Towards Designing PCM-Conscious Database Systems

A THESIS
SUBMITTED FOR THE DEGREE OF
Master of Science (Engineering)
IN THE
Faculty of Engineering

BY

Vishesh Garg

INDIAN INSTITUTE OF SCIENCE
भारतीय विज्ञान संस्थान

Department of Computational and Data Sciences
Indian Institute of Science
Bangalore – 560 012 (INDIA)

March 2016

DEDICATED TO

My Beloved Master

Shri Parthasarathi Rajagopalachari

Acknowledgements

No thesis acknowledgment can begin without thanking one's advisor – not out of compulsion but out of gratitude for the things that one gets to learn from him/her. In my case it's no different, as I consider working under Prof. Jayant Haritsa a turning point in my life. If it was not for him, I would perhaps have never known what research is all about, what should our attitude be towards work, and how we should keep striving for perfection in everything that we do.

Next, I would like to thank my lab members – Anshuman, Srinivas and Rafia among several others – who provided critical feedback for my work and acted as support in trying times during my stay here.

Finally, I would like to thank my family who encouraged me to pursue higher studies even after spending a considerable time working in the industry.

Abstract

Phase Change Memory (PCM) is a recently developed non-volatile memory technology that 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. 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.

Database systems, by virtue of dealing with enormous amounts of data, are expected to be a prime beneficiary of this new technology. Accordingly, recent research has investigated how database engines may be redesigned to suit DBMS deployments on PCM, covering areas such as indexing techniques, logging mechanisms and query processing algorithms. Prior database research has primarily focused on computing architectures wherein either a) PCM completely replaces the DRAM memory ; or b) PCM and DRAM co-exist side-by-side and are independently controlled by the software. However, a third option that is gaining favor in the architecture community is where the PCM is augmented with a small hardware-managed DRAM buffer. 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. With most of the query processing research being preoccupied with the first two models, this third model has remained largely ignored. Moreover, even in this limited literature, the emphasis has been restricted to exploring execution-time strategies; the compile-time plan selection process itself being left unaltered.

In this thesis, we propose minimalist reworkings of current implementations of database operators, that are tuned to the DRAM_HARD model, to make them PCM-conscious. We also propose novel algorithms for compile-time query plan selection, thereby taking a holistic approach to introducing PCM-compliance in present-day database systems. Specifically, our contributions are two-fold, as outlined below.

First, 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. Our customized algorithms and techniques for each of these operators are designed to significantly reduce the number of writes while simultaneously saving on execution times. For instance, in the case of sort operator, we perform an in-place partitioning of input data into DRAM-sized chunks so that the subsequent sorting of these chunks can finish inside the DRAM, consequently avoiding both intermediate writes and their associated latency overheads.

Second, we redesign the query optimizer to suit the new environment of PCM. Each of the new operator implementations is accompanied by simple but effective write estimators that make these implementations suitable for incorporation in the optimizer. Current optimizers typically choose plans using a latency-based costing mechanism assigning equal costs to both read and write memory operations. The asymmetric read-write nature of PCM implies that these models are no longer accurate. We therefore revise the cost models to make them cognizant of this asymmetry by accounting for the additional latency during writes. Moreover, since the number of writes is critical to the lifespan of a PCM device, a new metric of write cost is introduced in the optimizer plan selection process, with its value being determined using the

above estimators.

Consequently, the query optimizer needs to select plans that simultaneously minimize query writes and response times. We propose two solutions for handling this dual-objective optimization problem. The first approach is a heuristic propagation algorithm that extends the widely used dynamic programming plan propagation procedure to drastically reduce the exponential search space of candidate plans. The algorithm uses the write costs of sub-plans at each of the operator nodes to decide which of them can be selectively pruned from further consideration. The second approach maps this optimization problem to the linear multiple-choice knapsack problem, and uses its greedy solution to return the final plan for execution. This plan is known to be optimal within the set of non interesting-order plans in a single join order search space. Moreover, it may contain a weighted execution of two algorithms for one of the operator nodes in the plan tree. Therefore overall, while the greedy algorithm comes with optimality guarantees, the heuristic approach is advantageous in terms of easier implementation.

The experimentation for our proposed techniques is conducted on Multi2sim, a state-of-the-art cycle-accurate simulator. Since it does not have native support for PCM, we made a major extension to its existing memory module to model PCM device. Specifically, we added separate data tracking functionality for the DRAM and PCM resident data, to implement the commonly used read-before-write technique for PCM writes reduction. Similarly, modifications were made to Multi2sim's timing subsystem to account for the asymmetric read-write latencies of PCM. A new DRAM replacement policy called N-Chance, that has been shown to work well for PCM-based hardware, was also introduced.

Our new techniques are evaluated on end-to-end TPC-H benchmark queries with regard to the following metrics: number of writes, response times and wear distribution. The experimental results indicate that, in comparison to their PCM-oblivious counterparts, 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 the case of Query 19, for instance, we obtained a 64% savings in writes, while the response time came down to two-thirds of the original.

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.

Publications based on this Thesis

1. Vishesh Garg, Abhimanyu Singh, Jayant R. Haritsa,
   "Towards Making Database Systems PCM-Compliant",
   Proc. of 26th Intl. Conf. on Database and Expert Systems Applications (DEXA), Valencia, Spain, September 2015 (to appear)

2. Vishesh Garg, Abhimanyu Singh, Jayant R. Haritsa,
   "On Improving Write Performance in PCM Databases", Technical Report, TR-2015-01, DSL/SERC, Indian Institute of Science
   http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2015-01.pdf

Contents

List of Figures

List of Tables

Keywords

Phase Change Memory, PCM, Database Operators, Sort, Hash Join, Group-By, Query Optimizer, Query Executor, Multi-Objective Query Optimization

Chapter 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 [32, 46, 25]. 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.

In the recent years, chip manufacturers have come up with new PCM prototypes and products signalling the advent of PCM-based systems. IBM, for example, has come up with a PCM prototype that is about 275 times faster than regular solid state disk (SSD) devices [6]. Companies like Micron [5] have made PCM chips available to original equipment manufacturers

(OEM) to be included in their products. Similarly, Samsung has started shipping a PCM-inclusive multi-chip package (MCP) that is intended for the mobile handset market [4]. These developments indicate that the transition of present computing systems to a PCM inclusive hardware is indeed imminent. Thus, it is imperative for database systems to be geared up for this transition if they are to utilize PCM to the fullest potential.

Algorithm design for database query execution in a PCM environment represents a departure from the conventional design principles based on symmetric read and write behaviours. For instance, the glaring performance gap between reads and writes can now be exploited by trading writes for reads. Current query execution, being rooted in symmetric I/O assumption, can be grossly sub-optimal in this new paradigm. Thus, PCM compliant query execution calls for a significant transformation in the hitherto established perspective on query execution algorithm design. There has been similar research undertaken earlier for database query execution on flash disks [36]. However, PCM differs from flash in some key aspects. Firstly, flash supports block addressability whereas PCM is byte addressable. Secondly, the read latency gap between Flash and DRAM is quite large (32X) whereas the read latencies of PCM and DRAM are almost comparable. These differences deem Flash suitable techniques sub-optimal for PCM. 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. [12, 40]).

Our Work

In this thesis, we propose minimalist reworkings of current implementations of database operators that are tuned to the DRAM_HARD model (described in detail in Section 1.2). 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 [37], 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 [12] 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, and incorporating these estimators in the query optimizer's cost model. Two novel query optimization algorithms are proposed, that consider both writes and response times in their plan selection process. 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.

1.1 Phase Change Memory

PCM is an upcoming non-volatile memory technology that is composed of phase change materials such as chalcogenide glass. The cells of this material are amenable to quickly and reliably switching between crystalline and amorphous states which have different degrees of electrical resistance. This switch is thermally induced by means of electrical pulses and can be invoked

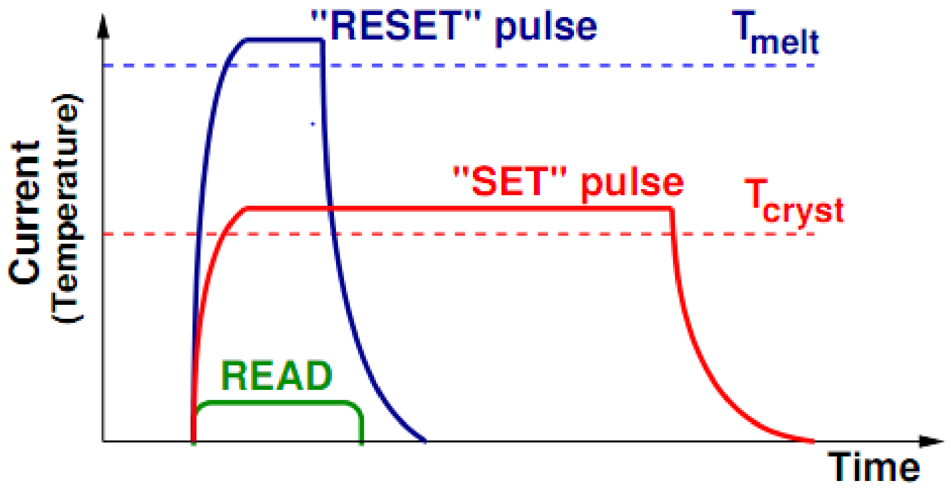


Figure 1.1: Programming the PCM cell [12]

a significantly large number of times. The resulting variations in resistance is used to store bit information in the cells – amorphous state characterized by high resistance representing *bit 0*, while crystalline with low resistance denoting *bit 1*. In fact, the difference in resistance between these two states is about 5 orders of magnitude, which can be further exploited by using intermediate states to denote multiple bits per cell [34]. Figure 1.1 is a schematic diagram showing the SET and RESET operations on PCM.

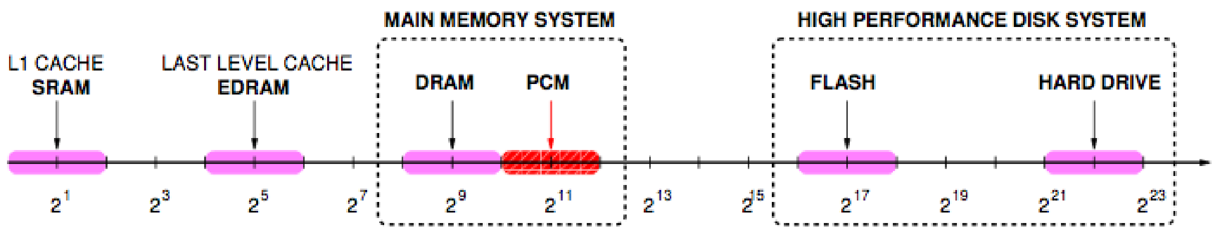


Figure 1.2: Typical access cycles for different memories [32]

The read latency of PCM is almost comparable to DRAM. Typically, this number is about 2^{11} cycles for a page access, as can be seen in Figure 1.2. Moreover, it is byte-addressable and consumes orders of magnitude less idle power than DRAM. On the density scale, it offers 2-4X the density of DRAM, while exhibiting superior scaling capabilities to suit shrinking chip dimensions [29]. This indicates that PCM is expected to be cheaper compared to DRAM when produced in large volumes [32]. Thus, PCM promises to bridge the gap between the DRAM and the Hard Disk in terms of both access latency and capacity. In fact, PCM is currently considered the most rapidly progressing memory technology under the class of storage class memory [8], an umbrella term that encompasses other memories characterized by similar properties of being random-accessible, non-volatile, fast and low cost – such as spin-torque-transfer RAM (STT-RAM) [22] and Memristor [35].

On the flip side, however, PCM comes with serious write limitations. A PCM write is 4-20X slower than a PCM read and consumes much larger energy. Furthermore, a PCM cell can tolerate a limited number of writes – the number typically being around 10^8 – beyond which it becomes unusable. Recent research has therefore sought to alleviate the adverse effects of writes by means of wear-levelling [32] and other custom techniques [43, 14].

Table 1.1 shows the characteristics of PCM as compared to DRAM, NAND Flash and HDD.

Table 1.1: Comparison of memory technologies [32], [25], [3], [12]

	DRAM	**PCM**	**NAND Flash**	**HDD**
Read energy	0.8 J/GB	1 J/GB	1.5 J/GB	65 J/GB
Write energy	1.2 J/GB	6 J/GB	17.5 J/GB	65 J/GB
Idle power	~100 mW/GB	~1 mW/GB	1-10 mW/GB	~10 mW/GB
Endurance	∞	$10^6 - 10^8$	$10^4 - 10^5$	∞
Page size	64B	64B	4KB	512B
Page read latency	20-50ns	$\sim 50ns$	$\sim 25\mu s$	$\sim 5ms$
Page write latency	20-50ns	$\sim 1\mu s$	$\sim 500\mu s$	$\sim 5ms$
Write bandwidth	~GB/s per die	50-100 MB/s per die	5-40 MB/s per die	~200 MB/s per drive

1.2 Architectural Model

The prior database work has primarily focused on computing architectures wherein either (a) PCM completely replaces the DRAM memory [12]; or (b) PCM and DRAM co-exist side-by-side and are independently controlled by the software [40]. We hereafter refer to these options as **PCM_RAM** and **DRAM_SOFT**, respectively.

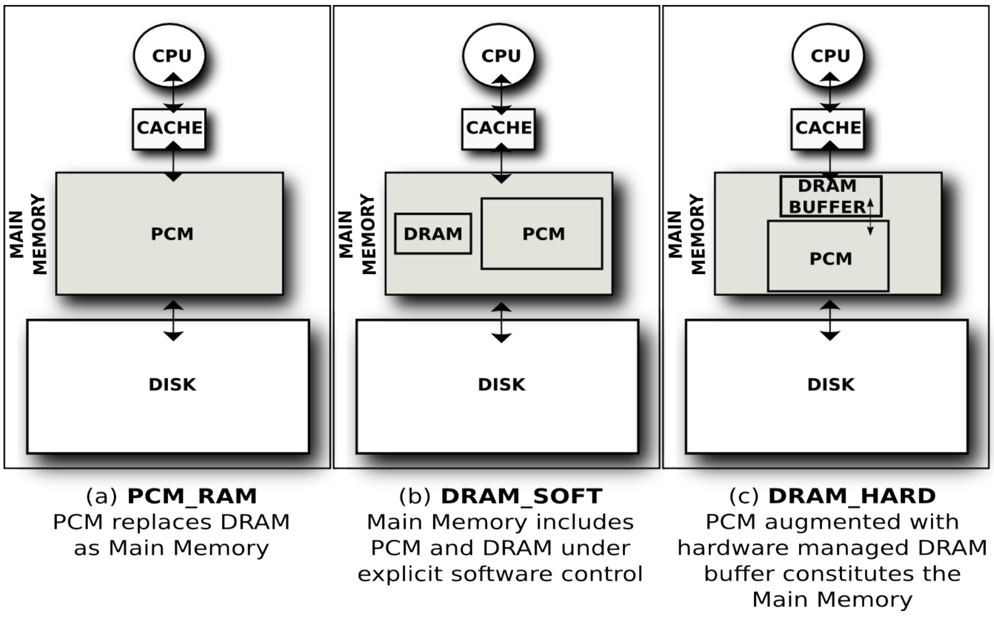

CPU
CACHE
MAIN MEMORY
PCM
DISK
(a) PCM_RAM
PCM replaces DRAM as Main Memory
CPU
CACHE
MAIN MEMORY
DRAM
PCM
DISK
(b) DRAM_SOFT
Main Memory includes PCM and DRAM under explicit software control
CPU
CACHE
MAIN MEMORY
DRAM BUFFER
PCM
DISK
(c) DRAM_HARD
PCM augmented with hardware managed DRAM buffer constitutes the Main Memory

Figure 1.3: PCM-based Architectural Options [12]

However, a third option that is gaining favor in the architecture community, and also mooted in [12] from the database perspective, is where the PCM is augmented with a small hardware-managed DRAM buffer [32]. 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 1.3.

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 [32]. 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.

1.3 Problem Framework

We model the DRAM_HARD memory organization shown in Figure 1.3 (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.

Table 1.2: Notations Used in Operator Analysis

Term	**Description**
D	DRAM size
K	DRAM Associativity
B	PCM Block Size
T_{PCM}	PCM Latency
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
F	Load-factor of buckets in hash table
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 assume that writes to PCM happen at the granularity of a memory-word (whose size we assume to be 4B) and are incurred only when a data block is evicted from DRAM to PCM. A *data-comparison write (DCW)* scheme [43] 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* [19] 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 thesis.

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 5.8 indicates the impact of this assumption to be only marginal. Further, our experimentation is restricted to single query workloads.

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 work, 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 1.2.

1.4 Organization

The remainder of this thesis is organized as follows: The related literature is reviewed in Chapter 2. The design of the new PCM-conscious database operators, and an analysis of their PCM writes, are presented in Chapter 3. The details of our physical implementations of these operators along with the simulator extensions for PCM are covered in Chapter 4. The experimental framework and simulation results are reported in Chapter 5. This is followed by a discussion in Chapter 6 on integration with the query optimizer. Finally, Chapter 7 summarizes our conclusions and outlines future research avenues.

Chapter 2

Survey of Related Literature

Over the past decade, there has been considerable PCM-related research activity on both the architectural front and the various application domains, including database systems. A review of the literature that is closely related to our work is presented here.

