Towards Making Database Systems PCM-Compliant

Vishesh Garg, Abhimanyu Singh, and Jayant R. Haritsa

Database Systems Lab, SERC/CSA
Indian Institute of Science, Bangalore, INDIA
{vishesh, abhimanyu, haritsa}@dsl.serc.iisc.ernet.in

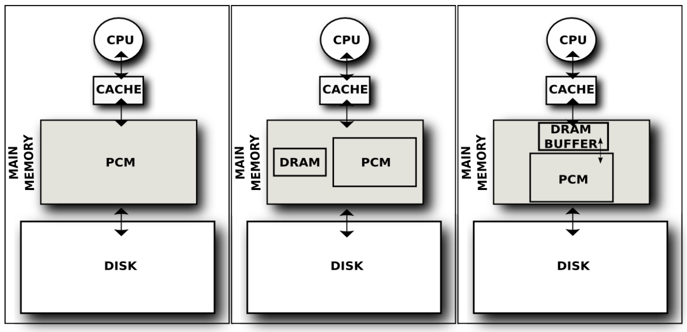
Abstract. Phase Change Memory (PCM) is a new *non-volatile* memory technology that is comparable to traditional DRAM with regard to read latency, and markedly superior with regard to storage density and idle power consumption. Due to these desirable characteristics, PCM is expected to play a significant role in the next generation of computing systems. However, it also has limitations in the form of expensive writes and limited write endurance. Accordingly, recent research has investigated how database engines may be redesigned to suit DBMS deployments on the new technology.

In this paper, we address the pragmatic goal of minimally altering current implementations of database operators to make them PCM-conscious, the objective being to facilitate an easy transition to the new technology. Specifically, we target the implementations of the "workhorse" database operators: *sort*, *hash join* and *group-by*, and rework them to substantively improve the write performance without compromising on execution times. Concurrently, we provide simple but effective *estimators* of the writes incurred by the new techniques, and these estimators are leveraged for integration with the query optimizer.

Our new techniques are evaluated on TPC-H benchmark queries with regard to the following metrics: number of writes, response times and wear distribution. The experimental results indicate that the PCM-conscious operators collectively reduce the number of writes by a factor of 2 to 3, while concurrently improving the query response times by about 20% to 30%. When combined with the appropriate plan choices, the improvements are even higher. In essence, our algorithms provide both short-term and long-term benefits. These outcomes augur well for database engines that wish to leverage the impending transition to PCM-based computing.

1 Introduction

Phase Change Memory (PCM) is a recently developed non-volatile memory technology, constructed from chalcogenide glass material, that stores data by switching between amorphous (*binary 0*) and crystalline (*binary 1*) states. Broadly speaking, it is expected to provide an attractive combination of the best features of conventional disks (persistence, capacity) and of DRAM (access speed). For instance, it is about 2 to 4 times denser than DRAM, while providing a DRAM-comparable read latency. On the other hand, it consumes much less energy than magnetic hard disks while providing substantively smaller write latency. Due to this suite of desirable features, PCM technology is expected to play a prominent role in the next generation of computing systems, either augmenting or replacing current components in the memory hierarchy [10, 15, 7].



(a) **PCM_RAM** PCM replaces DRAM as Main Memory
(b) **DRAM_SOFT** Main Memory includes PCM and DRAM under explicit software control
(c) **DRAM_HARD** PCM augmented with hardware managed DRAM buffer constitutes the Main Memory

Fig. 1: PCM-based Architectural Options [3]

A limitation of PCM, however, is that there is a significant difference between the read and write behaviors in terms of energy, latency and bandwidth. A PCM write, for example, consumes 6 times more energy than a read. Further, PCM has limited write endurance since a memory cell becomes unusable after the number of writes to the cell exceeds a threshold determined by the underlying glass material. Consequently, several database researchers have, in recent times, focused their attention on devising new implementations of the core database operators that are adapted to the idiosyncrasies of the PCM environment (e.g. [3, 12]).

Architectural Model

The prior database work (which we have analyzed in detail in [5]) has primarily focused on computing architectures wherein either (a) PCM completely replaces the DRAM memory [3]; or (b) PCM and DRAM co-exist side-by-side and are independently controlled by the software [12]. We hereafter refer to these options as **PCM_RAM** and **DRAM_SOFT**, respectively.

However, a third option that is gaining favor in the architecture community, and also mooted in [3] from the database perspective, is where the PCM is augmented with a small hardware-managed DRAM buffer [10]. In this model, which we refer to as **DRAM_HARD**, the address space of the application maps to PCM, and the DRAM buffer can simply be visualized as yet another level of the existing cache hierarchy. For ease of comparison, these various configurations are pictorially shown in Figure 1.

