Write-limited sorts and joins for persistent memory

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

To mitigate the impact of the widening gap between the memory needs of CPUs and what standard memory technology can deliver, system architects have introduced a new class of memory technology termed persistent memory. Persistent memory is byte-addressable, but exhibits asymmetric I/O: writes are typically one order of magnitude more expensive than reads. Byte addressability combined with I/O asymmetry render the performance profile of persistent memory unique. Thus, it becomes imperative to find new ways to seamlessly incorporate it into database systems. We do so in the context of query processing. We focus on the fundamental operations of sort and join processing. We introduce the notion of write-limited algorithms that effectively minimize the I/O cost. We give a high-level API that enables the system to dynamically optimize the workflow of the algorithms; or, alternatively, allows the developer to tune the write profile of the algorithms. We present four different techniques to incorporate persistent memory into the database processing stack in light of this API. We have implemented and extensively evaluated all our proposals. Our results show that the algorithms deliver on their promise of I/O-minimality and tunable performance. We showcase the merits and deficiencies of each implementation technique, thus taking a solid first step towards incorporating persistent memory into query processing.

1. INTRODUCTION

Persistent memory is a new class of memory technology that has the potential to deliver on the promise of a universal storage device. That is, a storage device with capacity comparable to that of hard disk drives; and access latency comparable to that of random access memory (DRAM). Database systems, as one of the prime consumers of this technology, must be prepared for this transition if they are to sustain the high performance users have come to expect. Therefore, database developers need to optimize query processing operations for persistent memory. Likewise, it is necessary to introduce abstractions that will incorporate this technology in an informed way into the processing stack of database systems. As this technology rapidly evolves, the abstractions should be resilient to future trends and be system- and user-tunable.

Proceedings of the VLDB Endowment, Vol. 7, No. 5

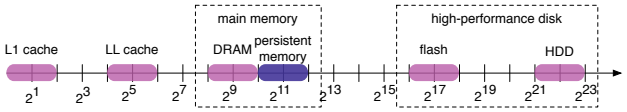


Figure 1: Typical access latency in processor cycles on a 4GHz processor (figure adapted from [22])

The need for persistent memory is practical. The increase of the number of cores per CPU dictates that the memory system primarily scale in terms of capacity as it must cater for the working sets of all concurrently executing processes across all cores; and secondarily in terms of data transfer rate to keep up with the increased demand. The growth rate of the number of cores per CPU is higher than the growth rate of DRAM capacity, and that gap only widens [13]. System architects have thus worked on memory technologies that deliver performance comparable to DRAM but at much higher capacities. Persistent memory (also referred to as non-volatile memory) is an umbrella term encompassing all such efforts (*e.g.*, phase-change memory). In terms of access latency, persistent memory sits between DRAM and block-based flash memory, as shown in Figure 1. Thus, persistent memory is a new level in the memory hierarchy, the design space of which is only now starting to be explored.

There are various technical reasons why persistent memory warrants a study of its own. Foremost, persistent memory is byte-addressable. This is in stark contrast to block-addressable flash memory. The block-oriented techniques that have been proposed for leveraging flash memory are inapplicable (see, *e.g.*, [17] for a review of such techniques). Then, persistent memory latencies are closer to DRAM than flash memory. The read latency is only 2-4 times slower than DRAM compared to the 32 times slower-than-DRAM latency of flash [22]. At the same time persistent memory exhibits the write performance problems of flash memory: writes are more than one order of magnitude slower than DRAM, and thus more expensive than reads [17]. Persistent memory cells also have limited endurance, which dictates wear-leveling data moves across the device to increase its lifetime, thereby further amplifying write degradation. Thus, persistent memory should be treated neither as byte-addressable DRAM nor as block-addressable flash memory; while it exhibits some of the merits and deficiencies of both.

The properties of persistent memory require revisiting existing work and optimizing it for the new medium. It is imperative that new algorithms and techniques are developed if database systems are to make the best possible use of this new technology; otherwise, they are doomed to the suboptimal performance that stems from false assumptions. Our key objective is to optimize writes as they manifest the performance problems due to both byte addressability and write/read asymmetry. The byte addressability of persistent memory renders flash-centric, block-based techniques inappli-

cable; while main-memory techniques do not differentiate between write and read cost asymmetry. To address these issues we will present a host of techniques that we term *write-limited*, which aim to seamlessly incorporate persistent memory in the data management stack. We will focus on key processing operations, namely sorts and joins, that are necessary for high-performing query evaluation. At the same time, we will present abstractions to introduce persistent memory into the system in ways that allow both the system and the developer to optimize performance.

Contributions and organization. Our contributions and the structure of the rest of this paper are as follows:

- We devise sort and join algorithms that minimize I/O by trading expensive writes for cheaper reads (Section 2). Additionally, the algorithms allow the developer to tune their write intensity for a small hit on performance.
- We provide ways to implement these algorithms by proposing a flexible API (Section 3.1). Our API records a blueprint of each algorithm's computation and enables the system to dynamically decide whether to trade writes for reads.
- We present four alternative implementations to incorporate persistent memory into the processing stack of a query processor (Section 3.2). The implementations conform to our proposed API and adhere to a common abstraction. They have been selected to showcase the duality of persistent memory as a non-volatile storage medium with performance characteristics close to volatile memory.
- We experimentally evaluate our algorithms and implementation alternatives in a variety of scenarios (Section 4). Our results show that it is indeed possible to have efficient sort and join algorithms that minimize the number of write operations without compromising performance. Our results also quantify the impact of implementations on performance and point out the subtleties in incorporating persistent memory in the data processing stack.

Finally, we present related work in Section 5 and conclude and identify future work directions in Section 6.

2. ALGORITHMIC FRAMEWORK

Our algorithms are based on trading writes for reads. There are two classes of algorithms. In the first class the computation is split into two parts: (*a*) a write-incurring part; and (*b*) a write-limited part that performs minimal writes. Such algorithms allocate different portions of the input to the write-incurring and the write-limited parts. The portion allocation can be either informed, through a cost model that minimizes the total I/O cost; or user-driven, allowing the user to set the write intensity of the algorithm. The second class of algorithms is based on lazy processing. Lazy algorithms keep track of the penalty being paid by performing extra reads and the manifested savings. Once the penalty plus the cost of generating an intermediate result exceed the savings, the algorithms generate the intermediate result and revert to being lazy.

Throughout the presentation we assume that persistent memory I/O takes place in units we term *buffers*. Though persistent memory is byte-addressable, most systems will perform I/O in larger chunks to amortize costs. These chunks are not as big as standard database pages (*i.e.*, four or eight kilobytes) but are equal to some small multiple of the word size. Typically, they will be equal to the cacheline size (*i.e.*, 64 or 128 bytes). Reading a chunk costs r cost units, while writing it costs w units; $\lambda = {}^{w}/_{r}$ is the write/read cost ratio; $\lambda > 1$. We will also be doing away with ceiling and floor functions. Doing so, though not strictly correct mathematically, simplifies the analysis: as the buffer size is small, the error margin in omitting floor and ceiling functions is quite small too. We will start with sorting before expanding to join processing.

2.1 Sorting algorithms

2.1.1 Segment sort

The starting point is traditional external mergesort. External mergesort proceeds by splitting the input into chunks that fit in main memory, using an in-memory algorithm to sort the values in the chunk, and then writing the sorted chunks to disk as a run. Runs are then merged in passes to produce the sorted output. The number of merging passes is dictated by the amount of available memory. Assume there are M buffers available for sorting; for a relation T of $|T|$ buffers, the size of each run will be M for a total number of ${}^{|T|}/_{M}$ runs. During the merging phase we can have at most M runs open; the number of merging passes will be equal to $\log_M |T|$. In each merging pass the input will be fully read and written; the cost of each pass will be $r+w = r(1+\lambda)$. The total cost of the algorithm is then $|T|r(1+\lambda)+\log_M |T|r(1+\lambda) = |T|r(1+\lambda)(\log_M |T|+1)$.

Consider now a generalization of selection sort which, at a cost of extra reads, writes each element of the input once at its final location. For a memory budget of M buffers, this algorithm works in multiple passes, generating a run during each pass. During the first pass it scans the input to identify the M minimum values. This can be achieved by maintaining a heap of values, *e.g.*, a max-heap when sorting in ascending order. For each value $t \in T$ either: (*a*) t is less than the current maximum, so the value belongs to the current run; or (*b*) t is greater than or equal to the maximum, so the value belongs to the next run. This is reminiscent of run generation during external mergesort with replacement selection. When the input is exhausted the contents of the heap are sorted and written. When writing we keep track of the maximum element and its position in the input (which is recorded whenever the maximum heap element is updated). During the next scan two more conditions are added for an element to be inserted into the heap: (*a*) its value must be greater than or equal to the maximum of the previous run; and (*b*) its position must be greater than the position of the maximum element of the previous run. These conditions ensure there is no overlap between runs. All subsequent iterations check all four conditions before adding an element to a run. For an input T the algorithm will perform $|T|{}^{|T|}/_{M}$ read passes over the input and $|T|$ writes for a total cost of $|T|{}^{|T|}/_{M}r+|T|w = r|T|({}^{|T|}/_{M}+\lambda)$.

