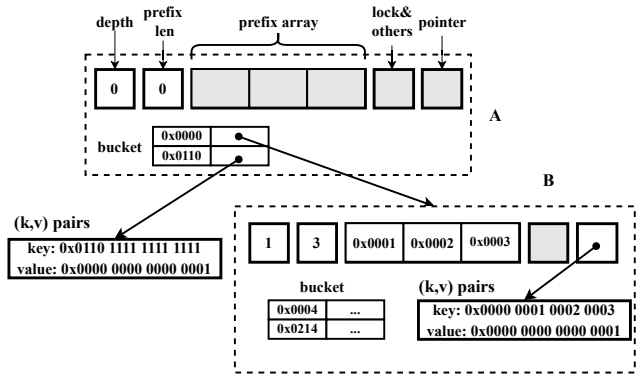


Fig. 7. An example of ERT structure. The tree uses a span of 16 bits.

be a small portion. In fact, after n times of directory doubling, the header results in just $O(n/2^n)$ data writes relative to the directory itself.

We remark that the global depth G of extensible hashing in each tree node is automatically adjusted according to the key distribution, following the extendible hashing algorithm. It does not affect point query performance as we use the MSBs of the subkey to directly index the directory to access the segment. On the other hand, it may affect the efficiency of range queries. With a larger global depth, each segment in ERT represents a smaller range so that it provides a better partial ordering of the keys in the node, thus achieving better range query performance.

3.3 ERT Operations

In this section, we illustrate the point query, range query and insert operations in ERT in detail.

Point query. Each point query starts from the root node. In each node, it first compares the key with the compressed prefix in the node header. If the prefix matches, it searches in the extendible hash to find the child. The child can be either another internal node or a leaf node. If the child is an internal node, it continues to search in the same way. If the child is a leaf node, it compares the key with the full key in the leaf node and returns the value if they are the same. Note that the full key is always stored in the leaf node for verification. We dynamically upgrade a leaf node into a subtree when new records are inserted and the subkey is not unique any more (see below for insertion). To summarize, records can be stored in two ways in ERT. First, a record can be found in a bucket after we traverse the tree and reach a leaf node. The leaf node could be at any intermediate level of the tree, i.e., with path compression, if its current subkey is unique (case 1). Second, to also support variable-sized keys, especially the scenario when one key is exactly the prefix of another, we reserve a `pointer` field in the node header (case 2).

For example, assume the ERT structure shown in Figure 7 uses a span of 16 bits. Given a key 0x0110 1111 1111 1111, because the `prefix length` of the root node A is 0, we use the first 16 bits (0x0110) as the subkey to locate one of its children. We find there is a match with the subkey in the bucket, and we reach a leaf node, which means no other keys in the structure starting with 0x0110. We compare the queried key with the full key stored in the leaf and find they match, so we can return the value (case 1 above). For another case, assume we want to search for the key 0x0000 0001 0002 0003 0004 0005. We first search in the root node A and traverse to node B in the next level. Because node B compresses a prefix of 3 spans (`prefix length` is 3), i.e., 0x0001 0002 0003 (`prefix array`), which matches with the queried key, we would continue the

search with the next subkey, 0x0004. Finally, when we search for another key 0x0000 0001 0002 0003, because it is exactly the same as the compressed prefix in node B, it is stored using the `pointer` field in the header (case 2 above).

Range query. We use the subkey itself instead of its hash value to index the node hash table. Because all the segments are selected by the subkey MSBs, the segments are sorted, e.g., subkeys in segment 00 are smaller than subkeys in segment 01. This provides the potential to support range queries. The buckets within a segment, and the records in each bucket, are however not sorted.

When performing a range query, we first use the begin and end keys of the range to locate their corresponding segments at the leaf level of ERT. When the two keys have a common prefix, we only need to traverse the top tree levels once. All records in the segments between the two edge segments, i.e., those in the subtree under the point when the begin and end keys first diverge, are guaranteed to be in the queried range. The records in the begin and end segments, however, need to be scanned to filter away those outside the range, resulting in some overheads.

Because of the shallow tree depth, ERT has few random accesses (to traverse the tree) plus several large sequential accesses (to scan the segments and filter the records). This is in contrast to existing index designs like B+Trees and radix trees, which have larger tree heights and smaller node sizes, leading to multiple random accesses followed by a sequence of small sequential accesses. Trading more sequential accesses for fewer random accesses is usually beneficial, but it could result in a moderate performance drop at very small ranges. We further discuss the balance between the cost of segment filtering and the benefits of sequential accesses in Sections 3.4 and 4.4.

Insert/update/deletion. We describe the volatile version of modifying ERT here, and leave the crash consistency and concurrency issues to Section 3.5. To insert/update a record into ERT, we traverse the radix tree in the same way as lookup. If we locate a leaf node at any level, we need to further verify whether the inserted key and the existing key match exactly. If so, the insertion becomes an update (a.k.a., upsert). Otherwise, the two only share a common prefix, and we need to upgrade the leaf node to a new subtree, typically with just one node (Figure 5), to store both records. If we do not find a child node either because of a subkey miss at any level or because we exhaust all levels, we insert the record to the current level. The insertion may cause segment splitting and/or directory doubling in the node structure, which would proceed in the same way as Section 3.2 and Figure 6 have illustrated.

ERT also supports path compression as radix trees. When we reach a node and find a mismatch against the stored `prefix array` in the header, we need to decompress the path with a node split. Figure 8 shows an example. The tree initially has the root node A, whose subkey 0x0000 points to a child node B. B stores a compressed path 0x0001 0002 0003. When we next try to insert a new key 0x0000 0001 0004 0005, we traverse from A to B, and find only a partial match with the compressed path in node B. Hence we create a new intermediate node C with the common prefix 0x0001, which then links to the two children: the original node B through the subkey 0x0002, and the inserted record through the subkey 0x0004. Accordingly, we change the `depth` from 1 to 3 and the `prefix length` from 3 to 1 in node B.