2.1 Architectural techniques

On the architectural side, wear levelling algorithms are proposed in [31] that rotate the lines within a circular buffer each time a certain write threshold is reached. The circular buffer holds an empty line to facilitate the rotation of lines during each such round. To keep track of the start of the first block and the location of the gap block, two additional variables are used. Their techniques avoid a storage table lookup by providing direct mapping function between logical and physical blocks. A randomized algorithm is also introduced to handle the case when the writes are spatially concentrated to enable wear levelling across the entire PCM.

PCM buffer management strategies to reduce latency and energy consumption have been discussed in [25]. Using a narrow size of the PCM row buffer is advocated, coupled with the usage of multiple such buffers. The narrower buffer size helps in decreasing the PCM write energy and latency, while multiple buffers are advantageous in terms of write coalescing and spatial locality.

Techniques to reduce writes by writing back only modified data to PCM upon eviction from

LLC/DRAM are presented in [43, 25, 46]. These techniques recommend preceding each write operation with a read operation for the same location, in order to compare the existing data word with the modified one. Since the read operation is much faster than a write, the additional overhead of read is subsumed in the write savings obtained by avoiding redundant writes.

A hybrid memory design which calls for PCM augmentation with a small DRAM buffer is proposed by [32]. The recommended size of the DRAM buffer is about 3% of the PCM memory size. The buffer is meant to act as a page cache to the PCM memory by buffering frequently accessed PCM pages, thus hiding the large latency of PCM. Also, since pages that are modified repeatedly and in quick succession are expected to be resident in the DRAM, it helps in saving PCM writes. To save writes even further, they recommend writing back only the modified ('dirty') cache lines to the main memory. Further, swapping at the block-level is used to achieve wear levelling.

The observation that a PCM write for a SET operation is slow whereas that for the RESET operation consumes almost a read-comparable latency, is leveraged in [31] to alleviate the problem of slow writes. Whenever a cache line gets dirty, the corresponding entire PCM line is proactively issued a SET command, thereby allowing it a large time-window to finish. In this manner, when the dirty cache-line gets evicted, the only operations left are the RESET operations which consume much less time.

A novel write-sensitive cache-line replacement policy called N-chance is proposed in [19]. The policy uses N as a parameter that can be configured by the hardware designers. At the time of cache-line replacement, this policy examines, starting from the first LRU line, the N least recently used lines. The first clean (non-modified) cache-line is chosen as the replacement candidate. If no clean line is found among the N candidates, the LRU entry itself is chosen for replacement. Clearly, when N = 1, the policy behaves just like the LRU replacement policy. Thus, this technique gives priority to non-dirty entries to dirty ones for eviction, thus helping in saving PCM writes. Experimental evaluation of this techniques found N = K/2 (K is the cache associativity) giving the best results among all values of N.

A scheme called Flip-N-Write is proposed by Cho et al. in [14]. The fundamental observation behind this scheme is that, given two words A and B, the minimum value among the two Hamming distances – of A to B and that of A to $\overline{B}$ – is bounded by $B/2$, where B is the number of bits in a word. Their scheme employs a special bit called the “flip-bit” for this purpose that accompanies each memory word, and that indicates whether the currently stored memory word is the actual word or its complement. At the time of updating the modified word to PCM memory, instead of immediately writing the new word to the given memory location, it first checks the Hamming distance of the modified word with the original word. If the distance is greater than $B/2$, it stores the complement of the modified word and sets the flip-bit to ON. Otherwise, the modified word is stored as it is, and the flip-bit is set to OFF. As a result, this technique restricts the maximum bit writes per word to $B/2$.

Another technique to reduce writes to PCM main memory is proposed in [20]. Their technique relies on data migration among the scratch-pad memories belonging to the various cores of an embedded chip multiprocessor. This migration comprises of shared data values, thus helping in the reduction of writes by preventing PCM write-backs. Program analysis techniques are used to determine the time of data migration and the corresponding target scratch-pad memory for data placement. The problem of data migration is modelled as a shortest path problem and an approach is outlined to find the optimal data migration path with minimal cost for both dirty and clean data blocks. Additionally, they also propose a technique of data re-computation which discards data that would have otherwise been written to PCM, instead choosing to re-compute the data value when the need for it arises.

Use of PCM in the context of three-dimensional (3D) die stacking is studied in [44]. The high power density of 3D stacking and its consequent heat generation, while being harmful for DRAM, can be used fruitfully for programming PCM cells. The work details a hybrid PCM-DRAM memory architecture that simultaneously exploits the high speed, infinite lifespan and power-efficient write access of DRAM along with the low idle-power read access of PCM. The technique uses an OS-level paging scheme that considers memory reference characteristics of the programs. It maintains multiple LRU queues to classify “hot-modified” and “cold-modified” pages, which are updated using counters that track the frequency of page updates. The hot-modified pages are moved to DRAM to reduce the write traffic to PCM, thus leading to energy savings and increase in PCM lifetime.

A different variation of hybrid PCM-DRAM page placement policy is proposed in [33]. The basic underlying observation is that a relatively small subset of pages used by an application are typically performance critical and can be accommodated in DRAM. Further, this subset of pages may change during the duration of program run, requiring active tracking by the system. The policy uses PCM and DRAM to hold mutually exclusive sets of pages. Page placement is determined by the hardware through monitoring of the access patterns of the applications currently executing on the system. Preference is given to place performance-critical and frequently written pages in DRAM, while the rest of the pages are accommodated in PCM.

A unique memory hierarchy organization consisting of NVM for both last-level cache and main memory is advocated in [45]. Their approach avoids logging and copy-on-write overheads for updates, by using a multi-version memory hierarchy to allow direct in-place modification of main memory data structures. In case of an update to the last-level NV cache, the corresponding cache-line holds the updated data, while the old version of the data is maintained in the NV main memory. Once this cache line gets evicted, the main memory is automatically updated with the new data. The performance due to such a hierarchy is shown to closely resemble that of the system without such persistence support. The authors also develop a software interface along with a set of hardware extensions to render atomicity and consistency support to this kind of a hierarchy.

2.2 Programming models and APIs

Another body of work targets persistence in PCM and other non-volatile memories, while making them accessible directly through CPU loads and stores, by putting them on the memory bus. A system called BPFS [16] provides an interface similar to a file system for managing PCM. A technique of *short-circuit shadow paging* is proposed which can perform in-place update of data in some cases, while requiring to copy just a subset of the data used by the common shadow paging technique to achieve consistency. The technique relies on two key hardware primitives to be available - atomicity of 8-byte writes and epoch barriers. Epoch barriers ensure that a set of writes is flushed to persistent memory before flushing the next set of writes, thereby bringing some degree of ordering in the writes, which are otherwise entirely dependent on when the data is flushed from cache.

A lightweight heap-based abstraction to persistent memories is provided by NV-Heaps [15] that supports ACID based transactions. These heaps are internally stored as files in the PCM memory and are portable across machines, while the DRAM is used to hold volatile data. A critical drawback of these hybrid systems is that they are particularly prone to issues such as dangling pointers. This may arise when, for instance, a pointer in PCM points to a memory location in DRAM which becomes invalid on a system restart. The framework thus implements features like garbage collection to avoid memory leaks as well as puts in place safety measures to combat inconsistency issues. The underlying processor support requirements for NV-Heaps are similar to those for BPFS.

Another work contemporary to NV-Heaps is Mnemosyne [41] that provides a programming interface to SCMs. The interface provides applications with the ability to declare variables as static so that they persist their values across system restarts. Additionally, it provides the facility of allocating persistent regions and heaps which automatically map to NVM. However, in contrast to NV-Heaps and BPFS, it provides persistence primitives that use instructions that are *already* available in most processors, namely non-temporal stores, fences and cache flushes. These primitives are then used to provide ACID transactions over these regions. This comes in the form of a lightweight transaction mechanism which provides APIs for direct modification of persistent variables and supports consistent updates to NVM.

Persistence support for arbitrary data structures is provided in REWIND [9] which is available in the form of a user-mode library. This library can be utilized to perform transactional updates which can be invoked directly from user code. To guarantee recoverability, it uses write-ahead logging – implemented by means of a doubly linked list – that supports recoverable operations on itself. A combination of techniques including lightweight logging, batching log data, persistent memory fences and non-temporal updates are employed to achieve good performance. Furthermore, it also outlines a two-level logging mechanism that uses an additional atomic AVL-tree structure built on top of the linked list, to aid in faster recovery.

A fast failure recovery mechanism for main memories comprised entirely of NVM type storage is discussed in [27]. The main idea is to defer flushing of transient data residing in processor cache and registers to NVM till the time of system failure. This "flush on fail" approach relies on *residual energy window* – a small period of time during which the DC supply to the system is maintained post a failure event is signalled – that is provided by the system power supply. Due to this saving of the temporal state of the system, it can quickly resume the state just before failure without having to rebuild it from scratch by referring to the logs. Thus, it makes system failure to appear as a suspend followed by a resume event. Delaying flushing of committed data to NVM renders an additional advantage of saving the high write latencies of such memories for intermediate updates, while also hiding those writes to increase NVM lifetime.

The problem of making the most cost-effective use of NVMs is addressed in [23]. Due to the high-costs of presently available NVM devices, instead of replacing the entire main memory with NVM, it advocates using NVM only for the logging subsystem to extract maximum returns on investment. For fast logging, it provides logging data structures that are updated directly via hardware-supported memory references, which avoids the overheads associated with a software based interface. The conventional central log buffering functionality suited for block-based devices is jettisoned in favor of per-transaction logging, leading to highly concurrent logging performance. Further, to mitigate the high latency incurred in direct writes to NVMs, two log persistence policies – flush-on-insert and flush-on-commit – are offered. These policies allot the log entry initially in volatile memory and later flush it to NVM, either immediately when the volatile entry is fully written, or asynchronously before the actual data is committed.

2.3 Application-level optimizations

Turning our attention to the database front, for the PCM_RAM memory model, write reduction techniques for index construction and for hash join are proposed in [12]. They recommend keeping the keys unsorted at the leaf nodes of the index. While this scheme saves on writes, the query response times are adversely impacted due to the increased search times. Similarly, for partitioning during hash join, a pointer based approach is proposed to avoid full tuple writes. Since we assume database to be PCM-resident, this partitioning step is obviated in our algorithms.

Some recent research work builds on top of the B^+ tree structure presented in [12]. For instance, avoiding sorting overhead during split of unsorted leaf nodes is proposed in [13]. They suggest a B^+ tree variant called the *sub-balanced unsorted node scheme*. This tree allows unbalanced splits using a random element as pivot, which avoids write and latency overheads incurred by an explicit sort step to find the median element. Additionally, they recommend delaying update of the pointers from the parent node during splits by using overflow chaining at the node being split. Such a scheme is shown to help in avoiding writes due to intermediate updates.

Another such work [11] targets alleviating the search overheads in unsorted B^+ tree index nodes. A small indirection array called the "slot-array" is advocated as a part of each node that indicates the sorted order of the index entries. This makes the simple binary search possible within the entries of the nodes, speeding up the key search process. A variety of combinations for node structure such as bitmap only, bitmap + slot-array as well as slot-array only nodes are evaluated for their advantages and drawbacks. Further, the paper also discusses how to perform consistent updates to the B^+ tree index without the overheads of shadow copying or undo-redo logging. This is achieved by means of atomic writes and the less expensive redo-only logging.

Allowing controlled imbalance in the B^+ tree to leverage the asymmetric I/O properties of NVM is is proposed in [39]. Such a tree, named *unbalanced B^+ trees or uB^+ trees*, defers tree balancing to a later period in time, until the overhead due to the imbalance crosses a certain threshold, the threshold being determined by the relative read and write latencies. Once the read penalty starts outweighing write savings, the tree goes through another round of balancing. In this manner, the uB$^+$ tree trades writes for extra reads to achieve performance and PCM lifetime benefits.

A B^+ tree that pre-allocates node space by predicting future key insert positions is proposed in [21]. It uses a novel prediction mechanism that takes current key distribution into account to come up with accurate estimates of future insertions; thereby helping in alleviation of write cost due to node splits.

For the DRAM_SOFT memory model, two classes of sort and join algorithms are presented in [40]. The first class divides the input into “write-incurring” and “write-limited” segments. The write-incurring part is completed in a single pass whereas the write-limited part is executed in multiple iterations. In the second class of algorithms, the materialization of intermediate results is deferred until the read cost (in terms of time) exceeds the write cost. Our work fundamentally differs from these approaches since in our DRAM_HARD model, there is no explicit control over DRAM. This means that we cannot selectively decide what to keep in DRAM at any point of time. It also implies that we may ultimately end up obtaining much less DRAM space than originally anticipated, due to other programs running in parallel on the system. As shown in Chapter 5, our algorithms have been designed such that even with restricted memory availability, they perform better than conventional algorithms in terms of writes.

At a more specific level, the sorting algorithms proposed in [40] employ a heap that may be constantly updated during each pass. If the available DRAM happens to be less than the heap size, it is likely that the updated entries will be repeatedly evicted, causing a large number of writes. Secondly, the join algorithms proposed in [40] involve partitioning the data for the hash table to fit in DRAM. However, since the results are written out simultaneously with the join process, and the result size can be as large as the product of the join relation cardinalities, it is likely that the hash table will be evicted even after partitioning.

Sorting algorithms for DRAM_SOFT model are also discussed in [38]. They split the input range into buckets such that each bucket can be sorted using DRAM. The bucket boundaries are determined using hybrid histograms having both depth-bound and width-bound buckets, the bound being decided depending upon which limit is hit later. The elements are then shuffled to group elements of the same bucket together, followed by sorting of each bucket within the DRAM. The sorting methodology used is quicksort or count-sort based on whether the bucket is depth-bound or width-bound respectively. A major drawback with this approach is that there is a high likelihood of an error in the approximation of the histogram, leading to DRAM overflow in some of the buckets. This would lead to additional writes since the overflowing buckets need to be split into adequately small fragments. Besides, the construction of the histogram itself may incur a number of writes.

2.4 Optimizations for flash memory

Finally, there has also been quite some research on speeding up query execution in *flash-resident* databases. For instance, incorporation of the flash read-write asymmetry within the query optimizer is discussed in [7]. The operators covered by them include scan, sort, hash join, apart from addressing materialization and re-scanning operations. The consequent cost model for these operators includes four parameters – sequential and random page accesses as well as number of read and write operations. Their focus however is restricted to modifying the operator's latency cost modelling to suit the flash environment, while no changes are made to the optimization process itself to consider the number of writes as a *separate* metric.

The effect of flash memory on the performance of indexes and joins is analysed in [26]. Aspects of index considered in the study include cost of key insertion and lookup, besides evaluation of the overall speed-up of using an index traversal over a sequential scan during query processing. For joins, four common external join algorithms are investigated – standard nested loops, index nested loops, index-based sort-merge and hash join. Their experiments indicate that flash offers significant advantages only at low predicate selectivities due to its superior random read performance over disks.

On similar lines to previous work, fundamental observations for disk based query processing are examined for their validity in the flash environment in [18]. The authors perform their study in the context of *ad-hoc* joins, i.e joins that do not use any index. Four such join algorithms, namely nested-loops, sort-merge, grace and hybrid hash join, are implemented on both flash and HDD and their performance is compared. The authors conclude that blocked I/O offers the same performance advantages for flash as it does for disks, whereas CPU performance begins to play as major a role for the choice of join algorithm as do its I/O requirements.

The use of a column-based layout has been advocated in [36] to avoid fetching of unnecessary attributes during scans. The paper presents a new join algorithm, called *FlashJoin*, that aims to reduce the I/O cost of join evaluation by minimizing the number of passes over the participating tables. The column-based layout is also leveraged for this join by restricting the columns fetched to only those that are participating in the join. Full tuple materialization is deferred to as late as possible in the plan tree. The authors recommend external merge sort for data that cannot fit in the DRAM.

Likewise, previous research on flash memory spans many other areas including caching optimizations, indexing techniques, transactional logging as well as wear-levelling. There have been a very few techniques that are geared towards write reduction at the operator execution level – they are also applicable to a PCM setting and are complementary to our work. For instance, the work in [7] can be applied to PCM for revising the latency cost model, though it has to be adapted to the idiosyncrasies of the PCM environment. Similarly, the techniques of a column-based layout and late materialization of join tuples discussed in [36] would also be advantageous for PCM to bring about reduction in writes. However, all of them differ from the work in this thesis in three key aspects.

First, none of the previous database research on flash-based systems has considered an equivalent of a DRAM_HARD memory organization with flash, i.e. where flash plays the role of main memory, augmented with a small DRAM buffer as an additional level of cache. Hence, the algorithms have been designed keeping explicit control over DRAM in mind. Second, flash supports writes only at the granularity of a page (4 KB). Further, data can only be written to a page that is in an *erased* state, and hence immediate in-place update to a page is not possible. The erase granularity of flash is much larger – typically 256 KB block at a time. In the DRAM_HARD model of flash organization, this implies that even a single-byte update, when evicted from DRAM to flash, would amplify to a page size write! Therefore, write-efficient algorithms for flash suited to DRAM_HARD model have to be strictly designed keeping locality of reference for writes in mind, and hence do not come with the flexibility of design choices available with PCM; something which we have leveraged in our operator implementations for PCM. This also explains why there has been a very limited work on query execution algorithms for flash memory. Thirdly, we have also looked at integrating the *writes* metric in the query optimizer. To the best of our knowledge, none of the previous database work on flash has considered this aspect of query processing.