Let us now combine the two algorithms into a new one, which we term *segment sort*. Let $x \in (0,1)$ be the fraction of the input that will be sorted using external mergesort; the remaining $(1-x)\%$ of the input will be turned into a longer run using selection sort. We call x the *write intensity* of the algorithm. The input is split into two *segments*, each processed by a different algorithm. Runs will be merged using the standard merging phase of external mergesort. We assume external mergesort will execute first though this restriction can be easily lifted. Let us further assume that we will materialize the output (though it may well not need be materialized if it is to be pipelined to subsequent operators). The total cost S^h of this algorithm will be dependent on x and will be given by Eq. 1.

$$\begin{aligned} S^h(x) =& x|T|r(1+\lambda)+(1-x)|T|r({}^{(1-x)|T|}/_{M}+\lambda) \\ &+|T|r(1+\lambda)\log_M({}^{x|T|}/_{2M}+1) \end{aligned} \tag{1}$$

The first factor of the sum is the cost of generating the runs through replacement selection in external mergesort; the second factor is the cost of generating the longer run through selection sort; the third factor is the cost of merging all runs assuming that external mergesort generates runs that are, on average, twice the amount of main memory. To simplify the analysis assume that $\log_M({}^{x|T|}/_{2M}+1) \approx$

$\log_M(x|T|/2M)$, which is true for large values of $|T|$. After factoring common terms the cost of the algorithm is given by Eq. 2.

$$\begin{aligned} S^h(x) =& |T|r(x+\lambda) + |T|^2 r/M(x^2 - 2x + 1) \\ &+ |T|r(\lambda+1)\log_M(x|T|/2M) \end{aligned} \tag{2}$$

Our aim is to minimize $S^h(x)$; that is, $S^h_x(x) = 0$, or: $|T|r + (2x-2)|T|^2 r/M + \frac{(\lambda+1)}{\ln M}\frac{1}{x}|T|r = 0$; Factoring out $|T|r$ and since $|T|r \neq 0$ we need to solve Eq. 3 for x.

$$2(x-1)\frac{|T|}{M} + \frac{(\lambda+1)}{\ln M}\frac{1}{x} = 0 \tag{3}$$

The resulting quadratic equation has the two solutions given by Eq. 4. The second solution is clearly negative, so the plus-sign solution is the only admissible value for x.

$$x = \frac{-\ln M|T| \pm \sqrt{\ln M\,(\ln M|T|^2 + 2|T|M\ln M - \lambda M^2)}}{M\ln M} \tag{4}$$

Sanity checking. Apart from the second derivative $S^h_{xx}(x)$ being positive making this a minimum value, a few other constraints must hold. Firstly, the square root in Eq. 4 must be positive, which, after factorization, results in $\lambda < \frac{\ln M|T|(|T|+2M)}{M^2}$. Assume that $|T| = \beta M$ for some value $\beta > 1$. The inequality is rewritten as $\lambda < \beta(\beta+2)\ln M$ which holds for all realistic values of λ. Secondly, $x \in (0,1)$ must hold. For $x > 0$ to hold the numerator must be positive, so:

$$\ln M|T| < \sqrt{\ln M\left(\ln M|T|^2 + 2|T|M\ln M - \lambda M^2\right)}$$

must hold. Both sides are positive so we square them:

$$\ln^2 M|T| < \ln M\left(\ln M|T|^2 + 2|T|M\ln M - \lambda M^2\right)$$

and, after simplification, the inequality holds if $\lambda < \frac{2\ln M|T|}{M}$. Assuming again that $|T| = \beta M$ we obtain that $\lambda < 2\beta\ln M$ must hold. This is again true for most realistic values of λ, though it is a tighter bound than before. Finally $x < 1$ must hold, which means that:

$$\sqrt{\ln M\left(\ln M|T|^2 + 2|T|M\ln M - \lambda M^2\right)} < \ln M(M + |T|)$$

must be true. After squaring both sides and simplifying the result is that $\lambda > -\ln M$ must hold, which is always true. From the above we conclude that for the algorithm to be applicable $\lambda < 2|T|/M\ln M$ must hold (obtained by substituting β with $|T|/M$).

Choosing segment algorithms and generalizing. We have so far assumed that the first segment of the file is sorted using external mergesort and the second using selection sort; this may well be inversed. In terms of the chosen percentage it is likely that x will be greater than 0.5; otherwise the quadratic contribution of the selection sort scans will quickly surpass the savings due to avoiding writes. One can devise a second version of segment sort that does not minimize response time; rather, it does not surpass a specified number of writes. If we set x to zero then external mergesort is not executed at all and the algorithm performs the minimum number of writes: as many as there are buffers in T. We can relax this minimality requirement and allow a variable number of extra writes by manually setting x. Roughly, each percentile of the input allocated to external mergesort will result in corresponding extra writes: it will need to be sorted using external mergesort, while the results of the two sorted segments will need to be merged for the final output.

2.1.2 Hybrid sort

We introduce a variant of segment sort, shown in Algorithm 1, that is reminiscent of hybrid hash join. The memory M is split into the selection region R_s and the replacement selection region R_r. The selection region R_s is first filled up with input records and turned into a max-heap, which will contain the smallest records of the input. Once the selection region is full the rest of the input is scanned. Each new record $t \in T$ is inserted either into the selection region or in the replacement selection region. Let m be the maximum in the R_s heap. If $t \leq m$, then t is one of the smallest values encountered so far. So we extract m from the R_s region and replace it with t. We then insert m into the replacement selection region R_r. If $t > m$ it is inserted into the replacement selection region.

Algorithm 1: hybridSort(T, M)

```
input : Relation T to be sorted; memory M for the two regions; x
        percentage of M to be allocated to the selection region
output: T', the sorted version of T
|R_s| = ⌊xM⌋; |R_r| = M − |R_s|;
read |R_s|/|t| records into R_s and turn them into a max heap;
while t ≠ null do
    if t < R_s.max then
        m = R_s.pop(); insert t into R_s; t = m;
    if R_r.current.size() + R_r.next.size() < |R_r| then
        insert t into R_s.current;
        if R_r.current.size() = |R_r| then heapify(R_r.current) ;
    else
        n = R_r.current.pop(); write n to current run;
        if t ≥ n then R_r.current.push(t) ;
        else insert t into R_r.next ;
        if R_r.current is empty then
            close current run and start new run;
            R_r.current = R_r.next; R_r.next = ∅;
            heapify(R_r.current);
sort R_s and write to output;
sort R_r.current and write to current run;
sort R_r.next and write to a new run;
merge all remaining runs;
```

The replacement selection region R_r is organized as the two-heap structure of external mergesort with replacement selection. R_r is split into two parts: R_r.current for the current run and R_r.next for the next run. Initially, the whole of the R_r region is allocated to R_r.current and it is organized as a min-heap. New records to be inserted into the region go into R_r.current until the heap is full (*i.e.*, it occupies all its allotted space). From then on, for each new record t to be inserted into R_r we pop the minimum value n from R_r.current and place it in the current run. If $t \geq n$ it belongs to the current run so we push it into R_r.current. If $t < n$ then it belongs to the next run, so we reduce the current run's heap size by one element, increase the space allocated to the next run by one element, and insert t there. At some point R_r.current will be empty. We then: (*a*) close the current run, (*b*) open a new run, (*c*) heapify the space allocated to the next run, (*d*) turn that heap into the current heap, (*e*) set the space allocated to the next run to zero, and (*f*) continue as before.

2.1.3 Lazy sort

The lazy sort algorithm is based on the second phase of segment sort. The optimal algorithm for write minimization is cycle sort [10], which performs exactly one write for each element of the input. However, it does not constrain the number of reads. Our write-limited sort algorithms, given a budget of M buffers, continuously scan the input to extract the next set of minimum values to be appended to the output; each scan processes the entire input. An alternative would be to extract not only the set of minimum values from the input, but also the set of values that are necessary to produce the next set of minimum values. This is possible to achieve by

Algorithm 2: lazySort(T, M)

input : Relation T to be sorted; memory M for the heap
output: T', the sorted version of T; T_i is a potential intermediate result

```
1  n = 1; maxKey = ⊤; maxPos = ⊥;
2  while t ≠ null do
3      clear M; t = first record of T;
4      if n ≥ ⌊|T|λ/M(λ+1)⌋ then materialize = true ;
5      p = 0;
6      while t ≠ null do
7          if maxKey ≤ t ≤ M.max.val and p > maxPos then
8              top = M.pop(); insert (t, p) into M;
9              if materialize then append top.val to T_i ;
10         advance t; p++;
11     maxKey = M.max.val; maxKey = M.max.pos;
12     sort M and append to T';
13     if materialize then T = T_i; n = 0;
14     n++;
```

the lazySort() algorithm of Algorithm 2. The algorithm tracks the current iteration (*i.e.*, the number of full scans it has performed so far), the benefit of not materializing the input for the next scan, and the penalty it has paid by rescanning the input. In each iteration the algorithm compares the cost of materializing the next input to the cost of rescanning the current input. If the rescanning cost exceeds the materialization cost, then the algorithm materializes the next input; else it proceeds as before. Let n be the current iteration; up to this iteration, $(n-1)M$ buffers have been extracted from the input; during this iteration M further buffers from input T will be extracted; thus, the remaining input is equal to $|T|-nM$ buffers. The cost of writing that is $(|T|-nM)\lambda r$. If it is not written, then during the next iteration nM extra buffers will be read. Therefore, the algorithm should materialize the input when Eq. 5 holds.

$$(|T|-nM)\lambda r \leq nMr \Rightarrow n = \left\lfloor {}^{|T|\lambda}/_{M(\lambda+1)} \right\rfloor \tag{5}$$

This process is progressive: after materialization, n is recomputed as $|T|$ has changed; the algorithm then reverts to being lazy.