There are several practical advantages of the DRAM_HARD configuration: First, the write latency drawback of PCM_RAM can be largely concealed by the intermediate DRAM buffer [10]. Second, existing applications can be used *as is* but still manage to take advantage of both the DRAM and the PCM. This is in stark contrast to the DRAM_SOFT model which requires incorporating additional machinery, either in the program or in the OS, to distinguish between data mapped to DRAM and to PCM – for example, by having separate address space mappings for the different memories.

Our Work

In this paper, we propose minimalist reworkings, that are tuned to the DRAM_HARD model, of current implementations of database operators. In particular, we focus on the "workhorse" operators: *sort*, *hash join* and *group-by*. The proposed modifications are not only easy to implement but are attractive from the performance perspective also, simultaneously reducing *both* PCM writes and query response times. The new implementations are evaluated on Multi2sim [11], a state-of-the-art architectural simulator, after incorporating major extensions to support modelling of the DRAM_HARD configuration. Their performance is evaluated on *complete* TPC-H benchmark queries. This is a noteworthy point since earlier studies of PCM databases had only considered operator performance in isolation. But, it is possible that optimizing a specific operator may turn out to be detrimental to downstream operators that follow it in the query execution plan. For instance, the proposal in [3] to keep leaf nodes unsorted in B$^+$ indexes – while this saves on writes, it is detrimental to the running times of *subsequent* operators that leverage index ordering – for instance, *join filters*. Finally, we include the metric of *wear distribution* in our evaluation to ensure that the reduction in writes is not achieved at the cost of skew in wear-out of PCM cells.

Our simulation results indicate that the customized implementations collectively offer substantive benefits with regard to PCM writes – the number is typically brought down *by a factor of two to three*. Concurrently, the query response times are also brought down by about *20–30 percent*. As a sample case in point, for TPC-H Query 19, savings of 64% in PCM writes are achieved with a concomitant 32% reduction in CPU cycles.

Fully leveraging the new implementations requires integration with the query optimizer, an issue that has been largely overlooked in the prior literature. We take a first step here by providing simple but effective statistical *estimators* for the number of writes incurred by the new operators under uniform data distribution scenarios, and incorporating these estimators in the query optimizer's cost model. Sample results demonstrating that the resultant plan choices provide substantively improved performance are provided in our experimental study.

Overall, the above outcomes augur well for the impending migration of database engines to PCM-based computing platforms.

Organization

The remainder of this paper is organized as follows: We define the problem framework in Section 2. The design of the new PCM-conscious database operators, and an analysis of their PCM writes, are presented in Sections 3, 4 and 5. Our experimental framework and the simulation results are reported in Sections 6 and 7, respectively. This is followed by a discussion in Section 8 on integration with the query optimizer. Finally, Section 9 summarizes our conclusions and outlines future research avenues.

2 Problem Framework

In this section, we overview the problem framework, the assumptions made in our analysis, and the notations used in the sequel.

Table 1: Notations Used in Operator Analysis

Term	Description
D	DRAM size
K	DRAM Associativity
N_R, N_S	Row cardinalities of input relations R and S, respectively
L_R, L_S	Tuple lengths of input relations R and S, respectively
P	Pointer size
H	Size of each hash table entry
A	Size of aggregate field (for group-by operator)
N_j, N_g	Output tuple cardinalities of join and group-by operators, respectively
L_j, L_g	Output tuple lengths of join and group-by operators, respectively

We model the DRAM_HARD memory organization shown in Figure 1 (c). The DRAM buffer is of size D, and organized in a *K-way set-associative* manner, like the L1/L2 processor cache memories. Moreover, its operation is identical to that of an *inclusive* cache in the memory hierarchy, that is, a new DRAM line is fetched from PCM each time there is a DRAM miss. The last level cache in turn fetches its data from the DRAM buffer.

We assume that the writes to PCM are in word-sized units (4B) and are incurred only when a data block is evicted from DRAM to PCM. A *data-comparison write (DCW)* scheme [14] is used for the writing of PCM memory blocks during eviction from DRAM – in this scheme, the memory controller compares the existing PCM block to the newly evicted DRAM block, and selectively writes back only the modified words. Further, *N-Chance* [4] is used as the DRAM eviction policy due to its preference for evicting non-dirty entries, thereby saving on writes. The failure recovery mechanism for updates is orthogonal to our work and is therefore not discussed in this paper.

As described above, the simulator implements a realistic DRAM buffer. However, in our write analyses and estimators, we assume for tractability that there are no conflict misses in the DRAM. Thus, for any operation dealing with data whose size is within the DRAM capacity, our analysis assumes no evictions and consequently no writes. The experimental evaluation in Section 7.3 indicates the impact of this assumption to be only marginal.

With regard to the operators, we use R to denote the input relation for the *sort* and *group-by* unary operators. Whereas, for the binary *hash join* operator, R is used to denote the smaller relation, on which the hash table is constructed, while S denotes the probing relation.