Chapter 3

Operator Execution Algorithms

In this chapter, we present PCM-conscious algorithms for *sort*, *hash join* and *group-by* operators. These form the "workhorse" operators in most modern-day database systems. The algorithms use existing techniques to reduce writes while concurrently reducing response times for queries. In all these algorithms, the common underlying theme is trading writes for extra reads, besides localising writes so that they complete inside the DRAM without intermediate evictions.

Each algorithm is accompanied by the estimated number of writes that it is expected to incur at a 4B-word granularity. Note that, similar to the currently prevalent latency cost models, these estimators are incapable of predicting the exact number of writes. This is because the actual writes are highly dependent on a variety of factors that cannot be predicted using available database statistics – such as the *order* of input tuples in case of operators such as sort, or the runtime behavior of the DRAM. Also, while perfectly accurate estimators are certainly desirable, it is not a prerequisite for incorporation into query optimizers. The reason being that the estimators are meant to indicative of the relative costs of the various algorithms, and are therefore useful as long as their inaccuracy is within an acceptable degree – something which we examine in Chapter 5. These estimators are utilized later for integration with the redesigned optimizer in Chapter 6.

In the next section, we begin with detailing the sort operator. This is followed by the hash join operator in Section 3.2 and eventually by the group-by operator in Section 3.3.

3.1 The *Sort* Operator

Sorting is among the most commonly used operations in database systems, forming the core of operators such as *merge join*, *order-by* and some flavors of *group-by*. The process of sorting is quite write-intensive since the commonly used in-memory sorting algorithms, such as *quicksort*, involve considerable data movement. In the single pivot quicksort algorithm with n elements, the average number of swaps is of the order of $0.3nln(n)$ [42]. There are other algorithms such as *selection sort* which involve much less data movement, but they incur *quadratic* time complexity in the number of elements to be sorted, and are therefore unsuitable for large datasets.

The main advantage associated with the quicksort algorithm is that it has good average-case time complexity and that it sorts the input data in-place. 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. Thereafter, there would be no further evictions during swapping and the remaining sorting process would finish inside the DRAM itself.

sorting algorithm to converge fast to partition sizes below DRAM size with fewer number of swaps. For uniformly-distributed data, these requirements are satisfied by *flashsort* [28]. On the other hand, for data with skewed distribution, we propose a variant of flashsort called *multi-pivot flashsort*. This algorithm adopts the pivot selection feature of the quicksort algorithm into flashsort in order to tackle the skewness in data.

Both these algorithms are discussed in detail in the following sections.

3.1.1 Data with uniform distribution

The flashsort algorithm 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.* A brief description of each of these phases is as follows:

3.1.1.1 Classification phase

The classification phase divides the input data into equi-range partitions comprising of contiguous and disjoint ranges. That is, if p partitions are required (where p is an input parameter), the difference between the minimum and the maximum input values is divided by p. Subsequently, each tuple is mapped to a partition depending on in which range the value of the sorting attribute of the tuple lies. Specifically, a tuple with attribute value v is assigned to $Partition(v)$, computed as

$$Partition(v) = 1 + \lfloor \frac{(p-1)(v - v_{min})}{v_{max} - v_{min}} \rfloor$$

where v_{min} and v_{max} are the smallest and largest attribute values in the array, respectively. The number of tuples in each such partition is counted to derive the boundary information. 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.

3.1.1.2 Permutation phase

The Permutation phase moves the elements to their respective partitions by leveraging the information obtained in the Classification phase. The elements are swapped in a cyclic manner to place each element inside its partition boundary with a single write step per element.

3.1.1.3 Short-range Ordering phase

The resulting partitions, each having size less than D, are finally sorted in the Short-range Ordering phase using quicksort. Note that, by virtue of their size, these partitions are not expected to incur any evictions during the process of sorting.

3.1.2 Data with non-uniform distribution

In the case when the data is non-uniformly distributed, the equi-range partitioning used by flashsort fails to produce equi-sized partitions. This is because the number of tuples in each range is now dependent on the skew of the data. We therefore propose an alternative algorithm, called *multi-pivot flashsort*, which uses multiple pivots instead to partition the input tuples. These pivots are randomly-chosen from the input itself, in the same manner as conventional quicksort selects a single pivot to create two partitions. The chosen pivots are subsequently leveraged to partition the input during sorting.

The modified phases of this alternative implementation of the flashsort algorithm, along with their pseudo-codes, are described next.

3.1.2.1 Classification phase

In the Classification phase, we divide the input relation into p partitions, where $p = \lceil \frac{N_R L_R}{D} \rceil$, using $p-1$ random tuples as pivots. Since the pivots are picked at random, the hope is that each partition is approximately of size D. These pivots are then copied to a separate location and sorted. Subsequently, we scan through the array of tuples in the relation, counting the number of elements between each consecutive pair of pivots. This is accomplished by carrying out, for each tuple in the array, a binary search within the sorted list of pivots.

In spite of the random choice of pivot values, it is quite possible that some partitions may turn out to be larger than the DRAM. We account for this possibility by conservatively creating a larger number of initial partitions. Specifically, the number of partitions is $p = \lceil c \times \frac{N_R L_R}{D} \rceil$, where $c \geq 1$ is a design parameter similar to the one used in the flashsort algorithm. Subsequently, we consider each pair of adjoining partitions and coalesce them if their total size is within the DRAM size, after leaving some space for bookkeeping information.

While the above heuristic approach is quite effective, it still does not guarantee that all the resultant partitions will be less than DRAM size. The (hopefully few) cases of larger-sized partitions are subsequently handled during the Short-range Ordering phase.

The pseudo-code for the Classification phase is outlined in Algorithm 1.

Algorithm 1 Classification Phase

```
array[] is the array of input tuples
c is a design parameter $\geq 1$

1: p = $\lceil c \times \frac{N_R L_R}{D} \rceil$
2: randIndex[] = generate $p - 1$ random indexes
3: pivot[] = array[randIndex];
4: sort(pivot[])
5: size[] = 0...0                                  ▷ size of sub-arrays
6: partitionStart[] = 0...0                        ▷ starting offset of each partition
7: for i=1 to $N_R$ do
8:     partition = getPartition(array[i])
9:     size[partition]++
10: end for                                        ▷ Time complexity=$N_R \times log_2 p$
11: cumulative = 0
12: for i=1 to $p$ do
13:     cumulative = cumulative + size[i]
14:     partitionStart[i+1] = cumulative
15: end for                                        ▷ Time complexity=$p$
16: return partitionStart[]
```

3.1.2.2 Permutation phase

The Permutation phase uses the information gathered in the Classification phase to group tuples of the same partition together. A slight difference from flashsort here is that the attribute value now needs to be compared against the sorted list of pivots to determine the partition of the tuple. The pseudo-code for the Permutation phase is shown in Algorithm 2. The maximum number of writes is bounded by $N_R L_R$, corresponding to the worst case where *every* tuple has to be moved to its correct partition.

Algorithm 2 Permutation Phase

partitionStart[] is obtained from Classification Phase
nextUnresolvedIndex[] indicates the next position to be examined for each partition

```
 1: nextUnresolvedIndex[] = partitionStart[]
 2: for i=1 to $N_R$ do
 3:     curPartitionCorrect = getPartition(array[i])
 4:     if i between partitionStart[curPartitionCorrect] and partitionStart[curPartitionCorrect+1] then
 5:         nextUnresolvedIndex[curPartitionCorrect] = i+1
 6:         continue
 7:     else
 8:         firstCandidateLoc = i
 9:         presentCandidate = array[i]
10:         flag = 1
11:         while flag do
12:             targetPartitionStart = nextUnresolvedIndex[curPartitionCorrect]
13:             targetPartitionEnd = partitionStart[curPartitionCorrect + 1]
14:             for k=targetPartitionStart to targetPartitionEnd do
15:                 nextPartitionCorrect = getPartition(array[i])
16:                 if k between partitionStart[nextPartitionCorrect] and
17:                     partitionStart[nextPartitionCorrect + 1] then
18:                     continue
19:                 else if k == firstCandidateLoc then
20:                     flag = 0                                ▷ Indicates it is a cycle
21:                 end if
22:                 swap(presentCandidate, array[k])
23:                 nextUnresolvedIndex[curPartitionCorrect] = k+1
24:                 curPartitionCorrect = nextPartitionCorrect
25:                 break
26:             end for
27:         end while
28:     end if
29: end for                                     ▷ Time complexity=$N_R \times log_2 p$
```

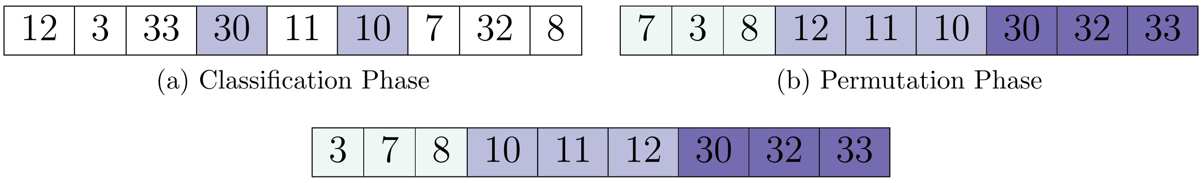
3.1.2.3 Short-range Ordering phase

Finally, each of the partitions are sorted separately using conventional quicksort to get the final PCM sorted array. For partitions that turn out to be within the DRAM size, the Short-range Ordering phase is completed using conventional quicksort. On the other hand, if some larger-sized partitions still remain, we recursively apply the multi-pivot flashsort algorithm to sort them until all the resulting partitions can fit inside DRAM and can be internally sorted.

Algorithm 3 Short-range Ordering Phase

```
1: for i=1 to p do
2:     if size[p] < D then
3:         quicksort (partition p)
4:     else
5:         multi-pivot flashsort (partition p)
6:     end if
7: end for
```

Figure 3.1 visually displays the steps involved in the multi-pivot flashsort of an array of nine values. First, in the Classification phase, 30 and 10 are randomly chosen as the pivots. These pivots divide the input elements into 3 different ranges: (< 10), (≥ 10, < 30), (≥ 30). The count of elements in each of these ranges is then determined by making a pass over the entire array – in the example shown, three elements are present in each partition. Then, in the Permutation phase, the elements are moved to within the boundaries of their respective partitions. Finally, in the Short-range Ordering phase, each partition is separately sorted within the DRAM.



(a) Classification Phase

(b) Permutation Phase

(c) Short-range Ordering Phase

Figure 3.1: Multi-Pivot Flashsort

3.1.3 PCM write analyses

In the quicksort algorithm, given an array with randomly permuted N_R tuples, for simplicity of analysis let us assume the chosen pivot in each phase of recursion is the median of the partition tuples. Since the tuples are arranged randomly, the probability that the tuple is in the right partition is 1/2. In other words, $N_R/2$ tuples are expected to be incorrectly placed.

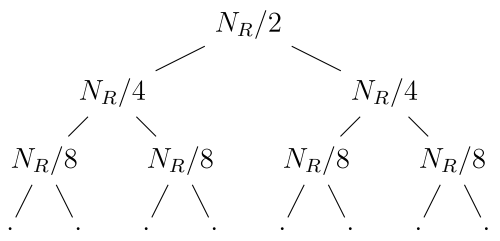


Figure 3.2: Recursion tree for quicksort swaps

In the next level of the recursion tree, there will be about $N_R/4$ incorrectly placed tuples for both the partitions, again totalling to $N_R/2$. This total of $N_R/2$ tuples would continue at each level of the recursion tree. This recursion tree is shown in Figure 3.2.

If Hoare partitioning [17] is used for partitioning, wherein each misplaced element is swapped with another misplaced element in the opposite partition, moving the elements to their right partitions will incur $N_R/2 \times L_R$ bytes of writes at each level. This trend of writes will continue until the size of the partition reaches to less than D, i.e when the level $l = \lceil log_2(\frac{N_R L_R}{D})\rceil$. Post this, there would be a total writes of $N_R L_R$ bytes incurred when all the individual partitions finish sorting within DRAM and are written out. Hence, we get the total word-writes as:

$$\begin{aligned} W_{sort\_conv} &= \frac{\sum_{i=1}^{l}(N_R/2 \times L_R) + N_R L_R}{4} \\ &= \frac{0.5 N_R L_R \lceil log_2(\frac{N_R L_R}{D})\rceil + N_R L_R}{4} \\ &= \frac{N_R L_R (0.5\lceil log_2(\frac{N_R L_R}{D})\rceil + 1)}{4} \end{aligned} \tag{3.1}$$

In the flashsort algorithm, 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 $N_R L_R$ bytes of 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 writes of $N_R L_R$ bytes upon eventual eviction of sorted partitions to PCM.

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

$$W_{sort\_uniform} = \frac{2N_R L_R}{4} = \frac{N_R L_R}{2} \tag{3.2}$$

The write analysis of multi-pivot flashsort follows that of the flashsort algorithm. A negligible number of writes would be incurred during the copying and sorting of the pivots. As mentioned, the writes during Permutation phase would be below $N_R L_R$ bytes. The creation of additional partitions by choosing extra pivots, and their subsequent coalescing, increases the likelihood that each partition is below DRAM size – akin to that in flashsort. Therefore the total word-writes is again estimated to be

$$W_{sort\_non\_uniform} = \frac{N_R L_R}{2} \tag{3.3}$$

3.1.4 Response time analyses

To arrive at an estimate of response time, we can divide the total time into read, computation and write times.

During reads in conventional quicksort algorithm, there will be $\frac{N_R L_R}{B} \times T_{PCM}$ cycles incurred for each level of the recursion tree until we reach level l calculated earlier. Post that, there will be no further rounds of misses as the data for subsequent partitions would fit inside DRAM. Based on the time complexity of quicksort algorithm, the computation time can be estimated as $N_R log_2 N_R$. Accounting for the additional time consumed during writes will add another $4 \times W_{sort\_conv} \times T_{PCM}$ cycles (since each word-write corresponds to 4 bytes). Thus the total cycles incurred can be calculated as

$$\begin{aligned} T_{sort\_conv} &= \sum_{i=1}^{l}(\frac{N_R L_R}{B} \times T_{PCM}) + N_R log_2 N_R + \frac{4W_{sort\_conv}}{B} \times T_{PCM} \\ &= \frac{(N_R L_R(1.5\lceil log_2(\frac{N_R L_R}{D})\rceil + 1)}{B} \times T_{PCM} + N_R log_2 N_R \end{aligned} \tag{3.4}$$

In the flashsort algorithm, during the Classification phase, as the data is read serially with the counters being updated, there will be $\frac{N_R L_R}{B} \times T_{PCM}$ cycles incurred. The Permutation phase will require another $\frac{N_R L_R}{B} \times T_{PCM}$ cycles as each individual partition is read serially. Finally, $\frac{N_R L_R}{B} \times T_{PCM}$ cycles will be also be needed for the Short-range Ordering phase, as the data required for sorting each individual partition can fit in DRAM and hence needs to be read just once.

For computation, both Classification and Permutation phases will consume N_R cycles each. If there are $p = \frac{N_R L_R}{D}$ partitions, each partition in Short-range Ordering phase will incur $\frac{N_R}{p} log_2 \frac{N_R}{p}$ cycles, adding an overall overhead of $N_R log_2 \frac{N_R}{p}$ cycles. Finally, $\frac{4W_{sort\_uniform}}{B} \times T_{PCM}$ cycles will be required for writing the blocks to PCM. Hence, the total time for flashsort would be

$$\begin{aligned} T_{sort\_uniform} &= \frac{3N_R L_R}{B} \times T_{PCM} + 2N_R + N_R log_2 \frac{N_R}{p} + \frac{4W_{sort\_uniform}}{B} \times T_{PCM} \\ &= \frac{5N_R L_R}{B} \times T_{PCM} + 2N_R + N_R log_2 \frac{D}{L_R} \end{aligned} \tag{3.5}$$

Calculation of response time for multi-pivot flashsort algorithm follows the same pattern as that of the flashsort algorithm. The only difference is in the computation time for Classification and Permutation phases, where N_R cycles is replaced by $N_R log_2 p$ cycles, owing to the comparisons involved with pivots for finding the associated partitions. Thus the overall cycles are given by:

$$\begin{aligned} T_{sort\_non\_uniform} &= \frac{3N_R L_R}{B} \times T_{PCM} + 2N_R log_2 p + N_R log_2 \frac{N_R}{p} + \frac{4W_{sort\_non\_uniform}}{B} \times T_{PCM} \\ &= \frac{5N_R L_R}{B} \times T_{PCM} + 2N_R log_2 N_R + N_R log_2 \frac{L_R}{D} \end{aligned} \tag{3.6}$$

3.2 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 is accompanied by a 4-byte 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 of P bytes each time a new entry of H bytes is inserted.

Our first modification is to use a well-known technique of allocating space to hash buckets in units of *pages* [24]. 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. In order to facilitate this, we can choose $B' \geq \frac{N_R}{256}$; where B' denotes the number of buckets in the hash table. 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.