Deletion in ERT operates similarly, with the additional actions of memory reclamation when nodes and segments become empty, which we discuss in Section 3.5.

3.4 Parameter Selection Tradeoffs

ERT integrates radix trees and extendible hashing. We now discuss how to carefully select their key parameters to realize the most benefits from both and fulfill the best potential of PM.

Radix tree span. The length of the subkey at each radix tree level has a major impact on both search performance and space efficiency. A large span reduces the tree height and lookup latency. The extreme case is a single-level tree, and ERT degenerates to a hash table at the root. However,

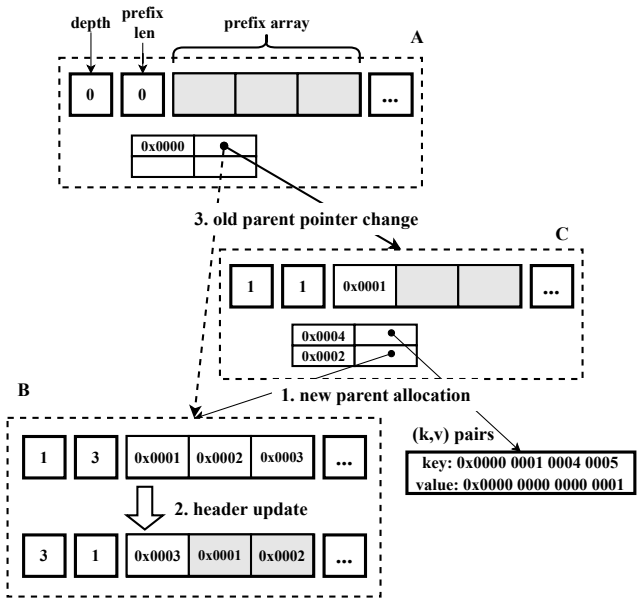


Fig. 8. Path decompression in ERT. Assume a span of 16 bits.

without hashing keys in ERT, a single extensible hash table is highly vulnerable to pathological cases. For example, with a skewed data set of only small-value keys, e.g., 100M keys ranging from 0 to 100M, the keys will collide in the first few segments with the common prefix $00\ldots0_2$ and cause frequent segment splitting and directory doubling, until the global depth is large enough to distribute the keys to enough segments. This leads to substantial memory space waste. The repetitive segment splitting and directory doubling, as well as the associated memory allocations, also incur significant performance degradation.

ERT addresses this issue by dividing the key into multiple bit slices, and using a radix tree to compactly represent the common slices. Assuming 8-byte keys, if we use a four-level tree, each node only deals with a 16-bit subkey, and the directory could at most extend to 64 slots if each segment holds 1024 records. This also limits the times of directory doubling and avoids its overheads. Section 4.4 presents our detailed experimental exploration on subkey length selection, and we find that a span around 32 bits achieves a good balance among search performance, insertion cost, and space efficiency. There also exist opportunities to achieve better performance by statically tuning the span choice to a specific workload and data set, if we can obtain their statistics through profiling before constructing the index. A more complex approach may further adjust the span dynamically to better adapt to workload behaviors. We leave these directions as future work.

Node segment and bucket sizes. Real PM devices exhibit different read/write granularities, e.g., 64 bytes (i.e., one cache line) for a read and 256 bytes for a write. Correspondingly, the common read and write operations in ERT should match the device properties. Fortunately, having two levels of segments and buckets in the ERT node allows us to decouple such read and write granularities and optimize each individually.

First, when searching for a record in ERT, we are able to use the MSBs and LSBs in the subkey to locate only a single bucket. The subkeys in the bucket are compared sequentially to the query, so a

bucket is mostly read in its full. We hence set the bucket size in the ERT node to 64 bytes to match the PM read granularity and achieve the best search performance.

Second, the segment size should be a multiple of the minimum PM write granularity, 256 bytes, to reduce write amplification. When splitting a segment, the new segment can be written once if its size matches the PM write granularity. Note that we do not need to update the old segment because of the lazy deletion introduced in Section 3.2. While a larger segment size than the minimum 256 bytes can reduce the number of segment splitting and directory doubling, it may increase the cost of each split since more keys are migrated. This can potentially increase the tail latency when accesses are blocked behind a split. Additionally, the segment size in ERT exhibits a new tradeoff that affects the range query performance. With our ERT node structure that uses the MSBs of the real subkey as the segment-level index, the ordering of the records between different segments is preserved, but the records in each individual segment are not sorted. On a small range query, the cost of filtering edge segments would dominate if too large segments are used. On the other hand, when the range query is wide, a larger segment size could result in more sequential accesses and reduce jumps across segments. In Section 4.4, we empirically find that 16 kB is a good segment size to reach a balance among the above effects. This result matches with the findings in CCEH [35].

3.5 Other Features

Crash consistency. ERT provides the same crash consistency guarantee as other PM indexing structures. When inserting a record, we first write the value, call a memory fence (`sfence` in our implementation), then write the key followed by another memory fence, and finally persist them using a cache line flush (`clwb`). This results in two fences and one cache line flush for each persist operation. When the (sub)key does not exceed 16 bits, we embed it with the 48-bit virtual-address pointer, and write and persist the 8-byte key-value pair together. This reduces one memory fence.

For SMOs in the radix tree (node split, etc.), we add a `depth` field to each node header to guarantee failure atomicity during path decompression, similar to WORT [22]. As illustrated in Figure 8, we first allocate the new intermediate node C, write its header and the pointers to both children, and then persist it (step 1). Next, we update the header of node B with one atomic 8-byte write, changing the `depth` from 1 to 3 (meaning the current compressed prefix 0x0003 is the 3rd subkey counted from 0) and updating the `prefix array` and `prefix length` to the new prefix (step 2). Finally, we set the pointer in node A to point to node C. If a crash happens before step 2, the split essentially has not started and we only need to reclaim the allocated node C during recovery (Section 3.5). If a crash happens between steps 2 and 3, the `depth` field allows us to detect the transient inconsistency and then recover [22]. Specifically, we detect that the `depth` mismatches with the subkey position, and recover the previous header in node B.