2.2 Join processing

2.2.1 Hybrid Grace-nested-loops join

Grace join and standard nested loops join can be straightforwardly combined for equi-join processing. The computation is split into two phases: a write-inducing phase based on Grace join and a read-only phase based on nested loops. Given inputs T and V with $|T| \leq |V|$, let x be the percentage of T and y the percentage of V that will be processed using Grace join. We are given a memory budget of M buffers for the computation and assume that Grace join is applicable, *i.e.*, $M > \sqrt{f|T|}$ where f is the increase in the sizes of partitions due to building a hash table for them during the second phase of Grace join. The number of partitions is ${}^{|T|}/_{M}$; λ is the write to read ratio of the medium. The algorithm progresses as follows. First, $x|T|$ records are scanned and partitioned; let $T_x \subset T$ be that part, and $T_{1-x} = T - T_x$ be what remains. Similarly, $y|V|$ records are scanned and partitioned, where V_y corresponds to that part of V and $V_{1-y} = V - V_y$ is what remains. The partitioned parts of the inputs will be processed using Grace join, which means that they will be scanned one more time, for a total of two reads and one write per part of each input. Thus, the total cost of this phase is $2(rx|T|+y|V|)+\lambda r(x|T|+y|V|) = r(2+\lambda)(x|T|+y|V|)$. In the second phase we need to compute three partial join results for the complete result: $T_x \bowtie V_{1-y}$, $T_{1-x} \bowtie V_y$, and $T_{1-x} \bowtie V_{1-y}$. The first partial join result can be piggybacked onto the Grace join computation. When processing partition p of T, we also scan V_{1-y}. The cost will be $({}^{rx|T|}/_{M})(1-y)|V|$ since we iterate over the number of partitions and each partition has size approximately equal to M. The remaining two partial results are the equivalent of $T_{1-x} \bowtie V$ for a cost of $r(1-x)|T| + {}^{r(1-x)|T|}/_{M}|V|$, *i.e.*, scanning T_{1-x} and performing block nested loops between T_{1-x} and V with a block size of M. The total cost J^h of the computation after factorization and simplification is given by Eq. 6.

$$\begin{aligned} J^h(x,y) = r\big(&(2+\lambda)(x|T|+y|V|) \\ &+ (1-x)|T| + {}^{|T||V|}/_{M}(1-xy)\big) \end{aligned} \tag{6}$$

Eq. 6 is parametrized on x and y; we want to minimize $J^h(x,y)$ under the constraints that $x,y \in (0,1)$ and $\lambda > 1$. We compute the first partial derivatives $J^h_x(x,y)$ and $J^h_y(x,y)$ and solve:

$$\begin{aligned} J^h_x(x,y) = 0 &\Rightarrow r(2+\lambda)|T| - r|T| - \frac{r|T||V|}{M}y = 0 \\ &\Rightarrow y = {}^{M}/_{|V|}(\lambda+1) = y_h \end{aligned} \tag{7}$$

$$\begin{aligned} J^h_y(x,y) = 0 &\Rightarrow r(2+\lambda)|V| - \frac{r|T||V|}{M}x \\ &\Rightarrow x = {}^{M}/_{|T|}(\lambda+2) = x_h \end{aligned} \tag{8}$$

We now need the second derivatives to test the critical point. We compute $J^h_{xx}(x,y) = J^h_{yy}(x,y) = 0$ and $J^h_{xy}(x,y) = J^h_{yx}(x,y) = -r|T||V|/M$. The result of the second derivative test yields $J^h_{xx}(x_h,y_h)J^h_{yy}(x_h,y_h) - \left[J^h_{xy}(x_h,y_h)\right]^2 = -({}^{r|T||V|}/_{M})^2 < 0$, which means that the point (x_h,y_h) is a saddle point and not an extremum. However, plotting the function is enough to indicate what happens around the saddle point and thereby guide the choice of x and y. In Figure 2 we plot the cost function as we vary x and y, and as the cardinality ratio between the two inputs $({}^{|T|}/_{|V|})$ and the write inefficiency of the medium scale. We assume that $|T| \leq |V|$ and that $M > \sqrt{1.2|T|}$, *i.e.*, Grace join is applicable and a hash table for a partition is 20% larger than the partition itself. The results are represented as heatmaps with a lighter shade denoting a lower cost and thus better performance. We do not show the actual value as it is irrelevant: we are more interested in trends. The plots indicate certain heuristics for choosing x and y. If the inputs are similarly sized and the medium is not too inefficient,

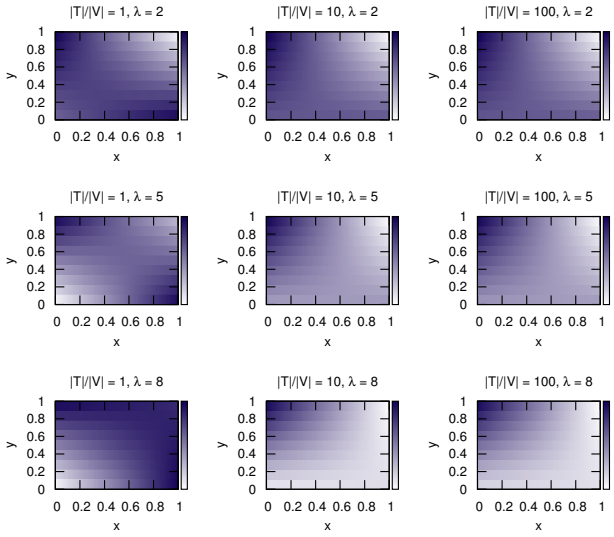


Figure 2: Representation of the hybrid Grace-nested-loops join cost function; a lighter shade denotes better performance

iteration	Standard hash join reads	Standard hash join writes	Lazy hash join reads	Lazy hash join writes	Lazy hash join savings	Lazy hash join penalty
1	$m(M+M_T)$	$(m-1)(M+M_T)$	$m(M+M_T)$	0	$(m-1)(M+M_T)\lambda r$	0
2	$(m-1)(M+M_T)$	$(m-2)(M+M_T)$	$m(M+M_T)$	0	$(m-2)(M+M_T)\lambda r$	$(M+M_T)r$
3	$(m-2)(M+M_T)$	$(m-3)(M+M_T)$	$m(M+M_T)$	0	$(m-3)(M+M_T)\lambda r$	$2(M+M_T)r$
...	...	...	...	...	...	...
i	$(m-i+1)(M+M_T)$	$(m-i)(M+M_T)$	$m(M+M_T)$	0	$(m-i)(M+M_T)\lambda r$	$(i-1)(M+M_T)r$

Table 1: The progress of standard hash join compared to lazy hash join

then we are better off using large values for x and y, *i.e.*, employing Grace join; this is intuitive as Grace join is more efficient than nested loops. If the inputs have similar sizes then the decisive factor is λ, the write to read ratio of the medium. As λ grows the advantage shifts to nested loops. On the other hand, as the ratio between input sizes changes, we can start gradually employing nested loops as the evaluation algorithm. This can be proportional, *e.g.*, moving along the diagonal of each individual plot, *i.e.*, $x \approx y$, as shown in the middle row of plots of Figure 2; alternatively, choosing values in the bottom right triangular region of each plot, *e.g.*, $x+y=1$ and $x \geq y$ is a good rule of thumb.

2.2.2 Segmented Grace join

Let us now assume that we do not account for each input independently, but instead operate at a partition level. Given a number of partitions k we choose to materialize only some number x of them and continuously iterate over the rest of the inputs to process the remaining $k-x$ partitions. The algorithm first scans both inputs and offloads k partitions. Assuming inputs T and V with $|T| \leq |V|$ and also assuming that our memory budget M is greater then $\sqrt{f|T|}$, *i.e.*, Grace join is applicable, then $k = \lceil |T|/M \rceil$. The total cost J^s of the algorithm is given by Eq. 9. The first two factors account for Grace join. We scan the input to extract the x partitions; we offload these partitions; and then read them back to process their partial join. We therefore fully scan T and V and then write and read $x(|T|+|V|)/k$ buffers, where $|T|/k$ (resp. $|V|/k$) is the size of each partition of T (resp. V). The last factor is the cost of iterating over both inputs $k-x$ times to process the remaining partitions.

$$J^s(x) = r(|T|+|V|) + rx(1+\lambda)\left(\frac{|T|+|V|}{k}\right) + r(k-x)(|T|+|V|) \tag{9}$$

The cost is parametrized on x: the number of partitions that will be written. After factoring common terms, Eq. 9 can be rewritten as:

$$J^s(x) = r(|T|+|V|)\,(1+(\lambda+1)x/k + k(1-x))$$

The cost of Grace join is $r(|T|+|V|)(\lambda+2)$; this algorithm performs better if $1+(\lambda+1)x/k + k(1-x) < \lambda+2$ holds, or:

$$x < \frac{(\lambda+1-k)k}{\lambda+1-k^2} \tag{10}$$

Eq. 10 ensures that Segmented Grace join outperforms Grace join. Regardless of outperforming Grace join, the choice of x is a knob by which we alter the write intensity of the algorithm.