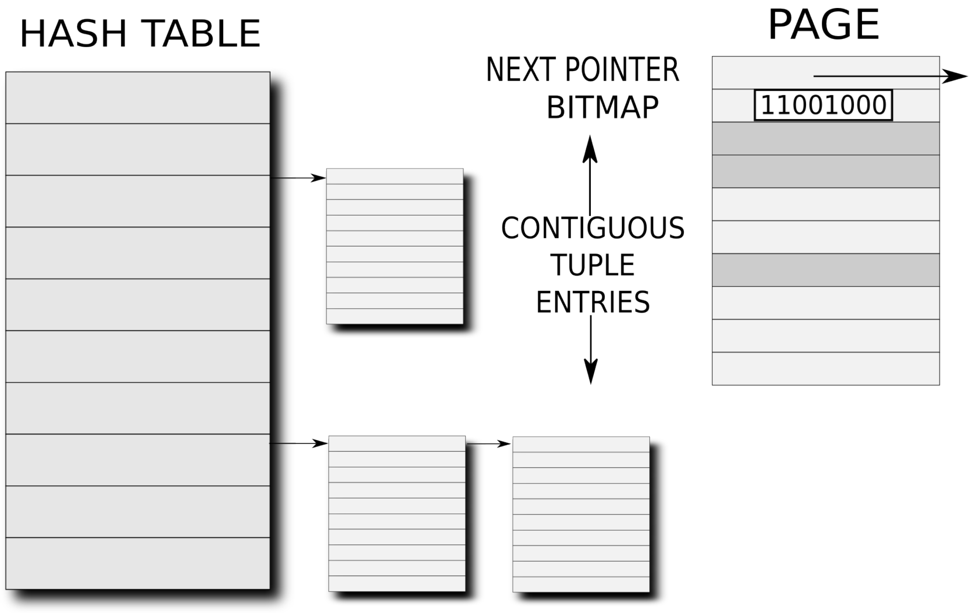


Figure 3.3: Paged Hash Table

Figure 3.3 displays the hash table organization with the proposed optimizations.

3.2.1 PCM write analyses

We ignore the writes incurred while initializing each hash table bucket since they are negligible in comparison to inserting the actual entries. The total writes for conventional hash join is given by

$$W_{hj\_conv} = \frac{N_R \times (H + P + 4) + N_j \times L_j}{4} \tag{3.7}$$

If the size of the page is given by S_{page}, there will be $E_{page} = \frac{S_{page}}{H}$ entries per page. There would now be a 1-byte hash value per entry, along with 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

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

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

$$W_{hj} \approx \frac{N_R \times (H + 1) + N_j \times L_j}{4} \tag{3.8}$$

3.2.2 Response time analyses

As the bulk of the time in hash join is spent in I/O, we divide the response time for hash join into just read and write times. During the build phase, $\frac{N_R \times L_R}{B} \times T_{PCM}$ cycles will be spent in reading tuples of relation R. Similarly, $\frac{N_S \times L_S}{B} \times T_{PCM}$ cycles will be consumed in reading tuples of relation S during the probe phase.

For conventional hash join, as each hash entry in the bucket is dynamically allocated and hence can land on any PCM block, searching for a matching entry in a bucket would involve fetching a PCM block for each entry. This would incur an overhead of $N_S \times F \times T_{PCM}$ cycles. Retrieving the matching R tuples – assuming each tuple of relation S joins with a single tuple from relation R – would require another $N_S \times \lceil \frac{L_R}{B} \rceil \times T_{PCM}$ cycles. The total cycles consumed will therefore be

$$\begin{aligned} T_{hj\_conv} &= (\frac{N_R L_R}{B} + N_S(\frac{L_S}{B} + F + \lceil \frac{L_R}{B} \rceil) + \frac{4W_{hj\_conv}}{B}) \times T_{PCM} \\ &= (\frac{N_R(L_R + H + P + 4) + N_j L_j}{B} + N_S(\frac{L_S}{B} + F + \lceil \frac{L_R}{B} \rceil)) \times T_{PCM} \end{aligned} \tag{3.9}$$

In PCM-conscious hash join, since each page is a contiguous block of memory, the cycles consumed for search inside a hash table bucket will be $N_S \times \lceil \frac{H \times F}{min(S_{page}, B)} \rceil \times T_{PCM}$. Moreover, $\frac{F}{256} \times \lceil \frac{L_R}{B} \rceil \times T_{PCM}$ cycles will be incurred for fetching the tuples of relation R including those due to false positives – owing to the 1-byte limited hash value length. Hence, the overall time taken would be

$$\begin{aligned} T_{hj} &= (\frac{N_R L_R}{B} + N_S \times (\frac{L_S}{B} + \lceil \frac{H \times F}{min(S_{page}, B)} \rceil + \frac{F}{256} \times \lceil \frac{L_R}{B} \rceil) + \frac{4W_{hj}}{B}) \times T_{PCM} \\ &= (\frac{N_R(L_R + H + 1) + N_j L_j}{B} + N_S \times (\frac{L_S}{B} + \lceil \frac{H \times F}{min(S_{page}, B)} \rceil + \frac{F}{256} \times \lceil \frac{L_R}{B} \rceil)) \times T_{PCM} \end{aligned} \tag{3.10}$$

3.3 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 $N_g \times L_g$ bytes.

3.3.1 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 3.2 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.

3.3.1.1 PCM write analyses

In hash table based group-by, the number of separate entries in the hash table would be N_g. Similar to the analysis of writes for conventional hash join in Equation 3.7, creation of these entries would incur writes of $N_g \times (H + 4 + P)$ bytes. Writes due to updates of the aggregate field would be $N_R \times A$ bytes. This would give a total word-writes of:

$$W_{gb\_ht\_conv} = \frac{N_g \times (H + 4 + P) + N_R \times A + N_g \times L_g}{4} \tag{3.11}$$

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

$$W_{gb\_ht} = \frac{N_g \times (H + 1) + N_R \times A + N_g \times L_g}{4} \tag{3.12}$$

3.3.1.2 Response time analyses

The calculations for response time for hash-based grouping are similar to those for hash-join in Section 3.2.2. For simplicity of calculation, we assume that the number of distinct values of the group-by column is small compared to the overall cardinality of the table, and that all of them appear at the beginning of the table. In that case, the number of entries in each bucket is fixed once those initial tuples are read, and subsequent tuples will only update the aggregate value.

There will be $\frac{N_R L_R}{B} \times T_{PCM}$ cycles incurred for reading tuples of table R. For probing the table, another $N_R \times F \times T_{PCM}$ cycles will be consumed. Finally, fetching the actual tuple corresponding to the entry would incur $N_R \times \lceil \frac{L_R}{B} \rceil$ cycles. Thus, the overall time consumed will be

$$\begin{aligned} T_{gb\_ht\_conv} &= (\frac{N_R L_R}{B} + N_R \times F + N_R \times \lceil \frac{L_R}{B} \rceil) + \frac{4W_{gb\_ht\_conv}}{B}) \times T_{PCM} \\ &= (N_R(\frac{L_R + A}{B} + F + \lceil \frac{L_R}{B} \rceil) + \frac{N_g(H + P + L_g + 4)}{B}) \times T_{PCM} \end{aligned} \tag{3.13}$$

For PCM-conscious hash-based group-by, the expression for the number of PCM blocks required for searching for a matching tuple inside a bucket would change to $N_R \times \lceil \frac{(H+A) \times F}{min(S_{page}, B)} \rceil$. Moreover, there will be some false-positives due to the limited hash value length, due to which the number of tuples fetched for matching entries would change to $N_R \times \frac{F}{256} \times \lceil \frac{L_R}{B} \rceil$, thereby resulting in the response time expression of

$$\begin{aligned} T_{gb\_ht} &= (\frac{N_R L_R}{B} + N_R \times \lceil \frac{(H + A) \times F}{min(S_{page}, B)} \rceil + N_R \times \frac{F}{256} \times \lceil \frac{L_R}{B} \rceil + \frac{4W_{gb\_hj}}{B}) \times T_{PCM} \\ &= (N_R(\frac{L_R + A}{B} + \frac{F}{256} \times \lceil \frac{L_R}{B} \rceil + \lceil \frac{(H + A) \times F}{min(S_{page}, B)} \rceil) + \frac{N_g(H + L_g + 1)}{B}) \times T_{PCM} \end{aligned} \tag{3.14}$$

3.3.2 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 3.1 to use *pointers* in both the Permutation and Short-range Ordering phases, subsequently leveraging these pointers to perform aggregation on the sorted tuples.

3.3.2.1 PCM write analyses

For conventional sort-based grouping, the writes would be akin to those incurred for sorting, plus an additional writes of $N_g \times L_g$ bytes incurred due to writing out the grouped tuples. Thus, following the same analysis as in Equation 3.1, the total word-writes would be:

$$W_{gb\_sort\_conv} = \frac{N_R L_R (0.5\lceil log_2(\frac{N_R L_R}{D})\rceil + 1) + N_g \times L_g}{4} \tag{3.15}$$

Coming to the calculations of writes for flashsort based group-by, the full tuple writes of $2N_R L_R$ bytes which were incurred in the flashsort scheme, are now replaced by $2N_R \times P$ bytes 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

$$W_{gb\_sort} = \frac{2N_R \times P + N_g \times L_g}{4} \tag{3.16}$$

3.3.2.2 Response time analyses

The response time calculation for conventional sort-based group-by is similar to that of sort in Section 3.1.4, with an additional read of the entire list of tuples – amounting to $\frac{N_R L_R}{B}$ PCM blocks – to compute the aggregates. There will also be a computation time overhead of N_R for the aggregation pass over the sorted tuples. The response time will therefore be

$$\begin{aligned} T_{gb\_sort\_conv} &= \sum_{i=1}^{l} (\frac{N_R L_R}{B} \times T_{PCM}) + \frac{N_R L_R}{B} \times T_{PCM} + N_R + N_R log_2 N_R + \frac{4W_{gb\_sort\_conv}}{B} \times T_{PCM} \\ &= \frac{N_R L_R (\lceil 1.5 log_2(\frac{N_R L_R}{D})\rceil + 2) + N_g L_g}{B} \times T_{PCM} + N_R(1 + log_2 N_R) \end{aligned} \tag{3.17}$$

To compute the response time for flashsort based group-by operator, we divide the response time into read, computation and write times – similar to the analyses of the sort operator in Section 3.1.4. During the Classification phase, $\frac{N_R L_R}{B}$ blocks will be read from PCM. Afterwards, during the Permutation phase, $\frac{N_R(L_R+P)}{B}$ PCM blocks will be required for reading both tuples and pointers to those tuples. Finally, another set of $\frac{N_R(L_R+P)}{B}$ PCM blocks will be fetched during the Short-range Ordering phase. Note that there are no additional block reads required for aggregation, as it will be done for each partition while it is in DRAM during the Short-range Ordering phase. The computation time requirements of the group-by operator will be $3N_R + N_R log_2 \frac{N_R}{p}$, which is obtained after accounting for the additional time required for the aggregation pass over sorted data. Hence, the overall time consumed by sort-based group-by operator will be

$$\begin{aligned} T_{gb\_sort} &= \frac{N_R(3L_R + 2P)}{B} \times T_{PCM} + 3N_R + N_R log_2 \frac{N_R}{p} + \frac{4W_{gb\_sort}}{B} \times T_{PCM} \\ &= \frac{N_R(3L_R + 4P) + N_g L_g}{B} \times T_{PCM} + N_R(3 + log_2 \frac{D}{L_R}) \end{aligned} \tag{3.18}$$

Chapter 4

Implementation Details

This chapter covers the details of our actual physical implementations for the experimental evaluation of our proposed optimizations for PCM. We describe the library of operator algorithms that we created to compare the performance of our implementations with their corresponding PostgreSQL counterparts. This includes aspects such as programming language, parameter values and the choice of standard library routines.

Since PCM memory is as yet not available commercially to end-users, we have taken recourse to a simulated hardware environment to evaluate the impact of the PCM-conscious operators. For this purpose, we chose Multi2sim [37], an open-source application-only[1] simulator. Further, due to its lack of native support for PCM, we made a major extension to its existing memory module to model PCM device.

In the next sections, we present a detailed discussion on each of these aspects of our implementations.

4.1 Operators Library

We wrote both our library as well as the query code in *C* programming language – the total implementation involving about 5K lines of code. Since applications written in C generally execute faster than those written in its object-oriented counterparts such as C++ and Java,

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

this ensured that the data-intensive query executions finished in manageable running times despite orders of magnitude slowdowns introduced by a simulator.

The next sections describe each of our operator implementations in sequence.

4.1.1 Sort

For the native version of sort, we used the PostgreSQL source code, which essentially utilizes the common quicksort algorithm for sorting data that can fit in main memory. Pivot selection in this implementation is based on the *median-of-3* rule, in which the median element among three randomly selected elements is chosen as pivot, to increase the probability of balanced partitioning.

As mentioned in earlier chapters, the fundamental building block of our sorting operator was the flashsort algorithm. Since no open-source library for flashsort was available, we implemented a fresh version of the algorithm for our evaluation. Later, we added customizations to this algorithm to build separate versions for uniform and non-uniform data distributions. Ideally, this distribution should be identified using the histograms built on the columns of the database. However, since these statistics are available only within the database system, we separately supplied this information to our library to invoke the correct version of the algorithm. To sort the DRAM sized partitions obtained after the classification step of both the versions, the standard *qsort()* routine was used.

We now describe the details of the two implementations of the flashsort algorithm.

4.1.1.1 Uniform Data Distribution

This version partitioned the data on the based of ranges. It used two passes – one to determine the smallest and the largest elements in the input array and the other to count the number of elements belonging to each partitions. The partition count was chosen to be twice the ratio of PCM to DRAM size in order to leave space for additional data structures. This was followed by the in-place permutation and short-range ordering steps, as described in Section 3.1. Determining the partition of an element during each of these phases was instantaneous since it required a direct hash of the value of the element.

4.1.1.2 Non-Uniform Data Distribution

This version used random pivots to partition the input data. To generate the random indices for pivot selection, we used the *rand()* function available in the standard C library. This function was in turn seeded using a constant argument to the *srand()* function to ensure that the experimental results were repeatable. The selected pivots were copied to a separate location in PCM and sorted using the *qsort()* function. In contrast to the previous implementation, finding the partition of an element was slightly slower as required a binary search through the pivots.

4.1.2 Hash Join

The version of hash join used by PostgreSQL is the hybrid hash join algorithm which builds a hash table for the current batch of join tuples, while simultaneously adding tuples for subsequent batches to overflow buckets represented as files on disk. A new dynamically allocated memory space is allotted for each hash table entry and is connected to existing entries in that bucket using a pointer. Moreover, the load factor used for deciding the bucket count in the hash table is fixed at a constant value of 10. For our library, we implemented the simple hash join algorithm while retaining the hash table structure used by the native PostgreSQL implementation, since the join tuples in our case can fit entirely in PCM.

The PCM-conscious version of hash join differed from the native version in that the hash table used pages to allocate memory to each bucket. Each such page had contiguous slots that could accommodate 32 entries, and the same number was used for our load factor. Further, a 4 Byte bitmap was allocated at the head of each page, each bit in the bitmap corresponding to a particular page-slot. Single-byte hash values were used, and separate space was reserved in each page to host these values in immediate succession.

For both the implementations, we used a common hash function of *SHA-256* to map tuples to buckets. This was done to ensure fairness in comparison of the two approaches.

4.1.3 Group-By

The group-by operator in PostgreSQL leverages the implementations of the previous two operators in order to perform the aggregation. Depending on the type of grouping among sort and hashed, the appropriate primitives – sorting in case of sort-based, and hash table construction and probing in case of hash-based – are invoked. Inside our library also, we reused most of the code of sort and hash join operator implementations in group-by. Some customizations were made specific to the group-by operator, for instance, using pointer-based sorting for PCM-conscious grouping using sort. These specific customizations of the are described next.

4.1.3.1 Sort-based Grouping

Sorting for PCM-conscious group-by was carried out using 4-Byte pointers to the actual tuples, and these pointers were eventually used to compute the final aggregates. Though these pointers consumed extra memory space, it was a very small fraction of the space occupied by the actual tuples. Moreover, once the aggregate computations were over, the PCM memory associated with both the pointers and the actual sorted tuples was reclaimed immediately.

4.1.3.2 Hash-based Grouping

For hash table construction for group-by, provisions were made in each hash table entry to store the additional aggregate field. Secondly, in contrast to the hash table probing for a join attribute requiring search in the entire bucket (since multiple inner relation tuples can have identical join column values), the search stopped as soon as the matching value of the aggregated column was found.

4.2 Simulator Enhancements

Adding PCM support to Multi2sim required modifications to several of its components. This is because the behaviour of PCM is quite unlike all existing components of the memory hierarchy. For instance, changes were required to the timing subsystem of the simulator to support asymmetric read and write latencies. Similarly, implementation of the data-comparison-write (DCW) [43] scheme required tracking multiple versions of the data for the same memory location. Furthermore, extensions were necessary to the cache replacement policy to support N-chance [19] replacement. Clearly, all this required a significant programming effort, the enhancements running into nearly 6K lines of code, besides requiring complex debugging of simulator crashes.

In the following sections, we describe the low-level details of each of these enhancements.

4.2.1 Hybrid Main Memory

We implemented PCM as another layer of memory between the DRAM and the Hard Disk. The size of the existing DRAM was reduced to a fraction of the PCM size. Additionally, its semantics were modified so that it acts as a buffer for PCM blocks, like an additional level of cache between the L2 cache and the PCM in an inclusive memory hierarchy. The organization of the DRAM was also altered to a set-associative structure.