We apply an additional optimization here. When updating the `prefix array` of node B in step 2, we store the decompressed prefix (0x0001 0002 in Figure 8) in the obsolete space right after the valid prefix (0x0003). When we recover node B to its original state, this stale prefix allows us to directly restore the previous prefix without computing the largest common prefix from the node itself as done in WORT [22]. While the common prefix computation is fast in WORT, ERT stores much more keys in one node and in an unsorted way. So avoiding such computation can save significant work.

Insertion may also cause the creation of a subtree in ERT (Section 3.3). We first build and persist the entire subtree to PM. Only after that, we atomically direct the parent's pointer to the new subtree root. Finally we persist this pointer.

SMOs of the extendible hashing based node in ERT are essentially the same as CCEH [35]. Directory doubling is achieved by atomically swapping the pointer in the parent to the newly allocated and persisted directory. Segment splitting updates the directory slots from right to left,

and persists at every time crossing a cache line boundary. System crashes and power failures may happen in the middle of these directory slot updates. When recovering, we scan the directory from left to right, and check whether the strides of continuous slots match with their local depths and the global depth. For detailed description please refer to the CCEH paper [35].

Concurrency. To support thread-safe, concurrent accesses to ERT, we use fine-grained locking for both read and write operations. We protect each bucket, each segment, and each directory with individual reader/writer locks, respectively. We also apply a small-key optimization to further improve concurrency when possible. Specifically, if the span is no more than 16 bits, we can embed the subkey with the 48-bit pointer in one 64-bit space, so that a record can be atomically modified in the bucket using an 8-byte compare-and-swap (CAS), without the need of a bucket lock.

The locks at the segment layer are quite expensive in the original CCEH design. Operations may frequently compete for the same segment lock when there are only a small number of segments. In contrast, ERT applies hand-over-hand locking to only lock a subtree that is currently under operation, and releases the parent node after traversing to the child level. This enables more parallelism opportunities not only across different segments, but also across different subtrees.

DASH [30], an optimized version of CCEH, used lock-free access to the segments, i.e., with lock-free reads but fine-grained locks for writes. However, this scheme is incompatible with lazy deletion and in-place updates, and has to use copy-on-write-based segment splitting which is particularly expensive on PM. In contrast, the extra parallelism from ERT's tree structure reduces the necessity for such lock-free optimizations. Therefore, we prefer reader/writer locks in ERT, and leave further investigation of lock-free access as future work.

Memory safety. To prevent memory leakage and ensure memory safety, we need to reclaim memory spaces both during normal operations and when recovering after a crash.

During normal operations, because ERT uses read/write locks, it does not need complicated memory reclamation mechanisms such as epoch-based approaches [14] as required by lock-free designs. We can safely reclaim memory spaces in the following three circumstances. First, when performing directory doubling, we allocate a new space, and free the old one after updating the pointer in the parent level. Second, after deleting the last record in a segment, we can set the directory slot of this segment to point to its buddy segment, and decrease the buddy's local depth by one. Then we can free the empty segment. Third, when a leaf node in ERT becomes empty, we can free it after clearing the pointer in its parent node.

Memory safety under a system crash is much more complex and challenging. The PM allocator should first support crash consistency itself to ensure its metadata are always consistent. The system may also crash after we successfully allocate a node or segment but have not linked it to the main structure. Typically, there are two kinds of approaches, logging-based allocators [36, 43] and post-crash garbage collectors [5, 7, 9]. Determining the best among these methods is orthogonal to our work. For simplicity, we let ERT use a post-crash garbage collector to minimize runtime overheads.

4 EVALUATION

In this section, we experimentally compare ERT against five state-of-the-art tree-based PM indexes (FAST&FAIR [18], LB+Trees [29], WORT [22], WOART [22], and ROART [32]) to demonstrate the advantages and tradeoffs of our design.

4.1 Experimental Setup

System. We run all experiments on a server with two Intel Xeon Gold 6240 processors. Each processor runs at 2.6 GHz, and has 18 physical cores with hyperthreading enabled and a 24.75 MB L3 cache. The memory system consists of 252 GB DDR4 DRAM and 4×128 GB Intel Optane DCPMM.

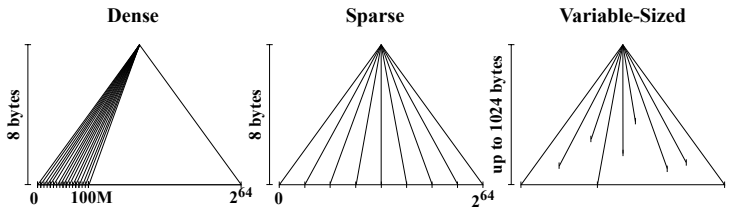


Fig. 9. Three synthetic data sets in our evaluation. All the keys in the Dense and Sparse data sets have a fixed length of 8 bytes. In the Variable-sized data set, the key lengths are between 8 and 1024 bytes.

The system runs Ubuntu 18.04. We manage the Optane DCPMM through a DAX file system mapped to a pre-defined address space. All data and metadata are allocated to the Optane DCPMM.

Data sets. We use both synthetic and real-world data sets. We fix the value sizes in all data sets to 8 bytes to represent pointers to database records. To investigate how robust ERT is against different key distributions, we generate three synthetic data sets, each having 100 million keys. As illustrated in Figure 9, keys in the Dense data set are randomly selected from the range 0 to 100M, thus sharing long common prefixes. Duplicated keys in the data set are treated as upserts during the initial load, and there are ≈ 60M unique keys in Dense. Keys in the Sparse data set follow a uniformly random distribution between 0 and 2^{64}. For the Variable-sized data set, we generate random strings with lengths between 8 and 1024 bytes.

