Efficient Sorting and Join on NVM-Based Hybrid Memory

Yongping Luo[1], Zhaole Chu[1], Peiquan Jin[1,2](✉), and Shouhong Wan[1,2]

[1] University of Science and Technology of China, Hefei 230027, Anhui, China
jpq@ustc.edu.cn

[2] Key Laboratory of Electromagnetic Space Information, China Academy of Science, Hefei 230027, Anhui, China

Abstract. Non-volatile memory (NVM) as a new kind of future memories has a number of special properties such as non-volatility, read/write asymmetry, and byte addressability. This makes it difficult to directly replace DRAM with NVM in current memory hierarchy. Thus, a practical way is to construct hybrid memory composed of both NVM and DRAM. Such hybrid memory architecture introduces many new challenges for existing algorithms. In this paper, we focus on improving sorting and join algorithms for DRAM-NVM-based hybrid memory. In particular, we start with a theoretical study on the data placement issue in DRAM-NVM-based hybrid memory systems and propose an optimal data placement model to store data structures in DRAM and NVM during the sorting process. We present the theoretical proof to the optimal data placement model to ensure the correctness of the model. Further, based on the optimal data placement model, we propose a new NVM-aware sorting algorithm named NVMSort that adopts heap structures to accelerate the sorting process. Compared with traditional sorting algorithms, NVMSort is write-friendly and more efficient on DRAM-NVM-based hybrid memory. We further apply NVMSort into the traditional merge-sort join algorithm to optimize merge-sort join on DRAM-NVM-based hybrid memory. We conduct comparative experiments with existing sorting algorithms including HeapSort and QuickSort. The results show that NVMSort is much faster than the classical Heapsort and QuickSort. In addition, NVMSort is more NVM-friendly as it can reduce more NVM writes. When integrated into the traditional merge-sort join algorithm, NVMSort also achieves the best performance.

Keywords: Non-volatile memory · Sorting · Data placement model · Sort join

1 Introduction

The performance of traditional computer systems is highly limited by the high latency between DRAM and disks. This has been widely regarded as the "storage wall" problem [1]. Although building in-memory systems [2] seems to be a possible solution to the storage-wall problem, the volatility of DRAM makes it difficult to tackle data consistency and durability. Recently, the advance in non-volatile memory (NVM) [3] brings new opportunities to build NVM-based in-memory systems that can not only support in-memory data processing but also ensure data consistency and durability. On the other hand, traditional algorithms such as sorting algorithms and database join schemes, which are designed either for disks or for DRAM, need to be revisited to suit for the special properties of NVM.

NVM has some special properties [4–6]. First, differing from DRAM, it is non-volatile, meaning that all data written into NVM will not be lost when the host computer is shut down. Second, differing from magnetic disks or solid-state drives (SSD) [7] that only support block-based data accesses, NVM is byte addressable, which is similar to DRAM. Third, NVM has a high level of density, which is comparable to SSD and higher than NVM. To this end, NVM has the advantages of both disks and DRAM. Moreover, NVM is more like a new kind of memory, but not a new type of disk. However, NVM also has some limitations compared to DRAM and disks. Firstly, the read and write latencies of NVM are not balanced. Particularly, NVM has the similar read latency as DRAM, but its write latency is higher than that of DRAM. In addition, the endurance of NVM is limited, meaning that after a certain number of writes (~10^8 at present), NVM will become unstable. Thus, algorithms running on NVM have to be write-friendly.

Due to the special features of NVM, existing in-memory algorithms such as sorting algorithms must be re-designed to make them efficient on NVM. However, it is not a trivial task. There are a few challenges that need to be carefully considered. First, as NVM has a lower write speed than DRAM, currently it is more reasonable to construct a hybrid memory system composed of DRAM and NVM. In this hybrid-memory environment, how to optimally place data in DRAM and NVM is a new and challenging issue. Second, traditional external sorting algorithms like Merge Sort [8] are designed toward reducing I/O operations on block-based disks or SSDs and are not suitable for byte-addressable NVM. Traditional in-memory sorting algorithms [8] like Quick Sort or Heap Sort do not consider the high-write-latency of NVM as well as the reduction of NVM writes for endurance. Thus, they cannot be applied to NVM. Here, the challenge is that reducing NVM writes may lower the sorting performance of the algorithm. Therefore, we need to devise new sorting schemes that are not only time efficient, but also write friendly.

In this paper, we focus on re-designing sorting algorithms on NVM and further improving the sort join algorithm in database systems. We aim to devise a new sorting algorithm that is not only time-efficient but also NVM friendly. Specially, we start with a theoretical study on the data placement issue in DRAM-NVM-based hybrid memory systems and propose an optimal data placement model to store data structures in DRAM and NVM during the running of an algorithm. We present the theoretical proof to the optimal data placement model to ensure the correctness of the model. Further, based on the optimal data placement model, we propose a new sorting algorithm on NVM as well as a new sort join algorithm. In summary, we make the following contributions in this paper:

(1) We study the data placement issue on DRAM-NVM-based hybrid memory systems and present an optimal data placement model for algorithms running on hybrid memory. We theoretically prove that the proposed model can offer the best data placement during the execution of algorithms.