These extensions required addition of a fresh module, beside setting up new L2 to DRAM and DRAM to PCM communication channels, in the Multi2sim memory configuration file. Changes were also required in the memory subsystem to ensure that the block requests from the cache to PCM are routed via the DRAM buffer.

4.2.2 New DRAM Replacement Policy

The default policies available in Multi2sim for cache line replacement were restricted to least recently used (LRU), first in first out (FIFO) and random block replacement. We added the novel N-chance replacement policy to this pool that is more suited to future PCM-based hardware. This is because this policy gives eviction preference to non-dirty lines over dirty ones, thereby saving on expensive writes.

To implement this, we maintain an LRU queue of the DRAM blocks that is updated on each block access. At the time of eviction, we sequentially examine the blocks in the LRU queue starting from the oldest block, halting either when a clean block is found or when the number of blocks examined exceeds N. If no clean block is found, we choose the oldest entry itself as the candidate block for replacement. Since setting N to half the block associativity was found to give the best results in [19], we used the same value in our implementation.

4.2.3 Tracking DRAM-PCM Data

Another major feature addition involved duplicating the entire address space of the applications to keep track of updates. Current architectural simulators just maintain a dirty bit for each cache line to indicate that a certain memory location corresponding to that line has been written to. However, this no longer suffices in a PCM setting since the exact modification to each block are necessary to determine the number of writes for that block.

To achieve this, we create a duplicate of a PCM block as soon as a request for that block is made by the DRAM. All the updates to that block are recorded in the duplicate while the original block is left intact. In this way, the contents of duplicate data block reflect the DRAM contents, while the original data block represents the data residing in PCM. After the block is evicted from DRAM, the duplicate is removed to avoid wastage of memory.

4.2.4 Data Comparison Write Scheme

This feature leveraged the separate data functionality discussed earlier, to read the original contents of the PCM memory location corresponding to the evicted DRAM data block, before writing back the updated contents. Following such an approach [43] is found to give significant write savings, especially in cases where the updates to the DRAM block are minimal.

Immediately upon eviction of a dirty DRAM block, we compare the evicted DRAM block's contents with those of the PCM version by means of the duplicate block mentioned earlier. If any of the bits of a particular memory word is found to differ, the entire word is written back to the PCM memory. Otherwise, the DRAM word is simply discarded without incurring any writes or their associated latency.

4.2.5 Asymmetric Read-Write Latencies

While making the latency measurements of PCM operations, a critical distinction that needs to be made is whether the operation is a read or a write. Furthermore, in case of a write, the latency is additionally dependent on the exact number of word modifications taking place. Hence, the timing simulation also requires the details of the actual data as its input to accurately determine the latency incurred in the process.

We modified the timing simulation to feature the extra latency of a PCM write as compared to a read operation. The number of modified words, as measured by the DCW scheme, were supplied to it and the latency calculations for each write were done on the basis of this input. Further, the fact that a PCM read precedes every PCM write was also accounted by inserting additional delay during each write request.

4.2.6 Wear Distribution

A critical but often overlooked metric in application-level research for PCM write optimization is wear distribution. Most such studies assume the presence of an underlying architectural wear-levelling mechanism that is capable of taking care of skewed writes. However, in rare cases is the overhead of invoking such routines accounted for. Clearly, an application that performs its write operations in an evenly distributed manner is likely to avoid the inherent latency of wear-levelling, while an application with skewed writes is expected to incur such a penalty.

Therefore, we ensured that we add wear distribution among the set of metrics on which we evaluate our algorithms. Our measurement of wear distribution included introducing multiple counters in the simulator, one corresponding to each 256B block, which were incremented on each PCM write. Note that, although a finer granularity of measurement (at a byte level) was also possible, the number of counters required for such a degree of measurement required a colossal amount of memory for a 4 GB address space.

4.2.7 Intermediate Statistics

For analysing effect of individual operators on the performance of queries, it was necessary to get intermediate performance numbers during the execution of the queries. This provision was missing in Multi2sim since it was only restricted to get the final execution statistics. Also, since the queries themselves execute atop the simulator, it was tricky to get the same query to communicate with the simulator to get these numbers.

We devised an inter-process communication mechanism using a disk file based approach. This file could be written by the running program and periodically probed by the simulator.

When an operator finished its part in the execution of a query, it wrote a command to the simulator in this file to dump the statistics, and this command would be carried out by the simulator in the next probe cycle. In this manner, we could obtain individual per-operator figures for the TPC-H benchmark queries that we executed as part of our experimental evaluation.

Chapter 5

Experimental Evaluation

This chapter details our experimental setup along with the results of our evaluation of the proposed techniques. The initial part outlines our experimental settings in terms of the hardware parameters, the database and query workload, and the performance metrics on which we evaluate the PCM-conscious operator implementations. Afterwards, our experimental results on benchmark queries are presented, which is followed by an analysis of the savings due to each of the operators individually. To simulate the effect of multiple processes running in parallel, we subsequently measure the effect of reduced DRAM availability on the performance of our implementations. Finally, we validate the write estimators proposed in Chapter 3 by comparing the estimated writes with the actual writes incurred during query execution.

5.1 Architectural Platform

We evaluated the algorithms on Multi2sim in cycle-accurate simulation mode. 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 5.1. These values are scaled-down versions, w.r.t. number of lines, of the hardware simulation parameters used in [31] – 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

Table 5.1: 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)

levels in the hierarchy are maintained as per the original configurations in [31].

5.2 Database and Queries

For the data, we used the default 1GB database generated by the TPC-H [2] 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 [1] implementations. To this library, we added the PCM-conscious versions described in the previous sections. Afterwards, for each of the queries, a custom end-to-end execution code was written, that made function calls into the above libraries. Further, the code also created and maintained intermediate data structures that held the inter-operator tuples during execution.

Due to the large overhead of writing complete execution code for each query, we restricted ourselves to three queries: Query 13 (Q13), Query 16 (Q16) and Query 19 (Q19), that cover a diverse spectrum of the experimental space. 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 Chapter 6. The three SQL queries and execution plans associated with them are shown next.

5.2.1 Queries

Q13

```
select c_count, count(*) as custdist
from ( select c_custkey, count(o_orderkey)
from customer left outer join orders on
c_custkey = o_custkey and
o_comment not like '%pending%accounts%'
group by c_custkey
) as c_orders (c_custkey, c_count)
group by c_count
order by custdist desc, c_count desc;
```

Q16

```
select p_brand, p_type, p_size,
count(distinct ps_suppkey) as supplier_cnt
from partsupp, part
where p_partkey = ps_partkey
and p_brand <> 'Brand#35'
and p_type not like 'ECONOMY BURNISHED%'
and p_size in (14, 7, 21, 24, 35, 33, 2, 20)
and ps_suppkey not in ( select s_suppkey
from supplier
where s_comment like '%Customer%Complaints%')
group by p_brand, p_type, p_size
order by supplier_cnt desc, p_brand, p_type, p_size;
```

Q19

```
select sum(l_extendedprice* (1 - l_discount)) as revenue
from lineitem , part
where
( p_partkey = l_partkey and p_brand = 'Brand#23'
and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
and l_quantity >= 5 and l_quantity <= 5 + 10
and p_size between 1 and 5 and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON' )
or
( p_partkey = l_partkey and p_brand = 'Brand#15'
and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
and l_quantity >= 14 and l_quantity <= 14 + 10
and p_size between 1 and 10 and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON')
or
( p_partkey = l_partkey and p_brand = 'Brand#44'
and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
and l_quantity >= 28 and l_quantity <= 28 + 10
and p_size between 1 and 15 and l_shipmode in ('AIR', 'AIR REG')
and l_shipinstruct = 'DELIVER IN PERSON' );
```

5.2.2 PostgreSQL Query Plan Trees

We obtained the execution plans for the three TPC-H queries using the PostgreSQL version 9.4.2. Our PostgreSQL server was hosted on a SUN Ultra 24 machine with the following hardware configuration:

Processor: Intel Core 2 Quad-Core 3.0 GHz processor

Cache: 128 KB L1, 12 MB L2

Memory: 8 GB memory

Disk: 750 GB, 7200 RPM

OS: Ubuntu Linux 12.04 (64-bit)

The corresponding plans are shown in Figures 5.1 – 5.3. The same plans were used in our code to execute the queries.

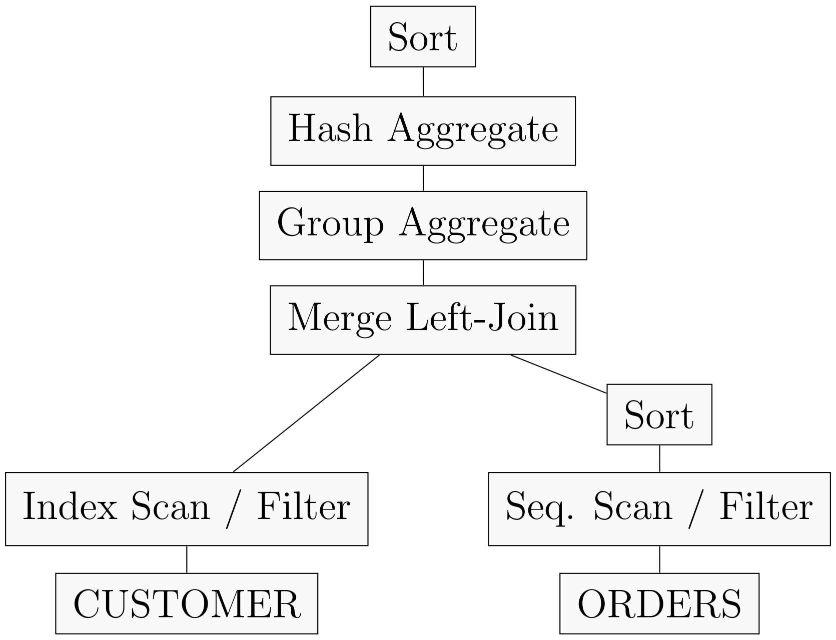


Figure 5.1: Q13 Plan

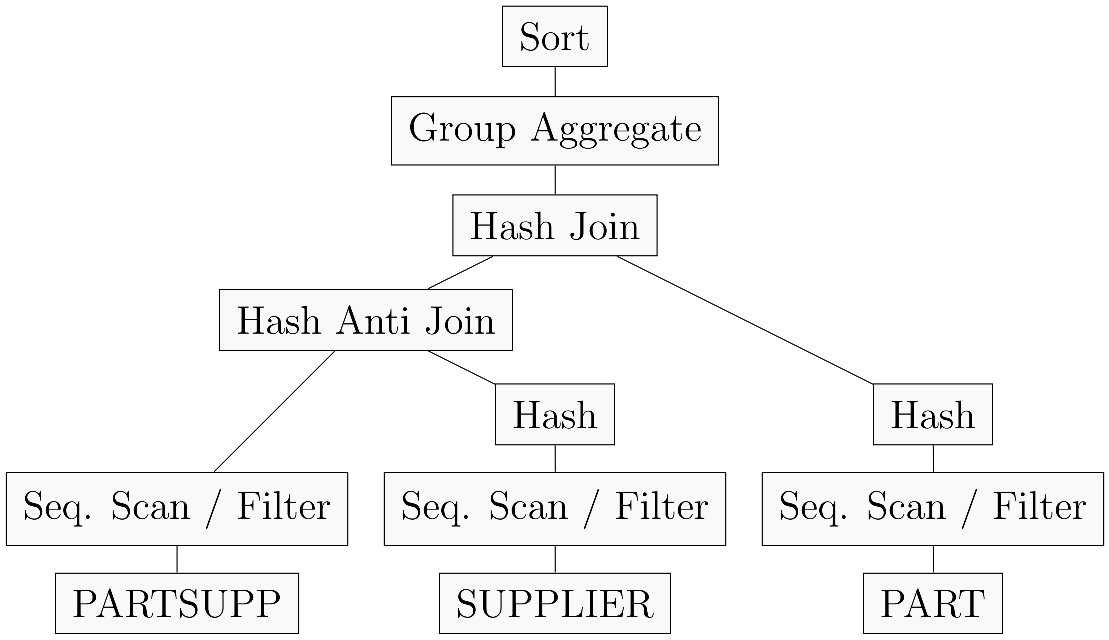


Figure 5.2: Q16 Plan

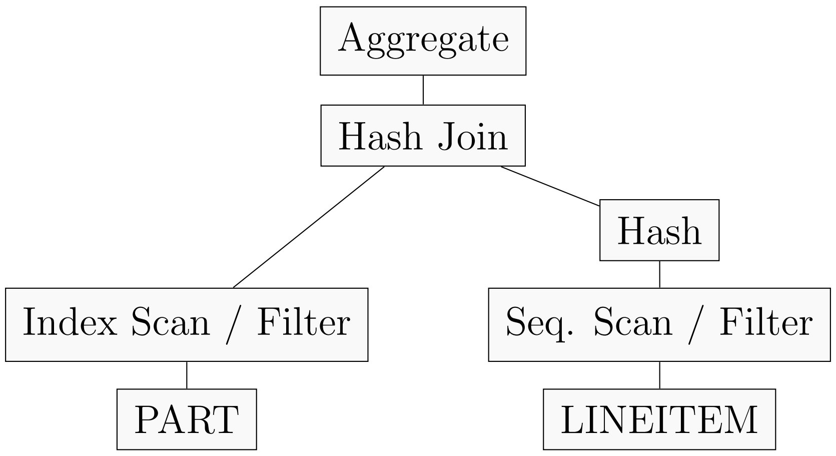


Figure 5.3: Q19 Plan

5.3 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.

5.4 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 Figures 5.4-5.7. In addition to the standard TPC-H with uniform data distribution, we also show results for the sort operator implementation on a skewed version of TPC-H, generated using a Zipfian distribution [10] with a skew factor of $Z = 1$. 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 5.4, 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.

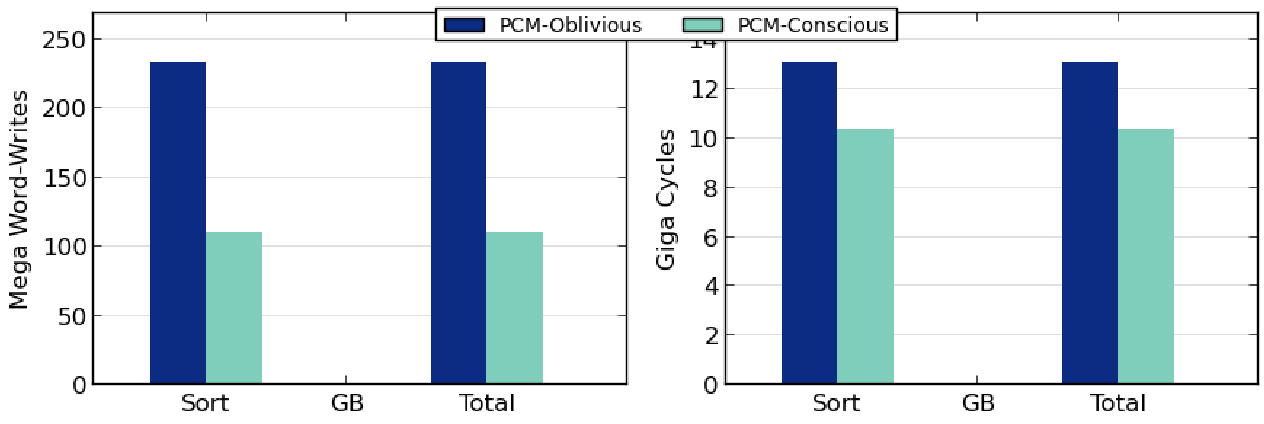


Figure 5.4: Q13 Performance

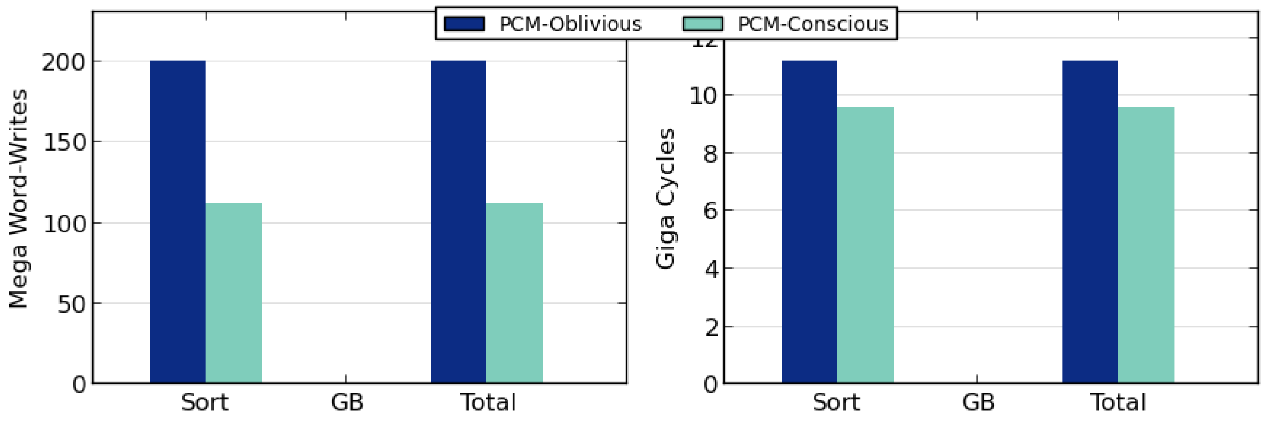


Figure 5.5: Q13 Performance (skewed TPC-H)

For Q13 execution on skewed TPC-H, for which we used the multi-pivot flashsort algorithm, the corresponding performance numbers (Figure 5.5) are comparatively lesser. Specifically, savings of 44% and 14% are observed in writes and cycles, respectively.