In this paper, we assume that all input relations are *completely PCM-resident*. Further, for presentation simplicity, we assume that the sort, hash join and group-by expressions are on singleton attributes – the extension to multiple attributes is straightforward.

A summary of the main notation used in the analysis of the following sections is provided in Table 1.

3 The *Sort* Operator

The *quicksort* algorithm is the most commonly used sorting algorithm in database systems. In the single-pivot quicksort algorithm with n elements, the average number of swaps is of the order of $0.3nln(n)$ [13]. If the initial array is much larger than the DRAM size, it would entail evictions from the DRAM during the swapping process of partitioning. These evictions might lead to PCM writes if the evicted DRAM lines are *dirty*, which is likely since elements are being swapped. If the resulting partition sizes continue to be larger than DRAM, partitioning them in turn will again cause DRAM evictions and consequent writes. Clearly, this trend of writes will continue in the recursion tree until the partition sizes become small enough to fit within DRAM.

From the above discussion, it is clear that it would be desirable for the sorting algorithm to converge fast to partition sizes within DRAM size with fewer number of swaps. For uniformly-distributed data[1], these requirements are satisfied by *flashsort* [8]. Specifically, flashsort can potentially form DRAM-sized partitions in a *single* partitioning step with at most N_R swaps. The sorting is done in-place with a time complexity of $O(N_R log_2 N_R)$ with constant extra space.

The flashsort algorithm proceeds in three phases: *Classification*, *Permutation* and *Short-range Ordering*. The Classification phase divides the input data into p partitions, where p is an input parameter. Specifically, an element with value v is assigned to $Partition(v)$, computed as $1 + \lfloor \frac{(p-1)(v-v_{min})}{v_{max}-v_{min}} \rfloor$, where v_{min} and v_{max} are the smallest and largest values in the array, respectively. The number of elements in each such partition is counted to derive the boundary information. Next, the Permutation phase moves the elements to their respective partitions. Finally, the individual partitions are sorted in the Short-range Ordering phase to obtain the overall sorted array.

We choose the number of partitions p to be $\lceil c \times \frac{N_R L_R}{D} \rceil$, where $c \geq 1$ is a multiplier to cater to the space requirements of additional data structures constructed during sorting. In our experience, setting $c = 2$ works well in practice. The resulting partitions, each having size less than D, are finally sorted in the Short-range Ordering phase using quicksort.

PCM write analysis: Though the partition boundary counters are continuously updated during the Classification phase, they are expected to incur very few PCM writes. This is because the updates are all in quick succession, making it unlikely for the counters to be evicted from DRAM during the update process. Next, while in the Permutation phase, there are no more than $N_R L_R$ writes since each tuple is written at most once while placing it inside its partition boundaries. Since each partition is within the DRAM size, its Short-range Ordering phase will finish in the DRAM itself, and then there will be another $N_R L_R$ writes upon eventual eviction of sorted partitions to PCM.

Thus, the number of word-writes incurred by this algorithm is estimated by

$$W_{sort} = \frac{2N_R L_R}{4} = \frac{N_R L_R}{2} \tag{1}$$

[1] In [5], we present a modified flashsort algorithm, called *multi-pivot flashsort*, for skewed data.

4 The *Hash Join* Operator

Hash join is perhaps the most commonly used join algorithm in database systems. Here, a hash table is built on the smaller relation, and tuples from the larger relation are used to probe for matching values in the join column. Since we assume that all tables are completely PCM-resident, the join here *does not* require any initial partitioning stage. Instead, we directly proceed to the join phase. Thus, during the progress of hash join, writes will be incurred during the building of the hash table, and also during the writing of the join results.

Each entry in the hash table consists of a pointer to the corresponding build tuple, and the hash value for the join column. Due to the absence of prior knowledge about the distribution of join column values for the build relation, the hash table is expanded dynamically according to the input. Typically, for each insertion in a bucket, a new space is allocated, and connected to existing entries using a pointer. Thus, such an approach incurs an additional pointer write each time a new entry is inserted.

Our first modification is to use a well-known technique of allocating space to hash buckets in units of *pages* [6]. A page is of fixed size and contains a sequence of contiguous fixed-size hash-entries. When a page overflows, a new page is allocated and linked to the overflowing page via a pointer. Thus, unlike the conventional hash table wherein each *pair* of entries is connected using pointers, the interconnecting pointer here is only at page granularity. Note that although open-addressing is another alternative for avoiding pointers, probing for a join attribute value would have to search through the *entire table* each time, since the inner table may contain *multiple* tuples with the same join attribute value.

A control bitmap is used to indicate whether each entry in a page is vacant or occupied, information that is required during both insertion and search in the hash table. Each time a bucket runs out of space, a new page is allocated to the bucket. Though such an approach may lead to space wastage when some of the pages are not fully occupied, we save on the numerous pointer writes that are otherwise incurred when space is allocated on a per-entry basis.