(2) We present a new sorting algorithm named NVMSort that is optimized for DRAM-NVM-based hybrid memory. NVMSort is based on the optimal data placement model and adopts heap structures to accelerate the sorting process. Compared with traditional sorting algorithms, NVMSort is more efficient on DRAM-NVM-based hybrid memory. In addition, NVMSort can reduce more NVM writes. We further integrate the NVMSort algorithm into the traditional merge-sort join algorithm to accelerate the join process on the hybrid memory.
(3) We conduct extensive experiments to verify the performance of NVMSort and NVMSort-based join algorithm. Compared with QuickSort and HeapSort, the proposed NVMSort has the best sorting performance. In addition, its NVM writes are much fewer than that of QuickSort and HeapSort. Overall, NVMSort reaches a better trade-off between time performance and NVM writes. When integrated into the merge-sort join algorithm, NVMSort also shows better time performance and is more NVM-friendly than its competitors.

2 Related Work

Recently, the big data concept leads to a special focus on the use of main memory. However, the increasing capacity of main memory introduces many problems, such as increasing of total costs and energy consumption [9]. Both academia and industries are looking for new greener memory media. Emerging NVM technologies, such as Phase Change Memory (PCM) and Resistive Memory (ReRAM), can provide faster persistence than traditional disks and flash memory. NVMs can provide similar read latency but higher write latency than DRAM. Like flash memory, the write endurance of NVM is limited. Thus, reducing write operations to NVM is critical for software system design [4–6].

NVM can provide better support for data durability than DRAM does. Further, it differs from other media such as flash memory in that it supports byte addressability. However, NVM has some limitations [3, 4], e.g., high write latency, limited lifecycle, slower access speed than DRAM, etc. Therefore, it is not a feasible design to completely replace DRAM with NVM in current computer architectures. A more exciting idea is to use both NVM and DRAM to construct hybrid memory systems, so that we can utilize the advantages from both media [10, 11]. NVM has the advantages of low energy consumption and high density, and DRAM can afford nearly unlimited writes. Specially, NVM can be used to expand the capacity of main memory, whereas DRAM can be used as a buffer for NVM. Presently, both the architectures are hot topics in academia and industries. Many issues need to be further explored. The biggest challenge for DRAM-NVM-based hybrid memory systems is that we have to cope with heterogeneous memories. In this paper, we also focus on the architecture of hybrid memory systems.

A few prior works [12, 13] have explored algorithms for asymmetric read-write costs in emerging NVMs within the context of databases. Chen et al. [12] presented analytical formulas for PCM latency and energy, as well as algorithms for B-trees and hash joins that are tuned for PCM. For example, their B-tree variant does not sort the keys in a leaf node nor repack a leaf after a deleted key, thereby avoiding the write cost of sorting and repacking, at the expense of additional reads when searching. Similarly, Viglas [13] traded off fewer writes for additional reads by rebalancing a B+ -tree only if the cost of rebalancing has been amortized.

To the best of our knowledge, few studies have been focused on the improvement of fundamental sorting and joint algorithms on NVM. In a word, there are only two previous works that are related to this study. The first study [14] presented a write-limited sorting algorithm but it was toward in-memory sorting. On the contrary, this study is toward traditional disk-based sorting and join, i.e., the involved relations are initially stored in disks. The second work [15] proposed a cost model for sorting on storage devices with asymmetric read and write latencies. However, these related works are both towards page-based storage devices, such as flash-memory-based SSDs. Although flash memory also has limited write endurance and low write latency, it is much different from NVM, because NVM can be used as main memory while flash memory can only be used as secondary storage.

3 Data Placement Model on Hybrid Memory

In this section, we study the data placement issue on DRAM-NVM-based hybrid memory. This issue is raised because of the existence of the two heterogeneous memories in hybrid memory systems. Thus, when a sorting algorithm is running, we have to decide where any intermediate data should be placed. Below we first introduce the concepts and problem definitions of the data placement issue. Then, we present the optimal data placement model as well as its proof for hybrid memory.

3.1 Basic Concepts

Logical Data Structures (DS) need to be placed on physical memory space. In a hybrid memory system, the memory space of a data structure can be allocated completely from DRAM, completely from NVM or partially from the two memory devices. A *data placement model* (DPM) cares only about how the physical memory allocation of data structures affects the total memory read and write cost. If not mentioned specially, all the memory units we refer to afterwards represent one cacheline, which is the unit of one memory reference, typically 64 bytes in current computer systems. The read and write time to a unit, as well as other involved symbols, are summarized in Table 1.

If one algorithm uses t data structures, which can be referred as $\Phi = \{DS_i | i = 1, 2, \ldots t\}$. For any data structure DS_i, we refer the read times of each unit of DS_i to be m_i and write times to be n_i. When the unit is allocated on DRAM, the total read and write cost (memory cost) of that unit is $C_i^d = m_i r^d + n_i w^d$; when the unit is allocated on NVM, the memory cost is $C_i^p = m_i r^p + n_i w^p$. If we swap the unit from NVM to DRAM, the memory cost of that unit will decrease by $C_i^p - C_i^d$, we name it the cost gain C_i^g of the unit, represented by (1).

$$C_i^g = C_i^p - C_i^d \tag{1}$$

Moreover, we define the memory allocation scheme of Φ in an algorithm as a data placement scheme. In a data placement scheme, if DS_i occupies B_i^d DRAM units and B_i^p