Turning our attention to Q16 in Figure 5.6, 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.

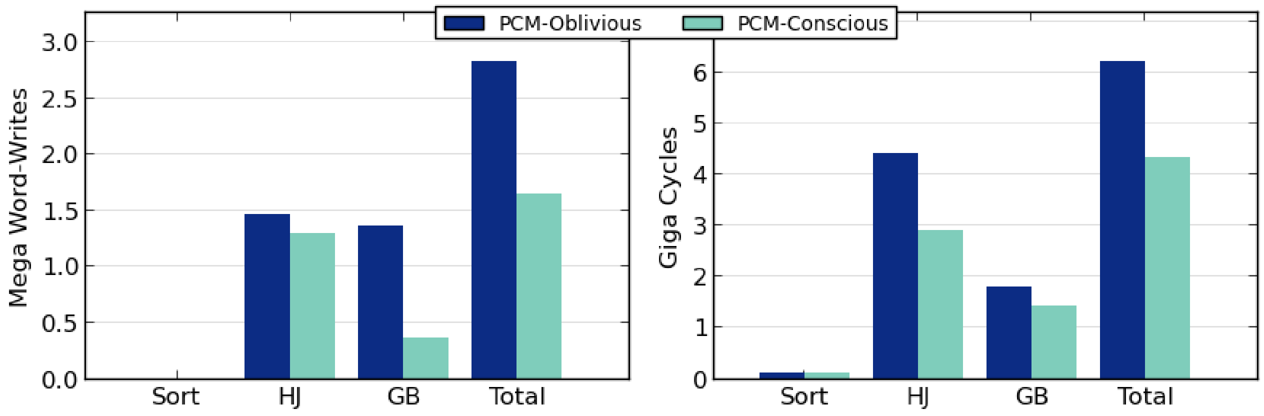


Figure 5.6: Q16 Performance

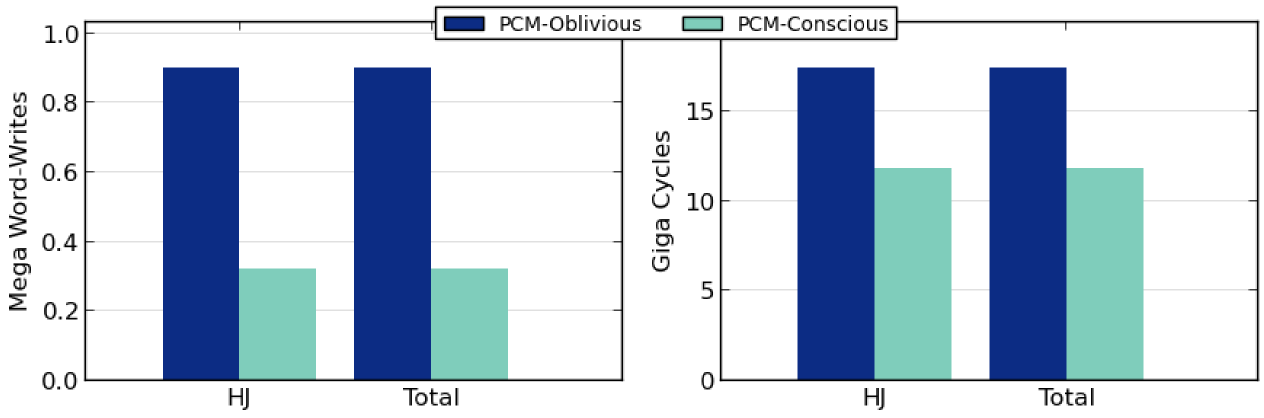


Figure 5.7: Q19 Performance

Finally, moving on to Q19 in Figure 5.7, where hash join is essentially the only operator, the savings are around 64% with regard to writes and 32% in cycles.

5.5 Operator-wise Analysis

We now analyse the savings due to each operator independently and show their correspondence to the analyses in Sections 3.1–3.3 .

5.5.1 Sort

For Q13 execution on uniform TPC-H, as already mentioned, we observed savings of 53% in writes and 20% in cycles, since the number of tuples to be sorted was high. Similarly, on skewed TPC-H, these figures were 44% (writes) and 14% (cycles). In the case of Q16, the data at the sorting stage was found to be much less than the DRAM size as the number of tuples coming out of the group-by operator was low. 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.

5.5.2 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 5.6. 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 5.7. The source of the enhancement was the 3 bytes of writes saved due to single-byte hash values[1], and additionally, the page-based aggregation of hash table entries.

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

5.5.3 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 5.6. 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 5.4.

5.6 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 Figures 5.8 - 5.10, 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.

5.7 DRAM size sensitivity analysis

We now move on to covering the scenario wherein the available DRAM size at runtime is lesser than that anticipated prior to the query execution – for example, due to concurrent query processing. To model this scenario, we experiment with DRAM sizes of 512 KB, 1 MB and 2 MB. Note that, for each of these cases, all the algorithms continue to assume 4 MB as the

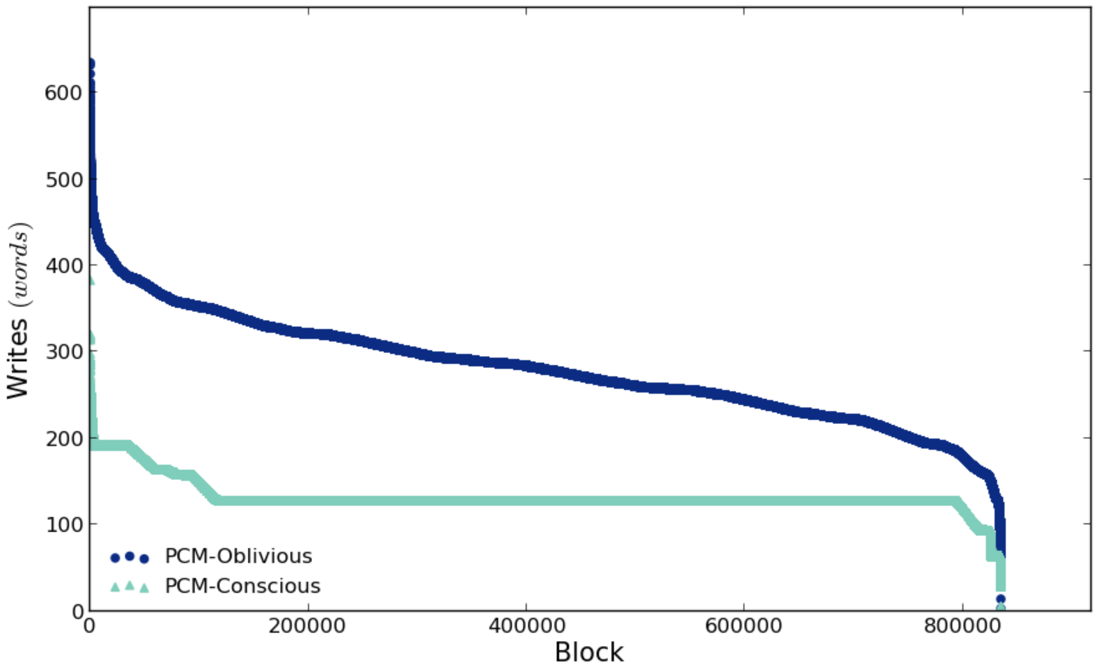


Figure 5.8: Q13 wear distribution

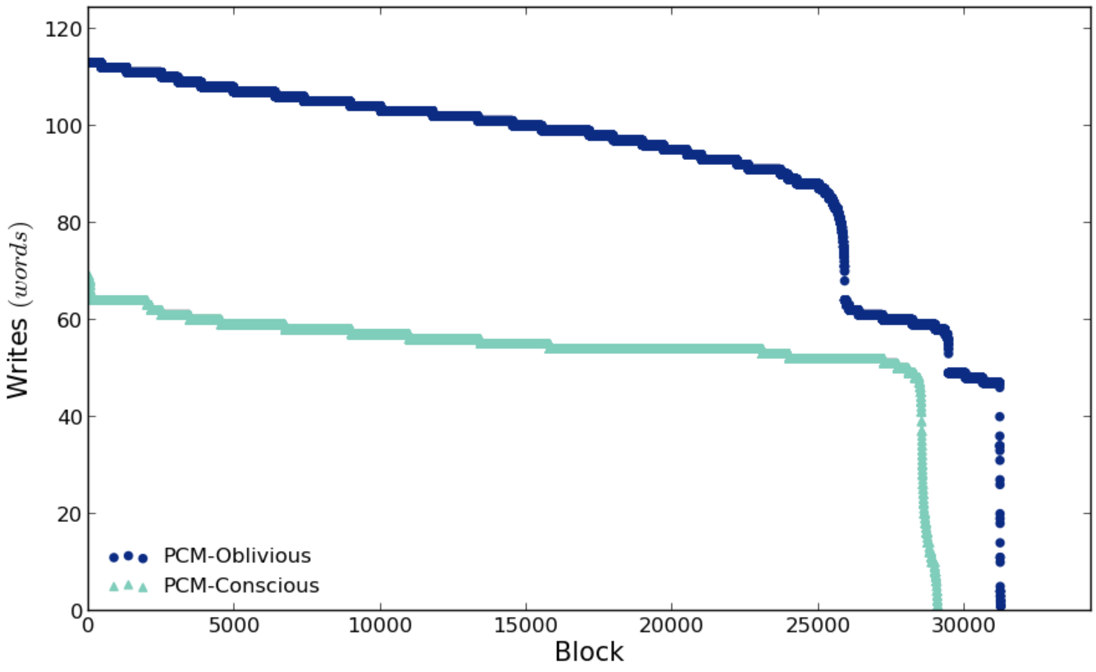


Figure 5.9: Q16 wear distribution

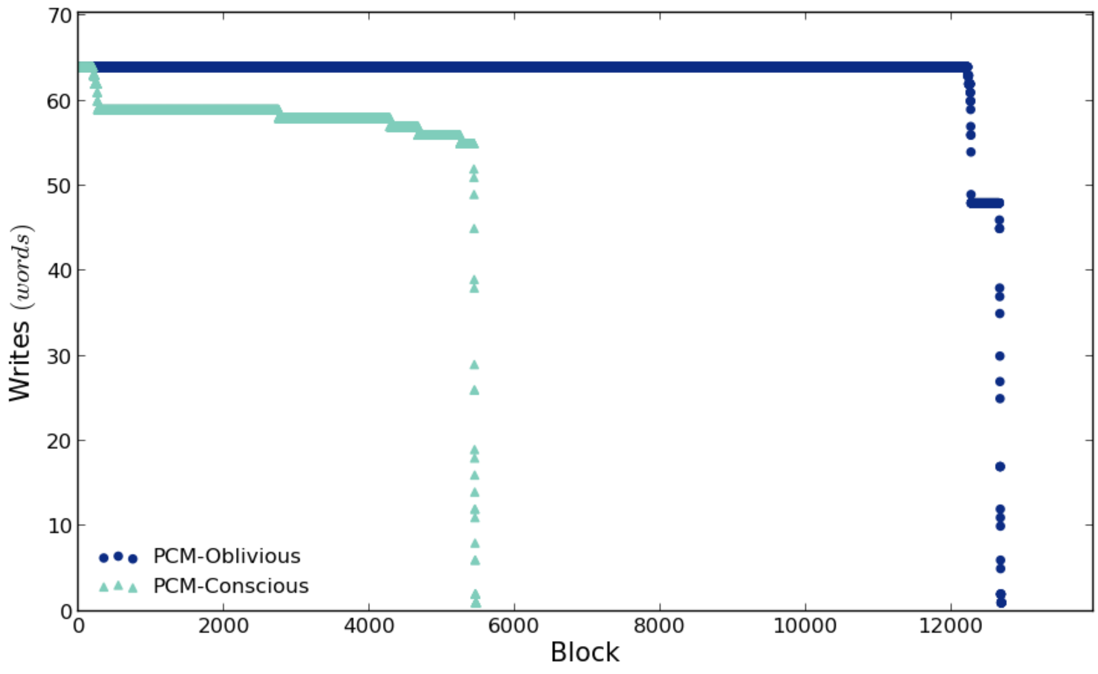


Figure 5.10: Q19 wear distribution

available DRAM size, oblivious to the actual reduction in the availability of DRAM, and hence continue to execute accordingly. The results are shown in Figures 5.11, 5.12 and 5.13 for the Sort, Hash Join and Group By operators, respectively.

In Figure 5.11, we observe that the Sort operator provides average savings for Q13 of about 47% in Writes and 20% in Cycles.

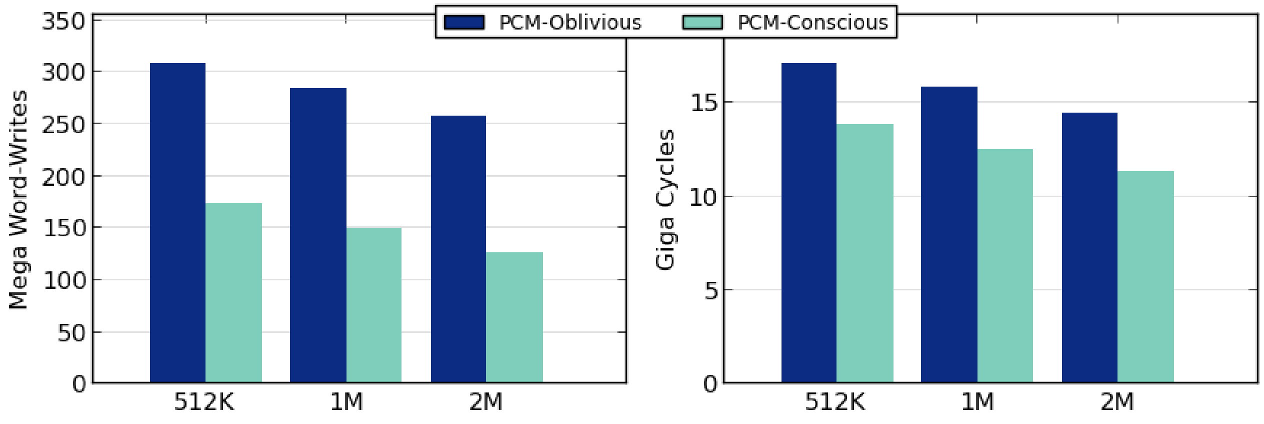
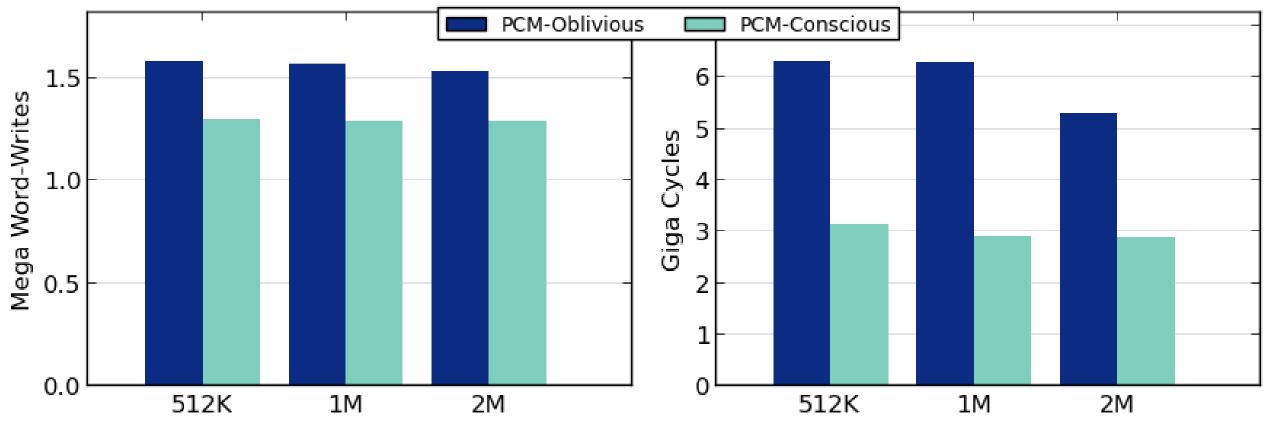
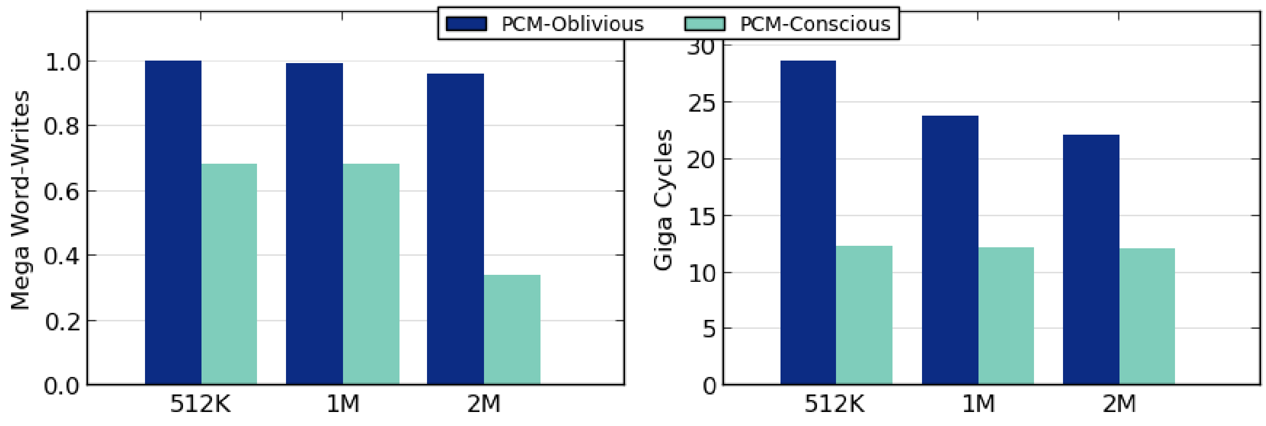


Figure 5.11: Sort (Q13)

In case of Q16, the hash join operator rendered 17% improvement in writes and 50% in running time. The hash join in Q19 continued to benefit with savings of about 42% in writes and again 50% in running time.



