A Write Efficient PCM-Aware Sort

Meduri Venkata Vamsikrishna*, Zhan Su, and Kian-Lee Tan

School of Computing, National University of Singapore
meduri@cwi.nl
{suzhan,tankl}@comp.nus.edu.sg

Abstract. There is an increasing interest in developing Phase Change Memory (PCM) based main memory systems. In order to retain the latency benefits of DRAM, such systems typically have a small DRAM buffer as a part of the main memory. However, for these systems to be widely adopted, limitations of PCM such as low write endurance and expensive writes need to be addressed. In this paper, we propose PCM-aware sorting algorithms that can mitigate writes on PCM by efficient use of the small DRAM buffer. Our performance evaluation shows that the proposed schemes can significantly outperform existing schemes that are oblivious of the PCM.

1 Introduction

Design of database algorithms for modern hardware has always been a prominent research area. Phase Change Memory (PCM) is a latest addition to the list of modern hardware demanding the design of PCM-friendly database algorithms. With PCM being better than flash memory (SSD) in terms of write endurance, read and write latency, focus has shifted to exploring the possibilities of exploiting PCM for databases. Because of the high density and lower power consumption of PCM as compared to DRAM, it is evident that PCM might be an alternative choice for main memory [2, 3]. However, as PCM has a low write endurance and high write latency as compared to DRAM, it is essential to design algorithms such that they do not incur too many writes on PCM and thus prevent the hardware from getting worn out soon.

In-memory sorting is a write-intensive operation as it involves huge movement of data in the process of ordering it. Quick sort is the most expensive in terms of data movement though it is the best in time complexity. Selection sort involves fewest data movement but it has quadratic time complexity and also incurs a lot of scans on the data. In this paper, we design write-aware in-memory sorting algorithms on PCM. Like [3], we also use a small DRAM buffer to alleviate PCM writes using efficient data structures. We assume that the data movement between DRAM and PCM can happen seamlessly. This can be achieved through a hardware driven page placement policy that migrates pages between DRAM and PCM [6].

* Please note that this work was done while the author was at the National University of Singapore.

Our first sort algorithm constructs a histogram that allows us to bucketize the in-memory data such that either the depth or width of each bucket is DRAM-size bound. This is an important heuristic that we introduce to make our algorithm write-efficient as well as run-time efficient. Quick sort is employed on buckets that are depth bound and counting sort is used to sort the buckets that are width bound such that minimum writes are incurred on PCM. We further minimize PCM writes aggressively with an improved version of our algorithm. In this variant, we construct the histogram even before the data is read into PCM by sampling data directly from the disk.

If the unsorted data on disk sorted doesn't entirely fit into the PCM at one go, external sort is performed which involves getting the data in chunks to main memory one by one, sorting the chunk, creating the runs on disk and finally merging those runs. We show in the experiments that our algorithm performs well even in the scenario where the entire data is not memory resident.

There have been few works in adapting database algorithms for PCM. In [1], a B+ tree index structure that incurs fewer writes to PCM (which is used as main memory) is proposed. Though the idea to reduce memory writes is beneficial, the B+ tree nodes (at least the leaves) are kept unsorted to obtain fewer modifications during insertion and deletion of data to the index. This leads to more expensive reads. An algorithm to adapt hash join is also proposed in [1] which minimizes the writes from the partitioning phase on PCM by storing the record ids of the tuples (or the differences between consecutive record ids) in the hash partitions. The records are in-place accessed during the join phase using the record ids. The algorithm also aims at achieving fewer cache misses.

PCM-aware sorting is of significance in the context of databases as it is used in many query processing and indexing algorithms. Our work to produce a PCM aware and efficient sorting algorithm can help alleviate the heavy read exchange the existing B+ tree index algorithm in [1] does. It can also be extended to obtain a PCM-aware sort merge join algorithm. To our knowledge, this is the first report work on sorting algorithms in memory-based PCM.

We present our basic and advanced PCM sorting algorithms in Sections 2.3 and 2.4 respectively. We report results of an extensive performance study in Section 3.

2 PCM-aware sorting algorithm

Our goal is to design efficient sorting algorithms that incur as few writes on PCM as possible. As context, we consider external multi-way merge sort that comprises two main phases: (a) generating sorted runs, and (b) merging the runs into a single sorted file. Our key contribution is in the first phase where PCM-aware in-memory sorting is proposed.

For our hybrid architecture, we divide the main memory into three sections. PCM is divided into two partitions - a large chunk to hold the incoming unsorted data and a small chunk for use by a histogram. We refer to the first partition as

sort-chunk, and the second partition as hist-chunk. The DRAM forms the third partition.

