Sorting on Byte-Addressable Storage: The Resurgence of Tree Structure

Ying Zheng
National University of Singapore
zheng@comp.nus.edu.sg

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

The tree structure is notably popular for storage and indexing; however, tree-based sorting such as tree sort is rarely used in practice. Nevertheless, with the advent of byte-addressable storage (BAS), the tree structure captures our attention with its write-once property. This property aligns well with BAS's asymmetric read-write characteristics. In this paper, we seek to answer the question: Can tree-based sorting algorithms outperform existing algorithms in the hybrid DRAM-BAS system? To address this, first, we conduct a comprehensive study to assess the compatibility of existing sorting algorithms with such hybrid memory systems and explore the challenges. We then delve into various design dimensions of tree-sort algorithms which leads to an optimized variant, *TSort*. Finally, a comparative analysis is conducted among three different sets of sorting algorithms, including in-place sorts, external sorts, and tree-based sorts. The results indicate that TSort not only challenges the traditional negative perceptions of the tree structure in sorting but also exhibits excellent performance. It outperforms all other counterparts across diverse datasets, whether uniformly distributed or skewed, in most cases.

PVLDB Reference Format:
Ying Zheng and Kian-Lee Tan. Sorting on Byte-Addressable Storage: The Resurgence of Tree Structure. PVLDB, 17(6): 1487-1500, 2024.
doi:10.14778/3648160.3648185

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/iamzhengying/sort.

1 INTRODUCTION

Ordering is often intricately linked to efficiency, indicating that the sorting algorithm plays an important role in diverse data-intensive applications [3, 14, 38, 41, 45, 53]. For example, in database management systems, sorting is the foundation of many query operations, such as ORDER BY, JOIN, DISTINCT, and GROUP BY. Therefore, a large amount of research has been devoted to designing efficient sorting algorithms.

Recently, byte-addressable storage (BAS) technologies [8, 15, 46, 51] have received lots of attention as potential alternatives to current storage media. They have impressive access performance compared to SSDs, and are more cost-effective and scalable than DRAM [18, 20, 49]. Given these attributes, the hybrid DRAM-BAS memory stands out as the most promising next-generation storage device [29, 47]. However, existing algorithms may not be able to adapt well to the following inherent characteristics of BAS to perform optimally:

- (A) Asymmetric read and write performance. BAS exhibits a more significant difference in read and write performance. Specifically, its sequential write is up to 4x worse than sequential read and even worse than random read [49]. Also, it suffers from limited write endurance [17]. Both result in writes being more expensive than reads.
- (B) Byte addressability. BAS supports byte-level small-size access, which makes its random access cheaper than that of traditional disks. Meanwhile, its random read performance is comparable to sequential read performance when the access size is larger than 256B [49]. Therefore, BAS is particularly tolerant of random read scenarios.
- (C) Constrained concurrency. BAS shows constrained performance on concurrent operations, especially for writes. With a non-optimal degree of parallelism, its write performance may drop to half of its peak [12, 37]. Its read, on the other hand, exhibits a more predictable performance, which initially improves as the degree of parallelism increases and then stabilizes after reaching a certain threshold. In short, BAS's maximal access bandwidth is determined jointly by access mode, access size, and the degree of parallelism.
- (N) Non-volatility. Although BAS's non-volatility ensures data persistence, it also implies that in-place processing is at risk of data corruption when a crash occurs.

Table 1 lists how well several existing sorting algorithms cope with the above characteristics. We add a fifth column (D) for DRAM to indicate the ability of the algorithm to leverage additional DRAM to enhance performance. For example, IPS^{4}o (in-place) [4, 5] is not write-efficient (no checkmark in column A), takes advantage of BAS's byte addressability (which allows it to run directly on BAS), is multithreaded (although it does not specifically focus on thread management), poses risk of data corruption (as it operates on the original dataset), and does not utilize main memory. On the other hand, B*Sort [30] is a write-limited scheme that leverages the BAS's property of byte addressability and is not affected by crashes (since the original dataset is read-only, and the sorted data is stored as a separate tree structure), but it does not utilize main memory. The table also distinguishes between two types of algorithms: value-based sorting schemes, which sort the entire record, and pointer-based schemes, which operate on (key, pointer)-pairs. The latter schemes require making a copy of (key, pointer)-pairs before sorting, and retrieving the value part to complete the sort. So, they naturally do not corrupt the original dataset in the event of a crash. Interestingly,

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 6 ISSN 2150-8097.
doi:10.14778/3648160.3648185

none of these schemes effectively address all five characteristics. Given the unique characteristics of BAS, there is a clear need to identify, if not develop, efficient BAS-friendly sorting algorithms.

Among all data structures, the tree structure catches our attention, although it is often overlooked by most sorting algorithm designers. Intrinsically, tree structures exhibit asymmetry between read and write operations during construction, with far more reads than writes. This inherent asymmetry aligns well with the unique read-write characteristics of BAS, which inspired us to explore the performance of tree-based sorting algorithms. The traditional and well-known tree-based sorting algorithm, tree sort [35, 36], does the sorting by constructing a binary search tree. It then traverses the constructed tree to obtain a globally ordered sequence. Despite its theoretically appealing $O(n \log n)$ best-case time complexity (where n is the number of data elements), tree sort has limitations in practice. Its frequent random accesses during construction result in a large number of cache misses. Additionally, it builds a very tall tree when handling large-scale datasets, which leads to significant traversal overhead. The multithreaded implementation of tree sort is also inefficient: the tree construction process typically involves mutex locks, which incurs substantial locking overhead and thread blocking. Furthermore, implementing an efficient multithreaded global tree traversal process is challenging due to ordering constraints. Finally, tree sort struggles with skewed datasets; when sorting such data, it may degenerate into a linked list.

Considering the above limitations, we decide to explore the potential of the tree-sort algorithm in the following design dimensions.

(1) Dimension #1: no batching (N) v.s. batching. The batching technique [10, 39] buffers a batch of data elements and inserts them into the tree together, which can be further classified as single-buffer batching (B) and double-buffer batching (D) [28].
(2) Dimension #2: global tree (G) v.s. multiple subtrees. Instead of building one global tree, the multitree scheme [30, 34] constructs a set of range-continuous trees and locates the target tree through hash-based (H) or binary-based (B) searches.
(3) Dimension #3: single thread (S) v.s. multiple threads. Because of BAS's property of constrained concurrency, we shall discuss both general multithreading (M') and augmented BAS-aware multithreading (M) [6, 54] implementations.
(4) Dimension #4: Synchronous manner (S) v.s. asynchronous manner (A). This dimension indicates whether threads are working together for a particular operation in the multithreaded implementation [23].

In our experiments, the variant with BHMA exhibits the best performance, which we denote as TSort. To evaluate the efficiency of TSort, we conduct a comprehensive experimental study among in-place sorting algorithms (including Introsort, IPS^{4}o, and selection sort), external sorts (such as traditional external merge sort [25] and WiscSort [6]), and tree-based sorts (like B*-sort and TSort). Our evaluation results reveal that TSort outperforms all other algorithms across both uniformly and skewed distributed datasets in most cases. From another perspective, this study revives the ever-forgotten tree structure in the field of sorting, opening up new design directions for sorting algorithms on BAS devices. In summary, the contributions of this paper include:

Table 1: Algorithm's compatibility to the ABCN(D) design principle. (Note: "✓" - the scheme supports the feature well, "◯" - there is room for improvement.)

Algorithm	A	B	C	N	D
Value-based sorting schemes					
IPS^{4}o (in-place) [4, 5]		◯	◯		
B*-Sort [30]	✓	◯		✓	
External merge sort [25]	✓		◯	✓	✓
segment sort [42]	◯	◯			
NVMSort [32]	◯	◯			✓
MONTRES-NVM [24]	✓	◯		✓	✓
NVMSorting [11]	✓	◯		✓	✓
Pointer-based sorting schemes					
IPS^{4}o (optimized)		✓	◯	✓	
PMSort [16]	✓	✓		✓	
WiscSort [6]	✓	✓	◯	✓	✓

- According to the characteristics of BAS devices, we comprehensively study and analyze the problems faced by existing conventional and BAS-friendly sorting algorithms.
- We provide the first thorough analysis of tree-sort algorithms on the BAS device across different design dimensions to explore its potential.
- We systematically conduct a rigorous empirical study among in-place sorting algorithms, external sorting algorithms, and tree-based sorting algorithms to identify their relative strengths and weaknesses in different scenarios.

The rest of this paper is organized as follows. In the next section, we provide background knowledge and motivations for this work. Then, in Section 3, we detail the design dimensions of tree-based schemes. Section 4 offers a comprehensive analysis of tree-sort algorithms with various design dimensions. Section 5 presents a thorough performance evaluation among three sets of sorting algorithms. Afterward, we review several existing BAS-related algorithms in Section 6. Finally, this paper is concluded in Section 7.

2 BACKGROUND AND MOTIVATION

In this section, we present an overview of in-place and external sorting algorithms, their BAS-based implementations, as well as the motivation for studying tree-sort algorithms.