Secondly, we can reduce the writes incurred due to storing of the hash values in the hash table by restricting the length of each hash value to just a single byte. In this manner, we trade-off precision for fewer writes. If the hash function distributes the values in each bucket in a perfectly uniform manner, it would be able to distinguish between $2^8 = 256$ join column values in a bucket. This would be sufficient if the number of distinct values mapping to each bucket turn out to be less than this value. Otherwise, we would have to incur the penalty (in terms of latency) of reading the actual join column values from PCM due to the possibility of false positives.

PCM write analysis: We ignore the writes incurred while initializing each hash table bucket since they are negligible in comparison to inserting the actual entries. Assuming there are E_{page} entries per page, there would now be one pointer for each E_{page} set of entries. Additionally, for each insertion, a bit write would be incurred due to the bitmap update. The join tuples would also incur writes to the tune of $N_j \times L_j$. Thus, the total number of word-writes for hash join would be

$$W_{hj} = \frac{N_R \times (H + \frac{P}{E_{page}} + \frac{1}{8}) + N_j \times L_j}{4}$$

Since in practice both $\frac{P}{E_{page}}$ and $\frac{1}{8}$ are small as compared to H,

$$W_{hj} \approx \frac{N_R \times H + N_j \times L_j}{4} \tag{2}$$

5 The *Group-By* Operator

We now turn our attention to the group-by operator which typically forms the basis for aggregate function computations in SQL queries. Common methods for implementing group-by include *sorting* and *hashing* – the specific choice of method depends both on the constraints associated with the operator expression itself, as well as on the downstream operators in the plan tree. We discuss below the PCM-conscious modifications of both implementations, which share a common number of *output* tuple writes, namely $N_g \times L_g$.

5.1 Hash-Based Grouping

A hash table entry for group-by, as compared to the corresponding entry in hash join, has an additional field containing the aggregate value. For each new tuple in the input array, a bucket index is obtained after hashing the value of the column present in the group-by expression. Subsequently, a search is made in the bucket indicated by the index. If a tuple matching the group-by column value is found, the aggregate field value is updated; else, a new entry is created in the bucket. Thus, unlike hash join, where each build tuple had its individual entry, here the grouped tuples share a common entry with an aggregate field that is constantly updated over the course of the algorithm.

Since the hash table construction for group-by is identical to that of the hash join operator, the PCM-related modifications described in Section 4 can be applied here as well. That is, we employ a page-based hash table organization, and a reduced hash value size, to reduce the writes to PCM.

PCM write analysis: From the above discussion, it is easy to see that the total number of word-writes incurred for the PCM-conscious hash-based group-by is given by

$$W_{gb\_ht} = \frac{N_g \times H + N_R \times A + N_g \times L_g}{4} \tag{3}$$

5.2 Sort-Based Grouping

Sorting may be used for group-by when a fully ordered operator such as *order by* or *merge join* appears downstream in the plan tree. Another use case is for queries with a *distinct* clause in the aggregate expression, in order to identify the duplicates that have to be discarded from the aggregate.

Sorting-based group-by differs in a key aspect from sorting itself in that the sorted tuples do not have to be written out. Instead, it is the aggregated tuples that are finally passed on to the next operator in the plan tree. Hence, we can modify the flashsort algorithm of Section 3 to use *pointers* in both the Permutation and Short-range Ordering phases, subsequently leveraging these pointers to perform aggregation on the sorted tuples.

PCM write analysis: The full tuple writes of $2N_RL_R$ which were incurred in the flashsort scheme, are now replaced by $2N_R \times P$ since pointers are used during both the Classification and Short-range Ordering phases. Thus, the total number of word-writes for this algorithm for uniformly distributed data would be

$$W_{gb\_sort} = \frac{2N_R \times P + N_g \times L_g}{4} \tag{4}$$

6 Simulation Testbed

This section details our experimental settings in terms of the hardware parameters, the database and query workload, and the performance metrics on which we evaluated the PCM-conscious operator implementations.

6.1 Architectural Platform

Since PCM memory is as yet not commercially available, we have taken recourse to a simulated hardware environment to evaluate the impact of the PCM-conscious operators. For this purpose, we chose Multi2sim [11], an open-source application-only[2] simulator.