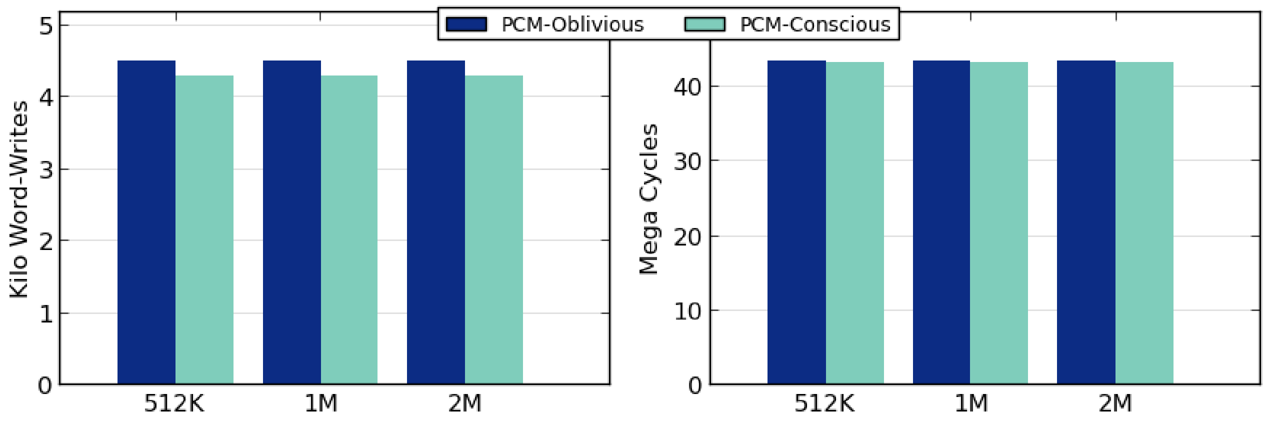
(a) Q16



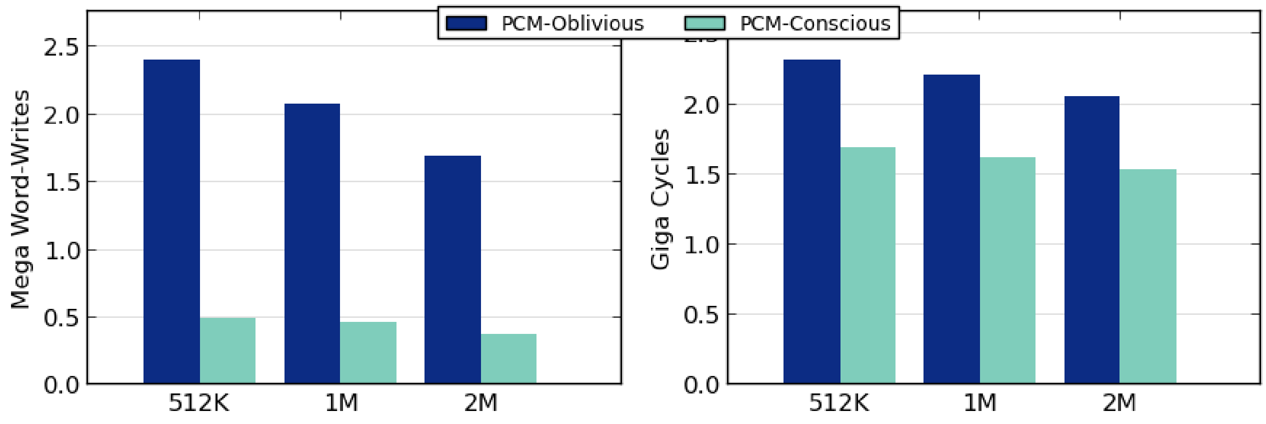
(b) Q19

Figure 5.12: Hash Join

For the group-by operator in Q16, writes savings of 78% and running time savings of 26% were observed for the group-by operator. The hash table for group-by being much smaller than all DRAM variants for Q13, the change in the DRAM size didn't reflect much of a change in writes and running time - yielding the same average of 4% and 1% for the two metrics respectively. Thus, we can conclude that the savings obtained by our algorithms are not restricted to the environment where the DRAM is exclusively available to one process. Instead, our al-



(a) Q13



(b) Q16

Figure 5.13: Group By

gorithms can handle the unpredictability in the DRAM availability due to multiple running processes equally well, thereby being sufficiently robust to volatile system behaviour.

5.8 Validating Write Estimators

We now move on to validating the estimators (presented in Sections 3.1 through 3.3) for the number of writes incurred by the various database operators.

5.8.1 Sort

The size of the *orders* table is approximately 214 MB. The flashsort algorithm incurred writes of 110.6M. On the other hand, the writes for multi-pivot flashsort algorithm were 112.1M. Replacing the values for $N_R L_R$ with the table size in Equations 3.2 and 3.3, we get the writes as

$$W_{sort\_uniform} = W_{sort\_non\_uniform} = (214 \times 10^6)/2 = 107\text{M}$$

Thus the estimate is close to the number of observed word-writes.

5.8.2 Hash Join

For the hash join in Q19, the values of N_R, H, N_j, L_j are 0.2M, 4 bytes, 120 and 8 bytes, respectively. Substituting the parameter values in Equation 3.8, the writes are given by:

$$W_{hj} = (0.2 \times 10^6 \times 5 + 120 \times 8)/4 \approx 0.25\text{M}$$

which is close to the actual word-writes of 0.32M.

5.8.3 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 3.16 results in:

$$W_{gb\_sort} = (2 \times 119056 \times 4 \ + 18341 \times 48)/4 = 0.46\text{M}$$

This closely corresponds to the observed word-writes of 0.36M.

A summary of the above results is provided in Table 5.2. 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.

Table 5.2: Validation of Write Estimators

Operator	**Estimated Mega Word-Writes** (e)	**Observed Mega Word-Writes** (o)	**Error Factor** ($\frac{e-o}{o}$)
Sort (uniform)	107	110.6	-0.03
Sort (non-uniform)	107	112.1	-0.05
Hash Join	0.25	0.32	-0.22
Group-By	0.46	0.36	0.27

Chapter 6

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 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 PCM writes. Having formulated the write estimators for each operator (as described in Sections 3.1 to 3.3), this modification is straightforward. All that it requires is to use those estimators to get an approximate count of the number of memory operations that are writes, and then multiply that count with the difference between the write and read latencies to derive the additional delay. Further, we also introduce a new metric of *write cost* in the operator cost model that represents the number of incurred writes for a plan in the PCM environment. Again, the same write estimators prove useful by approximating this figure apriori. 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 (DP) 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. In the next sections, we describe two algorithms that can be used to incorporate both the metrics in the optimizer. These algorithms vary in terms of their ease of implementation as well as their ability to come up with an optimal plan.

6.1 Heuristic-Propagation Algorithm

As we just discussed, propagation of the full set of sub-plans is an infeasible solution. To curb the multiplicative effect of plans at each operator node, we instead propose a heuristic-propagation algorithm that employs an algorithmic parameter, *local threshold* λ_l ($\geq \lambda$), at each internal node of the DP tree. 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 $c_i \leq (1+\lambda_l)c_o$. For the case when the same least WC is shared between multiple sub-plans, we choose the sub-plan with the lowest LC if their LC values differ. Otherwise, the propagated sub-plan is an arbitrary selection from these plans. We observed that setting $\lambda_l = \lambda$ delivered reasonably good results in this regard.

In light of these modifications, let us revisit Query Q13, for which the default plan is shown in Figure 5.1. With just the revised latency costs (i.e. $\lambda = 0$), the optimizer identified a new execution plan shown in Figure 6.1 wherein the merge left-join between the *customer* and *orders*

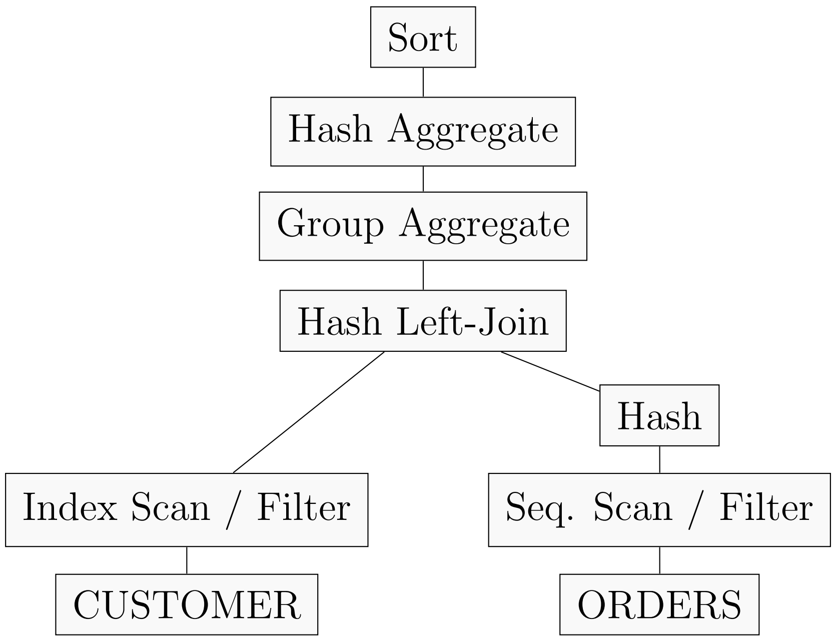


Figure 6.1: Q13 Plan with revised latency costs

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 6.3(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, 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 (Figure 6.2), 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 6.3(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). This is an example to highlight the potential benefits that can be obtained by tuning the cost models of database operators for PCM environment. The results clearly indicate that

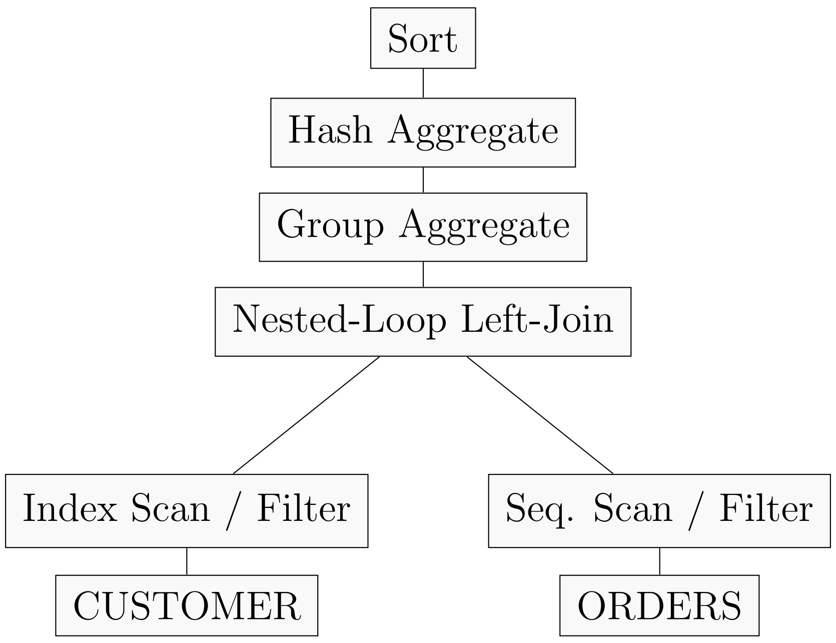


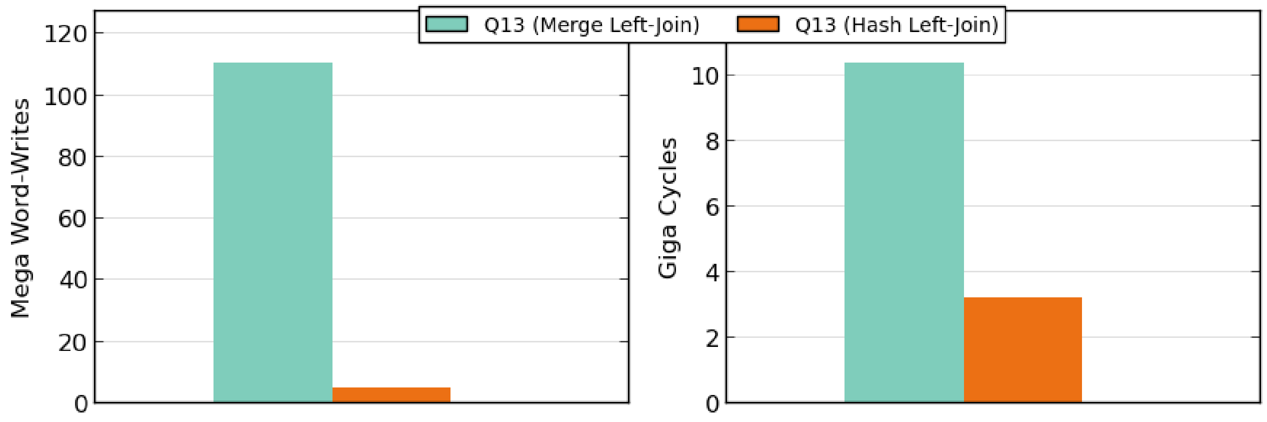
Figure 6.2: Q13 Plan with slack (λ) value of 5

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.

6.1.1 Experimental Results

We performed our experiments on PostgreSQL 9.2.4 database system, with data generated by the TPC-H [2] benchmark at a scale factor of 1. We made modifications to the query optimizer engine to accept different values of the the latency slack (λ) and return the WC-optimal plan corresponding to that slack. In Table 6.1, we summarize the results obtained by keeping the latency slack value as 1. Note that the set of base plans considered here is consisting of those plans that were proposed by the optimizer *after* accounting for the additional delay due to writes, i.e. with the revised LC model. We omit the results of those queries for which the WC-optimal plan was the same as the LC-optimal plan. In the table, the column *Time* refers to the time incurred in the optimization process.

As we can see, with a slack of 1, the optimizer gave a different plan for 2 queries – Q15 and Q18. For Q15, savings of 24% in WC were obtained at the cost of doubling the LC of the plan shown in Figure 6.4. This was obtained by switching hash join to nested-loop join at the



(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

Figure 6.3: Integration with Query Optimization and Processing Engine

topmost node. On the other hand, for Q18, minuscule savings of 1% in WC were derived by trading off 48% increase in LC – specifically by replacing the hash join between customer and (orders $\bowtie$ lineitem) tables by nested-loop join in Figure 6.5.

From the table, we can see that the time taken for optimization was very low for both the native optimizer as well as the modified optimizer for slack value of 1. Moreover, owing to the low times, there was a considerable fluctuation in the numbers for each run. As a result, the times we have presented in the table are those obtained for one of the sample runs, and are mainly to serve as a rough estimate for the time taken in the optimization process.