In the following subsection, we describe the run generation phase. We first look at a naive strategy, afterwhich we present our two proposed solutions.

2.1 A Naive Approach

Under the naive strategy, we fill the sort-chunk with as many tuples from disk as possible. Next, we adopt the scheme in [4] to generate an equi-depth histogram (in hist-chunk) using a single scan of the data in the sort-chunk.

The depth of each bucket in our equi depth histogram will be no greater than DRAM size. Once the histogram is constructed, the input data in the PCM buffers is shuffled according to the bucket ranges in the histogram. That means, the data belonging to each bucket range is brought together such that data belonging to bucket 0 is placed first, bucket 1 next and so on. So the input data is sequentially arranged according to the bucket intervals, still unsorted within each interval. Now each bucket is sent to the DRAM and a quick sort is done upon the bucket data in DRAM. The sorted data will be written to the sorted run on disk directly (assuming DRAM has direct access to the disk). Figure 1 illustrates the manner in which bucket transfer is done from PCM to DRAM.

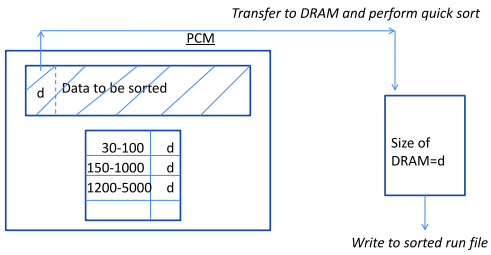


Fig. 1. PCM-aware sorting scheme (naive).

The PCM writes in this scheme will not exceed $2n + sizeof(Histogram)$ (including the disk fetches) where "n" denotes the total size of the input block of data fetched into PCM. This is because, histogram storage on PCM and shuffling of unsorted data according to bucket range contribute to writes.

2.2 Refining the naive sorting scheme

We observe that the naive scheme described above is insensitive to data skew. Where data is skewed, using quicksort is an overkill (and costly). Instead, we can potentially improve the performance by adopting the Counting Sort. Instead of swapping the elements to be sorted, it focusses on the count of each element. If the data range is small enough to fit in memory, a count array is constructed with

array index starting from minimum element of the input data to the maximum element. The count array determines the number of elements less than or equal to each data element. Thus it determines the final positions of each data element in the sorted array. Please refer to [10] for complete description of the algorithm.

One of the key disadvantages of Counting Sort is that it is effective only for small data ranges. However, this seemingly disadvantageous trait indeed benefits our scheme. For skewed data, our naive scheme fails to identify the skewedness and constructs several equi depth buckets. Whereas, if we can identify that the frequency of values in a certain range is higher and if the range manages to fit into DRAM, we can replace quick sort with counting sort for the elements of that range.

- The bucket data need not fit into DRAM. Instead if the count array is smaller than DRAM size, it can help us sort many elements in one go, thus saving time.
- This makes our histogram hybrid and compact: either its width or the depth is DRAM-bound.

In the example shown in Figure 2, DRAM size is 50 elements, and PCM holds an unsorted data block of 600 elements. These 600 data elements are distributed over three non-overlapping ranges as 50, 500 and 50 elements. It can be observed that the ranges of the first and third buckets are large spanning over 1000 element width, but the depth is just 50. On the other hand, the middle bucket which has just a range of 50 elements holds 500 elements because of several duplicates. Not paying attention to the skewed range of a small interval holding a whopping 500 values, an equi-depth histogram is constructed by our naive scheme with 12 buckets of 50 elements each since 50 is the available DRAM size. But if we can pay attention to the width of each bucket and strive to make it DRAM-contained, we can achieve a compact histogram with just 3 buckets. This is because the second bucket's width is 50 (<= size of DRAM) and hence a DRAM-contained count array can be constructed for the data in the second bucket. Buckets 1 and 3 are depth-bound, so quick sort is employed to sort them by transferring the bucket data to DRAM.

That means, while constructing a bucket we need to check whether the depth or width of a bucket is hitting the DRAM limit first as input (unsorted) data keeps getting added to a bucket. Whichever parameter hits the limit later is the one we use to decide the bucket boundaries. For example, if the data is sparsely distributed, bucket width crosses DRAM limit easily but it will take a while for its depth to hit the DRAM limit, in that case the bucket is depth-bound. In case of densely distributed data, it is the other way round and the bucket becomes width bound. So our histogram is robust to both sparsely and densely distributed data.

One more advantage in having a compact histogram is that, the amount of data that has to be transferred to DRAM is minimal. Whenever there is an opportunity to avoid data transfer, we grab it by employing counting sort. In the worst case where the entire data is uniformly distributed, our hybrid histogram

gracefully reduces to an equi-depth histogram since the need to use counting sort arrives only in the case of non-uniformly distributed data. Perhaps, a constraint in using counting sort is that it is basically applicable to sorting integers.