We also evaluate three real-world data sets, all of which contain 64-bit integer keys. The Facebook [41] data set is an upsampling of 100M Facebook's user IDs. The Amazon [1] data set includes 20M unique book IDs sold on Amazon. The Wiki [48] data set consists of 50M unique timestamps when Wikipedia articles are edited.

Workloads and measurements. For both the synthetic and real-world data sets, we build the indexes by inserting the key-value pairs one by one (i.e., the *insert* workloads). For the synthetic data sets, we generate the *point query* workloads by randomly selecting keys from them. A *range query* is constructed as a random key plus a configurable scan length. For the real-world data sets, we apply YCSB [10] to generate different mixtures of query types. Specifically, we use YCSB-A (update-heavy with 50% point queries and 50% updates), YCSB-C (read-only with 100% point queries), and YCSB-E (scan-heavy with 95% range queries and 5% inserts) with a zipf distribution. Experiments are single-threaded unless stated otherwise. We measure the throughput, latency, and memory consumption for each index when executing the above workloads. We repeat each experiment 3 times and report the average measurements in the figures.

Parameters. We set ERT's parameters following the discussion in Section 3.4 and the experiments in Section 4.4. The default radix tree span is set to 32 bits. The segment size is 16 kB. The initial global depth of each ERT node is set to 0. For the baseline indexing structures, we follow their default parameter settings from the papers or the implementations.

4.2 Performance on Synthetic Benchmarks

Figure 10 shows the throughput results of point queries, inserts, and range queries on the synthetic benchmarks. The purpose is to analyze the relative strengths and weaknesses of ERT in a simpler and more controllable experimental setting. The results of the real-world data sets are presented in the next subsection.

Point query. ERT outperforms the other five tree-based PM indexes with an over 2× speedup compared to the second best one in each of the three synthetic benchmarks. This is because ERT

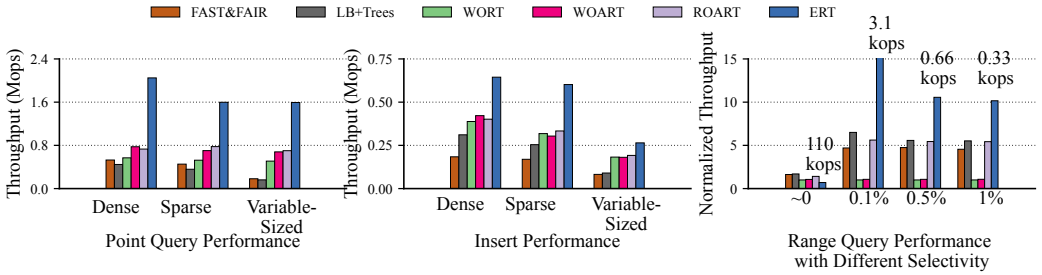


Fig. 10. Performance comparison on the synthetic benchmarks.

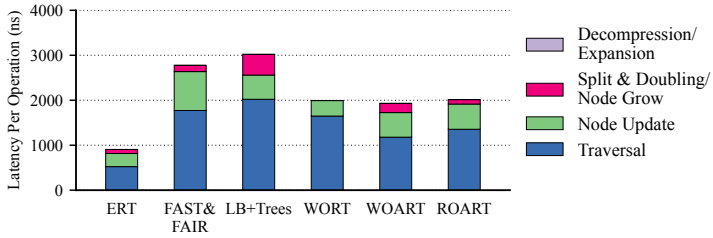
Table 1. Statistics of average visited nodes per search and average latency per node, on the Sparse data set.

	FAST&FAIR	WORT	WOART
Avg visited node number	4.375	6.297	3.364
Avg latency per node (ns)	198	118	181
	ROART	ERT	CCEH
Avg visited node number	1.983	1.002	1
Avg latency per node (ns)	395	346	250

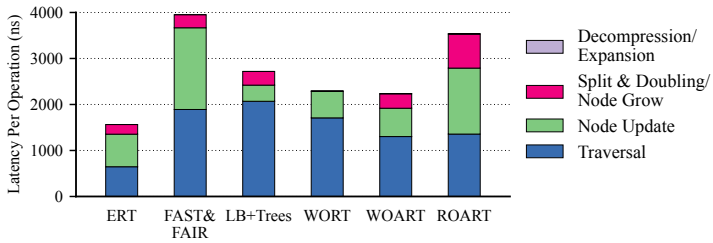
specifically focuses on optimizing the tree traversal efficiency. ERT not only has a small tree height but also supports fast in-node search. Such a performance advantage is robust across different key distributions. Table 1 shows that ERT only visits 1.002 nodes on average for point queries on the Sparse data set, which is close to what a hash table could accomplish. Although ERT's tree height grows linearly with the key length in theory, we observe that most of the keys in our Variable-sized data set are terminated (i.e., can be uniquely identified) in or very close to the root thanks to path compression and lazy expansion. Meanwhile, because ERT has larger nodes, its in-node search latency is higher than most previous designs in Table 1. However, the benefit of reducing the tree height outweighs such extra cost. ROART also tried to trade in-node search latency for a smaller tree height. ERT achieves a Pareto improvement over ROART in this tradeoff.

We note that the original implementations of FAST&FAIR and LB+Trees did not support variable-length keys. We modified their source code so that they can recognize pointers to keys stored outside the tree nodes (we only use the modified versions in the Variable-sized workload). As shown in Figure 10, this incurs significant pointer-dereferencing overheads.

Insertion. The performance for inserts generally follows the same trend as that for point queries. Compared to a lookup, ERT exhibits a relatively lower advantage over the others because of the cost of PM writes during an insert. Figure 11 shows a latency breakdown of the insert operation. For all the indexes in the experiment, the dominating component is tree traversal, followed by node update and node grow/split. Similar to point queries, ERT obtains most of its performance gain from the optimized tree traversal even for insertions. ERT is also very efficient at structural modifications (i.e., node grow/split) because it applies lazy deletion during segment splitting to avoid a full copy of the records in the node. Notice that the throughput advantage of ERT decreases for the Variable-sized data set because longer keys (up to 1024 bytes) lead to an increased cost in persisting the records, thus diluting the benefits of a more efficient tree design.