2.1 In-place Sorting Algorithm

In-place sorting algorithms like Introsort, selection sort, and IPS^{4}o are widely used in main memory (DRAM). Introsort is a hybrid of quicksort, heapsort, and insertion sort. It begins with quicksort and switches to heapsort when the recursion depth exceeds a level based on the number of data elements being sorted, and uses insertion sort for small partitions for efficiency. Selection sort is commonly used for small datasets. It iteratively scans the unsorted part of the input sequence, selects the smallest element (assuming a non-descending sort), and places it in the correct position in the sorted part. IPS^{4}o is a state-of-the-art (parallel) sorting algorithm implemented based on

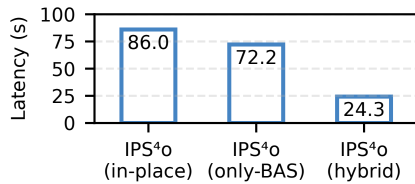


Figure 1: Performance of different IPS^{4}o implementations (400 million data elements; 1/4 relative DRAM capacity).

samplesort and optimized for modern hardware architectures. It integrates ordered sequence detection methods to avoid performance deterioration when dealing with skewed datasets.

The introduction of BAS with its byte addressability naturally raises the question of whether these in-place sorting algorithms can be directly migrated from DRAM to BAS. Unfortunately, such a straightforward migration brings significant challenges. When applied to BAS directly, traditional in-place algorithms exhibit poor performance. One of the primary reasons is the write amplification caused by redundant "values" in (key, value)-pairs. These values are irrelevant to the sorting process but introduce many additional writes. Although this phenomenon also exists in DRAM, its inefficiency is masked by DRAM's considerable write performance. However, BAS's relatively inefficient write performance makes this problem evident. To mitigate redundancy and optimize BAS access, we adopt pointer-based in-place sorting algorithms. Initially, we extract the keys and addresses (pointers) of the (key, value)-pairs from the dataset and store them on BAS. Subsequently, in-place sorting algorithms process only the generated (key, pointer)-pairs rather than the original (key, value)-pairs. Once the sorting is complete, a sequence of sorted (key, pointer)-pairs is obtained, from which the sorted (key, value)-pairs can be reconstructed by sequentially accessing the corresponding pointers. In addition to reducing redundancy, this approach also naturally addresses the issue of data corruption in the event of a system crash.

Taking IPS^{4}o as an example, Figure 1 illustrates a performance comparison among its different implementations on BAS. The in-place version implements sorting based on (key, value)-pairs, i.e., it is a straightforward migration of the traditional IPS^{4}o. The other two versions are based on the (key, pointer)-pair implementation. While the only-BAS version reconstructs the (key, value)-pairs directly on BAS as they are generated, the hybrid version uses DRAM to buffer (key, value)-pairs and write out a batch of sorted data elements each time. When sorting 400 million uniformly distributed data elements, the hybrid version performs about 3x faster than the only-BAS version and 3.4x faster than the in-place version. This is because the hybrid version not only reduces redundancy by using (key, pointer)-pairs but also avoids small-size write operations with the help of DRAM. Given the efficiency of the hybrid version implementation (utilizing both pointer-based mechanism and DRAM), all algorithm implementations in this paper adopt it by default.

2.2 External Sorting Algorithm

When dealing with datasets that exceed the capacity of main memory and are stored on secondary storage (e.g., SSD, HDD), the algorithm of choice is typically the external sort [25]. It efficiently manages large datasets by dividing them into multiple memory-sized runs. These runs are then loaded into memory, sorted using an in-place sorting algorithm, and written back to secondary storage. Once all runs are processed, these ordered runs are merged into a globally sorted sequence with the help of DRAM. Because of the similarity of the traditional DRAM-disk storage structure to the current hybrid DRAM-BAS structure, external sort can also be migrated directly. However, such direct migration leads to poor performance, mainly because it is not ABCN(D)-compliant in several aspects as shown in Table 1. In response to this challenge, the state-of-the-art BAS-friendly external merge sort, WiscSort, was proposed [6]. Compared to the traditional method, it introduces the pointer-based mechanism and multithreading management. By optimizing parallel implementation at the device level, WiscSort achieves a 2x improvement over the traditional external merge sort.

2.3 Tree structure

The tree structure is prominently utilized in the field of indexing [33, 43] and storage [9, 31]. There are also numerous tree-based algorithms designed specifically for BAS devices [22, 40, 50, 52]. However, it is seldom used for sorting, especially large-scale datasets, due to high maintenance costs. The state-of-the-art BAS-friendly tree-based sorting algorithm is B*-sort [30]. It is designed to mitigate performance degradation caused by data skew and improve the worst-case performance by optimizing the traversal process. This method theoretically and dynamically divides the binary search tree into multiple subtrees. It stores meta-information about these subtrees in a linked list. When inserting a data element, B*-sort sequentially scans the linked list to identify the target subtree, then directly traverses this subtree instead of the entire global tree. However, its scalability is problematic in practice. As the number of subtrees grows, scanning the linked list becomes expensive.

Despite the poor performance of B*-sort, the potential of tree-sort algorithms remains, especially on the hybrid DRAM-BAS structure. This optimism is rooted in the fact that tree-sort's asymmetric read-write operations perfectly match with BAS's asymmetric read-write performance. More specifically, tree construction involves a large number of random reads and a limited number of writes; fortunately, BAS tolerates random reads despite its poor write performance. Therefore, curiosity prompts us to explore the answers to the following two questions: What existing optimization techniques are crucial to BAS-friendly tree-sort algorithms? And is it possible for a tree-sort algorithm to outperform its counterparts? The details of the exploration are presented in subsequent sections.

3 TREE-BASED SORTING ALGORITHM

In this section, we explore the design dimensions of tree-based sorting algorithms, including batching, multitree schemes, multithreading techniques, and concurrent implementations. For clarity, we divide the tree-sort process into three stages: preparation, construction, and traversal. The main task of the preparation stage is to obtain meta-information about trees, during which both tree ranges and tree roots are determined. The construction stage is dedicated to building trees, where the original (key, value)-pairs are converted to (key, pointer)-pairs and globally ordered search trees are built on BAS. Lastly, in the traversal stage, a globally

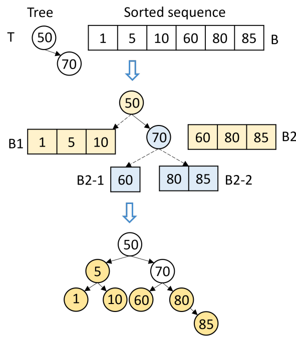


Figure 2: An example of batch insertion.

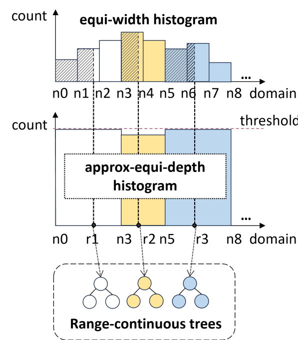


Figure 3: Hash-based multi-tree scheme.

ordered (key, value)-pair sequence is reconstructed by traversing trees and accessing associated pointers. More details about design dimensions are described below.

3.1 Batching

The first dimension concerns batching strategies [10, 39]: no batching (N), single-buffer batching (B) or double-buffer batching (D). These strategies primarily work in the construction stage. When no data elements are batched (N), the algorithm sequentially scans the unsorted data elements stored on BAS and then inserts them directly into the tree one at a time. During this process, no DRAM resources are needed. An alternative approach is to insert a batch of data elements each time. Here, a batch is first gathered into the DRAM buffer and then inserted into the tree together with only one tree traversal. To elaborate, when the DRAM buffer is full, the scheme first sorts the data elements inside using an in-place sorting algorithm, and then does batch insertion. During batch insertion, it compares the data elements in the buffer with the tree node of the target tree, and partitions the buffer into two sub-buffers; data elements of one sub-buffer are to be inserted into the left branch, and the elements of the other to the right. If one branch is empty, the scheme constructs a balanced subtree from the sub-buffer's data elements and integrates this subtree into the vacant branch. Otherwise, it repeats the above operations until an empty branch is found. The balanced subtree is constructed by iteratively selecting the median data element of the sub-buffer as the root node, with data elements on either side forming the left and right nodes.

Figure 2 gives an example demonstrating how batch insertion works. The tree root node 50 divides the sorted sequence in buffer B into two parts (B1 and B2). B1, whose data elements are smaller than node 50, goes to the left branch and B2 goes to the right. Since the left branch is empty, a small batch of data elements in B1 can be inserted directly. Specifically, these three data elements are coalesced to form a balanced subtree with node 5 as the root. Then this subtree is linked directly to the left branch of node 50 to complete the insertion of B1. For B2, because the right branch is not empty, it is further divided into B2-1 and B2-2 by node 70 and inserted into the tree in the same way as B1 when an empty branch is found. Once all data elements in buffer B are inserted into tree T, the process of batch insertion is completed.