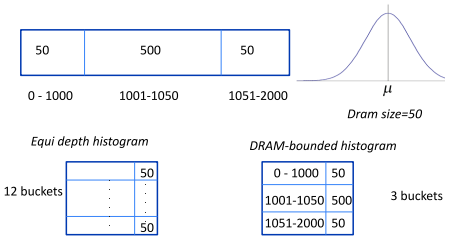


Fig. 2. Compact Histogram achieved using Counting Sort for normal data distribution.

2.3 Algorithm 1: Basic PCM-aware sorting

Our scheme consists of the following steps:

- Construction of the hybrid DRAM-bounded width / depth histogram
- Shuffling the input data block according to bucket range
- Since the histogram constructed is not 100% accurate, efforts should be made to combat it
- Transfer of data / range to DRAM to perform quick / counting sort

Once the hybrid histogram is constructed (Refer Figure 3), the algorithm transfers the bucket data to DRAM and performs quick sort if the depth $\leq$ DRAM size. On the contrary, if the width $\leq$ DRAM size, the count array is built in DRAM and accordingly the data is moved within PCM to its final places by looking up to the count array. It should be noted that we do not use a separate output array for counting sort, rather the data on PCM is moved in-place to reach the sorted order.

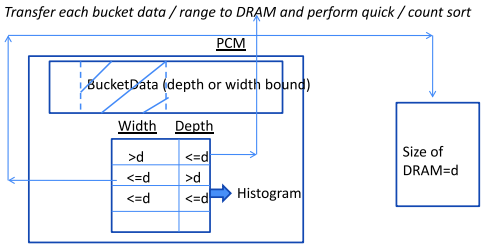


Fig. 3. Basic PCM-aware sort scheme.

Construction of Histogram We follow a similar way of histogram construction as [5]. A scan of the input elements in PCM is performed to uniformly pick up "d" random elements, two of them being the minimum and maximum of the scanned block. The reason for including maximum and minimum values in the sample is to let them participate in fixing the bucket boundaries of the first and the last buckets respectively. Else, we would miss some potential values we need to sort. With "d" being the DRAM size, the sampled elements are sorted inside DRAM using quicksort. Now the sorted sample array present in DRAM is scanned to construct the required histogram.

The constructed histogram is for the sample array and it needs to be scaled up for the original PCM resident data. It should be understood that the width (range) of each bucket in the histogram is appropriate because we made sure we did not even miss including the original data's minimum and maximum values while fixing the bucket ranges. The only difference is that, since we are operating on the sampled array, the maximum allowed depth of each bucket (in cases where the width has already exceeded the DRAM limit) is $\frac{d^2}{sizeof|inputelements|}$ as against the regular value of d. Here $inputelements$ refers to the original unsorted data residing in the PCM. Since the histogram is constructed on PCM, the depth values of all the buckets are scaled up before storing them in the histogram.

Firstly, PCM writes are incurred by the histogram after its construction (because it is stored on PCM). The number of writes is equal to the size of the histogram.

Kolmogorovs statistic gives a theoretical support to sampling. It fixes a bound of 0.05 error for sample size of 1024 tuples and 740 tuples with confidence 99% and 95% respectively([5]).

Shuffling of data according to histogram buckets The unsorted data held in the PCM buffer is re-arranged such that all the elements belonging to the same bucket are brought to contiguous locations though unsorted. This is used in our naive scheme as well (refer section 2.1). The easiest way to do this is to create a new array and to transfer the elements from the original array to the new array according to the requirement. But we choose to do in-place shuffling to be memory conscious.

If the total number of PCM elements are "n", at max, there can only be "n" moves and thus "n" integer writes, i.e, $n * sizeof(int)$ byte writes. On the other hand if we are sorting tuples, the number of writes would rather be $n * sizeof(tuple)$.

Appropriating the hybrid histogram We make use of Kolmogorovs statistic to reduce the error in bucket depths, followed by a rigorous bucket correction procedure.

We introduce a tightening factor Δ into the bucket depth to reduce the possibility of errors. As errors in the histogram are induced because of miscalculating bucket depths, we apply the tightening factor to depth bound and not width bound. Instead of DRAM size "d", the depth bound is fixed to $(1-\Delta)*d$, where

Δ could range from 0.05 to 0.1 meaning 5% to 10% error is accommodated. Even though this bound tightening is applied, bucket depths can still cross the DRAM size with a small probability. To correct it we apply iterative splitting and shuffling on the erring buckets till we reach a correct hybrid histogram.