(a) On the Dense data set.



(b) On the Sparse data set.

Fig. 11. Latency breakdown for insert operations.

Table 2. Latency breakdown for range query operations, on the Dense data set.

Selectivity = 0.1%	ERT	LB+Trees	ROART
Tree traversal (ns)	2908	4526	2480
Segment scan (ns)	159370	315258	354811
Selectivity ≈ 0 (worst case for ERT)	ERT	LB+Trees	ROART
Tree traversal (ns)	155	1053	1652
Segment scan (ns)	4217	1340	934

Range query. We generate range queries with different selectivities ($\approx 0, 0.1\%, 0.5\%, 1\%$) on the Dense data set to benchmark the indexes (the observations are similar on the other data sets). The rightmost subfigure in Figure 10 shows the results. Notice that the y-axis shows normalized throughput because of the large differences in absolute throughput under different selectivities. We observe that ERT excels at longer range scans. The throughput, however, drops significantly when the range only contains a few items. Therefore a selectivity of ≈ 0 is the worst-case scenario for ERT. This is because the smallest ordering unit in ERT is a segment (i.e., ordering is not guaranteed within a segment). Although the range only contains a few entries, ERT has to scan the entire segment to fetch them. On the other hand, relatively larger segments are beneficial to sequential scans, and thus improving the performance of wider-range queries. The latency breakdown in Table 2 confirms our diagnosis. One can further speed up range queries in ERT by using a smaller segment size, but it in turn hurts the insert performance, as we will discuss in Section 4.4.

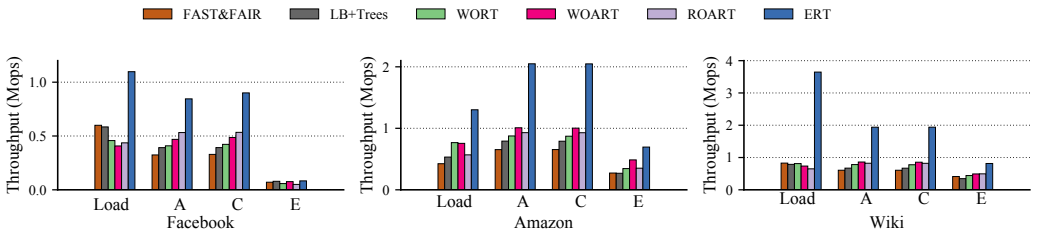


Fig. 12. Performance comparison on the real-world benchmarks.

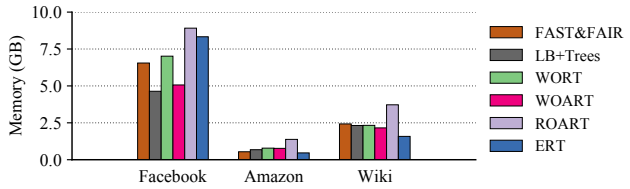


Fig. 13. Memory usage on the real-world workloads.

4.3 Performance on Real-World Benchmarks

We present the results of executing the YCSB benchmarks with the real-world data sets on ERT and the baselines in Figure 12. For `Load` and `YCSB-C` (i.e, read-only), the throughput results agree with what we have observed in the synthetic benchmarks. Across the data sets, all radix-tree-based indexes perform better in `Amazon` and `Wiki` compared to `Facebook` because `Amazon` and `Wiki` contain keys with denser distributions. We also note that ERT performs exceptionally well at loading the `Wiki` data set. This is because the keys in `Wiki` are sorted. Because we use the real keys to index the buckets in ERT's extendible hashing scheme, inserting an ordered sequence of keys benefits from better cache performance due to the sequential pattern.

For `YCSB-A` (update-heavy), the results are similar to those in `YCSB-C` because the procedure of a (value) update operation overlaps largely with that of a lookup where tree traversal remains dominant. ERT's performance advantage is compromised in `YCSB-E` (scan-heavy) because the ranges are small (< 100 items) and, therefore, it approaches the worst case in Figure 10.

Figure 13 shows the memory usage after loading the data sets into each index. ERT is more memory-efficient than the baselines in `Amazon` and `Wiki` because these two data sets have denser key distributions. Denser keys share common prefixes that facilitate compression in radix trees. Moreover, a denser key distribution leads to an improved load factor for extendible hashing used in each node (recall that we use the real key bits to index the hash table). Hence, a lower tree height and a higher load factor within the nodes allow ERT to consume less memory compared to other radix-tree variants for relatively dense keys. For a sparse key distribution such as in `Facebook`, ERT also achieves comparable memory efficiency.

We also evaluated the scalability of ERT using a read-intensive workload (95% point queries and 5% inserts) and a write-intensive workload (50% point queries and 50% inserts) derived from YCSB. We choose FAST&FAIR and ROART to represent the B+tree family and the radix tree family, respectively. As shown in Figure 14, ERT consistently outperforms the baselines as we increase the number of threads. ERT has the same trend in scalability as the baselines because they use similar locking schemes.

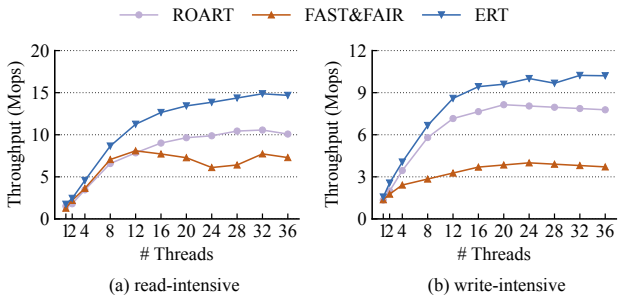


Fig. 14. Multi-threaded scalability comparison.