Although the batching scheme incurs extra DRAM reads and writes, it reduces overhead on repeated accesses during multiple traversals. Moreover, it efficiently constructs a more balanced subtree for a given dataset, thus reducing the tree height and subsequent traversal cost. Importantly, this mechanism makes it possible to store some data elements, which are adjacent to each other in the tree, in consecutive memory addresses, thereby improving cache utilization in subsequent traversals. It achieves this as follows: whenever a subtree for a sub-buffer is to be inserted, its data elements are written to a contiguous memory space reserved on BAS that belongs to the tree. In addition, our investigation extends to both single- and double-buffering mechanisms [28]. Both work similarly (as described above), but the double-buffer batching approach facilitates parallelism - while data elements are being inserted from one buffer, the other can be loaded simultaneously. However, splitting the DRAM resources over two buffers means a smaller batch of elements is inserted each time and thus may incur higher traversal costs compared to the single-buffer batching scheme.

3.2 Multitree schemes

The second dimension considers the choice between employing a single global tree (G) or multiple range-continuous (sub)trees [30, 34]. The global tree scheme requires minimal or no time in the preparation stage but more time in the subsequent two stages for the taller tree. Conversely, the multitree scheme requires time for preparation and identifying the target tree but benefits from the shorter tree structure. Since it is computationally inefficient to create multiple range-continuous trees dynamically (as shown in B*-Sort), we opt for a simpler and more effective approach that predetermines the number of trees (n) before the sorting process. This approach raises two main issues: (1) How to determine the range of the (sub)trees in the preparation stage; (2) How to efficiently identify appropriate target trees during the construction stage. To address the above questions, we study two distinct multitree strategies: the binary-based (B) and the hash-based (H) schemes.

For the binary-based scheme (B), we use the systematic sampling method to extract a set of data elements from the dataset at fixed periodic intervals. Then, we sort the sampled data elements and construct a range-partitioning vector [13]. This vector splits the dataset into n (predetermined number) equally spaced and range-continuous parts. Each part corresponds to a tree, such that each tree is expected to hold approximately the same number of data elements. Finally, we can easily identify the target tree for the data elements by applying a binary search on the constructed range-partitioning vector.

To enhance the target tree search efficiency, we also examine a hash-based scheme as shown in Figure 3. In the preparation stage, we first define a large number ($>> n$) of equi-width buckets. Then, we count the number of data elements that fall into these buckets with a dataset scan and build an equi-width histogram. Afterward, we strategically combine adjacent buckets of the equi-width histogram and build an approx-equi-depth histogram. The guiding principle behind this process is to combine as many buckets as possible while ensuring that the number of data elements within each partition does not exceed a threshold (i.e., the average number of data elements per partition). Finally, we map each partition to a

tree. Once these trees are initialized, a hash-based mapping between data elements and trees is essentially established.

3.3 Multithreading techniques

The third dimension explores the thread-related models: single-threading (S), (conventional) multithreading (M'), and BAS-aware multithreading (M) [6, 54]. The single-threading (S) and multithreading (M') models are straightforward, with the former executing all three stages using a single thread and the latter employing all the threads throughout the entire process. BAS-aware multithreading (M) is a parallel implementation based on the features of BAS. BAS, unlike traditional storage devices, does not always benefit from maximizing thread utilization because of its constrained concurrency. Specifically, as the degree of parallelism increases, BAS's write performance drops significantly, sometimes to half of its peak [12, 37]. Its read performance, on the other hand, follows a more predictable trend, which initially improves and stabilizes after reaching a certain threshold. In addition, taking into account the device's internal prefetching and thread distribution mechanisms [54], BAS's maximal access bandwidth is determined jointly by the access mode, access size, and the degree of parallelism [44]. Therefore, it is non-trivial to effectively manage concurrent BAS-access operations.

Under a BAS-aware multithreading model (M) [6, 54], the thread manager allocates varying numbers of threads based on different access modes and access sizes. A preliminary study showed that a higher degree of parallelism is preferred for reads and a lower degree is optimal for writes; moreover, for large-size writes, the optimal degree of parallelism is lower than that for small-size writes. For example, inserting a (key, pointer)-pair to the tree is a small-size write. On the other hand, inserting a buffer of (key, value)-pairs is a large-size write. For the tree-sort algorithm, the final traversal stage involves large-size writes. Here, it needs to traverse the trees and retrieve the `value` component from the source dataset in order to produce the final (key, value)-pairs. The buffered large-size (key, value)-pairs are then written out to the BAS as the sorted output. In our experimental study where 32 threads are employed, all threads would be assigned for concurrent traversal reads but only 4 threads would be assigned for such large-size writes.

3.4 Concurrent implementations

The fourth dimension pertains to whether multithreading is implemented in a synchronous (S) or asynchronous (A) manner [23] during the construction stage. In a synchronous setup, all BAS-access threads move on to the subsequent tasks only after they have all successfully completed the current one. Any thread lagging behind causes other threads to be blocked. Specifically, the construction stage is divided into three substages, which are load, sort, and insert. In the load substage, all threads read from BAS, converting these data elements into (key, pointer)-pair format and storing them into the DRAM buffer. After all threads complete loading, the buffered data elements are sorted in parallel using an in-place sorting algorithm. In this way, data elements in the buffer that belong to the same tree are clustered together. The buffer is then (virtually) partitioned into multiple sub-buffers, each containing all data elements belonging to the same tree in this buffer. Afterward, each thread is assigned a set of sub-buffers and is responsible for inserting the data elements inside into the corresponding trees. All threads perform the batch insertion of their assigned data elements in parallel. Once all data elements in the entire buffer are fully inserted into the trees, the threads proceed to the next iteration, i.e., load new data elements into the buffer, etc. The process repeats until all data elements in the dataset are inserted into the trees.

In contrast, the asynchronous approach ensures that all BAS-access threads work independently. Once a thread completes its current task, it can seamlessly progress to the next one without waiting for others. To elaborate, each tree has its own sub-buffer in this approach. These sub-buffers are defined before the construction stage and hold the same size. From the view of a particular thread, this mechanism is implemented as follows: First, the thread reads a data element from BAS and places it into the appropriate DRAM sub-buffer. Then, it checks the status of the current sub-buffer. If the sub-buffer is full, it performs batch insertion for that sub-buffer. Otherwise, it does nothing. Once the above process is completed, this thread fetches the next data element from BAS and repeats the process until all assigned data elements are inserted into the trees. Although this approach avoids thread waiting between substages, the overhead caused by read-write interference is increased.

3.5 Algorithmic Space

Based on the various dimensions, a large number of tree-based sorting algorithms can be designed. Some of these are meaningless, e.g., if single-threading is used, then there is no need to consider the concurrent implementation dimension. We also expect schemes that are based on batching to be superior over those that process data elements one at a time; schemes that employ multiple subtrees should outperform those that build only a global tree; methods that use multithreading should in general perform better than single-threading models. In this paper, we implement 19 tree-sort variants - 3 "worst-case" variants that include one for no batching, one for global tree, and one for single threading; and the other 16 are combinations of the remaining 2 options from each dimension (after ignoring the "worst-case" options). According to the performance analysis in Section 4, the variant with the format "BHMA" stands out among all tree-sort implementations; we refer to it as *TSort*.

Here, we give an overview of how TSort works. (1) In the preparation stage, we scan the original file to obtain statistical information and build a hash mapping between the data elements and the trees on DRAM through approx-equi-depth histogram[1]. Here, all threads perform BAS-read operations concurrently for file scan (each thread reads different parts of the source file), and a subset of threads are assigned for the small-size BAS-write operations to create the tree roots on BAS[2]. (2) In the construction stage, first, we partition the entire DRAM buffer[3] into multiple sub-buffers to ensure that each tree has its own sub-buffer. All threads then independently construct the trees asynchronously[4] as described in Section 3.4. Once all threads complete their insertion tasks, a set of globally ordered trees is constructed on BAS. In this stage, multithreaded mixed BAS-read-and-write operations are needed, i.e., some threads perform BAS-read while others perform BAS-write

[1]TSort adopts hash-based multitree structure "H".
[2]TSort follows BAS-aware multithreading model "M".
[3]TSort takes a single-buffer batching scheme "B".
[4]TSort utilizes asynchronous multithread implementation "A".

Table 2: Default settings.

dataset size	200 million - 2000 million
data element	(1) 128B: 4B key + 124B value (2) 100B: 10B key + 90B value
relative DRAM space	1/25; 1/8; 1/4; 1/2
maximum # of threads	32
# of trees	0.025% of the dataset size
# of buckets	10x the number of trees

simultaneously. (3) In the traversal stage, we use the entire DRAM space as the write buffer. Moreover, multiple threads are able to traverse different trees together since we have global information about the trees from the preparation stage, such as the exact tree size and storage address. Specifically, these threads traverse the trees, retrieve the values, and store the (key, value)-pairs on DRAM. When the DRAM buffer is full, a large batch of sorted (key, value)-pairs is dumped into BAS. While the entire sorted (key, value)-pair sequence is generated on BAS, this algorithm ends. During this stage, tree traversal and value retrieval require multithreaded BAS-read operations, and final sorted (key, value)-pair generation needs multithreaded large-size BAS-write operations.