2.2.3 Lazy hash join

Given M memory buffers and two inputs T and V with $|T| < |V|$, standard hash join computes the join in $k = \lceil |T|/M \rceil$ iterations by partitioning the inputs in m partitions. During iteration i the algorithm scans T and hashes each $t \in T$ to identify its partition. If t belongs to partition i, the algorithm puts it in an in-memory hash table. If t belongs to any other partition it offloads it to the backing store. The algorithm then scans V and hashes each $v \in V$ to identify its partition. If v belongs to partition i it is used to probe the in-memory hash table; any matches are propagated to the output. If t does not belong to partition i, it is offloaded to the backing store. The algorithm iterates as above until both inputs are exhausted. Thus, M buffers from T and $M_V = \lceil |V|/k \rceil$ buffers from V are eliminated in each iteration. Assume now that the algorithm is lazy: when it comes across a record that does not belong to the partition currently being processed, it does not write it back. Instead, it pays the penalty of rescanning the input during the next iteration.

In Table 1 we show the progression of the lazy algorithm compared to standard hash join. In each iteration the algorithm earns savings; but doing so incurs a penalty during the next iteration. The savings in each iteration are equal to the portion of the input that is not written (hence the multiplication with λr). The penalty is equal to the portion of the input that would not have been read in comparison to standard hash join. The algorithm is better off as long as the savings surpass the penalty. When the savings are less than the penalty plus the cost of materializing the part of the input that will be processed in the remaining iterations, the algorithm should materialize an intermediate input. Therefore, the iteration n at which the penalty surpasses the savings is computed through Eq. 11.

$$nr > (k-n)\lambda r \Rightarrow n > k/(\lambda+1) \Rightarrow n = \lfloor k/(\lambda+1) \rfloor \tag{11}$$

The process is progressive. The algorithm periodically materializes intermediate inputs and then reverts to being lazy.

3. IMPLEMENTATION

Our implementation, shown in Figure 3, treats DRAM and persistent memory as distinct levels of the memory hierarchy. Our algorithms operate at the DRAM level and offload data to persistent memory for later processing; thus, DRAM is the equivalent of a bufferpool in a database system. Data is exchanged in cachelines (termed buffers in the algorithmic framework) between the two levels of memory. Our algorithms have a limited number of DRAM cachelines for their operation. A thin abstract persistence layer sits between DRAM and persistent memory to implement *persistent collections*: sources and/or intermediate results that the algorithms operate on. Persistent collections are organized in blocks that are larger than cachelines to further amortize the persistent memory I/O cost. However, the block size may well be equal to the cacheline size if so desired. In what follows, we present the library support needed for implementing our algorithms. We then focus on the persistence layer and give four methods to instantiate it.

3.1 Library support

Abstract API definition. The premise of write-limited algorithms is that some intermediate results need not be materialized but can be reconstructed from primary inputs. Therefore, we define an API to expose such opportunities to the runtime. Our only assumption is that every collection within a computation has a unique identifier. This can be enforced by the thin persistence layer of Figure 3. The materialization of any collection is by default deferrable unless

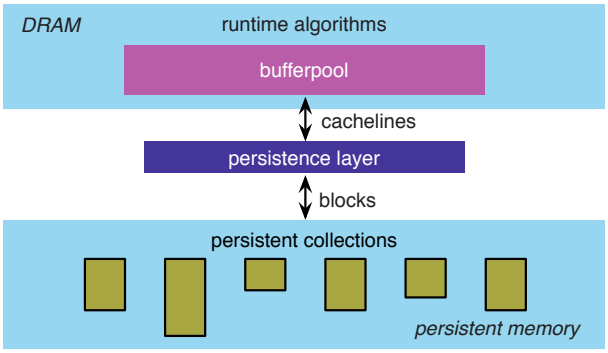


Figure 3: Implementation overview: a thin persistence layer sits between a traditional two-level hierarchy. The runtime algorithms issue calls to the persistence layer, which in turn applies them on collections hosted in persistent memory.

specified otherwise by the programmer. Collections that must be materialized are tagged as such when they are declared. We have a special type of collection that is purely in-memory; such collections are also tagged as such at declaration time. Our API has the following calls: (*a*) `split`(T,n,T_l,T_h): split collection T at position n into T_l and T_h; (*b*) `partition`$(T,h(),k,\langle T_i\rangle,\langle s_i\rangle = |T|/k)$: partition collection T into k partitions T_1 to T_k using $h()$ as the partitioning function; the size of each partition is expected to be s_1 to s_k respectively; the last argument is optional and if omitted each partition is expected to be of size $|T|/k$; (*c*) `filter`$(T,p(),f,T_p)$: filter collection T into T_p using predicate $p()$ and expect the output to be of size $f|T|$ where $f \in [0,1]$; (*d*) `merge`$(T_l,T_r,m(),T)$: merge collections T_l and T_r into T using $m()$ as the merging function. These primitives are enough to implement write-limited algorithms and enable the runtime to perform the optimizations we have described. This is achieved by tracking collection sizes and read/write operations; and dependencies across primary and deferred collections.

Runtime support. To track dependencies between collections we employ a control flow graph. The nodes of the graph are either collections or one of the API calls. Edges from collections to API call nodes mean the collection is the call's input; outgoing edges from API call nodes to collections mean the collection is an output. Each API call node is annotated with call-specific parameters. When the collection is accessed the runtime decides whether the collection should be materialized or not. Simply declaring a collection and how it is constructed does not materialize it; only access to a collection triggers its potential materialization. Upon access, the runtime estimates the number of reads and writes to construct the collection and decides whether deferring materialization is cost-effective. To materialize a collection we start from its oldest materialized ancestor and apply all the computations that construct it. The runtime enforces the constraint that no input is fully scanned twice to materialize its outputs. For instance, consider a `partition`() operation where the materialization of the first few output partitions is deferred. If upon access to a subsequent partition the runtime decides to materialize it, then it must decide to materialize or defer all remaining partitions and materialize the selected ones while it scans the input.

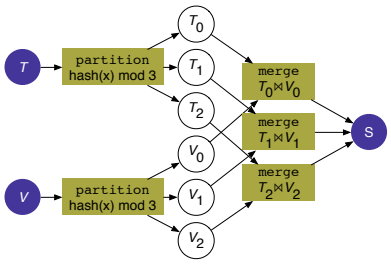


Figure 4: Example control flow graph

An example control graph is shown in Figure 4. The graph corresponds to the segmented Grace join algorithm of Section 2.2.1. Oval nodes are collections, while rectangular nodes are API calls. If a collection node's oval is filled, then this collection is tagged as materialized. Empty collection nodes are deferred. In Figure 4 the inputs T and V are materialized, as is S, the final output of the computation. Inputs T and V pass through a `partition`() operation to produce $T_0 - T_2$ and $V_0 - V_2$. Corresponding partitions are then merged through partial joins, with each partial result appended to S for the final output. Consider reconstructing V_0: the runtime can do so by walking the graph. V_0 depends on V so it can be reconstructed by partitioning V using function hash$(x) \mod 3$, where $x \in V$. The estimated cost of the computation as it results from the graph is $r(|T|+|V|)+(w+r)\sum_{i=0}^{2}(|T_i|+|V_i|)+|S|w$. Factoring out the output materialization cost, and assuming a ratio $\lambda = w/r$, the decision of deferring materialization comes down to choosing the number x of partitions to materialize. If we make the appropriate substitutions then the expression is rewritten as Eq. 9.

Implementation and use of the API. The API calls and the control flow graph act as a blueprint of an algorithm. Each algorithm manifests as a physical operator and is assigned an operator context: an encapsulation of the information necessary to dynamically optimize the operator. Collections accept an operator context as a construction parameter. The `C++` fragment in Listing 1 showcases these properties. `OpCtx` is the operator context type. The API calls are members of the operator context, which has two more methods: `assess`() and `produce`(). Both methods accept as parameter the identifier of a collection. The first method assesses the collection to decide whether it should be materialized; the second method produces the collection. It does so by walking the control flow graph of the operator, as we will shortly see. Collections can be queried on their state (in-memory, materialized, or deferred). When a collection is opened the operator context assesses if the collection should be materialized. If so, the context produces the collection.

```
enum cstatus_t { MEMORY, MATERIALIZED, DEFERRED };
class Collection {
private:
 std::string m_name; // collection name
 OpCtx* m_ctx;       // operator context
 cstatus_t m_status; // collection status
public:
 Collection(const std::string& name, OpCtx* ctx = 0,
            cstatus_t s = DEFERRED);
 ...};
void Collection::open() {
  if (m_status == DEFERRED && m_ctx) { m_ctx->assess(name); }
  if (is_materialized()) { m_ctx->produce(name); } }
```

Listing 1: Collection definition and access

Operators accept their context as a parameter when constructed. They provide a standard iterator interface, as well as an `evaluate`() method that records the control flow graph. This method is called at construction time. The implementation of the method uses the API calls presented earlier, with the additional argument of the operator context. For example, the fragment of Listing 2 records the graph of Figure 4. The `SGJ` class implements the algorithm as a physical operator. Its `evaluate`() method sets up the workflow by declaring the appropriate collections for the partitions; the operator's context `create_name`() method generates a unique identifier. After declaring collections, `evaluate`() makes an API call to the `partition`() primitive passing the hashing function (a reference to the `hash_of`() functor) and the rest of the required parameters. Partitions are pairwise joined afterwards; this is done through an iteration and successive `merge`() calls. One of the parameters to the