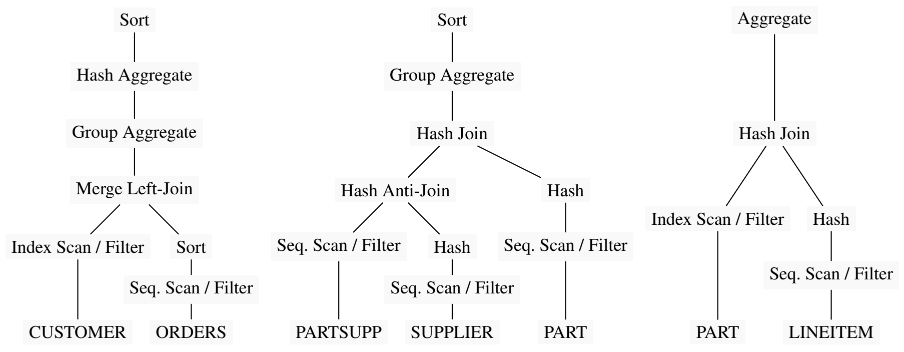
Table 2: Experimental Setup

Simulator	Multi2sim-4.2 with added support for PCM
L1D cache (private)	32KB, 64B line, 4-way set-associative, 4 cycle latency, write-back, LRU
L1I cache (private)	32KB, 64B line, 4-way set-associative, 4 cycle latency, write-back, LRU
L2 cache (private)	256KB, 64B line, 4-way set-associative, 11 cycle latency, write-back, LRU
DRAM buffer (private)	4MB, 256B line, 8-way set-associative, 200 cycle latency, write-back, N-Chance (N = 4)
Main Memory	2GB PCM, 4KB page, 1024 cycle read latency (per 256B line), 64 cycle write latency (per 4B modified word)

We evaluated the PCM-conscious algorithms on Multi2sim in cycle-accurate simulation mode. Since it does not have native support for PCM, we made a major extension to its existing memory module to model PCM with a hardware-controlled DRAM buffer as main memory. Furthermore, we added separate data tracking functionality for the DRAM and PCM-resident data, to implement the DCW scheme (Section 2) of DRAM line write-back to PCM. Likewise, we made several other enhancements to Multi2sim for PCM modelling, and these are enumerated in [5].

The specific configurations of the memory hierarchy *(L1 Data, L1 Instruction, L2, DRAM Buffer, PCM)* used for evaluation in our experiments are enumerated in Table 2. These values are scaled-down versions, w.r.t. number of lines, of the hardware simulation parameters used in [9] – the reason for the scaling-down is to ensure that the simulator running times are not impractically long. However, we have been careful to ensure that the *ratios* between the capacities of adjacent levels in the hierarchy are maintained as per the original configurations in [9].

[2] Simulates only the application layer without the OS stack.



(a) Q13 (b) Q16 (c) Q19

Fig. 2: Query execution plan trees

6.2 Database and Queries

For the data, we used the default 1GB database generated by the TPC-H [1] benchmark. This size is certainly very small compared to the database sizes typically encountered in modern applications – however, we again chose this reduced value to ensure viable simulation running times. Furthermore, the database is significantly larger than the simulated DRAM (4MB), representative of most real-world scenarios.

For simulating our suite of database operators – *sort*, *hash join* and *group-by* – we created a separate library consisting of their native PostgreSQL [2] implementations. To this library, we added the PCM-conscious versions described in the previous sections.

While we experimented with several of the TPC-H queries, results for three queries: Query 13 (Q13), Query 16 (Q16) and Query 19 (Q19), that cover a diverse spectrum of the experimental space, are presented here. For each of the queries, we first identified the execution plan recommended by the PostgreSQL query optimizer with the native operators, and then forcibly used the same execution plan for their PCM-conscious replacements as well. This was done in order to maintain fairness in the comparison of the PCM-oblivious and PCM-conscious algorithms, though it is possible that a *better* plan is available for the PCM-conscious configuration – we return to this issue later in Section 8. The execution plans associated with the three queries are shown in Figure 2.

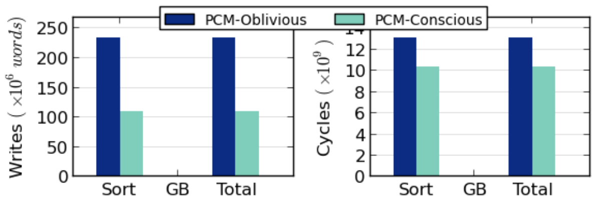
6.3 Performance Metrics

We measured the following performance metrics for each of the queries:

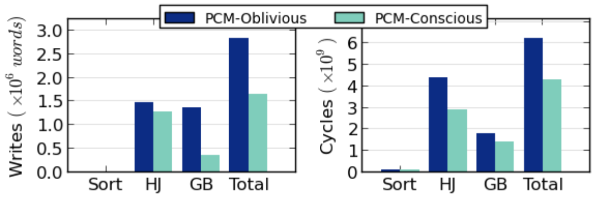
PCM Writes: The total number of word (4B) updates that are applied to the PCM memory during the query execution.

CPU Cycles: The total number of CPU cycles required to execute the query.