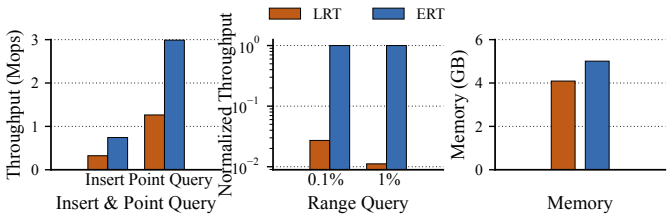


Fig. 15. Impacts of the hashing scheme choice in ERT. We replace extendible hashing in ERT with linear hashing to construct an alternative design named LRT.

4.4 Parameter Selection and Recovery

Hashing scheme. ERT uses extendible hashing for the node structure of the radix tree. We now explore alternative hashing schemes. As an example, we replace extendible hashing in ERT with linear hashing [28], and evaluate the performance and memory efficiency impacts. We use the `Sparse` data set as the representative. We set the parameters for linear hashing with a 75% split ratio and a 16-entry bucket size. As shown in Figure 15, the throughput of point queries decreases by nearly 3×. This is because linear hashing allows overflow: a query may have to traverse multiple linked overflow buckets, incurring extra random accesses. We find that each point query in our experiments traverses about three buckets on average in linear hashing. The overflow issue also affects the performance of insertion, where it needs to find an empty slot in a bucket. Another issue with linear hashing is that it performs the bucket split operation too frequently (it only splits one bucket at a time). As records keep getting inserted, it repetitively reaches the split ratio and triggers the split. This is particularly inefficient because of the expensive PM allocations. In contrast, ERT adopts a hierarchical structure and splits by segments rather than buckets, so that it has much fewer splits. Nevertheless, the finer split granularity of linear hashing improves the memory efficiency slightly compared to ERT.

For range queries, because linear hashing does not provide any ordering between keys, it has to scan all the keys in each node to identify those in the queried range. Such overheads significantly degrade the range query performance compared to ERT, by nearly two orders of magnitude. The performance gap is larger for larger ranges. There exist other order-preserving hashing schemes such as perfect hashing [12, 15] that could support efficient range queries, but they usually only work for static data sets, and incur significant overheads in dynamic settings.

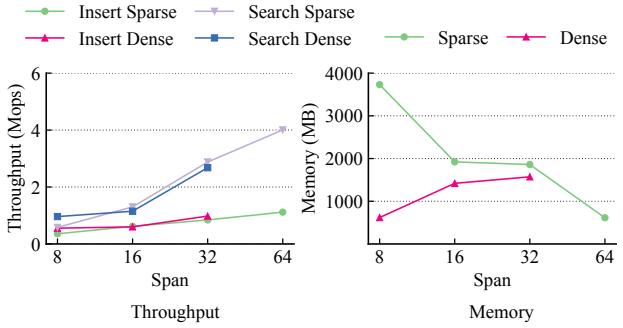


Fig. 16. Impacts of the radix tree span on ERT.

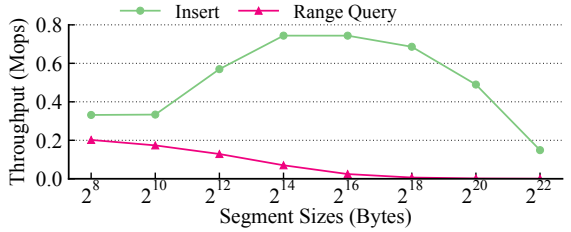


Fig. 17. Impacts of the segment size on ERT.

Span. As discussed in Section 3.4, the radix tree span is an important parameter that could affect the performance and memory efficiency of ERT significantly. We, thus, test ERT with different spans on both the `Dense` and `Sparse` data sets. The results are in Figure 16. Note that because the keys in the data sets are 8-byte integers, ERT reduces to single-node (root-only) for the 64-bit span. As the radix tree span increases, the tree height of ERT decreases correspondingly, leading to an improvement in lookup performance. The memory consumption changes depend heavily on the key distribution. Because we use the bits from the real keys instead of their hashes, the dense keys cause a large number of collisions and almost every collision causes a directory-doubling operation in extendible hashing. In fact, when the span reaches 64 bits for `Dense`, we do not have enough memory on the machine to hold the exponentially-growing directory (and thus the missing points in the figure). In contrast, the node sizes in ERT are relatively small with the `Sparse` data set. Pointers between nodes dominate the memory consumption in this case. Increasing the span reduces the number of nodes and, therefore, saves memory. We choose 32 bits as the default span for ERT because it is robust even in the worst-case dense keys. However, if the key distribution is reasonably sparse, increasing the span further can improve performance and memory efficiency simultaneously.

Segment size. Another important parameter in ERT is the segment size, which has an impact on the performance of inserts and range queries, as discussed in Section 3.4. We perform a parameter sweep on the segment size using the `Sparse` data set. We can see from Figure 17 that the insert throughput of ERT peaks at about 16 kB. A smaller segment size incurs excessive split operations, while a larger segment size increases the cost of each split. We also measure the throughput for the worst-case (i.e., only scans a few entries) range queries. As the segment size increases, the

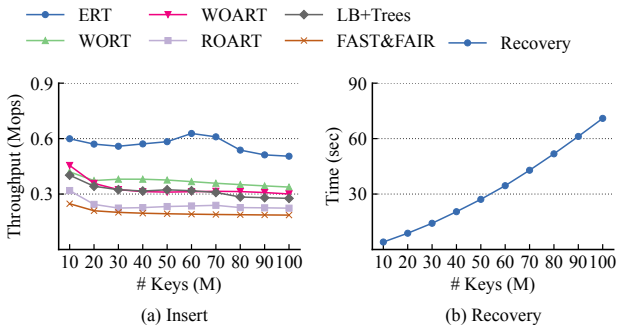


Fig. 18. Impacts of the number of keys on insert performance and recovery time.

performance of such a small-range query drops because ERT must scan an entire segment to answer the query. Based on the results in Figure 17, we set the default segment size to 16 kB because it achieves a good balance in the performance tradeoffs between inserts and small-range queries.