`merge()` call is the merging function. A functor implementing this function is shown at the end of Listing 2. It overloads the C++ function symbol and expects the two input and one output collections as parameters. The participating collections are opened so they can be assessed by the operator context and produced if necessary. Then, a hash table is built for the left collection and the right collection is used to probe the hash table for matches (omitted in the code).

```
class Operator {
protected:
 OpCtx* m_ctx; ...
public:
 Operator(OpCtx* ctx, ...): m_ctx(ctx), ... {}
 virtual void evaluate() = 0;
};
class SGJ: public Operator { ... };
void SGJ::evaluate() {
 // assumption: m_left and m_right are the two inputs;
 // m_output is the output collection
 std::vector<Collection*> lp;  std::vector<Collection*> rp;
 for (int i = 0; i < m_partitions; i++) {
   lp.push_back(new Collection(m_ctx->create_name(),
                               m_ctx, DEFERRED));
   rp.push_back(new Collection(m_ctx->create_name(),
                               m_ctx, DEFERRED)); }
 m_ctx->partition(m_left, hash_of(m_parts), m_parts, lp);
 m_ctx->partition(m_right, hash_of(m_parts), m_parts, rp);
 for (int i = 0; i < m_partitions; i++) {
   m_ctx->merge(*lp[i], *rp[i], partition_join(), m_output);
 } }
class partition_join {
 void operator()(Collection& l, Collection& r, Collection& s) {
   l.open(); r.open(); s.open(); // assess and produce
   // build a hash table for l
   while (r.next()) {
     // probe for matches and output into s
   } } };
```

Listing 2: Example definition of an operator

Optimization. We track the accumulated numbers of reads and writes per materialized collection during execution; and trigger materialization by using rules. For each materialized collection, the system maintains a running sum of the number of read cachelines for that collection. The sum is used to decide if it is cheaper to keep a collection deferred and construct it on demand by (re)applying operations; or it is cheaper to materialize it. Rules rely on detecting patterns stemming from the write optimizations of the algorithms. The rules are symbolically named and explained below:

(*a*) *multi-process*: if a collection is processed multiple times then it is materialized only if the number of times it is processed is greater than the write-to-read ratio; this rule applies to the segmented and hybrid sort and join algorithms.

(*b*) *eager-partition*: if the system decides to materialize one of the outputs of a `partition()` operation, then to amortize the write time, all remaining results are materialized; this rule applies to the segmented and hybrid join algorithms.

(*c*) *process-to-append*: intermediate results immediately appended to another collection are always deferred.

(*d*) *read-over-write*: for a deferred collection, compare the cost, C_m, of materializing it to the so-far accumulated read cost, C_r, of its input, plus the read cost, C_c, for constructing it. If $C_m \leq C_r + C_c$ then the collection is materialized and deferred in any other case; this rule applies to the lazy sort and join algorithms.

Consider assessing T_0 in Figure 4. Deferring it saves $|T|/3$ writes at the cost of $|T|$ reads; if $|T| < \lambda|T|/3$ where λ is the write/read ratio, T_0 is deferred. When computing the partial join between T_0 and V_0 the runtime knows, through the reference from a collection to its operator context, that T will be used to produce T_0 by reapplying the partitioning function. Moving on to T_1, the runtime compares $2|T|$ to $\lambda|T|/3$ since we use the accumulated read cost for any materialized source. If $2|T| > \lambda|T|/3$, then T_1 is materialized. If so, then under the *eager-partition* rule, the runtime materializes T_2 as well.

Extensions. We presented the optimization of single operators. However, it is possible to generalize the method to entire evaluation plans, assuming that the operators are connected through intermediate result collections. We have not tested this here as we focus on individual algorithms rather than on entire queries. Incorporating such functionality is straightforward but left for future work.

3.2 Incorporating persistent memory

A salient decision to make when incorporating persistent memory into the programming stack is whether to treat it as part of the filesystem, or as part of the memory subsystem. The first option fully addresses the persistence aspects, but implies the traditional boundary between main memory and secondary storage. The second option makes persistent memory part of the memory hierarchy treated as volatile; thus the system itself must guarantee persistence. Our goal is not to answer the question of which option is better. Rather, it is to showcase the performance of our algorithms under each option. We tested our algorithms over four implementation techniques, each driven by one of these options.

RAM disk. The first approach is to employ a memory-mounted filesystem. RAM disks are complete lightweight filesystems bypassing disk-related overheads. A RAM disk does not provide persistence between reboots, so it never incurs disk I/O; though it implements persistence semantics as long as the filesystem is mounted in main memory. A RAM disk bypasses the filesystem cache: writes and reads are synchronous to the portion of main memory allocated to the RAM disk. Persistent collections in this case are standard files and they are manipulated using filesystem calls. In typical filesystem fashion, files are organized in 512-byte records, which map to the block abstraction of Figure 3. We can increase the block size in the same way an operating system can increase the page size. This is a middle-of-the-road approach to bridging the mismatch between traditional block devices and byte-addressable persistent memory. The utility of this implementation is in identifying the pros and cons of using filesystem practices to access persistent memory.

Byte-addressable filesystem. The second implementation we tested was a filesystem optimized for persistent memory. We used Intel's PMFS, the kernel-level filesystem extension available for the GNU Linux kernel version 3.9 onwards.[1] Another option would be a filesystem like BPFS [5]. We decided to go for PMFS as it is a kernel-level filesystem; BPFS is implemented in user space and that carries additional overhead. Kernel-level filesystems are tightly integrated with the kernel and thus reduce the overhead of system calls, while, at the same time, allow the filesystem to access kernel-specific functionality. PMFS provides low-level fine-grained persistence primitives and implements file-level access through CPU load/store instructions, thereby minimizing overhead. Thus, PMFS pushes the file abstraction to its limits; doing away not only with operating system caching, but also with the block-level interface.

Dynamic arrays. The third option for a persistence layer is to substitute the runtime's memory allocator (*e.g.*, `malloc()`) with one that uses the non-volatile memory for allocations, as opposed to the system's heap (see *e.g.*, [4] for an approach). This affects the memory allocator, but not the way by which data structures allocate memory. The typical data structure to represent an expandable random-access collection of records is a dynamic array, or, in C++ terms, a `vector`. C++ `vector`s have an initial capacity; when that capacity is reached they allocate a memory chunk twice as big as their current capacity; copy the elements over; and release the memory they had previously occupied. The doubling of allocated memory and, more importantly, the copying of elements over are

[1] Available at `https://github.com/linux-pmfs`.

characteristic	value
processor	Intel Xeon E5420 (four cores)
clock speed	2.5GHz per core
I1 cache	32kB per core
D1 cache	32kB per core
L2 cache	2 × 6MB
memory	12GB DDR2, fully buffered

Table 2: Hardware performance characteristics

far from ideal for persistent memory as they incur a large number of writes. This is, however, how dynamic arrays work in most runtimes (*e.g.*, Java `Vectors` and `ArrayLists` operate similarly).

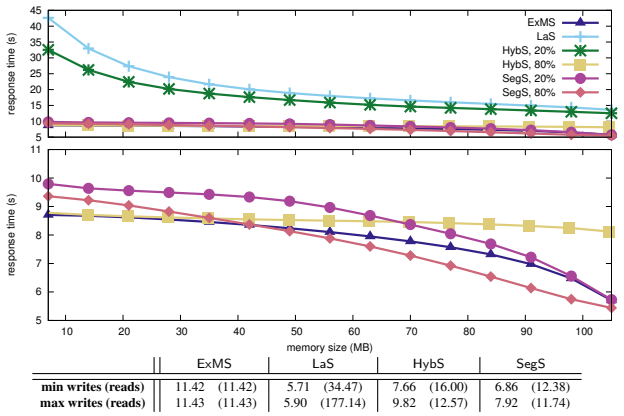
Blocked memory. Finally, we implemented the persistence layer as a mix of the previous options. We kept the interface of a dynamic array, but changed the memory profile of the array to a linked list of memory blocks. Accessor methods over the list of blocks provide byte addressability. Memory is allocated one block at a time with no copying upon expansion. This is effectively an in-memory file representation without the overhead of persistence, whether that is provided by the memory allocator or a filesystem substrate. The only overhead is reading from and writing to persistent memory.

4. EXPERIMENTAL STUDY

Implementation and hardware. We developed our algorithms in C++ and used the language's template mechanisms to eliminate any artificial bloat associated with type genericity. This means that for any data field access we do not perform function calls to retrieve values; we simply dereference a pointer, which aids the compiler and the runtime to better optimize the code and its execution. The code was compiled using g++ version 4.7.3 with the `-O3` optimization flag for maximum code efficiency. We used the 3.9 GNU/Linux kernel, as the public version of PMFS is available for that kernel source tree. Our hardware had the performance characteristics summarized in Table 2. Even though we used a quad-core CPU our implementation was single-threaded and did not make any use of parallelism. Our tests did not perform any disk I/O apart from the necessary for loading the data before processing (which we have factored out in our reported timings). We tested block sizes ranging from 512 bytes (the disk record size) to 8192 bytes. We found an improvement in response time of 10% on average when moving from 512 to 1024 bytes and insignificant improvements beyond that. We therefore report measurements for 1024-byte blocks.