4 EVALUATION OF TREE-SORT VARIANTS

In this section, we provide a comprehensive performance analysis of the 19 tree-sort variants. Some default settings of the experimental evaluation are listed in Table 2.

4.1 Setup

Platform. The algorithms are implemented in C++, together with the Intel Persistent Memory Development Kit (PMDK) [19] on a real BAS device (Optane DCPMM). The performance evaluation is conducted on a server with Linux version 5.4.0-163 and two 2.90GHz Intel(R) Xeon(R) Gold 6326 CPUs. Each CPU has 16 physical cores, with a shared 24MB L3 cache. Each CPU core has a 48KB L1 data cache, a 32KB L1 instruction cache, and a 1280KB L2 cache. The overall Optane DIMM space for the system is 2048GB and the DRAM capacity is 256GB, both are large enough to store all data elements and meet the storage requirements for all algorithms in this paper.

Datasets. The datasets used consist of a set of fixed-size (key, value)-pairs. Each pair includes a 4B key (randomly generated integer) and a 124B value. To measure the performance across different scenarios, both uniform- and skew-distributed datasets are generated. The uniformly distributed dataset has a size ranging from 200 million to 2000 million (key, value)-pairs, and its data elements are randomly generated. The skewed distributed dataset exhibits varied patterns [5, 30], including 100% ordered, 50% ordered, 25% ordered, reversed ordered, interleaved ordered, and gathered with outliers. In addition, a sort benchmark from [2] is used to demonstrate that these algorithms also work for non-integer keys. The data elements are randomly distributed and consist of 10B keys and 90B values. In this section, we only focus on datasets comprising 400 million 128B (key, value)-pairs with uniform, interleaved ordered, and 50% ordered distributions. The comprehensive evaluations with other datasets will be detailed in Section 5.

Implementation. For implementation, there are four aspects to state. (1) All multithreading implementations are run in parallel with a maximum of 32 threads. For the tree-sort algorithm with BAS-aware multithreading implementation, it uses 16 threads for small-size BAS-write (in the preparation stage), 4 threads for large-size BAS-write (in the traversal stage), and 32 threads for all other multithreaded operations (across different stages). (2) All implementations adopt the (key, pointer)-pair mechanism during sorting, and store the final sorted file in a (key, value)-pair format on BAS. (3) To eliminate the influence of noise on the conclusion and disregard the warm-up and cool-down phases, all experiments are repeated 10 times, and only the middle stable 4 runs are averaged and reported. (4) To assess the algorithm's dependence on DRAM capacity, and to simulate scenarios where input data size is substantial (far exceeding our current experimental platform) while the DRAM resource is limited, all algorithms are performed on varied relative DRAM space in the ratios: 1/25, 1/8, 1/4, and 1/2. For example, a relative DRAM space of 1/4 for 400 million data elements means 1/4*(400000000)*16 = 1.6 GB DRAM space is allocated. The relative DRAM space is calculated based on the size of (key, pointer)-pairs.

4.2 Uniformly Distributed Datasets

4.2.1 Dimension #1: batching. In order to understand the performance gain in the first dimension, we first look into the performance evaluation among NHMA, BHMA, and DHMA, which share the same settings in other dimensions. As shown in Figure 4a, the variants equipped with buffer (BHMA and DHMA) present impressive performance enhancements compared to the no-batching implementation (NHMA), with an average improvement of 6.21x. The no-batching implementation (NHMA) has two primary drawbacks: (1) There are numerous repeated scans during node insertion. Even if two data elements are positioned closely in the tree, separate traversal rounds are required to insert them both. (2) The no-batching scheme fails to exploit cache locality efficiently. When successively processing data elements belonging to different trees, the no-batching scheme needs to swap around among these trees frequently to perform insertions, thus inducing heavy cache misses. However, with the help of batching, we see a significant reduction in superfluous BAS reads and an enhancement in cache utilization, both of which improve performance. Additionally, batching schemes manage to store (some of) the data elements that are neighbors in the tree in contiguous spaces on BAS, thus reducing cache misses in subsequent traversals.

In comparing the two batching schemes, the double-buffer batching implementation performs worse than the single-buffer batching. Taking DHMA and BHMA as an example, despite efforts to alleviate thread blocking, DHMA exhibits a 12% performance degradation. This phenomenon is especially pronounced with lower DRAM capacity as shown in Figure 4a. This is because the reduced buffer capacity leads to increased traversal overhead, which overshadows the benefits of exploiting parallelism.

4.2.2 Dimension #2: multitree. In Figure 4a, the performance comparison between BGMA and BHMA/BBMA illustrates the substantial advantage brought by the multitree scheme, with performance enhancements exceeding 200x. Constructing multiple range-continuous trees, instead of a single global tree, distinctly shortens

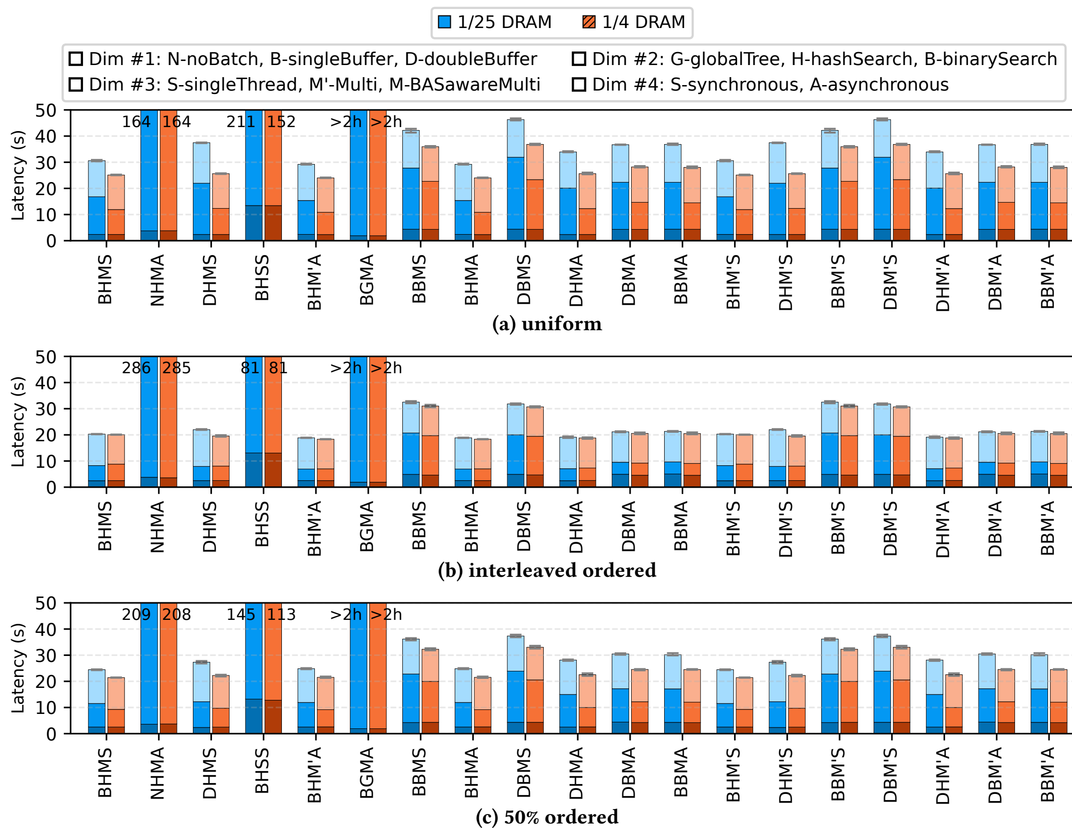


Figure 4: Performance of tree-sort variants when sorting 400 million data elements with different distributions. (Note: In each single bar, the colors changing from dark to light (from bottom to the top) represent the three different stages of tree-sort variants, which are preparation, construction, and traversal.)

the height of each individual tree, thus reducing the traversal overhead. Within the multitree scheme, exemplified by BHMA and BBMA, the hash-based scheme (BHMA) outperforms the binary-based scheme (BBMA) by 1.19x, and it benefits from both preparation and construction stages. In the preparation stage, it requires 1.68x less time than the binary-based scheme. In the construction stage, it locates the target tree with only constant time complexity for each data element, while the binary-based scheme requires a time complexity of $O(\log n)$, where n represents the number of trees. When dealing with large-scale datasets, the performance gap in the construction stage can be even larger.

Additionally, the hash-based scheme also manages to maintain more evenly balanced trees. Its average tree height is reduced by one level relative to the binary-based scheme. In detail, the hash-based scheme aims to initialize a set of dual-balanced trees, ensuring both inter-tree balanced and intra-tree balanced. In other words, all trees contain a similar number of data elements, and the left and right subtrees of each tree are similar in height. However, the binary-search-based scheme cannot guarantee the dual-balance property because of the lack of accurate global distribution information before construction.