Performance robustness and recovery time. Segment splitting and directory doubling are both expensive SMOs, and may incur substantial performance overheads when we insert a key into ERT under specific states of the index. We evaluate these effects using the `Sparse` data set, and measure the throughput after inserting every 10M keys as in Figure 18(a). ERT shows stable speedups. The result indicates that the SMO cost in ERT does not frequently happen, and could be well amortized over a large number of key insertions.

In Figure 18(b), we evaluate the recovery time of ERT with different numbers of keys in the index on the `Sparse` data set. We inject a crash after inserting a certain number of keys, and traverse the index to recover all the transient inconsistency. With 100M keys (about 6.4 GB total tree size), ERT recovers successfully in 70.95 seconds. This recovery performance is on the same order of magnitude as previous tree-based PM indexes [32, 37, 52]. Recovery is only needed infrequently and is not a critical performance bottleneck. Our current implementation only uses a single thread. Optimizations such as multi-threading and lazy recovery can be applied to further improve the recovery speed as future work.

5 RELATED WORK

Persistent memory. Emerging persistent memory (PM) technologies [16, 40, 53] have brought a revolution to the system memory and storage architecture. PM is byte-addressable and non-volatile, and it could be accessed through the memory bus with normal load and store instructions, instead of via the file system interface as an I/O device. PM offers a much higher capacity, but exhibits only 1/3 bandwidth and over 2× latency compared to DRAM [51]. In particular, writes to PM are commonly considered quite expensive, because they are slower than reads, and also wear out the device to result in limited endurance. Our work further shows that random reads could also be a performance bottleneck, and efficient PM data structures like ERT should pay attention to reducing random read accesses.

Fully leveraging the non-volatile benefit of PM requires software to ensure crash consistency. There are usually three general approaches, atomic in-place updates [18, 29, 35], logging [4, 50], and copy-on-write (CoW) [39]. Among them, in-place update is the most efficient because it avoids extra PM writes. But because typical hardware only guarantees 8-byte atomicity, proper use of

memory fence and cache line flush instructions is needed to support larger data granularities. ERT is designed to leverage in-place updates to minimize the crash consistency cost [22, 35].

The recent eADR technology has the potential to relax data persistence requirements [42, 55]. As long as data reach the eADR domain, they can be considered as persisted, instead of being forced onto the PM. However, because of the extra high battery cost and currently limited compatible platforms [45], we choose to design ERT in a more general scope. Nevertheless, ERT's optimizations on random read accesses would be more critical if the persist cost reduces with eADR.

Indexing structures. The most commonly used indexing structures include B+Trees [26, 33], radix trees [24], and hash tables [13, 38]. All these index types have recently been optimized on PM [17] to leverage its persistency and high capacity while tolerating more expensive writes and crash consistency overheads [3, 8, 18, 22, 29, 30, 32, 35, 37, 52, 58, 59].

B+Trees achieve relatively good search and insertion performance and have the most efficient range query support. Each B+Tree leaf node contains multiple records as a sorted array, enabling fast search and sequential reads when locating and returning the queried range. There have been many proposals extending the classic B+Trees, such as Blink Tree [23], Palm Tree [44], and Mass Tree [33]. Specifically on PM, a large body of proposals [3, 18, 29] employ in-place updates rather than logging/CoW to avoid redundant writes, and to reduce expensive cache line flushes and memory fences. To further optimize performance, some recent B+Tree designs [3, 8, 29, 37, 52] do not maintain keys as sorted in their leaf nodes, but use fingerprints or additional indirection slots for both fast search speed and reduced writes. However, the keys in their internal nodes are still sorted. Some designs also adopt selective persisting by leaving non-critical data on DRAM and rebuilding them after a restart [37, 52].

An alternative tree structure is the radix tree, which does not explicitly store keys in nodes, but instead uses each bit slice of the key to determine the next branch to traverse within a fixed-length child pointer array. Radix trees usually exhibit better search but worse range query performance than B+Trees [34]. Path compression and lazy expansion are often used in radix trees to improve memory utilization and cache efficiency, at the cost of introducing complex node split/merge operations. Adaptive Radix Tree (ART) [24] further improves the memory utilization of each node, by dynamically changing the node format among four node types based on its utilization. Though less popular in disk-based storage systems, radix trees are more suitable for PM compared to B+Trees, as demonstrated by recent proposals like WORT&WOART [22], DPTree [58], and ROART [32]. First, searching a radix tree does not involve key comparisons, but just directly following the pointers indexed by the key's bit slices, saving memory accesses. Second, with fixed-length child pointer arrays, no key re-sorting is needed for every insertion. Third, expensive tree rebalance operations are also not needed as the tree structure is deterministic given the key prefix distribution. Therefore, ERT uses a radix tree as its backbone structure, and optimizes the tree height and the in-node search design.

Finally, hash tables can achieve constant lookup time and are superior in terms of point query performance due to the flat structure. However, supporting range queries is difficult due to the random position of each record after hashing. Static hash tables also need to estimate the table size in advance and pre-allocate space to avoid overflow and under-utilization. This is difficult in reality, and may lead to frequent rehashing when too many collisions occur. In contrast, dynamic hash tables, such as extendible hashing [13], allocate and deallocate memory space in an incremental manner to avoid full table rehashing. In light of PM, dynamic hashing is particularly attractive because it avoids excessive writes to redistribute the entire table during rehashing [30, 35, 59]. ERT borrows the multi-granularity structure from CCEH [35], which is a PM-optimized variant of extendible hashing.

6 CONCLUSIONS

We proposed a novel indexing structure on PM, Extendible Radix Tree (ERT), that hierarchically combines the radix tree and an extendible hash table design. ERT provides range query and variable-sized key support with both high search and modification performance. The main performance benefits of ERT come from its much smaller tree height and the ability to do almost constant-time search within each tree node. ERT also uses extendible hashing for incremental and in-place updates without excessive PM writes. We conduct extensive parameter exploration to ensure the internal organization of ERT matches well with commercial PM read/write granularities, and also achieves a balanced tradeoff among range query performance, insert cost, and memory utilization. ERT significantly outperforms existing indexing structures, even on real-world data sets and challenging cases with variable-sized keys.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions. This work was supported by the National Natural Science Foundation of China (62072262) and Shanghai Qi Zhi Institute. Mingyu Gao is the corresponding author.