The extra writes incurred during histogram correction are dependant on the number of buckets which are corrected. Each time a bucket is split, it incurs a new bucket write on the histogram and the PCM data belonging to that bucket alone is shuffled incurring PCM writes. If "*new*" is the number of new buckets that are created because of additional splits, on an average it leads to $new * (hist\_bucket + PCM\_bucket\_data)$ writes on PCM because of histogram manipulation and bucket shuffling. These writes are few, because on an average, very few buckets are corrected and no bucket from our experiments encounters more than two rounds of additional splits.

Writes during quick / counting sort As mentioned in section 2.1, quick sort is done in DRAM for those buckets whose depth is DRAM bounded and the sorted elements can be written directly to the disk provided DRAM in the hybrid architecture has direct access to the disk incurring zero PCM writes. In cases where the bucket width is DRAM bounded, the count array is computed in the DRAM. Out of memory consideration, we perform in-place movement of data on PCM looking up to the aggregated count array present in DRAM. So the writes are again at max "n" provided there are "n" data elements. It is clear that because of counting sort, to get the sorted order a linear number of writes happen on PCM. But these writes are minimum and worthwhile given the speed and compact histogram we achieve because of counting sort using our hybrid scheme. Moreover if the data is uniformly distributed, our algorithm automatically reduces to our naive scheme.

Because we use counting sort for width-bound buckets, we achieve some linearity in time complexity. Since counting sort is used only in the case of non-uniform data distribution, suppose "r" buckets are depth bound and "$n - r$" buckets are width bound, the computation goes as $\sum_{i=0}^{r-1} O(n_i \log n_i) + \sum_{i=r}^{n-1} O(n_i + k_i)$ where k_i indicates the width of "i"th bucket as against a relatively expensive $\sum_{i=0}^{n-1} O(n_i \log n_i)$ incurred by a non-skew aware scheme.

2.4 Algorithm 2: Advanced PCM-aware sorting

The best running time is achieved by quick sort and the least memory writes are achieved by selection sort. But quick sort is worse in terms of writes because of numerous swaps and selection sort is bad in terms of the huge reads and long sorting time. We try to achieve an algorithm which is close to both the ideal attributes of least writes and running time in our algorithm 1.

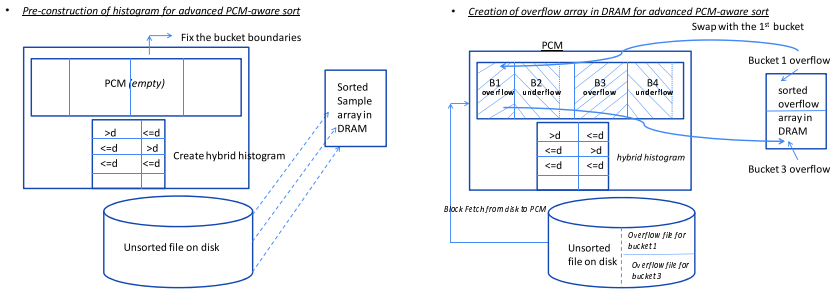
In this section, we design an algorithm that improves over the basic PCM aware sort scheme by aggressively reducing the PCM writes. The writes incurred in main memory because of comparisons and swaps have already been reduced using the basic PCM aware algorithm. The aspect that incurs any extra writes

(other than disk fetches to PCM) is histogram creation and shuffling. We already ensure that the histogram is always compact. But shuffling is write-intensive. Because our histogram is constructed after fetching the data from disk to memory, we need to shuffle the data in the memory to ensure the unsorted data belonging to each bucket gets collected together. This causes a number of writes directly proportional to the data size that is memory resident, because in the worst case each tuple (assuming that we are sorting tuples) has to move to get to its intended bucket.

Our idea is to construct the histogram even before the data is fetched into main memory (please refer Figure 4(a)), so that once the actual data from disk starts arriving, it can directly go to its respective bucket and thus avoid a shuffle. This can avoid PCM writes totally except for the disk fetch. The efficiency of this approach still depends on the accuracy of the histogram. And the accuracy of the histogram depends on the extent to which sampling helps us. But this incurs a lot of random reads from the disk.

So we adopt two steps to enhance the accuracy of our histogram while keeping the random reads from disk within limit.

- While fetching the data from the disk to DRAM to construct the sampling array for sorting, we drop all attributes in the tuples other than the sorting attribute.
- At the same time, we do not fill the entire DRAM with the sampling array. We just use a small fraction of the DRAM to construct the sample array. So this avoids the overhead of sorting too many tuples beforehand and the overhead of random reads from the disk.



(a) Pre-construction of histogram (b) Creation of Overflow array in DRAM

Fig. 4. The Advanced PCM-Aware Sort

It is important to note that in this advanced scheme, we do not need to rescan data and correct the histogram depths. Correction is done as the data is fetched from disk to PCM.