4.2.3 Dimension #3: multithread. BHMA, BHM'A, and BHSS in Figure 4a demonstrate the performance behavior of different models in the third dimension. Since the single-threaded implementation is the same for synchronous and asynchronous manners (the fourth dimension), these three comparators share the same settings in other dimensions. When the sorting is executed with multiple threads (BHMA or BHM'A), the latency can be reduced by about 7x on average because of the more efficient CPU utilization. For the multithreaded implementation, since BAS suffers from concurrency constraints, both the conventional multithreading model (M') and BAS-aware multithreading model (M) are implemented and evaluated to demonstrate the necessity of special consideration for BAS. Taking BHMA and BHM'A as an example, the BAS-aware multithreading implementation improves the performance by 5% on average. This is because it considers more parallel access details, such as access mode and access size, and assigns different numbers of threads for different operations. Its strategic allocation ensures that both BAS-read and BAS-write operations are executed at their peak efficiency. Comparing two multithreading schemes, large-size BAS-write shows the largest difference in the number of threads among all BAS access operations. Since the traversal stage contains more large-size BAS-writes, these two multithreading models show a more significant performance difference in this stage.

Table 3: Four-factor ANOVA table.

Component	1/25		1/4	
	PoV	F	PoV	F
D1	16.26	5623.22	2.55	515.04
D2	52.63	18 201.49	64.43	13 030.96
D3	3.66	1265.89	6.58	1331.24
D4	16.58	5734.98	16.83	3404.61
D1:D2	6.04	2087.56	1.58	319.37
D1:D3	0.01	2.10	0.08	16.04
D2:D3	0.03	11.82	0.09	17.64
D1:D4	1.32	455.82	0.37	73.98
D2:D4	2.94	1016.90	6.12	1238.10
D3:D4	0.15	53.46	0.48	98.09
D1:D2:D3	0.00	0.07	0.04	7.25
D1:D2:D4	0.21	72.92	0.48	96.97
D1:D3:D4	0.01	3.87	0.00	0.13
D2:D3:D4	0.01	4.83	0.13	26.55
D1:D2:D3:D4	0.01	3.60	0.00	0.04
Residual	0.14		0.24	

4.2.4 Dimension #4: concurrency. As shown in Figure 4a, BHMA exhibits a 1-second performance improvement over BHMS. Although both asynchronous and synchronous methods experience thread blocking, they are blocked for different reasons. The asynchronous implementation uses mutex locks to manage threads and protect shared resources. Therefore, thread blocking happens when one thread tries to access a resource but another thread is accessing it. The synchronous implementation, however, does not need to consider consistency issues, but its thread blocking occurs between substages of the construction stage. It attempts to reduce concurrent BAS-read and BAS-write operations to minimize read-write interference, but the resulting performance improvement is not significant. Meanwhile, its higher thread-blocking overhead leads to poorer overall performance than the asynchronous scheme.

4.2.5 Discussion. To statistically evaluate the significance of various dimensions and their interactions, a four-factor Analysis of Variance (ANOVA) [21] is employed. The validity of this analysis is predicated on the normal distribution of variables and the homogeneity of variances. To satisfy these assumptions, log and power transformations are applied to the data before analysis. Table 3 displays the computed Percentage of Variation (PoV) and F-value (F) for each dimension and their interactions, excluding the worst-case levels (no batching, single global tree, and single threading), under two DRAM settings (1/25 and 1/4). In this table, "Dn" represents Dimension #n and "Dx:Dy" denotes the interaction effect between Dx and Dy. The reference F-value from the F-distribution is $F_{[0.99,1,48]} = 7.19$, indicating that computed F-values exceeding this threshold denote a statistically significant effect on performance at a 99% confidence level. For clarity, PoV is also provided, offering an informal representation of each dimension's and their interactions' relative importance in percentage terms.

Regarding main effects observed in the first four rows of Table 3, all dimensions demonstrate statistically significant effects on performance. The second dimension, related to the multitree scheme, exhibits the highest F-value, signifying a dominant effect. Notably, the first dimension, the batching scheme, has varying effects depending on the DRAM capacities, which meets the expectations. With limited DRAM, buffer capacity becomes a critical bottleneck, making the choice between single-buffer and double-buffer batching schemes crucial, as double-buffer batching exacerbates the scarcity of DRAM. Conversely, with larger DRAM, capacity ceases to be the primary constraint, diminishing the importance of the batching scheme. In terms of interaction effects, it is observed that the interactions between the first and second dimensions, as well as between the second and fourth dimensions, are more pronounced than others. However, certain combinations, such as D1:D2:D3:D4 and D1:D3:D4, do not exhibit statistically significant interaction effects, indicating there are no combination effects for them on performance.

Table 4: The average (maximum) tree height.

Distribution	1/25		1/4	
	BHMA	BHMS	BHMA	BHMS
Uniform	22 (33)	22 (31)	17 (22)	17 (21)
50% Ordered	64 (118)	16 (28)	22 (31)	13 (21)
Interleaved Ordered	107 (118)	12 (22)	27 (32)	12 (22)

4.3 Skewed Distributed Datasets

From Figures 4a, 4b and 4c, we observe that all tree-sort algorithms, except for NHMA, have lower running time for skewed datasets compared to the case where the dataset is uniformly distributed. The relative performance among all tree-sort algorithms is not much affected by the skewed inputs. NHMA's performance, as expected, degenerates as the trees become taller due to the skewed input. For all the other algorithms, which are essentially batch-based schemes, a consistent reduction in latency is demonstrated in both the construction and traversal stages compared to the uniformly distributed input. Clearly, batching allows more balanced subtrees to be built for each batch of data elements, and contributes to maintaining relatively balanced and shorter trees overall. This significantly reduces the traversal overhead. Moreover, the batching schemes store neighboring data elements, which are from the same tree, contiguously on BAS, thus increasing cache utilization during subsequent traversal processes. This advantage becomes particularly pronounced when (partially) ordered data elements are processed together, i.e., in a skewed distribution.

Another interesting phenomenon that occurs when handling skewed distributed datasets is captured in the tree height. The difference here mainly appears in the fourth dimension. Taking BHMA and BHMS as an example, their tree heights under different scenarios are shown in Table 4. For the asynchronous (A) implementation (e.g., BHMA), the tree height depends heavily on both the DRAM capacity and the distribution of input datasets. When dealing with a dataset with skewed distribution, the tree height can be about 3-5x larger than in the uniform case if only 1/25 DRAM space is provided. If larger DRAM space is given, like 1/4, the tree height issue is mitigated, but still worse than in the uniform case. However,

the synchronous (S) implementation (e.g., BHMS) performs well in terms of tree height. It even builds a lower tree in the skewed case than in the uniform one. This contrasts sharply with the behavior of traditional tree structures, where skewness can lead to severely unbalanced left and right branches. The main reason for the significant difference in tree height between the asynchronous and synchronous schemes is the batch size used during batch insertion.

In detail, let us assume that the total DRAM capacity is M and the number of trees is k. Since the asynchronous implementation divides the DRAM space into multiple parts to ensure each tree has its own buffer, the maximum batch size for this scheme would be M/k. The synchronous implementation, on the other hand, treats the entire DRAM space as a global buffer when loading data elements from BAS, and (virtually) partitions it only before batch insertion. So its maximum batch size can be M in theory. As mentioned before, the batching mechanism helps a lot in constructing balanced trees; and its strengths would be even more evident with larger batch sizes. Therefore, the synchronous implementation enables more balanced trees and thus shorter trees. Moreover, the synchronous implementation is more likely to obtain a larger batch size in skewed scenarios than in uniform ones. Therefore, the difference in batch size between these two implementations becomes more obvious when handling skewed distributed data elements. This explains why the tree height difference between them is pronounced in skewed scenarios, but not as significant in uniform cases.

However, although the synchronous (S) scheme holds shorter trees, it surprisingly fails to outperform the asynchronous (A) scheme. As we mentioned, the synchronous (S) scheme benefits from the larger batch size, which, however, might reduce the efficiency of multithreading as well. Because the smallest execution unit for multithreaded batch insertion is a tree or a batch. More specifically, consider an extreme case where all elements in the buffer belong to the same tree. In this case, though a more balanced tree is built, only one thread is working when inserting all data elements in this buffer, thus slowing down the sorting process.

5 COMPARATIVE STUDY

In this section, we conduct an experimental study among three sets of sorting algorithms: in-place sorts, external sorts, and tree sorts. All algorithms apply the (key, pointer)-pair mechanism and utilize DRAM as a write buffer. We reiterate that the amount of DRAM space available is given by the ratio with respect to the dataset size of (key, pointer)-pairs (and not (key, value)-pairs). All experimental settings are the same as the ones mentioned in Section 4.1.

5.1 Preliminary Evaluation

Prior to processing large datasets, we undertake a preliminary evaluation for each category of algorithms. After that, a more extensive series of experiments are conducted only on the representative algorithms. The preliminary experiment involves a dataset composed of 40 million 128B (key, value)-pairs. All algorithms are processed on the hybrid DRAM-BAS structure, with 1/4 relative DRAM capacity.