Datasets and metrics. We developed a custom microbenchmark of sort and join operations, as we wanted to test our techniques in a controlled environment and not in the context of a full database server with the intricacies and complexity it introduces. We used a schema of ten eight-byte integer attributes for a total record size of 80 bytes. The key attribute followed the key value permutation of the Wisconsin benchmark [6]. The values of the remaining attributes were computed based on the key attribute through integer division and modulo computations. We instrumented the code to report the response time, and the numbers of cacheline reads and writes. For response time, we ran each operation ten times. We report the average; variance was less than 0.1%.

Methodology. To simulate the read and write latencies of persistent memory we followed the lead of the hardware community [24] and injected artificial delays after read and write operations. We did so at a cacheline granularity. To enforce the delay we used the hardware counters to invoke an idle loop of as many clock ticks as necessary for the desired latency. We used a 10ns read latency and a 150ns write latency [22, 24]; we further experiment with different latencies in the sensitivity analysis of Section 4.2.



	ExMS	LaS	HybS	SegS
min writes (reads)	11.42 (11.42)	5.71 (34.47)	7.66 (16.00)	6.86 (12.38)
max writes (reads)	11.43 (11.43)	5.90 (177.14)	9.82 (12.57)	7.92 (11.74)

Figure 5: Sorting performance for varying memory sizes; writes and reads in millions of cachelines

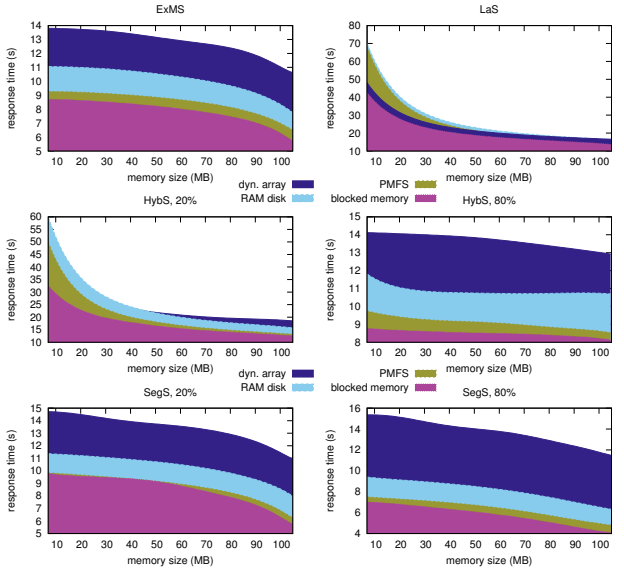


Figure 6: Performance comparison of sorting algorithms under the four different implementation alternatives

4.1 Performance analysis

We first compare the raw performance of the algorithms before analyzing their sensitivity to parameter values. We will be doing so for all four implementations and in a variety of settings.

4.1.1 Sorting

We start with an analysis of our sorting algorithms over a ten-million-record input. The algorithms are summarized as follows: (*a*) ExMS: standard external mergesort using replacement selection; (*b*) SegS: segment sort (Section 2.1.1); (*c*) HybS: hybrid sort (Section 2.1.2); and (*d*) LaS: lazy sort (Section 2.1.3) We first tested the impact of available main memory, as this affects the reduction in the number of writes. We varied the amount of available memory from 1% to 15% of the total input size. SegS is parametrized on the percentage of the input that will be sorted using external mergesort; likewise, HybS is parametrized on the percentage of the main memory that is used as the selection region. We call these percentages the write intensity of each algorithm; the more write-intensive the algorithm the better the performance, at the cost of extra writes.

Performance comparison. In Figure 5 we report the response

time for each algorithm and in the bottom table we give the minimum and maximum number of writes for each algorithm, along with the corresponding number of reads in parentheses. We focus on the blocked memory implementation as it had the minimal overhead (though we will return to this point shortly). We plot the response times of ExMS and LaS; and HybS and SegS for write intensities of 20% and 80%; we will further analyze the impact of write intensity in Section 4.2. We present the overall performance of all algorithms and a zoomed in picture of the four best performing algorithms as LaS and HybS for a 20% write intensity are disproportionately slow and make the performance differences of the remaining algorithms harder to see. Naturally, performance improves as more memory becomes available. Note that even though ExMS is optimized for symmetric I/O, its write-limited competition outperforms it from the beginning. For a small write intensity HybS and SegS incur about 35% fewer writes. As the write intensity grows the algorithms perform at most 15% fewer writes. HybS has comparable performance to ExMS; SegS outperforms HybS by about 30% on average. LaS has the worst response time. It has, however, the best write profile overall by performing about 50% fewer writes than ExMS and up to 30% fewer writes than the best version of SegS. Note also how the write-limited algorithms trade writes for reads: as the number of writes decreases, the number of reads increases. The exception is LaS which always performs approximately the same (and minimal) number of writes; the reduction in the number of reads is due to more memory being available (and hence longer sorted chunks being generated).

Implementation comparison. In Figure 6 we show the overhead of each implementation; each layer in the stack graph represents additional overhead. The blocked memory approach bears the minimal overhead. Its only penalties are the write and read costs of persistent memory. The PMFS implementation approximates the minimal overhead. Exposing byte addressability to the filesystem seems like a viable approach to introduce persistent memory functionality in the processing stack. The next best performing implementation is the RAM disk one. Even though it bypasses the caching overhead of a filesystem, the remaining filesystem overheads and primarily block-level access, suggest that introducing byte addressability at the filesystem level is crucial in order to use filesystem abstractions for manipulating persistent collections. The worst-performing implementation is the dynamic memory one. The reason is the write/read asymmetry of the medium is not exposed. Even though this implementation is still a main-memory based one, *i.e.*, it exhibits the same access overheads as blocked memory, its reallocation and data copying to improve memory access patterns result in excessive writes; this in turn hurts performance. Even for a not as write-intensive an algorithm as SegS with a 20% write intensity, the overhead of the dynamic array implementation may be up to 50% for low memory budgets; or go up to a factor of two for larger memory sizes. While the order of the implementations by performance merit is generally the same, there is one outlier: LaS. There, the memory-based approaches are better than the ones based on a filesystem. LaS bears the minimal number of writes. Thus, the number of expansion operations of the dynamic array is minimal, which in turn means that the write penalty is more-or-less amortized as it is not paid as frequently.

4.1.2 Join processing

We computed the join between a one-million-record (left) input and a ten-million-record (right) one. Each left input record joined with ten right input records. The algorithms are abbreviated as: (*a*) GJ: standard Grace join; (*b*) HJ: simple hash join; (*c*) NLJ: nested loops join; (*d*) HybJ: hybrid Grace-nested-loops join (Section 2.2.1); (*e*) SegJ: segmented Grace join (Section 2.2.2); and (*f*) LaJ: lazy join (Section 2.2.3). HybJ and SegJ are annotated with their write intensity. For HybJ this is the percentage of the left and right inputs handled using Grace join; for SegJ this is the percentage of the number of partitions materialized. For instance, HybJ, 50% - 80% means that 50% of the left input and 80% of the right input are handled using Grace join. The response time is on the *y*-axis and it is plotted against the available memory on the *x*-axis, which ranged from 1% to 15% of the left (smaller) input.

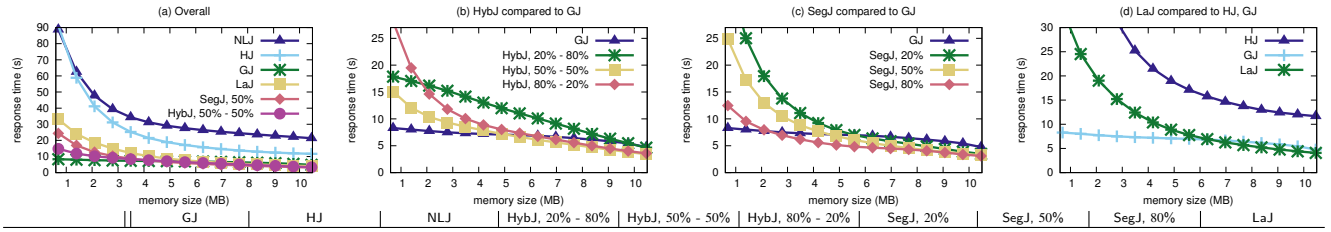
Performance comparison. We first compare the performance of the two versions of HybJ and SegJ for a 50% write intensity across the board; and the performance of LaJ; to NLJ, GJ and HJ in Figure 7(a). The write-limited algorithms quickly catch-up to GJ, the best performing I/O-optimized solution, and outperform it as available memory grows. In Figure 7(b) we compare the performance of HybJ to GJ. The performance of HybJ improves for different combinations of write intensity for its left and write inputs. The write intensity over the right input dictates performance: the more write-intensive the processing of the right input, the quicker the algorithm catches up with GJ for a given amount of memory. At the same time, one can have reasonable performance at a low write intensity over the right input. SegJ usually outperforms GJ and is only suboptimal for a low write intensity or a low available memory size, as shown in Figure 7(c). Note the tradeoff between write intensity and memory: we can obtain good performance at a low write intensity provided we are willing to use more memory. Finally, we focus on LaJ, effectively a variant of HJ. As shown in Figure 7(d) LaJ always outperforms HJ by up to a factor of three for small memory sizes. Also, it converges much sooner to the performance of GJ and surpasses it as available memory grows.