Errors and correction during memory fetch As the tuples are fetched into PCM, they go into their respective bucket boundaries which are computed from the histogram widths. But in this case, both widths and depths can be at fault unlike algorithm 1 which needs to correct wrong depths. Since the histogram is being pre-constructed, the minimum and maximum elements of the fetched block are not known. So the first bucket and the last bucket in our histogram can be erroneous with respect to the minimum boundary of the first bucket and the maximum range boundary of the last bucket. To accommodate the wrong estimate, we keep extending the border buckets' boundaries as elements come in. If a tuple with its sort attribute value $\leq$ minimum-boundary of bucket 1, the new tuple will now belong to bucket 1 and the minimum boundary is updated. The same applies to elements arriving beyond the last bucket as well. But there are two problems with this.

- The new elements can cause the depth to go beyond the estimate. If the depths cross DRAM size and if the bucket is depth-bound, sorting the bucket using DRAM is difficult.
- The new element can cause the width to go beyond the estimated boundary. If the bucket is width bound and if counting sort was planned to be employed, the bucket can no longer be sorted using DRAM as the count array is too big to fit into DRAM.

The width error is solved using a slack (Δ) 5% to 10% (same as the tightening factor in Section 2.3) which is sufficient for uniformly distributed data. But in the case of non-uniformly distributed data, the global maximum and minimum which we know from pre-computed database statistics are always included. This is because if the skew in data distribution is extremely high, it is impossible for slack to control width errors.

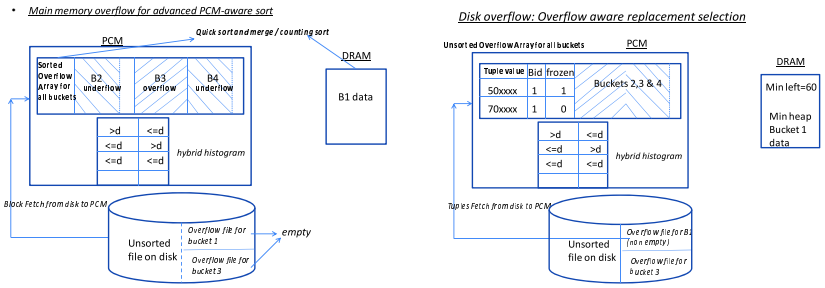
To correct depth errors, we introduce the construction of an overflow array. If there is a bucket whose depth was under-estimated, there should be some other bucket with an over-estimated depth. So if there is no space to accommodate some of the elements (tuples) belonging to a particular bucket due to an overflow, they can still find a place on the PCM in some of the holes created because of some underflowing buckets. But the management of these holes requires a lot of meta data and a very tedious way of maintaining bucket information. An overflow from some bucket may have to be distributed over multiple small underflows from several underflow buckets. To avoid such cumbersome management, we make use of a separate overflow array.

The overflow array is constructed in DRAM and is transferred to PCM once buckets start arriving to DRAM for their sort as shown in Figure 4(b). So all the tuples which overflow will at first be stored in DRAM in the overflow array. The number of overflow tuples is expected to be under DRAM size, given the accuracy of the histogram is not so bad. Otherwise, the overflow elements spill over to the disk in the individual overflow files, one for each bucket.

There are two cases that need to be handled during the actual sort of the buckets on PCM. The first of them is when the overflow is restricted to main memory alone, which means the overflow array doesn't exceed the DRAM size.

In such a case, the overflow array is sorted in DRAM first according to bucket id's and next according to the sort attribute, before transferring it to the PCM. Before the first PCM-resident bucket data moves over to DRAM for sorting, the sorted overflow is also prepared for movement to PCM in the void space. This swap of memory blocks takes place using some vacant input or output buffer as the intermediate swap media. The crucial condition for this to happen is that the first bucket is always depth-bound. Otherwise, if the bucket is width-bound, the count array needs to be constructed in DRAM and the overflow elements have no place to go. It is always theoretically possible to make a bucket depth bound by enumerating elements during histogram construction, than to make it width bound as the values of the elements and distribution are beyond our control. Once the overflow array is transferred to PCM, it will stay there till the sort of all the PCM-resident buckets finishes.

Sorting during in-memory overflow If the bucket is width bound, a counting array is constructed over the data present in PCM buffers as well as the overflow array and an in-place counting sort is performed on this aggregate array.In case of a depth bound bucket, the bucket data in DRAM is first quick sorted and merged with the already sorted overflow data belonging to the same bucket residing in PCM (See Figure 5(a)).



(a) Handling main memory overflow for a bucket.