5.1.1 Latency. As shown in Figure 5, within the in-place algorithms, IPS^{4}o stands out with its robust performance across both uniformly and skewed distributed datasets. Selection sort, as the only write-limited in-place algorithm (in this paper), performs worst

Figure 5: Preliminary performance evaluation among three various sets of algorithms when sorting 40 million data elements with different distributions.

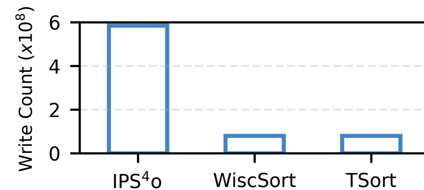
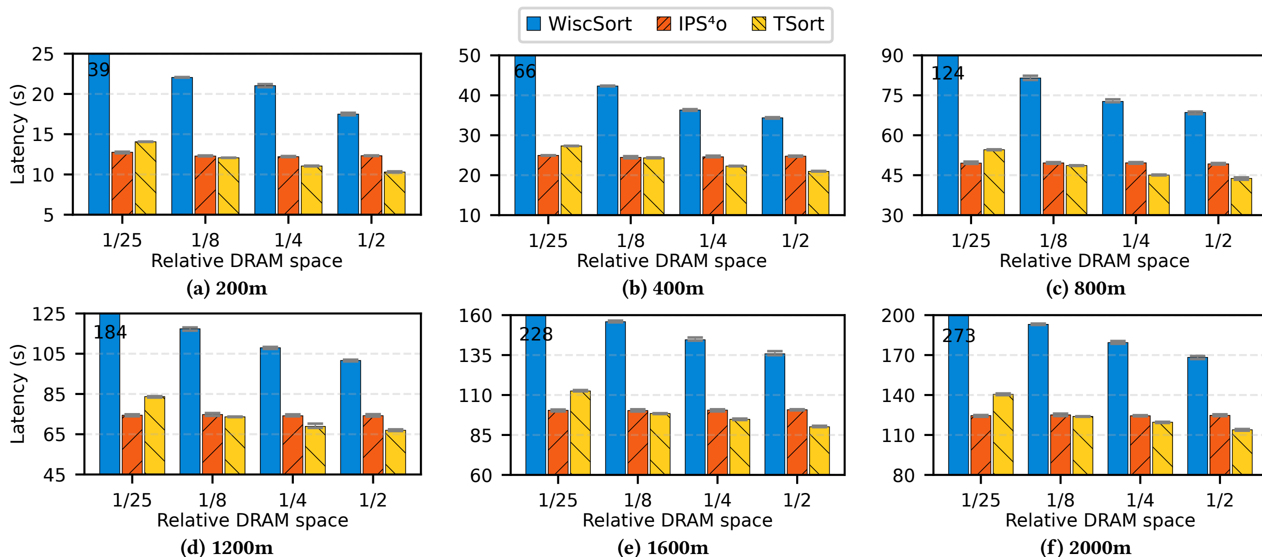


Figure 6: Write count of IPS^{4}o, WiscSort and TSort when sorting 40 million uniformly distributed data elements.

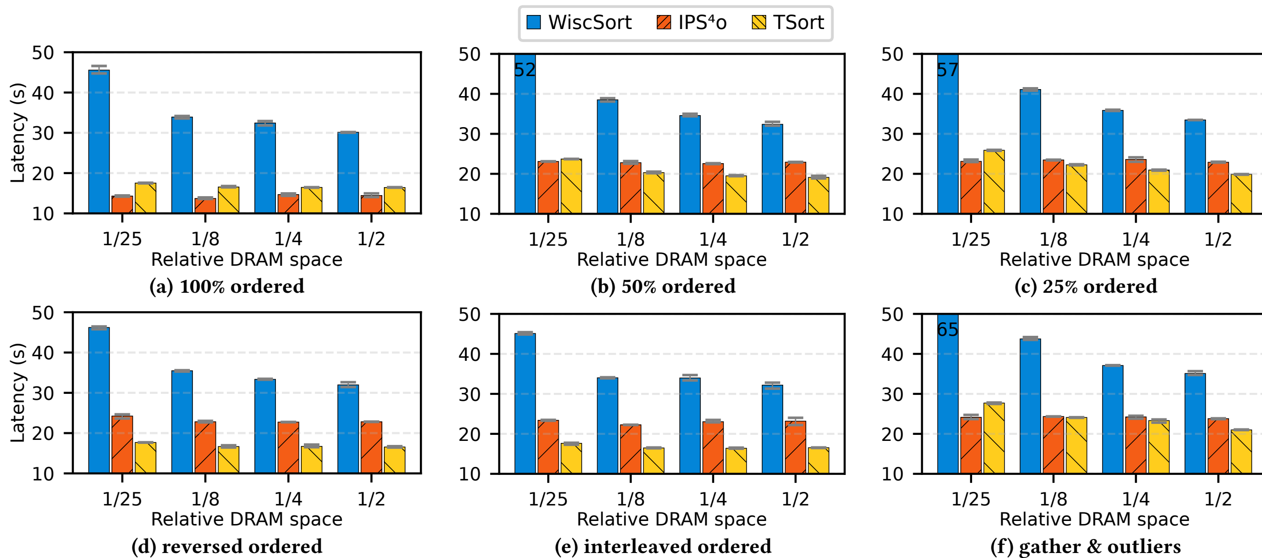
because of its $O(n^2)$ time complexity. Among external sort algorithms, WiscSort outperforms the traditional external merge sort because of its optimized BAS-aware thread management. For the tree-based algorithm category, TSort holds better performance, completing its sorting task in less than 3s. In stark contrast, B*-Sort fails to complete sorting even after an extensive 2-hour period.

5.1.2 Write count. BAS devices have two characteristics that are unfavorable to writes. First, their write bandwidth is significantly lower than the read bandwidth. Second, they have limited write endurance. As such, a BAS-friendly algorithm should not only be efficient but should also minimize the number of writes. As a result, write count is also a crucial metric to evaluate the performance of sorting algorithms on BAS. Figure 6 shows the write count for 3 representative algorithms of various sets, i.e., IPS^{4}o, WiscSort, and TSort. Notably, both external-based and tree-based algorithms exhibit exceptionally low write counts, with only two writes - one for (key, pointer)-pairs and another for sorted (key, value)-pairs. Therefore, they are friendly enough to BAS in terms of writing. Conversely, IPS^{4}o does not perform well on BAS regarding writes. Its write count is 7.3x higher than that of WiscSort and TSort. There is a trade-off associated with the in-place feature. Although the in-place scheme allows algorithms to maximize cache utilization, its frequent data element swapping incurs heavy write traffic, which is very unfriendly to BAS.



(a) 200m (b) 400m (c) 800m (d) 1200m (e) 1600m (f) 2000m

Figure 7: Performance of WiscSort, IPS^{4}o and TSort on uniformly distributed datasets with different workloads.



(a) 100% ordered (b) 50% ordered (c) 25% ordered (d) reversed ordered (e) interleaved ordered (f) gather & outliers

Figure 8: Performance of WiscSort, IPS^{4}o and TSort on datasets of size 400 million with different skewed distributions.

5.2 Uniformly Distributed Datasets

Based on the findings of the preliminary evaluation above, for this and all subsequent experiments, we restrict our comparative study to three representative schemes of the categories that perform best, i.e., IPS^{4}o representing in-place algorithms, WiscSort for external sort algorithms, and TSort for tree-based schemes. For our first set of comparative evaluation, we evaluate the three schemes on uniformly distributed datasets. The results are shown in Figure 7, illustrating the performance for processing different sizes of datasets ranging from 200 million to 2000 million. It can be observed that: (1) As the available DRAM capacity grows, the performance of both TSort and WiscSort improves, but TSort depends less on the DRAM resource. If we define the latency at 1/25 relative DRAM space as m, and the latency at 1/2 relative DRAM space as n, the DRAM dependency rate can be calculated by $|\frac{\frac{n}{m}-1}{\frac{1/2}{1/25}-1}|$. For all different workloads, the DRAM dependency rate of TSort is only 1/2 that of WiscSort. Furthermore, TSort outperforms WiscSort by up to 36%-55% on average for all different workloads and different relative DRAM capacities. Its superior performance and DRAM dependency rate are attributed to the tree structure and related techniques. (2) The performance of TSort and IPS^{4}o is comparable, and both are excellent. If more than 1/8 relative DRAM space (actually it is only less than 1.7% of the size of the original dataset, and

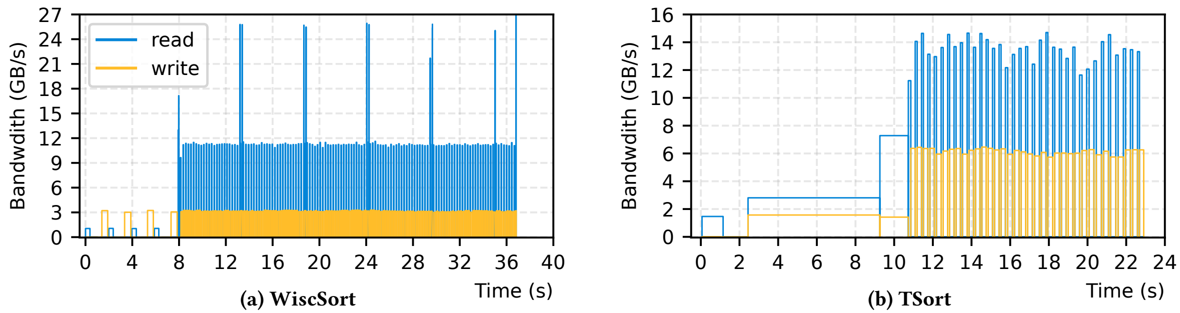


Figure 9: Comparison of BAS access bandwidth between WiscSort and TSort when sorting 400 million uniformly distributed data elements with 1/4 relative DRAM capacity provided.

ranges from 0.4GB to 4GB with different datasets) is provided, TSort outperforms IPS^{4}o. When 1/2 relative DRAM space is given (ranges from 1.6GB to 16GB), TSort exceeds IPS^{4}o by 12.6% on average. The fact that TSort is more write-efficient makes it an attractive BAS-friendly sorting scheme.