Table 1. Symbols in DPM

Symbol	Description
r^d	Latency of read a DRAM unit
w^d	Latency of write a DRAM unit
C^d	Memory cost of a unit if placed in DRAM
r^p	Latency of read an NVM unit
w^p	Latency of write an NVM unit
C^p	Memory cost of a unit if placed in NVM
C^g	Memory cost gain when swap unit from NVM to DRAM

NVM units, the memory cost C_i of DS_i is then $C_i^d B_i^d + C_i^p B_i^p$. Hence, the total memory cost C is calculated by (2)

$$C = \sum\nolimits_{i=1}^{t} \left(C_i^d B_i^d + C_i^p B_i^p \right) \tag{2}$$

3.2 Optimal Data Placement Model

Theorem 1. For $\forall x, y \in 1, 2, \ldots t$ that holds $C_x^g > C_y^g$, if the DRAM space of a hybrid memory system cannot accommodate both DS_x and DS_y, then the DRAM space must be allocated to DS_x preferentially. ■

Proof. Given that the total memory cost in the data placement scheme is minimal, we assume that although $C_x^g > C_y^g$ and DRAM is not large enough to accommodate both DS_x and DS_y, DRAM is needless to allocate to DS_x preferentially. Thus, there must be some DRAM spaces that are allocated to DS_y rather than DS_x, i.e. $B_x^p > 0$ and $B_y^d > 0$. Without loss of generality, we let $B_x^p > B_y^d$. Then, if we swap B_y^d units DS_y in DRAM with B_y^d units DS_x in NVM, we will get a new data placement scheme. Let C' be the new memory cost under this scheme, we have the following result.

$$\begin{aligned} C' = & \sum_{i=1,\ldots t \text{ and } i \neq x,y} \left(C_i^d B_i^d + C_i^p B_I^p \right) + C_x^d \left(B_x^d + B_y^d \right) + C_x^p \left(B_x^p - B_y^d \right) + C_y^d \left(B_y^d - B_y^d \right) \\ & + C_y^p \left(B_y^p + B_y^d \right) \\ & = \sum\nolimits_{i=1,\ldots t} \left(C_i^d B_i^d + C_i^p B_I^p \right) + B_y^d \left(C_x^d - C_x^p + C_y^p - C_y^d \right) = C + B_y^d \left(C_y^g - C_x^g \right). \end{aligned} \tag{3}$$

Since $B_y^d > 0$ and $C_x^g > C_y^g$, we can derive that $B_y^d \left(C_y^g - C_x^g \right) < 0$, resulting in $C^{'} < C$. This is in contrast with the above assumption that C is the minimal memory cost. So, we can conclude that DRAM space must be preferentially allocated to DS_x when $C_x^g > C_y^g$. ■

Theorem 2. Suppose that under a data placement scheme, the maximal memory cost is denoted as C_{max}. If $\forall x, y \in 1, 2, \ldots t$ that holds $C_x^g > C_y^g$ and the DRAM space cannot accommodate both DS_x and DS_y, the DRAM space must be preferentially allocated to DS_y. ■

The proof of Theorem 2 is similar to that of Theorem 1.

We name the data placement scheme with the minimum memory cost in Theorem 1 as the *optimal data placement scheme*. Accordingly, the scheme defined by Theorem 2 is regarded as the *worst data placement scheme*. According to the two theorems, the *optimal data placement scheme* allocates DRAM space to data structure with larger C^g value with higher priority, while the *worst data placement scheme* exactly does the opposite.

4 Sorting and Join with the Optimal Data Placement Model

The optimal data placement model is helpful to improve sorting and join algorithms. In this section, we present the new sorting and join algorithms that are based on the optimal data placement model. We first propose the NVMSort algorithm in Sect. 4.1, and then explore the NVMSort-based sort join algorithm in Sect. 4.2.

4.1 NVMSort

The sorting problem we consider is the standard comparison-based sorting with n records, each of which contains a key. We assume that the input is an unsorted array, and the output to be a sorted array which is sorted on their keys.

Based on the optimal data placement scheme, we propose an efficient sorting algorithm on hybrid memory systems. This algorithm is motivated by the read/write property of the classical HeapSort algorithm [8]. As Fig. 1 shows, during the execution of HeapSort, most writes are focused on a few portions of heap nodes. The following part will explain why heap structure has this read/write property and how we can leverage it to reduce the NVM writes.

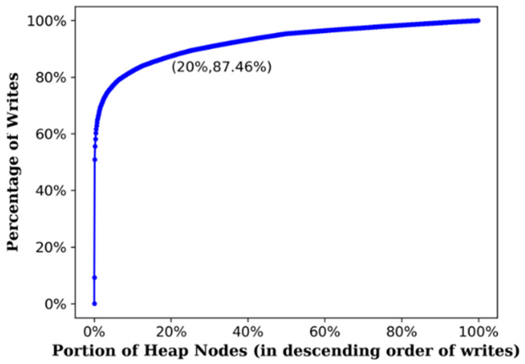


Fig. 1. Relationship between the portion of heap nodes and its percentage of total writes

Heap is physically stored in consecutive memory space like an array, while logically it is a binary tree with several constraints. Taking Min-heap as an example, the root node stores the minimal value and every node contains a smaller value than its child nodes. The HeapSort algorithm builds a heap at first; each time it replaces the root node with the last node in the heap, heapifies the heap, and outputs the original root to the result set until the heap goes empty. Note that every time we replace the root with new node, the heapify process will sift-down the new root until the heap meets Min-heap properties again. Therefore, every time the heap pops a root, the following heapify process does a couple of read/write operations to the heap memory space. Since the heap is a binary tree structure, if a node is close to the root, it is much likely to be read and written frequently. In other words, the nodes near to the root will have a higher C^g value than remote nodes. According to Theorem 1, if we place nodes close to the root into DRAM and nodes close to leaf into NVM, the total rea/write operations occurring on NVM as well as the total memory cost can be reduced.