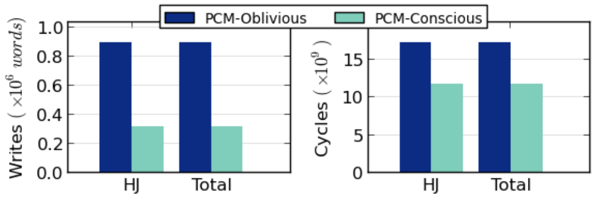
Wear Distribution: The frequency distribution of writes measured on a per-256B-block basis.



(a) Q13 Performance



(b) Q16 Performance



(c) Q19 Performance

Fig. 3: Performance of TPC-H queries

7 Experimental Results

Based on the above framework, we conducted a wide variety of experiments and present a representative set of results here. We begin by profiling the PCM writes and CPU cycles behavior of the native and PCM-conscious executions for Q13, Q16 and Q19 – these results are shown in Figure 3. In each of these figures, we provide both the total and the break-ups on a per-operator basis, with *GB* and *HJ* labels denoting group-by and hash join operators, respectively.

Focusing our attention first on Q13 in Figure 3(a), we find that the bulk of the overall writes and cycles are consumed by the sort operator. Comparing the performance of the Native (blue bar) and PCM-conscious (green bar) implementations, we observe a very significant savings (53%) on writes, and an appreciable decrease (20%) on cycles.

Turning our attention to Q16 in Figure 3(b), we find that here it is the group-by operator that primarily influences the overall writes performance, whereas the hash join determines the cycles behavior. Again, there are substantial savings in both writes (40%) and cycles (30%) delivered by the PCM-conscious approach.

Finally, moving on to Q19 in Figure 3(c), where hash join is essentially the only operator, the savings are around 64% with regard to writes and 32% in cycles.

7.1 Operator-wise Analysis

We now analyse the savings due to each operator independently and show their correspondence to the analyses in Sections 3–5 .

Sort. For Q13, as already mentioned, we observed savings of 53% in writes and 20% in cycles. In the case of Q16, the data at the sorting stage was found to be much less than the DRAM size. Hence, both the native and PCM-conscious executions used the standard sort routine, and as a result, the cycles and writes for both implementations match exactly.

Hash Join. Each entry in the hash table consisted of a pointer to the build tuple and a hash value field. New memory allocation to each bucket was done in units of pages, with each page holding up to 64 entries. A search for the matching join column began with the first tuple in the corresponding bucket, and went on till the last tuple in that bucket, simultaneously writing out the join tuples for successful matches. For Q16, we observed a 12% improvement in writes and 31% in cycles due to the PCM-conscious hash join, as shown in Figure 3(b). The high savings in cycles was the result of the caching effect due to page-wise allocation. These improvements were even higher with Q19 – specifically, 65% writes and 32% cycles, as shown in Figure 3(c). The source of the enhancement was the 3 bytes of writes saved due to single-byte hash values[3], and additionally, the page-based aggregation of hash table entries.

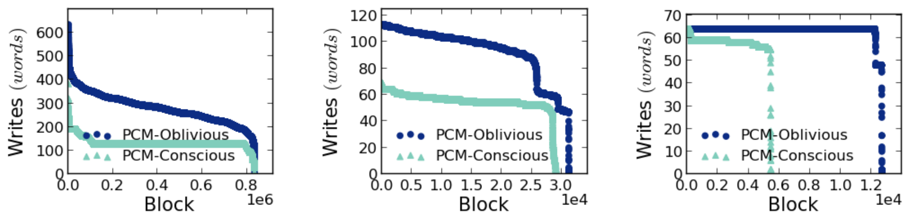
Group-By. In Q16, the aggregate operator in the group-by has an associated *distinct* clause. Thus, our group-by algorithm utilized sort-based grouping to carry out the aggregation. Both the partitioning and sorting were carried out through pointers, thereby reducing the writes significantly. Consequently, we obtain savings of 74% in writes and 20% in cycles, as shown in Figure 3(b). When we consider Q13, however, the grouping algorithm employed was hash-based. Here, the hash table consisted of very few entries which led to the overhead of the page metadata construction overshadowing the savings obtained in other aspects. Specifically, only marginal improvements of about 4% and 1% were obtained for writes and cycles, as shown in Figure 3(a).

7.2 Lifetime Analysis

The above experiments have shown that PCM-conscious operators can certainly provide substantive improvements in both writes and cycles. However, the question still remains as to whether these improvements have been purchased at the expense of *longevity* of the memory. That is, are the writes skewed towards particular memory locations? To answer this, we show in Figure 4, the maximum number of writes across all memory blocks for the three TPC-H queries (as mentioned earlier, we track writes at the block-level–256 bytes–in our modified simulator). The x-axis displays the block numbers in decreasing order of writes.

We observe here that the maximum number of writes is considerably more for the native systems as compared to the PCM-conscious processing. This conclusively demonstrates that the improvement is with regard to *both* average-case and worst-case behavior.