(b) Overflow aware replacement selection.

Fig. 5. Advanced PCM sort-aware scheme

Sorting for disk overflow For buckets whose overflow elements are on disk, neither quick sort nor counting sort is applicable. So, a variant of replacement selection is applied. Our algorithm is different from the conventional replacement selection as we have two main memory resident data structures that need to be maintained. One is the overflow array and another one is the minimum-heap constructed over the bucket data present in DRAM. But sorting the overflow array by constructing a heap on it is avoided as it incurs additional PCM writes.

It is important to note that the overflow array was previously sorted when it was formed inside DRAM. Now it is no longer sorted as new elements keep entering it from the disk. And moreover we have overflow elements belonging to several buckets keep arriving into the PCM-resident overflow array.

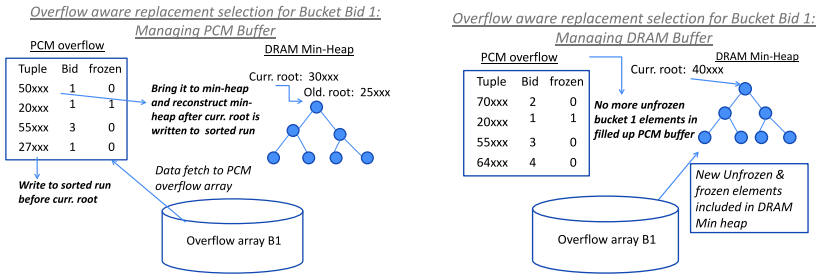
So in addition to bucket id (*bid*), we maintain one more field called "$frozen$" which accepts a boolean value (see Figure 5(b)). If an overflow element belonging to any bucket has left the overflow array, it makes space for the disk resident overflow tuples belonging to the current bucket. Though these elements arrive in unsorted order, they belong to the current run as along as they are $\geq$ the minimum value (or root) that has just left the min-heap. Else the element is marked as frozen and belongs to the next run.

Once there is no more space for any more disk elements to enter the PCM overflow array, the bucket elements present in the min-heap of DRAM will start initiating the replacement selection. The root of the min-heap will scan the PCM-resident overflow array to know if any elements $\leq$ itself belonging to the same bucket are present. If yes, those elements are sent out to the sorted run before the current root. Once the current root leaves the min-heap, another element from the overflow array is brought into the DRAM for min-heap reconstruction and the fetch of overflow elements from disk into the overflow array is repeated provided it has space. If the overflow array is filled with other bucket values and there no more elements in the overflow array belonging to the current bucket, our overflow based replacement selection reduces to the conventional replacement selection. And the elements from the disk are fetched straight into the DRAM heap. Finally the frozen elements in the overflow array and in the DRAM belonging to the current bucket will get unfrozen to resume min-heap reconstruction.

The bottomline is, to have a run at least as long as the traditional replacement selection and if possible a longer run, we utilize the overflow space in PCM and the DRAM buffer to implement our variant of replacement selection. So we manage two buffers, one in PCM and one in DRAM, while fetching data from disk for replacement selection. During this, frozen elements are created in both the buffers. Though the PCM overflow space is an asset that can be exploited to produce longer runs, we have to conduct a few additional scans (reads) on it to avoid PCM writes. Figure 6(a) portrays elements arriving from disk to the PCM buffer and being assigned a frozen value depending on their comparison with the min-heap root present in DRAM. Likewise, Figure 6(b) shows elements from disk directly arriving into the DRAM heap following the absence of unfrozen current bucket elements and space in the PCM buffer.

3 Performance Study

Our PCM simulator is actually DRAM based and uses the measures from [1] to simulate the PCM write latency on DRAM. By default our hybrid memory architecture reserves 3% of the simulatorś main memory obtained from actual DRAM for simulated DRAM and 97% behaves like PCM.