Next, we want to present a more formal analysis. The time complexity of building a heap is $O(n)$ while HeapSort owns a time complexity of $O(nlog(n))$; thus, we mainly focus on the read and write operations in the sorting process. Assume that a Min-heap has the size of n, the distance from a node to the root is the height of the node, and l is the max node height, we can derive that $n \approx 2^{l+1}$. Suppose x is the height of current last node in the heap; when the node becomes the new root and is sifted down till heap property holds again, $O(x)$ reads and writes are done to the heap, $O(1)$ reads and writes for each level. In one level, each node has the same height, along with the same probability for being moved up (which incurs $O(1)$ times of reads and writes). Since level h has about 2^h nodes, the probability of a node being moved up is $\frac{1}{2^h}$, then the expectation of read/write operations on each node is $O\left(\frac{1}{2^h}\right)$. Therefore, after all the node is pop out, the total expectation of the read/write operations on each node in level h can be represented by (4).

$$E(h) = \sum\nolimits_{i=h}^{l} O\left(\frac{1}{2^h}\right)2^i = O\left(\frac{1}{2^h}\right)\sum\nolimits_{i=h}^{l} 2^i = O\left(\frac{\left(2^{l+1}-2^h\right)}{2^h}\right) = O\left(\frac{n}{2^h}\right) \quad (4)$$

That means the nodes closer to root have higher expectation of read and write. For example, the root node has an expectation of $O(n)$ while a leaf node has an expectation of $O(1)$. Using Formula (1), we can derive that the C^g node in level h, as represented by (5).

$$C_h^g = O\left(\frac{n}{2^h}\right)\left(r^p + w^p\right) - O\left(\frac{n}{2^h}\right)\left(r^d + w^d\right) = O\left(\frac{n}{2^h}\right)\left(r^p + w^p - r^d - w^d\right) \quad (5)$$

By leveraging the *optimal data placement scheme*, we can optimize HeapSort in the follow manner: place the nodes close to the heap root into DRAM and maintain other nodes in NVM.

Following the above idea, we devise a new sorting algorithm named NVMSort, as shown in Algorithm 1. Like HeapSort, NVMSort is also based on a heap data structure. However, differing from the traditional HeapSort, we divide the heap nodes in NVMSort into two parts, namely *NearRoots* and *NearLeafs*. *NearRoots* can fill up into

DRAM, which are placed in DRAM. Meanwhile, *NearLeafs* are organized in NVM before performing a classical HeapSort procedure.

In this part, we calculate the theoretical performance of NVMSort to prove that NVMSort outperforms the classical HeapSort in reducing both memory cost and writes to NVM. We assume that in the memory the proportion of DRAM is ϵ and we need to sort n elements. According to Formula (4), The total read-write operations of HeapSort is $\sum_{\mathrm{i}=1}^{n} O\left(\frac{n}{2^{h(i)}}\right) = n\sum_{\mathrm{i}=1}^{n} O\left(\frac{1}{2^{\lfloor \log(i) \rfloor}}\right) = O(nlog(n))$, where $h(i)$ is the height of node i, i.e., $h(i) = \lfloor log(i) \rfloor$. Because HeapSort treats the hybrid memory as a uniform memory space, we can assume that the read/write operations are distributed evenly among the available memory space. Therefore, $O(nlog(n)) \cdot \epsilon$ read/write operations will occur on DRAM and $O(nlog(n)) \cdot (1-\epsilon)$ operations will occur on NVM. According to Formula (2), the total memory cost of HeapSort can be calculated by (6):

$$\begin{aligned} C_{HeapSort} &= O(nlog(n)) \cdot \epsilon \cdot (r_d + w_d) + O(nlog(n)) \cdot (1-\epsilon) \cdot \left(r_p + w_p\right) \\ &= O(nlog(n)) \cdot \left(\epsilon(r_d + w_d) + (1-\epsilon)\left(r_p + w_p\right)\right) \end{aligned} \tag{6}$$

Algorithm 1. NVMSort

```
Input: an unsorted data array $A$
Output: a sorted array $A$ in ascending order
1:   $NearRoot = DRAM$ space;
2:   $NearLeaf = (A$.size * Node_Size – $NearRoot)$ $NVM$ space;
3    $heap\_space = \{NearRoot, NearLeaf\}$;
4:   // make sure Nodes close to Root reside in NearRoot space
5:   $heap$ = Build_Max_Heap($heap\_space$, $A$);
7:   for $i = heap.length$ downto 2
8:       $A[i] = heap[1]$;
9:       $heap[1] = heap[i]$;
10:      $heap$.size = $heap$.size - 1;
11:      Max_Heapify($heap$);
12:  end for
13:  $A[1] = heap[1]$;
14:  return $A$;
```