[3] The hash values of all entries within a bucket are placed contiguously.



(a) Q13 (b) Q16 (c) Q19

Fig. 4: Queries wear distribution

7.3 Validating Write Estimators

We now move on to validating the estimators (presented in Sections 3 through 5) for the number of writes incurred by the various database operators.

Sort The size of the $orders$ table is approximately 214 MB. The flashsort algorithm incurred writes of 110.6M. Replacing the values for $N_R L_R$ with the table size in Equation 1, we get the writes as $W_{sort} = (214 \times 10^6)/2 = 107$M. Thus the estimate is close to the number of observed word-writes.

Hash Join For the hash join in Q19, the values of N_R, H, N_j, L_j are 0.2M, 5 bytes, 120 and 8 bytes, respectively. Substituting the parameter values in Equation 2, the writes are given by: $W_{hj} = (0.2 \times 10^6 \times 5 + 120 \times 8)/4 \approx 0.25$M which is close to the actual word-writes of 0.32M.

Group-By The values of the parameters N_R, L_R, P, N_g and L_g for Q16 are 119056, 48 bytes, 4 bytes, 18341 and 48 bytes, respectively. The grouping algorithm used was sort-based grouping. Using Equation 4 results in: $W_{gb_sort} = (2 \times 119056 \times 4 + 18341 \times 48)/4 = 0.46$M. This closely corresponds to the observed word-writes of 0.36M.

A summary of the above results is provided in Table 3. It is clear that our estimators predict the write cardinality with an acceptable degree of accuracy for the PCM-conscious implementations, making them suitable for incorporation in the query optimizer.

8 Query Optimizer Integration

In the earlier sections, given a user query, the modified operator implementations were used for the *standard* plan choice of the PostgreSQL optimizer. That is, while the execution engine was PCM-conscious, the presence of PCM was completely *opaque* to the optimizer. However, given the read-write asymmetry of PCM in terms of both latency

Table 3: Validation of Write Estimators

Operator	**Estimated Word-Writes (in millions)** (e)	**Observed Word-Writes (in millions)** (o)	**Error Factor** $(\frac{e-o}{o})$
Sort	107	110.6	-0.03
Hash Join	0.25	0.32	-0.22
Group-By	0.46	0.36	0.27

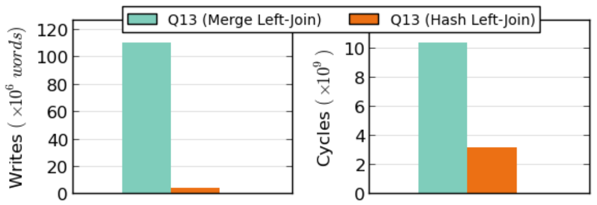
and wear factor, it is possible that alternative plans, capable of providing better performance profiles, may exist in the plan search space. To discover such plans, the database query optimizer needs to incorporate PCM awareness in both the operator cost models and the plan enumeration algorithms.

Current query optimizers typically choose plans using a latency-based costing mechanism. We revise these models to account for the additional latency incurred during writes. Additionally, we introduce a new metric of *write cost* in the operator cost model, representing the incurred writes for a plan in the PCM environment, using the estimators described in Sections 3 to 5. We henceforth refer to the latency cost and the write cost of a plan as **LC** and **WC**, respectively.

A new user-defined parameter, called the *latency slack*, is incorporated in the query optimizer. This slack, denoted by λ, represents the maximum relative slowdown, compared to the LC-optimal query plan, that is acceptable to the user in lieu of getting better write performance. Specifically, if the LC of the LC-optimal execution plan P_o is C_o and the LC of an alternate plan P_i is C_i, the user is willing to accept P_i as the final execution plan if $C_i \leq (1 + \lambda)C_o$. The P_i with the least WC satisfying this equation is considered the WC-optimal plan.

With the new metric in place, we need to revise the plan enumeration process during the planning phase. This is because the native optimizer propagates only the LC-optimal (and interesting order) plans through the internal nodes of the dynamic programming lattice, which may lead to pruning of potential WC-optimal plans. On the other hand, propagating the *entire* list of sub-plans at each internal node can end up in an exponential blow-up of the search space. As an intermediate option between these two extremes, we use a heuristic propagation mechanism at each internal node, employing an algorithmic parameter, *local threshold* λ_l ($\geq \lambda$). Specifically, let p_i and p_o be a generic sub-plan and the LC-optimal sub-plan at a node, respectively, with c_i and c_o being their corresponding LC values. Now, along with the LC-optimal and interesting order sub-plans, we also propagate p_i with the *least* WC that satisfies $c_i \leq (1 + \lambda_l)c_o$. We observed that setting $\lambda_l = \lambda$ delivered reasonably good results in this respect.