In the bottom table of Figure 7 we show the minimum and maximum cacheline writes of the algorithms, along with the corresponding number of reads in parentheses. It is evident that the write-limited algorithms perform fewer writes than the competition. At the same time, there is a tradeoff between writes and reads. Consider, for instance, an aggressive algorithm like SegJ at a low write intensity like 20%: the number of reads for the maximum number of writes is about one order of magnitude higher than the corresponding figure for GJ. By reducing the number of writes by a factor of two, however, the algorithm exhibits better performance overall. This motif is evident for all write-limited algorithms: they perform fewer writes than the competition, at times approximating the minimal number of writes that a read-intensive algorithm like NLJ guarantees. At the same time, however, they exhibit an inflated number of reads. But because of the write/read asymmetry of the medium this discrepancy does not compromise performance.

Implementation comparison. The results for the algorithms of Figure 7(a) and the four reference implementations are shown in Figure 8. The blocked memory implementation again has the smallest overhead, with the PMFS implementation closely following it. The dynamic memory implementation exhibits the highest overhead in the majority of cases, reaching up to a factor of two for an algorithm optimized for symmetric I/O like GJ. In general, the overheads of the alternative implementations over the blocked memory one are not always as high as was the case for sorting. For instance, the overhead is minimal for the SegJ algorithm with a 50% write intensity, or the LaJ algorithm as memory grows. This is not true for HybJ and a 50% write intensity over each input where the overhead rises to 50% for the dynamic array implementation.

4.2 Sensitivity analysis

We now analyze the sensitivity of the algorithms to their parameters. We begin with the write intensity, which is effectively a 'knob'



	GJ		HJ		NLJ		HybJ, 20% - 80%		HybJ, 50% - 50%		HybJ, 80% - 20%		SegJ, 20%		SegJ, 50%		SegJ, 80%		LaJ	
min writes (reads)	11.23	(11.31)	24.57	(25.14)	5.71	(97.71)	9.43	(39.54)	8.86	(21.78)	7.31	(68.46)	6.50	(44.90)	7.41	(34.86)	8.20	(29.03)	8.86	(42.14)
max writes (reads)	14.00	(14.57)	173.43	(174.00)	5.71	(677.71)	10.40	(62.54)	9.66	(29.71)	8.46	(217.89)	7.88	(158.40)	8.10	(95.38)	9.21	(33.57)	12.48	(149.89)

Figure 7: Performance of the join algorithms; writes and reads in millions of cachelines

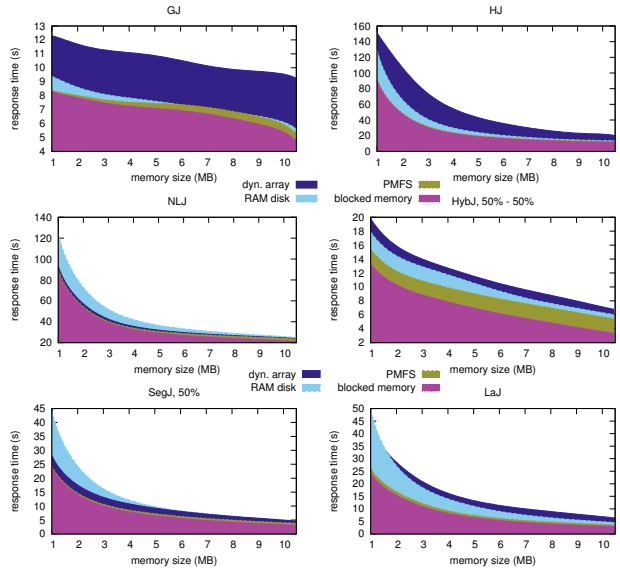


Figure 8: Performance comparison of join algorithms under the four different implementation alternatives

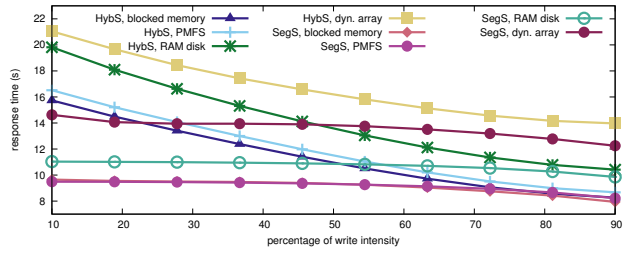


Figure 9: Impact of write-intensity on sorting algorithms

for specific algorithms. We then present the impact of the system-wide write/read asymmetry and the effectiveness of our cost model.

4.2.1 *Impact of write intensity*

For some algorithms, write intensity is tunable: it can either be chosen so that the algorithm is cost-optimal; or to bound the number of writes each algorithm performs with respect to its symmetric-I/O counterpart. In Figure 9 we report the impact of write intensity on the two sorting algorithms affected by this choice, *i.e.*, SegS and HybS. We report this impact on the four persistent memory implementations. The first conclusion is that the impact of write intensity is not as high on SegS as it is on HybS. The write intensity of SegS affects only the percentage of the input sorted using external merge-

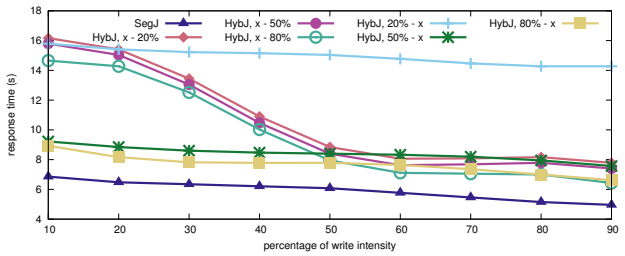


Figure 10: Impact of write-intensity on join algorithms

sort, with selection sort used for the rest of the input. This reduces the number of writes overall, but does not result in larger gains as intensity increases; SegS quickly reaches good performance at a lower intensity. Write intensity has a more pronounced effect on HybS. As write intensity grows, the performance of the algorithm improves substantially by up to 45%; SegS is only improved by up to 18%. Note the substantial overlap between the different algorithms, their write intensity, and the choice of implementation. For instance, for a low write intensity, even the worst implementation of SegS beats HybS on performance.

Switching to join evaluation, in Figure 10 we show the impact of write intensity on SegJ and HybJ, which are affected by this choice. HybJ is parameterized on the write intensity over each input individually. To aid presentation we keep the write intensity over one input constant and scale write intensity over the other; *e.g.*, HybJ, 50% - x denotes a 50% write intensity over the left input as we scale the write intensity over the right input. We report only for the blocked memory implementation as it carries the lowest overhead and to avoid cluttering the plots. The impact of write intensity on SegJ is gradual, with each increment improving performance up to about 20% in the end. For HybJ the determining factor is the write intensity over the left input. The performance for a fixed write intensity over the left input as write intensity over the right input varies is relatively stable. But as the write intensity of the left input grows, performance improves substantially to a maximum gain of up to 50%. This is due to the write intensity of the left input dictating the portion of the computation that will be performed with nested loops, or, more specifically, the number of full passes over the larger right input. The higher the intensity, the smaller the number of passes; and the better the performance. Note that a large write intensity is not necessary: a 50% write intensity over the left input is enough to give good performance. As the left input is the smaller one, this results in substantial write savings.

4.2.2 *Impact of write/read ratio*

Write-limited algorithms are designed for asymmetric write/read costs. We measured the performance of the algorithms by varying the write latency from 50ns to 200ns. We chose not to test different

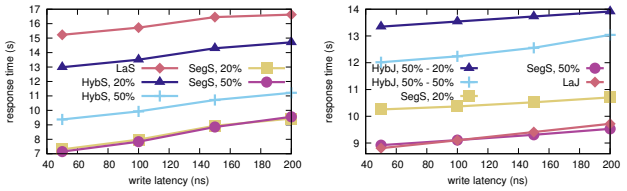


Figure 11: Impact of write latency on selected sorting (left) and join (right) algorithms

read latencies as read performance of persistent memory is generally good and does not vary as widely—nor is it the major point we address in this work. The performance of selected runs are shown in Figure 11 for sorting and join algorithms (left and right plots respectively). We report only for the blocked memory implementation as it carries the minimal overhead. We focus on no more than a 50% write intensity to avoid further penalizing the write profile of the algorithms. The write-limited algorithms are not adversely affected by higher write latencies. Even though write latency increases by up to 100% between successive points, the hit on performance is no more than 5%. The results confirm the resilience of our algorithms to write/read asymmetry.