While for NVMSort, the read/write operations to DRAM are all absorbed by *NearRoots*. The total number of operations is $\sum_{i=1}^{n\cdot\epsilon} O\left(\frac{n}{2^{h(i)}}\right) = n\sum_{i=1}^{n\cdot\epsilon} O\left(\frac{1}{2^{\lfloor log(i) \rfloor}}\right) = O(nlog(n\cdot\epsilon))$. Similarly, the read/write operations to NVM are concentrated on *NearLeaf*, which are $\sum_{i=n\cdot\epsilon+1}^{n} O\left(\frac{n}{2^{h(i)}}\right) = n\sum_{i=n\cdot\epsilon+1}^{n} O\left(\frac{1}{2^{\lfloor log(i) \rfloor}}\right) = O\left(nlog\left(\frac{1}{\epsilon}\right)\right)$. Thus, we can calculate the total memory cost of NVMSort by (7).

$$C_{NVMSort} = O(nlog(\mathrm{n}\cdot\epsilon))(r_d + w_d) + O\left(nlog\left(\frac{1}{\epsilon}\right)\right)\left(r_p + w_p\right) \tag{7}$$

According to the analysis above, HeapSort incurs $O(nlog(n))$ writes to NVM while NVMSort only makes $O(n)$ NVM writes. To this end, we can see that NVMSort is more write-friendly to NVM. Due to the read and write asymmetry of NVM chips, especially the write latency w_p of NVM is typically one order higher than the read latency r_p, the reduction of NVM writes in NVMSort can result in significant performance improving, when compared with HeapSort. When combined reads and writes, NVMSort can reduce the total access time of NVM by $O\left(nlog\left(\epsilon \cdot n^{1-\epsilon}\right)\right)\left(r_p + w_p\right)$. In the following experiments, ei demonstrate the performance of NVMSort to support the above analysis.

4.2 Sort Join with NVMSort

In this section, we discuss the applicability of NVMSort in traditional sort join. Sort join is one of the commonly used join algorithms in modern relational DBMSs. As the relations to be joined are supposed to be in external storage, e.g., SSDs or magnetic disks, traditional sort join usually employs the merge sort algorithm to sort relations residing in disks. The basic process of the merge-sort join consists of the following steps:

(1) Read pages into the memory;
(2) Sort the tuples in the memory;
(3) Write the sorted tuples as a run into the disk;
(4) Read all the first pages in each run, merge in memory, and write out to the file.

NVMSort can be used to optimize step (2) in the merge-sort join algorithm. In the implementation of traditional merge-join, we usually utilize QuickSort as the main-memory sort algorithm, while in the NVM-based hybrid memory QuickSort is not the best choice, as we have discussed before. We will experimentally demonstrate that NVMSort is more efficient than QuickSort as well as HeapSort when applied into the merge-sort join algorithm.

5 Performance Evaluation

In this section, we evaluate the efficiency of NVMSort by comparing it with other sort algorithms. The competitors include two traditional sort algorithms, including HeapSort and QuickSort [8]. We measure the total run time of each algorithm along with its total reads and writes to DRAM and NVM. The results show that our NVMSort has the best performance with limited writes to NVM.

5.1 Settings

All the experiments are performed on an Intel Core i5-8500 3.0 GHZ CPU. This CPU has 6 physical cores, with a 9 MB L3 cache. Each core has a private 256 KB L2 cache and 32 KB L1d cache and 32 KB L1i cache. The operation system is Ubuntu 18.04 with the kernel version of 5.2.8. We use the open-source persistent memory development kit [16] to simulate NVM on Linux. This library maps a memory region to on-disk file and ensure the atomicity and persistency on read/write operation to that region. In order to simulate the read-write asymmetric of mainstream NVM, we follow the lead of the hardware community [17] and inject artificial 240 ns delays after each write operation to that persistent region.

Each algorithm is allocated with the same amount of memory. The ratio between DRAM and NVM space is set to same for each algorithm. However, the classical HeapSort and QuickSort are unaware of the read-write asymmetric property of NVM. They use the hybrid memory space as a uniform memory space. NVMSort treats DRAM and NVM differently. It splits the heap structure into two parts before sorting; the formal part is put into DRAM and the latter part is placed in NVM.

We use three datasets to test the sorting algorithms, scaling from 100k elements (denoted as 100K below), 1 million elements (denoted as 1M) to 10 million elements (denoted as 10M). All data elements are tuples with 8 bytes key and 8 bytes payload, which is generated randomly and stored in the hybrid memory space.

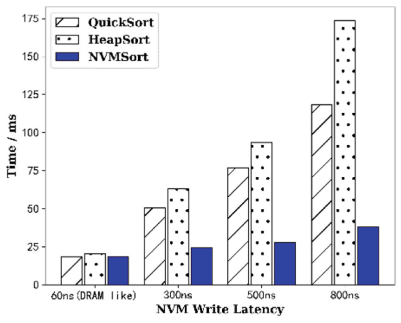
5.2 Sorting Performance

We mainly evaluate the run time of each sorting algorithm. In addition, we measure the write count on NVM during sorting, as reducing writes to NVM is one of the key objectives of NVMSort. In the following text, we will present the results on the small dataset as well as on the large dataset.