In light of these modifications, let us revisit Query Q13, for which the default plan was shown in Figure 2(a). With just the revised latency costs (i.e. λ = 0), the optimizer identified a new execution plan wherein the merge left-join between the *customer* and *orders* tables is replaced by a hash left-join. The relative performance of these two alternatives with regard to PCM writes and CPU cycles are shown in Figure 5(a). We observe here that there is a *huge difference* in both the query response times as well as write overheads between the plans. Specifically, the alternative plan reduces the writes by well over an order of magnitude! As we gradually increased the latency slack value,



(a) Performance of Alternative Plans

Metric	**Opt(PCM-O) Exec(PCM-O)**	**Opt(PCM-O) Exec(PCM-C)**	**Opt(PCM-C) Exec(PCM-C)**	**Opt(PCM-C) Exec(PCM-O)**
Mega Word-Writes	233.6	110.6	4.66	12.8
Giga Cycles	13.1	10.4	3.2	4.5

(b) Overall performance comparison

Fig. 5: Integration with Query Optimization and Processing Engine

initially there was no change in plans. However, when the slack was made as large as 5, the hash left-join gave way to a nested-loop left-join, clearly indicating that the nested-loop join provides write savings only by incurring a steep increase in latency cost.

To put matters into perspective, Figure 5(b) summarizes the relative performance benefits obtained as the database layers are gradually made PCM-conscious (in the figure, the labels Opt and Exec refer to Optimizer and Executor, respectively, while PCM-O and PCM-C refer to PCM-Oblivious and PCM-Conscious, respectively). For the sake of completeness, we have also added results for the case when the Optimizer is PCM-C but the Executor is PCM-O (last column). The results clearly indicate that future query optimizers for PCM-based architectures need to incorporate PCM-Consciousness at *both* the Optimizer and the Executor levels in order to obtain the best query performance.

9 Conclusion

Designing database query execution algorithms for PCM platforms requires a change in perspective from the traditional assumptions of symmetric read and write overheads. We presented here a variety of minimally modified algorithms for the workhorse database operators: *sort*, *hash join* and *group-by*, which were constructed with a view towards simultaneously reducing both the number of writes and the response time. Through detailed experimentation on complete TPC-H benchmark queries, we showed that substantial improvements on these metrics can be obtained as compared to their contemporary PCM-oblivious counterparts. Collaterally, the PCM cell lifetimes are also greatly extended by the new approaches.

Using our write estimators for uniformly distributed data, we presented a redesigned database optimizer, thereby incorporating PCM-consciousness in all layers of the database engine. We also presented initial results showing how this can influence plan choices, and improve the write performance by a substantial margin. While our experiments were conducted on a PCM simulator, the cycle-accurate nature of the simulator makes it likely that similar performance will be exhibited in the real world as well. In our future work, we would like to design write estimators that leverage the metadata statistics to accurately predict writes for skewed data. Additionally, we wish to design multi-objective optimization algorithms for query plan selection with provable performance guarantees.

Overall, the results of this paper augur well for an easy migration of current database engines to leverage the benefits of tomorrow's PCM-based computing platforms.

References

1. `http://www.tpc.org/tpch`.
2. `http://www.postgresql.org`.
3. S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proc. of 5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2011.
4. A. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse. Increasing PCM main memory lifetime. In *Proc. of 13th Conf. on Design, Automation and Test in Europe (DATE)*, 2010.
5. V. Garg, A. Singh, and J. R. Haritsa. On improving write performance in PCM databases. Technical Report TR-2015-01, DSL/SERC, IISc, `dsl.serc.iisc.ernet.in/publications/report/TR/TR-2015-01.pdf`, 2015.
6. P.-A. Larson. Grouping and duplicate elimination: Benefits of early aggregation. *Microsoft Technical Report*, 1997.
7. B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proc. of 36th Intl. Symp. on Computer Architecture (ISCA)*, 2009.
8. K.-D. Neubert. The flashsort1 algorithm. `http://www.drdobbs.com/database/the-flashsort1-algorithm/184410496`, 1998.
9. M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proc. of 42nd Intl. Symp. on Microarchitecture (MICRO)*, 2009.
10. M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of 36th Intl. Symp. on Computer Architecture (ISCA)*, 2009.
11. R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: A simulation framework for CPU-GPU computing. In *Proc. of 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.
12. S. D. Viglas. Write-limited sorts and joins for persistent memory. In *Proc. of 40th Intl. Conf. on Very Large Data Bases (VLDB)*, 2014.
13. S. Wild and M. E. Nebel. Average case analysis of java 7's dual pivot quicksort. In *Proc. of 20th European Symposium on Algorithms (ESA)*, 2012.
14. B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proc. of 2007 IEEE Intl. Symp. on Circuits and Systems (ISCAS)*, 2007.
15. P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proc. of 36th Intl. Symp. on Computer Architecture (ISCA)*, 2009.