REFERENCES

[1] Amazon. 2018. Amazon sales rank data for print and kindle books. https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD)*. ACM, Melbourne, Victoria, Australia, 1383–1394. https://doi.org/10.1145/2723372.2742797

[3] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565. https://doi.org/10.1145/3187009.3164147

[4] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348. https://doi.org/10.14778/3025111.3025116

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Amsterdam, The Netherlands, 677–694. https://doi.org/10.1145/2983990.2984019

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, Houston, TX, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[7] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. ACM, virtual [London, UK], 60–73. https://doi.org/10.1145/3381898.3397212

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

[9] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-oriented Recovery for Non-volatile Memory. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 153:1–153:22. https://doi.org/10.1145/3276523

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC)*. ACM, Indianapolis, Indiana, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[11] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-Value Stores Serving Large-scale Applications: The RocksDB Experience. In *Proceedings of the 2021 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Virtual Event, 33–49.

[12] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. 2020. RecSplit: Minimal Perfect Hashing via Recursive Splitting. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Salt Lake City, UT, USA, 175–185. https://doi.org/10.1137/1.9781611976007.14

[13] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems* 4, 3 (1979), 315–344. https://doi.org/10.1145/320083.320092

[14] Keir Fraser and Tim Harris. 2007. Concurrent Programming Without Locks. *ACM Transactions on Computer Systems* 25, 2 (2007), 5. https://doi.org/10.1145/1233307.1233309

[15] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with O(1) Worst Case Access Time. *Journal of the ACM* 31, 3 (1984), 538–544. https://doi.org/10.1145/828.1884

[16] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

[17] Xiangpeng Hao, Lucas Lersch, Tianzheng Wang, and Ismail Oukid. 2020. PiBench Online: Interactive Benchmarking of Persistent Memory Indexes. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2817–2820. https://doi.org/10.14778/3415478.3415483

[18] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 2018 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Oakland, CA, USA, 187–200.

[19] Intel. 2021. Intel Optane DC Persistent Memory Module. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 2016 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Atlanta, GA, USA, 385–398. https://doi.org/10.1145/2872362.2872392

[21] Michael Körber, Nikolaus Glombiewski, and Bernhard Seeger. 2021. Index-Accelerated Pattern Matching in Event Stores. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. ACM, Virtual Event, China, 1023–1036. https://doi.org/10.1145/3448016.3457245

[22] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 2017 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 257–270.

[23] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Brisbane, Australia, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[25] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 2019 ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Huntsville, ON, Canada, 447–461. https://doi.org/10.1145/3341301.3359628

[26] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Brisbane, Australia, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[27] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 2014 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Seattle, WA, USA, 429–444.

[28] Witold Litwin. 1980. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases (VLDB)*. IEEE Computer Society, Montreal, Quebec, Canada, 212–223.

[29] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[30] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[31] Zhihan Lv, Xiaoming Li, Haibin Lv, and Wenqun Xiu. 2020. BIM Big Data Storage in WebVRGIS. *IEEE Transactions on Industrial Informatics* 16, 4 (2020), 2566–2573. https://doi.org/10.1109/TII.2019.2916689

[32] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *Proceedings of the 2021 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Virtual Event, 1–16.

[33] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 2012 European Conference on Computer Systems (EuroSys)*. ACM, Bern, Switzerland, 183–196. https://doi.org/10.1145/2168836.2168855

[34] Donald R. Morrison. 1968. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM* 15, 4 (1968), 514–534. https://doi.org/10.1145/321479.321481

[35] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 2019 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Boston, MA, 31–44.

[36] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177. https://doi.org/10.14778/3137628.3137629

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

[38] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[39] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the 2014 European Conference on Computer Systems (EuroSys)*. ACM, Amsterdam, The Netherlands, 15:1–15:15. https://doi.org/10.1145/2592798.2592814

[40] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development* 52, 4-5 (2008), 465–480. https://doi.org/10.1147/rd.524.0465

[41] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, Amsterdam, The Netherlands, 36–53. https://doi.org/10.1145/3299869.3300075

[42] Steve Scargall. 2020. Persistent Memory Architecture. In *Programming Persistent Memory*. Springer, Santa Clara, CA, USA, 11–30.

[43] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. ACM, Kohala Coast, Hawaii, USA, 61–72. http://www.adms-conf.org/2015/adms15_schwalb.pdf

[44] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[45] Spyros Blanas. 2020. From FLOPS to IOPS: The New Bottlenecks of Scientific Computing. https://www.sigarch.org/from-flops-to-iops-the-new-bottlenecks-of-scientific-computing/.

[46] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the 2013 ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Farmington, PA, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[47] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 2020 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Athens, Greece, 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[48] Wikimedia. 2022. Wikimedia Downloads. https://dumps.wikimedia.org/.

[49] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, USA, 349–362.

[50] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *Proceedings of the 2021 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Virtual Event, USA, 346–359. https://doi.org/10.1145/3445814.3446730

[51] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 2020 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 169–182.

[52] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 2015 USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, USA, 167–181.

[53] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Istanbul, Turkey, 33–44. https://doi.org/10.1145/2694344.2694387

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, San Jose, CA, USA, 15–28.

[55] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: A Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1187–1200.

[56] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, San Francisco, CA, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[57] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the 2021 European Conference on Computer Systems (EuroSys)*. ACM, Online Event, United Kingdom, 194–209. https://doi.org/10.1145/3447786.3456237

[58] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434. https://doi.org/10.14778/3372716.3372717

[59] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 2018 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, USA, 461–476.

Received July 2022; revised October 2022; accepted November 2022