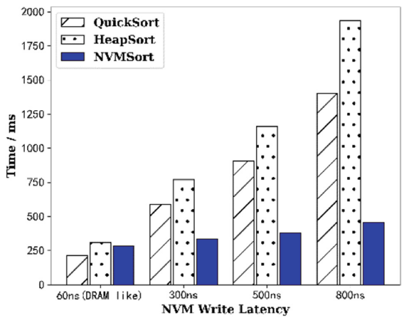
Scalability. Figure 2 shows the run time on the three datasets under different NVM write latency. In this experiment, the DRAM ratio is set to 0.2. We can see that NVMSort gets the best sorting performance when running on datasets scaling from 100K, 1M to 10M. Averagely, NVMSort is 1.5$\times$ faster than HeapSort and QuickSort. As NVM write latency gets higher, NVMSort exhibits more advantage than its competitors. This experiment shows that NVMSort can well suit different sizes of datasets, showing good scalability for real applications. Therefore, in the following experiments, we use the 1M dataset by default.

NVM Writes. Table 2 summarizes the DRAM reads/writes and NVM reads/writes of NVMSort, HeapSort, and QuickSort, when running on the 1M dataset with the DRAM ratio on 0.2. We can see that NVMSort has the fewest NVM writes among all algorithms; thus it is more NVM friendly than the other two algorithms. As NVM has limited write endurance, reducing writes to NVM is a critical issue in the design of NVM-aware sorting algorithms. NVMSort also has fewer NVM reads than QuickSort and HeapSort. The DRAM accesses of NVMSort is comparable to that of HeapSort. Although the total DRAM reads/writes of NVMSort is a little more than QuickSort, QuickSort has much more NVM reads/writes than NVMSort, resulting in poor time performance of QuickSort as shown in Fig. 2. On the other hand, this study does not aim to optimize DRAM operations.

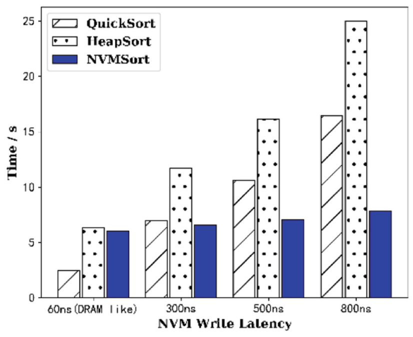
Sensitivity to the DRAM Ratio. In this experiment, we aim to measure the performance of NVMSort when varying the ratio of DRAM among the hybrid memory. The 1M dataset is used in this experiment. The NVM write latency is set to 300 ns. Figure 3 shows the run time of NVMSort and its two competitors, in which the DRAM ratio is varied from 0.1 to 0.4. Note that we do not increase the DRAM ratio to a much high value,



(a) 100K dataset



(b) 1M dataset



(c) 10M dataset

Fig. 2. Run time on different scales of datasets

Table 2. Comparison of DRAM read/writes and NVM read/writes

	DRAM		NVM	
	Reads (10^6)	Writes (10^6)	Reads (10^6)	Writes (10^6)
QuickSort	11	56	41	22
HeapSort	24	83	87	30
NVMSort	102	35	9	3

because NVM has higher density than DRAM. Thus, it is not likely that the DRAM size is over the NVM size in the hybrid memory system. As shown in Fig. 3, NVMSort keeps relatively high and stable performance when varying the DRAM ratio. This is because NVMSort can choose optimal data placement according to the DRAM and NVM usage during the sorting process. On the contrary, both QuickSort and HeapSort do not make any optimizations for NVM, resulting poor time performance.

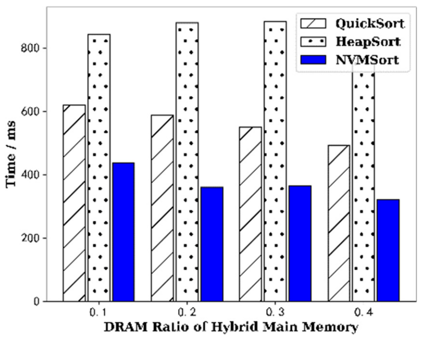


Fig. 3. Run time of three algorithms when varying the DRAM ratio

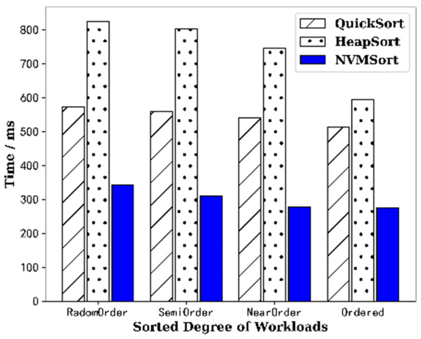


Fig. 4. Workload-change resistance of NVMSort, QuickSort and HeapSort

Workload-Change Resistance. In this experiment, we aim to see whether NVMSort can perform well on different kinds of workloads. We manually vary the sorted order of the elements in the 1M dataset and generate four kinds of workloads, which are named RandomOrder (totally unordered), SemiOrder (50% ordered), NearOrder (80% ordered), and Ordered (totally ordered). We then run NVMSort on these four kinds of workloads to verify its performance. Here, the DRAM ratio is also set to 0.2 and the NVM write latency is set to 300 ns. As Fig. 4 shows, NVMSort has stable time performance when the workload property changes from "totally unordered" to "totally ordered". In addition, NVMSort has the best time performance in all workloads, indicating that NVMSort is workload-aware due to its intrinsic optimal data placement scheme.