(a) Managing PCM buffer (b) Managing DRAM buffer

Fig. 6. Managing buffer for replacement selection

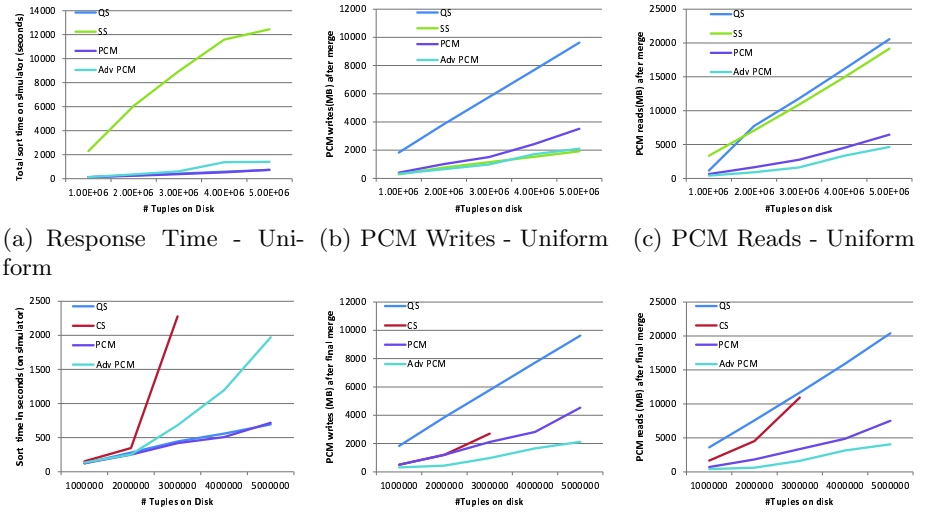
The experiments were run on a PC with Intel(R) Xeon(R) 2.33GHz CPU. All the experiments are performed with a default simulator memory size of 1,000,000 tuples with each tuple being 100 bytes wide. The experiments are conducted on data with uniform and non-uniform distribution. The default file size is kept at 1 million tuples.

Our basic and advanced PCM aware sort schemes were compared against quick sort, selection sort and counting sort. Our comparison is with the sort having best running time at one end and other existing sorts which can potentially provide few PCM writes at the other end of the spectrum. For fair comparison we allocate the total PCM + DRAM main memory to all other schemes we compare with. Due to space constraint, we only present representative results here.

3.1 Uniform distribution

Figure 7(a), 7(b) and 7(c) compare the various schemes in terms of their efficiency (time), total number of PCM writes (after the final merge) and total number of PCM reads (after the final merge) respectively. The data size varies from 1 to 5 million tuples, and the data are uniformly distributed data. Selection sort takes the longest sorting time as against quick sort which is the fastest. Beyond an unsorted disk file size of 5,000,000 tuples, selection sort takes longer than 4 hours of sort time. As expected, quick sort is the weakest in write endurance and incurs lot of writes and reads (during and after merge). Selection sort, though good in writes, performs poorly with respect to PCM reads. It is interesting to note that there are no readings for Counting Sort in Figure 7 owing to its inability to deal with uniform distribution. Counting sort creates numerous runs with tiny run size that it takes too long to finish run generation and merge. The runs are small because, the incoming block has its range too wide in the case of a uniform distribution. So counting sort ends up fetching small blocks to let their count array fit into the main memory.

PCM-aware sort(basic) comes close to quick sort in running time, and also good in reads but it is worse than selection sort with respect to PCM writes.



(a) Response Time - Uniform (b) PCM Writes - Uniform (c) PCM Reads - Uniform

(d) Response Time - Nonuniform (e) PCM Writes - Nonuniform (f) PCM Reads - Nonuniform

Fig. 7. Comparison among various sorting schemes

This can be attributed to the writes expended in shuffling of data after the construction of histogram. Since advanced PCM sort was designed with an aim to get rid of those writes, it performs as well as selection sort in terms of writes. It also incurs the least PCM reads.

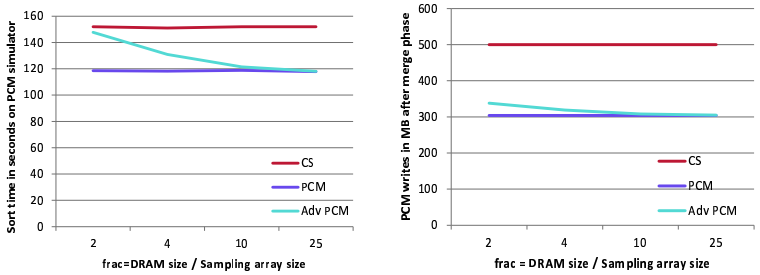
3.2 Non-uniform distribution

For non-uniform distribution, we use the normal distribution. The skew is set to a default value by fixing the values of mean and variance. While mean is set to $(min+max)/2$, standard deviation is set to $(max-min)/k$ by basic definitions. In our default settings, $k = 100$. Figure 7(d), 7(e) and 7(f) shows the response time, PCM write counts and PCM read counts for the various schemes. The values of selection sort are not plotted in Figure 7 because the performance of selection sort remains similar to that of the uniform distribution. Counting sort is applicable here as it is sensitive to the distribution of elements. However, in our experiments, counting sort sustains upto an unsorted disk file size of 3 million as it can be seen from the figures. Though the counting sort PCM writes during run creation are good (not shown owing to space constraint), the overall writes after the final merge phase are worse. This is because of the multiple merges counting sort undergoes owing to the multiple small runs it produced during run creation phase. The expensive merge phase also causes counting sort to have a long sorting time. Our basic PCM aware sort scheme performs well with respect to sort time and also incurs reasonably small PCM writes and reads. But our advanced PCM sort scheme, though with a penalty of extra sorting time

than our basic scheme, outperforms all other schemes in PCM writes and reads by aggressively reducing them to a minimum.