5.3 Skewed Distributed Datasets

Figure 8 depicts the performance of the three representative algorithms under various skewed distributed datasets, including (a) 100% sorted, (b) 50% sorted, (c) 25% sorted, (d) reversed sorted, (e) interleaved ordered, and (f) gathered with outliers. Each of these datasets comprises 400 million 128B (key, value)-pairs. Carefully comparing TSort's performance in Figure 8 with Figure 7b, we are surprised to find that TSort not only tolerates skewness but even thrives under it. This contrasts sharply with the behavior of the traditional tree structure, where skewness can lead to abysmal performance. The performance difference can be attributed to both the multitree scheme and batching mechanism, which together enable TSort to build trees with near-optimal balance and height and facilitate well the storage of data elements on BAS.

Delving into the first five subfigures of Figure 8 (i.e., 8a to 8e), which represent varying degrees of sorted subsequences, we can observe that: (1) TSort holds better performance in skewed cases than in uniform ones. Taking 8e and 7b as an example, the performance of TSort improves by 21% to 35% across different DRAM cases. The reason is the same as mentioned in Section 3.1. Storing data elements, which are neighbors in trees, together on contiguous addresses of BAS significantly reduces traversal cost because of better cache utilization, especially for (partially) ordered data elements. (2) Interestingly, the improvement of TSort's performance is more significant in scenarios with less DRAM space. Strictly speaking, it's not that TSort profits more from reduced DRAM but rather it saves more costs on random access. Although TSort benefits less from cache locality with smaller DRAM capacity, the ordered sequence from skewed datasets helps TSort compensate for this to some extent. In Figure 8, this compensation can obtain up to 35% performance improvement. (3) TSort yields more from datasets with more sorted subsequences. TSort mainly benefits from cache utilization when processing skewed datasets, so it is not sensitive to how the sequences are ordered by. It ensures performance gains as long as adjacent data elements are processed together. This trend is evident in Figures 8a, 8d, and 8e where TSort holds similar performance in 100%-ordered, reversed-ordered, and interleaved-ordered scenarios. And it gains more from the above scenarios than from the partially ordered ones that have shorter sorted subsequences as shown in Figures 8b and 8c. Moreover, it even outperforms IPS^{4}o with only 1/25 relative DRAM space in reversed-ordered and interleaved-ordered scenarios as shown in Figures 8d and 8e. (4) Although TSort gains a lot, IPS^{4}o gains more in 100%-ordered scenario. This is because IPS^{4}o benefits from skewed data in a different way than TSort. TSort gains from cache utilization, whereas IPS^{4}o profits from reduced data processing. It stops processing after detecting an ordered sequence. However, the reduced data processing doesn't give IPS^{4}o a large performance gain, which is only about 6%, as shown in Figures 8b-8e. This is because the performance bottleneck for IPS^{4}o is heavy BAS writes. However, in the 100%-ordered scenario (as shown in Figure 8a), the globally ordered sequence helps IPS^{4}o avoid frequent data element exchanges and reduces expensive BAS write operations, thereby improving performance by 42%. (5) WiscSort also gets a performance improvement in processing skewed data, and its improvement is between IPS^{4}o and TSort. Its improvement mainly comes from the in-place sorting process in the run generation stage and the comparison process in the merge stage. The performance gains in the former process are the same as IPS^{4}o, and in the latter process are through higher cache utilization.

Additionally, as depicted in Figure 8f, TSort is able to efficiently handle datasets where most data elements are gathered together but there are a few outliers. Its high tolerance to outliers is due to the reasonable control over the bucket interval. Such control ensures perfect histogram construction, allowing TSort to handle this specific data distribution well without compromising performance.

5.4 Bandwidth

Figure 9 compares the BAS access bandwidth between WiscSort and TSort. The measurements are captured while these two algorithms are sorting 400 million 128B (key, value)-pairs with a relative DRAM space of 1/4. It can be observed that although WiscSort can reach a higher peak access bandwidth, TSort holds a superior average bandwidth. During the traversal stage (corresponding to WiscSort's merge stage), TSort consistently reaches an impressive read bandwidth, nearing 14GB/s. In contrast, WiscSort hovers just below 12GB/s. Moreover, TSort displays a remarkable write bandwidth as

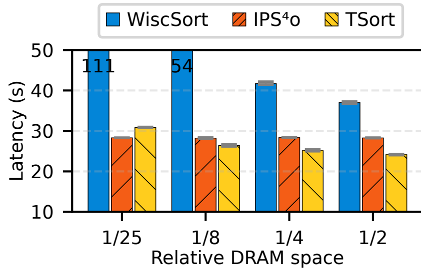


Figure 10: Performance of WiscSort, IPS^{4}o and TSort when sorting 400 million non-integer data elements.

well at the same stage, with performance approximately 2x that of WiscSort. It is worth noting that although TSort does not achieve high access bandwidth during the construction stage (from about 2.5s to 11s) because of small size accesses and read-write interference limitations, its performance is still comparable to that of WiscSort during the run generation stage (from about 0s to 8s). Taking similar times, TSort produces globally ordered sequences on BAS through the tree structure, while WiscSort generates only several locally ordered subsequences on BAS. Together with its bandwidth and DRAM space utilization advantages in the traversal stage, TSort ultimately provides better concurrency performance.

5.5 Non-integer Datasets

Figure 10 illustrates the performance of the three representative algorithms that process 400 million 100B (key, value)-pairs with non-integer keys. The relative performance among them is similar to that of processing integer keys. With all various sizes of DRAM, TSort outperforms WiscSort by 34%-72%. For very small relative DRAM capacity (e.g., 1/25), TSort performs slightly worse than IPS^{4}o since the latter's sorting process does not depend on DRAM. If a larger relative DRAM space is provided (e.g., 1/8, 1/4, or 1/2), TSort outperforms IPS^{4}o and the performance gain can exceed 14% when the relative DRAM capacity is 1/2.

5.6 Discussion and Future Work

Although Intel killed off the Optane DCPMM business in 2022 due to earning issues, research on it remains meaningful [18, 48]. There are two main reasons. First, the demand for new storage technologies to bridge the gap between DRAM and SSD is evident [6]. BAS technologies like Optane DCPMM and CXL-attached memory are promising solutions [26]. Hence, the exploration of BAS-friendly algorithms is of great importance. Second, the development of TSort is guided by the general characteristics of BAS devices and their hybrid structures, rather than being customized for a specific BAS device, like the Optane DCPMM. Future BAS iterations, including CXL-attached memory [1], are expected to inherit existing attributes such as byte addressability, asymmetric read-write performance, enhanced random-read performance, and more. Therefore, TSort would remain applicable to future BAS devices. Nevertheless, looking ahead, there are also several opportunities to further improve TSort. For instance, a more effective mapping approach might be utilized to optimize the hash process. In addition, more advanced tree structures, such as B$^+$-tree [7], could be integrated as well to construct a more robust multitree set.