NVM Efficiency. NVMSort is designed for reducing both sorting time and NVM writes. The above experiments present detailed measurement of NVMSort with respect to different measures. However, as NVMSort is designed not only for high time performance but also for reducing NVM writes, it is not apparently to see the overall performance of NVMSort in terms of time performance and NVM friendliness.

Thus, we further propose a metric named *NVM Efficiency* that combines both run time and NVM write reduction. The *NVM Efficiency* as the overall performance is defined as follows:

$$NVM\ \ Efficiency = P/W_{NVM}, \ \ where\ P = 1/t \tag{8}$$

Here, P represents the time performance of a sorting algorithm (t is the run time of the algorithm), and W_{NVM} is the NVM writes caused by a sorting algorithm. The *NVM Efficiency* represents the time performance per NVM write, which implies that higher time performance or less NVM writes will result in high NVM efficiency. As higher time performance and less NVM writes is actually the design goal of this study, the *NVM Efficiency* is suitable for measuring the overall performance of a sorting algorithm.

Figures 5 shows the NVM efficiency on the 1M datasets. Here, the time performance P is calculated with the time granularity of second. In this figure, we can see that as the DRAM ratio gets higher, all the sort algorithms have better NVM Efficiency due to more DRAM space can reduces costly NVM writes. Particularly, when the DRAM ratio is 0.1, the NVM Efficiency of NVMSort is 6× higher than QuickSort and 13× than HeapSort. As the DRAM ratio goes up to 0.4, NVMSort is 12×/22× higher than QuickSort/HeapSort in term of NVM Efficiency. That clearly shows that the proposed NVMSort algorithms maintains the highest NVM efficiency in various DRAM ratio, meaning that NVMSort has the best overall performance.

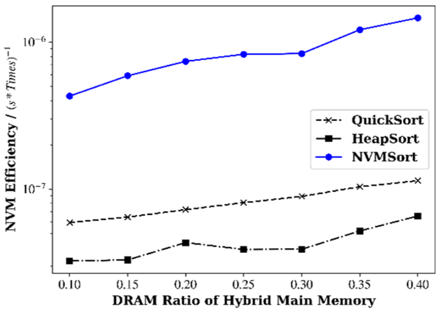


Fig. 5. NVM efficiency

To sum up, all the experiments show that our NVMSort is more efficient for the DRAM-NVM-based hybrid memory system compared with traditional sorting algorithms including HeapSort and QuickSort. This is mainly due to the new structural design in NVMSort, i.e., the heap-node organization and placement based on the optimal data placement model.

5.3 Join Performance

In this experiment, we compare the time performance as well as NVM writes of sort join algorithms. We implement the traditional merge-sort join algorithm and replace the in-memory sorting algorithm with NVMSort. This improved sort join is denoted as SMJ_NVMSort (meaning Sort-Merge-Join with NVMSort). Similarly, we get the other two sort join algorithms, which are denoted as SMJ_QuickSort and SMJ_HeapSort. In this experiment we also use the 1M dataset and the DRAM ratio is set to 0.2. The join algorithm runs on two relational tables R and S. Since we allocate total 12 MB memory (including 9.6 MB NVM and 2.4 MB DRAM, nearly 4:1) and assume that both relations cannot be hold in the memory, we set the size of R to 16 MB and the size of S to 160 MB. This setting is compatible with the general assumption of join algorithms in traditional relational database systems, in which the build relation R is smaller than the probe relation S.

The results are shown in Fig. 6. When the NVM write latency is close to that of DRAM, SMJ_NVMSort has similar performance with the other two algorithms. However, when the NVM write latency is set to be higher than that of DRAM, we can see that SMJ_NVMSort outperforms SMJ_QuickSort and SMJ_HeapSort. To this end, SMJ_NVMSort is more suitable for DRAM-NVM-based hybrid memory.

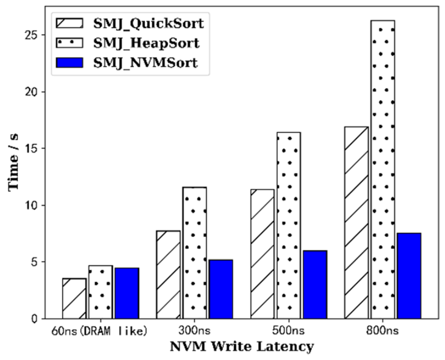


Fig. 6. Run time of three join algorithms

Table 3 shows the NVM reads/writes as well as the DRAM accesses of the three join algorithms. SMJ_QuickSort has more than 6 times of NVM writes than SMJ_NVMSort, while SMJ_HeapSort has over 9 times of NVM writes. Thus, SMJ_NVMSort is more NVM friendly than the other two algorithms. Although SMJ_NVMSort has a bit more DRAM accesses than its competitors, this does not influence the time performance of SMJ_NVMSort, as shown in Fig. 6. In addition, as DRAM has unlimited write endurance and is faster than NVM, increasing DRAM accesses is not acceptable in DRAM-NVM-based hybrid memory.