3.3 Varying the extent of sampling

Sampling array that is constructed in the DRAM for the construction of histogram can have major impact on our advanced PCM-aware scheme alone. Figures 8(a) and 8(b) show that advanced PCM aware sort scheme is the only sensitive scheme to this sampling size variation. This is because, the sample size determines the number of random reads that the advanced scheme does in advance to fetch the sample array from the disk to DRAM and eventually preconstruct the histogram. Overflow elements to disk increase if the sampling is poor. But here the overhead in random disk reads to construct a large sample array outweigh the savings obtained by accuracy from it. We can see the decline in the sorting time and PCM writes count of the advanced PCM sorting scheme as the sample array gets smaller. Counting sort is plotted just for reference.



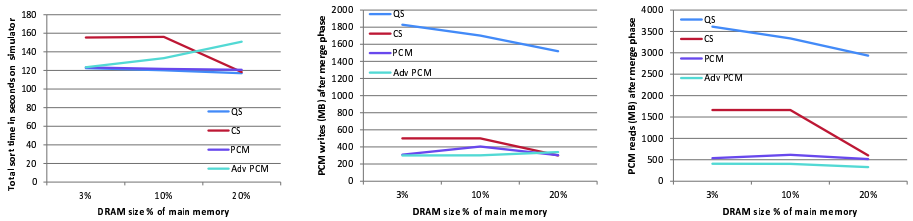
(a) Sort time (Non-Uniform) (b) Total PCM writes

Fig. 8. Effect of sampling

3.4 Varying DRAM size

DRAM size influences every scheme because it helps alleviate PCM writes in all the schemes. Figure 9 shows the results. We do not present the results for the selection sort because of its long running time. As we know counting sort can perform well in special cases when there is a large DRAM to fit the count array of the entire memory block. As shown in the result, counting sort is faster when a DRAM as large as 20% of the main memory is available. Beyond 10% of DRAM size, the sample array becomes large demanding more random reads from advanced PCM sort. This is unrealistic as such a large DRAM buffer is not good for a hybrid PCM architecture because it defeats the purpose of using PCM as main memory. Basic PCM sort performs well with respect to time, PCM reads and writes. As usual, though advanced PCM sort takes longer to sort, it

aggressively reduces PCM reads and writes. Counting sort performs poorly in PCM reads. Realistically, if we consider the interval of 3% to 10% for DRAM buffer size, advanced PCM sort emerges the overall winner.



(a) Sort time (b) Total PCM writes (c) Total PCM reads

Fig. 9. Effect of DRAM size (non-uniform distribution)

4 Conclusion

In this paper, we have proposed two PCM-aware sorting algorithms that can mitigate writes on PCM by efficient use of the small DRAM buffer. Our performance evaluation shows that the proposed schemes can significantly outperform existing schemes.

References

1. Shimin Chen and Phillip B. Gibbons and Suman Nath.: Rethinking Database Algorithms for Phase Change Memory. In CIDR., pp. 21–31 (2011)
2. Lee, Benjamin C. and Ipek, Engin and Mutlu, Onur and Burger, Doug.: Architecting phase change memory as a scalable dram alternative. In ISCA., pp. 2–13 (2009)
3. Moinuddin K. Qureshi and Vijayalakshmi Srinivasan, Jude A. Rivers.: Scalable high performance main memory system using phase-change memory technology. International Symposium on Computer Architecture.(2009)
4. Hamid Mousavi, Carlo Zaniolo.: Fast and accurate computation of equi-depth histograms over data streams. In EDBT., pp. 69–80 (2011)
5. Gregory Piatetsky-Shapiro, Charles Connell.: Accurate Estimation of the Number of Tuples Satisfying a Condition. SIGMOD Conference., pp. 256–276 (1984)
6. Ramos, Luiz E. and Gorbatov, Eugene and Bianchini, Ricardo.: Page placement in hybrid memory systems. In ICSC., pp. 85–95 (2011)
7. Goetz, M.A.: Internal and tape sorting using the replacement-selection technique. Commun. ACM(1963) 201-206
8. G. Graefe. Implementing sorting in database systems. ACM Computing Surveys (CSUR), 38(3), 2006.
9. D. Knuth. The Art of Computer Programming, volume 3 Sorting and Searching. Addison-Wesley, 2nd edition, 1998.
10. Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein.: Introduction to Algorithms, second edition.(2001)