6 RELATED WORK

Several sorting algorithms have been designed for BAS-related architectures in recent years, focusing on the I/O efficiency and performance optimization. Segment sort [42] is introduced alongside hybrid sort and lazy sort to minimize I/O overhead by achieving a balance between writes and reads. It divides the original file into two segments, which are sorted using read-less external mergesort and write-less selection sort, respectively. However, it degrades to pure external mergesort when running on practical BAS machines [16]. NVMSort [32] represents an enhanced iteration of traditional heapsort. It attempts to place frequently-accessed nodes, especially those proximal to the heap root, on DRAM, while others on BAS, to reduce the BAS access overhead. However, the real-machine implementation of NVMSort faces significant performance degradation due to huge cache misses. B*-Sort [30] stands out as a write-limited sorting algorithm based on a binary tree paradigm. It introduces a tunnel list structure that records some intermediate tree nodes to support start-from-middle (instead of root) traversal, thus shortening the traversal depth. Despite its innovative approach, scalability remains a challenge, primarily because the tunnel list structure necessitates a sequential scan for each insertion which is inefficient. NVMSorting [11], optimized for MONTRES [27] and MONTRES-NVM [24], adopts a different tactic. It aims to mitigate BAS access overhead by identifying and avoiding the repeated processing of self-ordered sub-sequences. PMSort [16] emerges as an adaptive sorting engine, combining various sorting approaches to ensure peak performance across different scenarios. WiscSort [6], meanwhile, refines the conventional external mergesort. It effectively manages threads to address the parallelism issue in the merge stage. However, these algorithms do not adhere to the ABCN(D) principle as mentioned in Table 1, thus struggling to perform well on real BAS devices with datasets of unknown distribution and when DRAM capacity is limited. In contrast, TSort skillfully handles the aforementioned issues and, as a result, exhibits excellent performance.

7 CONCLUSION

In this paper, we conducted an extensive analysis of tree-sort algorithms, particularly when operating on a hybrid DRAM-BAS structure. The results reveal that TSort emerged as the winner among all tree-sort variants, while in most cases it surpasses a set of other algorithms as well. This achievement responds positively to the question we posed at the outset of this paper. The tree algorithm indeed has the potential to outperform other widely used sorting methods. As an emerging BAS-friendly sorting algorithm, TSort challenges long-standing biases against the tree structure in sorting tasks, reinvigorating its pivotal role and potential in the field of sorting.

ACKNOWLEDGMENTS

This work was supported by a grant funded by the Singapore Ministry of Education (Title: inPMdb: An in-Persistent Memory Database System; WBS No: A8000082-00-00). We also thank the anonymous reviewers for their valuable comments and suggestions.

REFERENCES

- [1] 2022. Samsung Shows Off CXL Server Memory Expander. https://www.nextplatform.com/2022/08/23/samsung-shows-off-cxl-server-memory-expander/ [Last accessed on June 24, 2023].
- [2] 2023. Sort Benchmark. http://sortbenchmark.org/ [Last accessed on June 24, 2023].
- [3] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma, and Adrián Soto. 2021. Worst-Case Optimal Graph Joins in Almost No Space. In *Proceedings of the 2021 International Conference on Management of Data*. 102–114.
- [4] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms*, Vol. 87. 9:1–9:14.
- [5] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2022. Engineering In-Place (Shared-Memory) Sorting Algorithms. *ACM Trans. Parallel Comput.* 9, 1, Article 2 (2022), 62 pages.
- [6] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2023. WiscSort: External Sorting for Byte-Addressable Storage. In *Proceedings of the VLDB Endowment*, Vol. 16. 2103–2116.
- [7] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indices. *Acta Informatica* 1 (1972), 173–189.
- [8] E. Chen, D. Lottis, A. Driskill-Smith, D. Druist, V. Nikitin, S. Watts, X. Tang, and D. Apalkov. 2010. Non-volatile spin-transfer torque RAM (STT-RAM). In *68th Device Research Conference*. 249–252.
- [9] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies*. 17–32.
- [10] Li Chen, Rupesh Choubey, and Elke A Rundensteiner. 2002. Merging R-trees: Efficient strategies for local bulk insertion. *GeoInformatica* 6 (2002), 7–34.
- [11] Zhaole Chu, Yongping Luo, Peiquan Jin, and Shouhong Wan. 2021. NVMSorting: Efficient Sorting on Non-Volatile Memory. In *The 33rd International Conference on Software Engineering & Knowledge Engineering*.
- [12] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event China, 339–351.
- [13] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. 280–291.
- [14] Sahin Cem Geyik, Stuart Ambler, and Krishnaram Kenthapadi. 2019. Fairness-Aware Ranking in Search & Recommendation Systems with Application to LinkedIn Talent Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2221–2231.
- [15] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.
- [16] Yifan Hua, Kaixin Huang, Shengan Zheng, and Linpeng Huang. 2021. PMSort: An adaptive sorting engine for persistent memory. *Journal of Systems Architecture* 120 (2021).
- [17] Kaixin Huang, Yijie Mei, and Linpeng Huang. 2020. Quail: Using NVM write monitor to enable transparent wear-leveling. *Journal of Systems Architecture* 102 (2020), 101658.
- [18] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. In *Proceedings of the VLDB Endowment*, Vol. 16. 1249–1263.
- [19] Intel. 2023. Persistent memory development kit. (2023). https://pmem.io/pmdk/ [Last accessed on June 24, 2023].
- [20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).
- [21] Raj Jain. 1991. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Vol. 1.
- [22] Saeed Kargar and Faisal Nawab. 2023. Hamming Tree: The Case for Energy-Aware Indexing for NVMs. *Proceedings of the 2023 international conference on Management of data* 1, 2, Article 182, 27 pages.
- [23] Krishna M Kavi, Ben Lee, and Ali R Hirson. 1998. Multithreaded systems. In *Advances in Computers*. Vol. 46. 287–328.
- [24] Mohammed Bey Ahmed Khernache, Arezki Laga, and Jalil Boukhobza. 2018. MONTRES-NVM: An external sorting algorithm for hybrid memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 49–54.
- [25] Donald E Knuth. 1998. The art of computer programming, Volume 3: Sorting and Searching. *Addison-Westley* (1998).
- [26] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. 2023. NVM: Is it Not Very Meaningful for Databases?. In *Proceedings of the VLDB Endowment*, Vol. 16. 2444–2457.
- [27] Arezki Laga, Jalil Boukhobza, Frank Singhoff, and Michel Koskas. 2017. Montres: merge on-the-run external sorting algorithm for large data volumes on ssd based storage systems. *IEEE Trans. Comput.* 66, 10 (2017), 1689–1702.
- [28] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment* 9, 14, 1647–1658.
- [29] Yong Li, Lingfang Zeng, Guang Chen, Chunhua Gu, Fei Luo, Weichao Ding, Zhan Shi, and Joel Fuentes. 2022. A Multi-hashing Index for hybrid DRAM-NVM memory systems. *Journal of Systems Architecture* 128 (2022), 102547.
- [30] Yu-Pei Liang, Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Hsin-Wen Wei, and Wei-Kuan Shih. 2020. B*-Sort: Enabling write-once sorting for nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4549–4562.
- [31] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 553–571.
- [32] Yongping Luo, Zhaole Chu, Peiquan Jin, and Shouhong Wan. 2020. Efficient sorting and join on NVM-based hybrid memory. In *International Conference on Algorithms and Architectures for Parallel Processing*. 15–30.
- [33] Yongping Luo, Peiquan Jin, Qinglin Zhang, and Bin Cheng. 2021. TLBtree: A Read/Write-Optimized Tree Index for Non-Volatile Memory. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1889–1894.
- [34] Yannis Manolopoulos, Yannis Theodoridis, and Vassilis J Tsotras. 2000. Parallel External Sorting. *Advanced Database Indexing* (2000), 209–218.
- [35] W. A. Martin and D. N. Ness. 1972. Optimizing Binary Trees Grown with a Sorting Algorithm. *Commun. ACM* 15, 2 (1972), 88–93.
- [36] Keith McLuckie and Angus Barber. 1986. Binary Tree Sort. *Sorting Routines for Microcomputers* (1986), 58–67.
- [37] Jinyoung Oh and Youngjin Kwon. 2021. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*. 97–105.
- [38] Evaggelia Pitoura, Kostas Stefanidis, and Georgia Koutrika. 2022. Fairness in rankings and recommendations: an overview. *The VLDB Journal* 31 (2022), 431–458.
- [39] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen. 1996. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering* 8, 6 (1996), 975–984.
- [40] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies*. 69–82.
- [41] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
- [42] Stratis D Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment* 7, 5 (2014), 413–424.
- [43] 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. *Proceedings of the 2023 international conference on Management of data* 1, 1, Article 105, 26 pages.
- [44] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508.
- [45] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: sextuple indexing for semantic web data management. In *Proceedings of the VLDB Endowment*, Vol. 1. 1008–1019.
- [46] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.
- [47] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems.. In *USENIX Annual Technical Conference*. 349–362.
- [48] Minhui Xie, Youyou Lu, Qing Wang, Yangyang Feng, Jiaqiang Liu, Kai Ren, and Jiwu Shu. 2023. PetPS: Supporting Huge Embedding Models with Persistent Memory. In *Proceedings of the VLDB Endowment*, Vol. 16. 1013–1022.
- [49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies*. 169–182.
- [50] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-in-Place Key-Value Store for Modern Storage. *Proceedings of the VLDB Endowment* 16, 1, 99–112.
- [51] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. 2020. Resistive Random Access Memory (RRAM): an Overview of Materials, Switching

Mechanism, Performance, Multilevel Cell (mlc) Storage, Modeling, and Applications. *Nanoscale Research Letters* 15, 1 (2020), 90.

[52] 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, 1187–1200.

[53] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. 2001. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 international conference on Management of data*. 425–436.

[54] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 179–193.