Table 3. Comparison of DRAM read/writes and NVM read/writes

	DRAM		NVM	
	Reads (10^6)	Writes (10^6)	Reads (10^6)	Writes (10^6)
SMJ_QuickSort	141	755	468	252
SMJ_HeapSort	202	724	1093	380
SMJ_NVMSort	1178	409	117	44

6 Conclusions

NVM has become an alternative of next-generation memories. It is a trend to construct hybrid memory systems composed of DRAM and NVM in the future. In this paper, we studied the fundamental sorting issue on DRAM-NVM-based hybrid memory and proposed an efficient sorting algorithm called NVMSort. NVMSort is based on the optimal data placement model that proposes to maintain highly-written data structures in DRAM and others in NVM. We theoretically proved the correctness and efficiency of the optimal data placement model, based on which NVMSort was presented. We further integrated NVMSort into the traditional merge-sort join algorithm. We conduct extensive experiments on various datasets with different settings. The results suggest the high performance and NVM friendliness of NVMSort and the new sort-join scheme.

There are some future research directions that are worth further investigating. First, it is a promising issue to consider efficient buffer management schemes to improve the join performance on NVM-based hybrid memory [18, 19]. Second, it is valuable to study other join algorithms such as hash join that runs on NVM-based hybrid memory [20]. Third, the architecture of hybrid memory systems has a big impact on fundamental algorithms. There are also other kinds of architecture for hybrid memory systems, e.g., the hybrid storage involving DRAM, NVM, and SSD [21]. Thus, we will investigate sorting and join algorithms on other kinds of hybrid-storage architecture in the future.

Acknowledgement. This paper is partially supported by the National Science Foundation of China under the grant number no. 61672479.

References

1. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation, 2nd edn. Prentice Hall, Upper Saddle River (2010)
2. Faerber, F., Kemper, A., Larson, P., Levandoski, J.J., Neumann, T., Pavlo, A.: Main memory database systems. Found. Trends Databases **8**(1–2), 1–130 (2017)
3. Renen, A., et al.: Managing non-volatile memory in database systems. In: SIGMOD, pp. 1541–1555 (2018)
4. Psaropoulos, G., Oukid, I., Legler, T., May, N., Ailamaki, A.: Bridging the latency gap between NVM and DRAM for latency-bound operations. In: DaMoN, 13:1–13:8 (2019)
5. Chen, K., Jin, P., Yue, L.: A novel page replacement algorithm for the hybrid memory architecture involving PCM and DRAM. In: Hsu, C.-H., Shi, X., Salapura, V. (eds.) NPC 2014. LNCS, vol. 8707, pp. 108–119. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44917-2_10
6. Zhang, D., Jin, P., Wang, X., Yang, C., Yue, L.: DPHSim: a flexible simulator for DRAM/PCM-based hybrid memory. In: Chen, L., Jensen, C.S., Shahabi, C., Yang, X., Lian, X. (eds.) APWeb-WAIM 2017. LNCS, vol. 10367, pp. 319–323. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63564-4_27
7. Jin, P., Yang, C., Jensen, C.S., Yang, P., Yue, L.: Read/write-optimized tree indexing for solid-state drives. VLDB J. **25**(5), 695–717 (2016)
8. Cormen, T., Leiserson, C., Rivest, R., et al.: Introduction to Algorithms. MIT Press, Cambridge (2009)
9. Wang, Y., Li, K., Zhang, J., Li, K.: Energy optimization for data allocation with hybrid SRAM+NVM SPM. IEEE Trans. Circ. Syst. **65-I**(1), 307–318 (2018)
10. Salkhordeh, R., Mutlu, O., Asadi, H.: An analytical model for performance and lifetime estimation of hybrid DRAM-NVM main memories. IEEE Trans. Comput. **68**(8), 1114–1130 (2019)
11. Li, L., Jin, P., Yang, C., Wan, S., Yue, L.: XB+-tree: a novel index for PCM/DRAM-based hybrid memory. In: Cheema, M.A., Zhang, W., Chang, L. (eds.) ADC 2016. LNCS, vol. 9877, pp. 357–368. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46922-5_28
12. Chen, S., Gibbons, P.B., Nath, S.: Rethinking database algorithms for phase change memory. In: CIDR (2011)
13. Viglas, S.: Adapting the B+-tree for asymmetric I/O. In: ADBIS, pp. 399–412 (2012)

14. Viglas, S.: Write-limited sorts and joins for persistent memory. Proc. VLDB Endow. **7**(5), 413–424 (2014)
15. Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Gu, Y., Shun, J.: Sorting with asymmetric read and write costs. In: SPAA, pp. 1–12 (2015)
16. OFTC: Persistent Memory Development Kit, 25 March 2020. https://pmem.io/pmdk/
17. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: ASPLOS, pp. 91–104 (2011)
18. Wu, Z., Jin, P., Yang, C., Yue, L.: APP-LRU: a new page replacement method for PCM/DRAM-based hybrid memory systems. In: Hsu, C.-H., Shi, X., Salapura, V. (eds.) NPC 2014. LNCS, vol. 8707, pp. 84–95. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44917-2_8
19. Ou, Y., Härder, T., Jin, P.: CFDC: a flash-aware buffer management algorithm for database systems. In: Catania, B., Ivanović, M., Thalheim, B. (eds.) ADBIS 2010. LNCS, vol. 6295, pp. 435–449. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15576-5_33
20. Yang, L., Jin, P., Wan, S.: BF-join: an efficient hash join algorithm for DRAM-NVM-based hybrid memory systems. In: ISPA, pp. 875–882 (2019)
21. Jin, P., Yang, P., Yue, L.: Optimizing B+-tree for hybrid storage systems. Distrib. Parallel Databases **33**(3), 449–475 (2015)