4.2.3 Cost model validation

We have so far focused on the performance of write-limited algorithms. For these algorithms to be useful, however, they must be accompanied by a cost model capturing their performance. We will now validate in a limited setting the cost expressions of the write-limited algorithms. We used the ten-million-record sorting input and the one-million by ten-million-record join computation as we varied memory for a fixed 150ns write latency. We excluded the lazy algorithms, LaS and LaJ, from this study as their decisions are dynamic rather than static. That is, they monitor writes and reads and decide to materialize temporary results during run-time; in contrast to an optimizer deciding the best choice of algorithm at query compilation time. For each remaining algorithm, for each sort and join benchmark, and for each memory increment we estimated the cost of the algorithm using the cost expressions of Section 2, and ranked the algorithms according to their estimated performance. We then executed the algorithms and ranked them according to their true performance. We compared the two rank orders using Kendall's τ correlation coefficient [12]. The latter captures the agreement between two different orderings of a list of elements by looking at the concordant and discordant pairs of ranks for the same element. The correlation coefficient is a number in $[-1,1]$ with 1 denoting complete agreement; -1 denoting complete disagreement; and 0 implying independence.

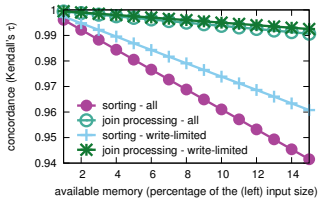


Figure 12: Concordance between estimated and true performance

We report the correlation coefficient in Figure 12 as we scale the amount of available memory as a percentage of the total input size (for sorting) or the left input size (for join processing). We show concordance for two cases per class of algorithm: if all algorithms are included (*i.e.*, algorithms optimized for symmetric I/O are used too) or if we focus only on write-limited algorithms. There is always high concordance between the estimated rank and the true rank. Concordance diverges as available memory grows since most algorithms then have comparable performance, thereby increasing the likelihood of a mistake. Concordance is higher for join processing than sorting as there is greater variation in performance and the cost expressions manage to differentiate between choices more effectively. Focusing only on the write-limited algorithms improves concordance for both sorting and join processing algorithms. This is due to: (*a*) fewer rank combinations being possible, and (*b*) the excluded algorithms always participating in groups of similarly performing algorithms; both factors result in higher concordance. Concordance is always above the 0.94 mark indicating that the cost estimates truly capture the relative performance of the algorithms and can be used as a solid basis for decision making.

4.3 Discussion

The results affirm that choosing algorithms or implementations when incorporating persistent memory into the I/O stack is not straightforward. It is a combination of various parameters and it comes down to what we want to optimize for. The algorithms do well in introducing write intensity and giving the developer, or the runtime, a knob by which they can select whether to minimize writes; or minimize response time; or both. It is also important that the majority of algorithms converges to I/O-minimal behavior at a low write intensity; *e.g.*, SegS and SegJ approximate or outperform their counterparts optimized for symmetric I/O from a 20% write intensity onwards. This confirms that one can have write-limited algorithms without compromising performance.

The cost models of the algorithms are necessary to choose an algorithm in an informed way. But it is not only the cost models that are important. The API of Section 3.1 is also conducive, as it allows the developer to defer decision making at compile-time; rather, decisions can be made at run-time. Perhaps even by optimizing for different objectives at different times, making it possible to autotune performance according to system-wide and potentially evolving policies; in addition to boosting performance during development if objectives are known *a priori*.

The results also suggest that one is better off using a memory representation for collections that borrows aspects of blocked storage and is not only optimized for main memory use. Consider the dynamic array representation of collections, which is optimized for main memory use by increasing spatial locality; and leveraging temporal locality to maximize performance. While array expansion in a main memory setting bears a one-off cost that is dwarfed by the benefits of improved locality, this is no longer the case for persistent memory and its asymmetric write/read costs. Thus, an implementation optimized for main memory is not the best choice for persistent memory. Treating persistent memory as block-addressable storage albeit mounted in main memory is not the best option either as it introduces significant overhead. A persistent collection implementation based on blocked memory shows the true potential of the hardware and the algorithms as it effectively bears zero overhead apart from the unavoidable penalties due to the write/read cost asymmetry. Whereas an implementation over a byte-addressable filesystem like PMFS gives the best of both worlds: true file-like persistence over a byte-addressable substrate at a low overhead in the majority of cases. It therefore makes sense to strive to optimize such implementations and further reduce their overheads. This can be achieved perhaps by additional hardware support, or better implementations of primitives. The goal should be to reach the ideal performance of blocked memory.

Finally, note that we studied asymmetry in terms of I/O response time. Asymmetry, however, also manifests in terms of power consumption [2]; or device degradation. Our algorithms are applicable then as well and the relative gains may be higher as the asymmetry is more pronounced under such metrics.

5. RELATED WORK

With persistent memory only now starting to emerge as a storage medium, related work in the area is rather limited. The closest area of research is flash memory, which has received considerable attention. There has been a host of techniques on improving the performance of the flash translation layer (FTL), which is the part of the flash controller that provides logical-to-physical address mapping, power-off recovery, and wear-leveling. Researchers have studied the FTL algorithms [3] and proposed various improvements on their performance based on block-level associativity [15], on-chip caching [1], page-level lazy updates [18], or wear-leveling [11]. On the software side, research has focused on flash-specific buffer-pool management schemes [14, 16, 20, 21], query evaluation techniques [7, 19, 23], and logging [8]. This work, while relevant, does not cater for byte addressability. The differences in block- *vs.* byte-level access suggest that considerable effort will be necessary to port these approaches to persistent memory.

In a database context, Chen *et al.* [2] explored how database algorithms need to be changed in the presence of persistent memory. They argued for a radical reimplementation of algorithms by eliminating data copying and using pointers to data in order to reduce memory stores. Our stance is different: we argue that we are better off limiting writes at a higher level. To that end we give ways to limit writes at the system and developer levels by exposing the workings of the algorithm through our API. The techniques of [2] then become orthogonal and may further improve performance.

The systems community has also addressed the persistence aspects of persistent memory. Coburn *et al.* [4] look to support heap-based allocation operations on non-volatile media; our abstraction of persistent collections may certainly benefit from such allocation primitives. Volos *et al.* [24] deal with the efficiency issues of supporting persistence and argue for a lightweight approach; this is a complementary issue to what we address here as it targets the persistent memory controller rather than the software side of the system. Finally, Condit *et al.* [5] discuss the intricacies of designing a persistent byte-addressable filesystem; we have used similar concepts in our implementation of persistent collections over PMFS.

6. CONCLUSIONS AND OUTLOOK

Persistent memory has the potential to become a universal storage device. We addressed some of the issues involved in incorporating persistent memory into database query engine design. We focused on two fundamental query evaluation operations, namely sorting and join processing. We adapted these operations for persistent memory and presented a family of write-limited algorithms that either minimize I/O; or are tunable by the developer and/or the system during run-time. We presented API and implementation primitives that enable the seamless integration of persistent memory into the processing stack of database systems. We extensively studied the performance of our proposals. Our results showed that write-limited algorithms deliver on their promise and outperform or, at worst, match the performance of traditional solutions.

One might extend this work to generalized algorithms (*e.g.*, [9]); or data structures (*e.g.*, indexes); or operations (*e.g.*, aggregation). Alternatively, one might focus on persistent-memory-specific solutions to support other aspects of database systems like transaction processing and recovery. In this work, we have studied an inclusive memory hierarchy where data is moved from persistent memory to DRAM to be processed. It would be interesting to see the trade-offs involved in using only persistent memory and doing away with DRAM altogether. This becomes especially important if the latency of persistent memory matches that of DRAM.

Acknowledgments

The author would like to thank the anonymous reviewers for their comments. This work was supported by the Intel University Research Office and the Software for Persistent Memories program.

7. REFERENCES

[1] A. Birrell *et al.* A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.

[2] S. Chen *et al.* Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[3] T.-S. Chung *et al.* System software for flash memory: A survey. In *EUC*, 2006.

[4] J. Coburn *et al.* NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS XVI*, 2011.

[5] J. Condit *et al.* Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[6] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future, 1993.

[7] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *DAMON*, 2009.

[8] R. Fang *et al.* High performance database logging using storage class memory. In *ICDE*, 2011.

[9] G. Graefe. New algorithms for join and grouping operations. *Computer Science - R&D*, 27(1):3–27, 2012.

[10] B. K. Haddon. Cycle-sort: A linear sorting method. *The Computer Journal*, 33(4):365–367, 1990.

[11] X.-Y. Hu *et al.* Write amplification analysis in flash-based solid state drives. In *SYSTOR*, 2009.

[12] M. Kendall. A New Measure of Rank Correlation. *Biometrica*, 30:81–89, 1938.

[13] L. Kevin *et al.* Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[14] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.

[15] J. Kim *et al.* A space-efficient flash translation layer for CompactFlash systems. *Trans. on Consumer Electronics.*, 2002.

[16] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *PVLDB*, 1(1), 2008.

[17] I. Koltsidas and S. D. Viglas. Data management over flash memory. In *SIGMOD*, 2011.

[18] D. Ma *et al.* LazyFTL: A page-level flash translation layer optimized for nand flash memory. In *SIGMOD*, 2011.

[19] D. Myers. On the use of NAND flash memory in high-performance relational databases. Master Thesis, MIT, 2007.

[20] Y. Ou *et al.* CFDC: a flash-aware replacement policy for database buffer management. In *DAMON*, 2009.

[21] S. Park *et al.* CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.

[22] M. K. Qureshi *et al. Phase Change Memory: from devices to systems*. Morgan & Claypool Publiishers, 2012.

[23] D. Tsirogiannis *et al.* Query processing techniques for solid state drives. In *SIGMOD*, 2009.

[24] H. Volos *et al.* Mnemosyne: lightweight persistent memory. In *ASPLOS*, 2011.