Table 6.1: Comparison of plans generated by Heuristic Algorithm ($\lambda = 1$)

Query	**LC-optimal Plan**			**WC-Optimal Plan**		
	Time(ms)	LC($\times 10^6$)	WC($\times 10^3$)	Time(ms)	LC($\times 10^6$)	WC($\times 10^3$)
Q15	12	0.06	2.1	20	0.12	1.6
Q18	15	237	3570	57	352	3525

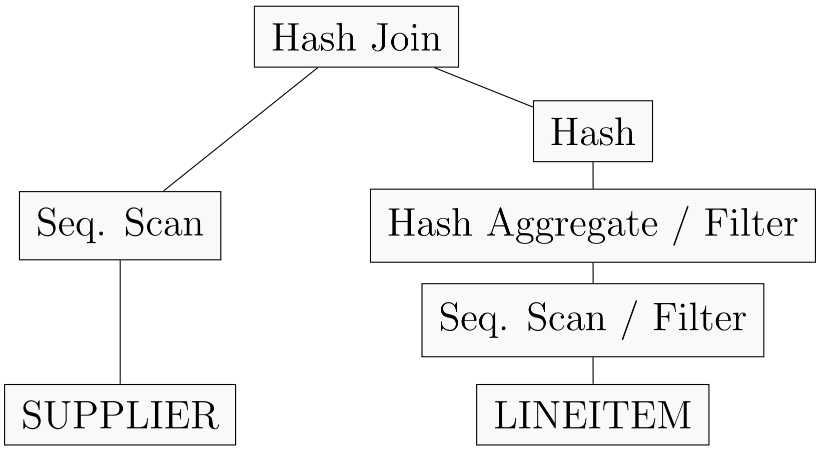


Figure 6.4: Q15 Plan

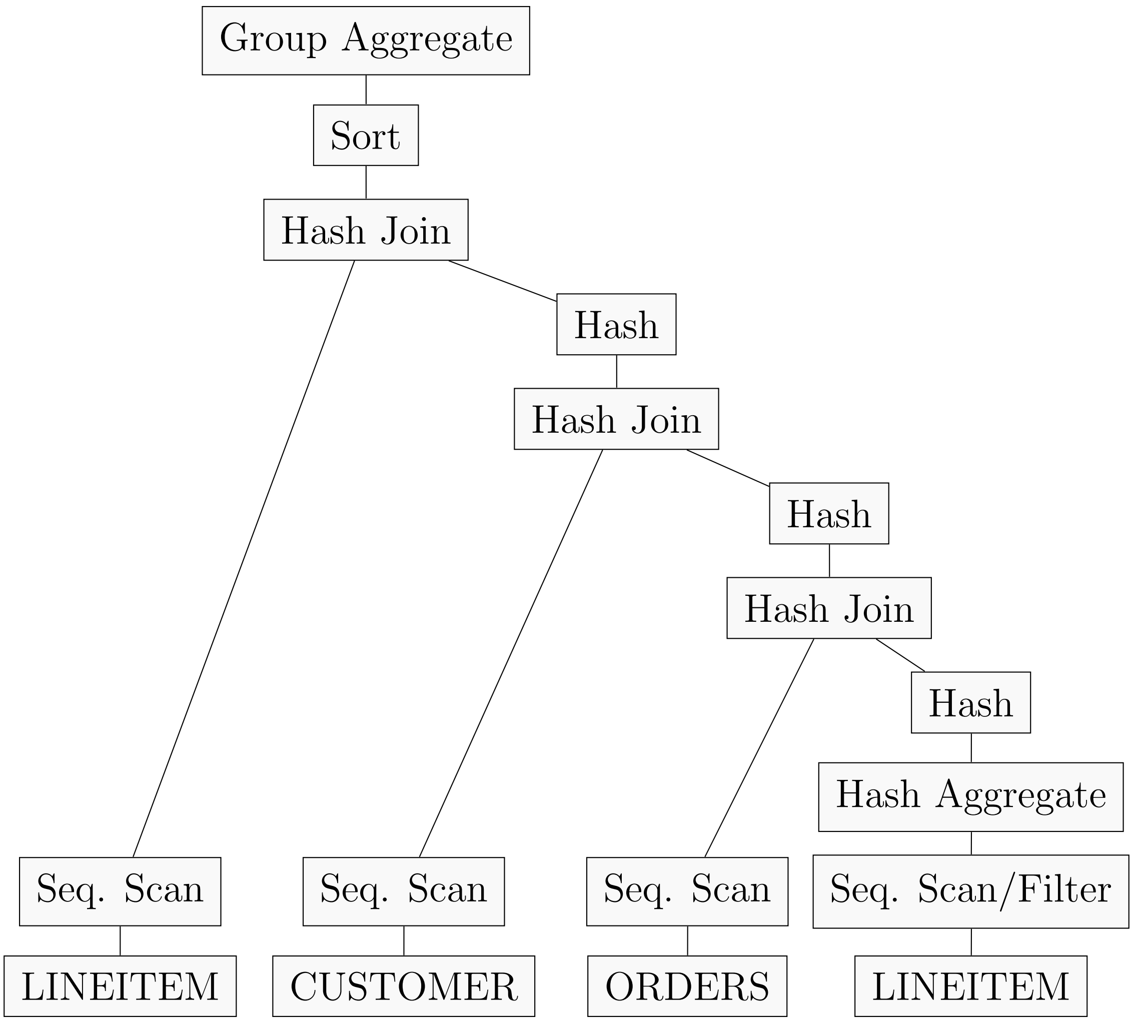


Figure 6.5: Q18 Plan

6.2 Greedy Algorithm

In order to identify a possible pattern in the nature of query plans with relatively low value of WC, we exhaustively enumerated the all plans for queries in the TPC-H benchmark and of examined their corresponding WC values. In all these experiments, we observed that plans having a lower WC had their join order identical to that of the LC-optimal plan. Using this observation for restricting the choice of plans to those having their join order matching the LC-optimal plan, we map the problem of finding the WC-optimal plan to the well known variation of the Knapsack Problem called the *Linear Multiple Choice Knapsack Problem (LMCKP)* [30].

Each item in LMCKP has a weight and a profit associated with it, and these items are divided into disjoint sets. The objective is to pick up one item from each set in such a manner that the total weight of the items is within the maximum weight allowed in the knapsack, while maximising the total profit obtained. Formally stated, the LMCKP problem is as follows:

Given multiple sets $N_1, N_2, ..., N_k$ of items, they are to be packed in a knapsack of capacity c. For each item $j \in N_i$, there is an associated profit p_{ij} and a weight w_{ij}. The objective is to choose a total of one item from each class so as to maximize the profit sum of the set of items, while having their weight sum to be within c. The Linear Multiple Choice Knapsack Problem (LMCKP) is thus formulated as:

$$\text{maximize } z = \sum_{i=1}^{k} \sum_{j \in N_i} p_{ij} x_{ij}$$

$$\text{subject to } \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \leq c,$$

$$\sum_{j \in N_i} x_{ij} = 1, i = 1, ..., k$$

$$0 \leq x_{ij} \leq 1,\ i = 1, ..., k, j \in N_i$$

All coefficients p_{ij}, w_{ij}, and c are positive integers, and the classes N_1,...N_k are mutually disjoint, class N_i having size n_i. x_{ij} represents the fraction of each item that is picked from the set. The total number of items is $n = \sum_{i=1}^{k} n_i$.

In our case, the sets N_i correspond to the *logical* operators (such as join, sort) in the plan tree of a query, with the items in a set mapping to the *physical* algorithms (such as hash join, merge join) for an operator. The weight w_{ij} of an item can be equated to the LC associated with an algorithm. The profit p_{ij} can similarly be mapped to a *Normalized Write Cost* (NWC), obtained by subtracting the WC value for each operator algorithm from a common large positive constant. The coefficient x_{ij} associated with each item $j \in N_i$ is similar to deciding whether a particular physical algorithm for executing an operator is utilized or not. Thus, given an LC limit $(1 + \lambda)C_o$ (corresponding to knapsack capacity c), LMCKP is equivalent to the problem of minimizing WC (by maximising NWC) while having the total sum of each operator's LC to be within the LC limit.

A critical point to note here is that there is no restriction on x_{ij} to have a binary 0 or 1 value. Therefore, one might think that all these variables can assume fractional values in the final solution. Since this corresponds to the weights of the algorithms at each operator node, it would seem to imply that the final plan could possibly have multiple algorithms executing a fraction of each operator. However, as proved in [30], the optimal solution to this problem can have *at the most two* of these variables taking a fractional value. Furthermore, both these variables will always belong to the *same* set N_f. For the rest of these variables, their value will either be 0 or 1. Since, according to the problem definition, the summation of these variables in each set should be exactly 1, the rest of the sets $N_i, i \neq f$ will have only one non-zero x_{ij}, the value of that variable being 1. In database terms this means that, in the WC-optimal plan, at the most one of the operators can have two algorithms performing a partial execution of the query. We return to this aspect in our final algorithm.

The LMCKP problem can be solved using a greedy algorithm [30] that provides an *optimal* solution in *polynomial* time complexity. We leverage the same greedy approach to solving the dual-objective optimization problem of query plan selection. Our algorithm uses two passes – the first pass closely resembling the conventional DP optimization process, while the second pass using the information gathered in the first to come up with the WC-optimal plan. Both these passes are described next.

First Pass

In the first pass, we keep track of the LC-optimal and an additional least WC plan (similar in the way of tracking interesting order plans) at each node, using the bottom-up DP algorithm. If they both are the *same*, we skip the second pass and output that plan as the solution. Otherwise, the LC-optimal cost C_o is used as an input to the second pass.

Second Pass

The second pass uses the C_o obtained in the first pass to derive the LC bound of $(1+\lambda)C_o$. Next, the NWC values are derived for each algorithm to map the query-optimization problem to LMCKP. Afterwards, it prunes the search space of operator algorithms by using the LP-domination principle. That is, for three algorithms r, s and t belonging to the same operator N_i, discard s if any of the following two conditions hold:

a) $LC_{ir} \leq LC_{is}$ and $NWC_{ir} \geq NWC_{is}$.

b) $\begin{vmatrix} LC_{is} - LC_{ir} & NWC_{is} - NWC_{ir} \\ LC_{it} - LC_{ir} & NWC_{it} - NWC_{ir} \end{vmatrix} \leq 0.$

Condition a) is equivalent to removing operator algorithm s if it is worse than r on both metrics – cycles and writes, such as P_4 dominating P_3 in Figure 6.6. Condition b), on the other hand, dictates that we discard an operator algorithm if a hybrid algorithm, that employs a weighted combination of two other algorithm, can give a better performance. For instance, in Figure 6.6, we can achieve any (LC, NWC) combination located on the line connecting P_1 and P_4 using a linear combination of P_1 and P_4 . Hence, the point P_2 can be discarded.

Overall, referring to the same figure, all the operator algorithms denoted by blue would be retained while those in red would be discarded. These new sets consisting of undominated operator algorithms for each operator in the plan tree are denoted by $R_i, i = 1, ..., k$. These are provided as an input to the greedy algorithm, the pseudo-code of which is outlined in Algorithm 4.

Algorithm 4 Greedy Algorithm for WC-Optimal Plan Selection

C_o is obtained from the first pass

R_i consists of LP-undominated operator algorithms λ is the slack value

```
 1: c = $(1+\lambda)C_o$
 2: Sort the operator algorithms in increasing order of LC for all $R_i, i = 1, ..., k$
 3: Choose the least LC algorithm from each $R_i$ (i.e. $x_{i1} = 1, x_{ij} = 0$) for $j = 2, ..., |N_i|, i = 1, ..., k$). Define $LC_{sum}$ and $NWC_{sum}$ as:
        $LC_{sum} = \sum_{i=1}^{k} LC_{i1}$ and
        $NWC_{sum} = \sum_{i=1}^{k} NWC_{i1}$, respectively.
 4: For all algorithms $j \neq 1$, we define the slope $\sigma_{ij}$ as
        $\sigma_{ij} = \frac{NWC_{ij} - NWC_{i,j-1}}{LC_{ij} - LC_{i,j-1}}$
 5: Sort the slopes $\sigma_{ij}$ in non-increasing order.
 6: while $LC_{sum} + LC_{ij} - LC_{i,j-1} \leq c$ do
 7:     Let $i, j$ be the indices corresponding to the next slope $\sigma_{ij}$ in $\{\sigma_{ij}\}$ .
 8:     $x_{ij} = 1, x_{i,j-1} = 0$
 9:     $LC_{sum} = LC_{sum} + LC_{ij} - LC_{i,j-1}$
10:     $NWC_{sum} = NWC_{sum} + NWC_{ij} - NWC_{i,j-1}$.
11: end while
12: if $LC_{sum} = c$ then
13:     return
14: else
15:     Let $\sigma_{ij}$ be the next slope in the list.
          /* The optimal solution has two fractional variables */
16:     $x_{ij} = \frac{c - LC_{sum}}{LC_{ij} - LC_{i,j-1}}$
17:     $x_{i,j-1} = 1 - x_{ij}$
18: end if
```

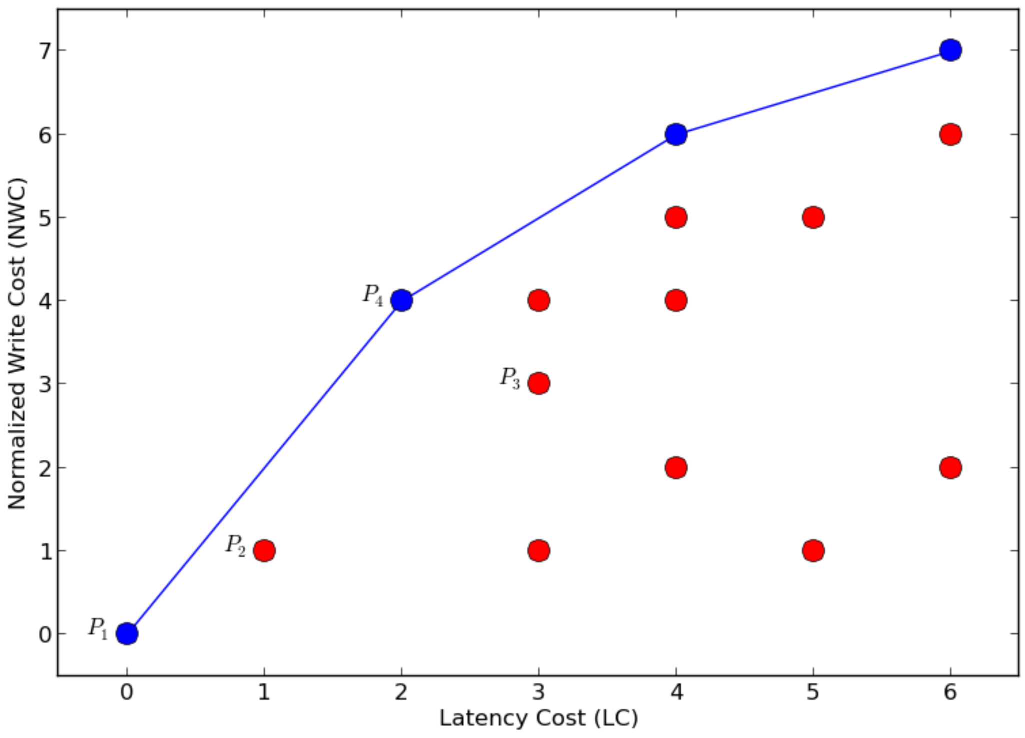


Figure 6.6: LP-undominated algorithms (blue)

As mentioned earlier, there exists a possibility that, for exactly one of the operators, the optimal solution contains a weighted combination of two algorithms. In that case, we perform a partial execution of that operator using the two algorithms; the extent of each of those executions being governed by their respective weights in the optimal solution. For instance, in case of join nodes, we can divide the inner relation into two portions, the ratio of which corresponds to the ratio of the weights obtained from the greedy algorithm. The outer relation is joined with one portion of the inner relation using the first algorithm, and with the other portion using the second algorithm.

6.3 Illustrative example comparing the algorithms

Let there be a query which has a single join order possible, with a plan tree structure corresponding to that shown in Figure 6.7.

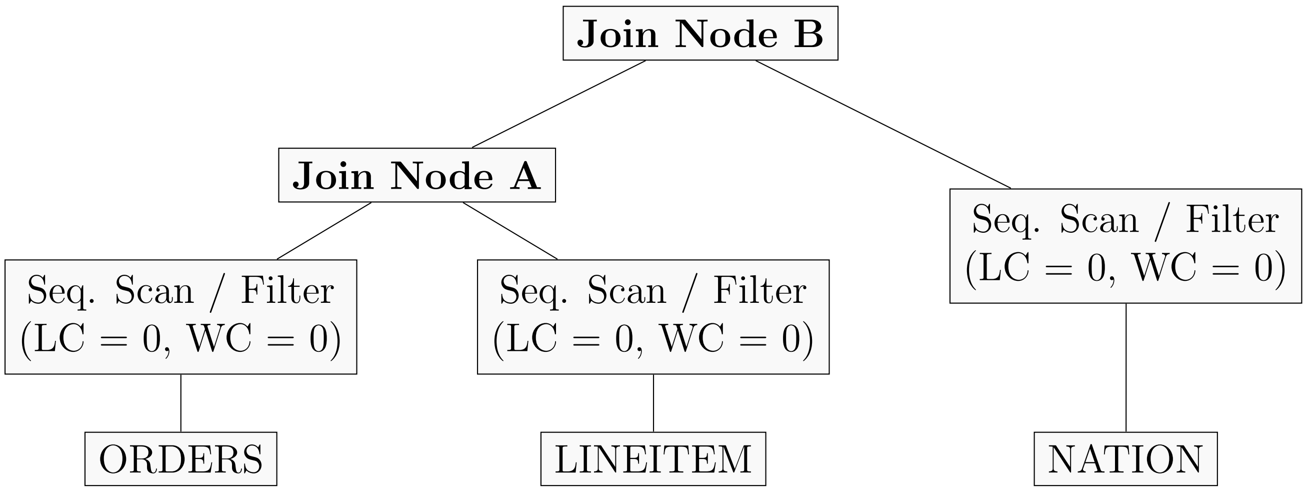

Join Node B
Join Node A
Seq. Scan / Filter (LC = 0, WC = 0)
Seq. Scan / Filter (LC = 0, WC = 0)
Seq. Scan / Filter (LC = 0, WC = 0)
ORDERS
LINEITEM
NATION

Figure 6.7: Query Plan Tree Structure

Let the algorithms available at join node A be the following:

Hash Join (A_{HJ}): LC = 100, WC = 300

Merge Join (A_{MJ}): LC = 150, WC = 200

Nested-loop Join (A_{NLJ}): LC = 200, WC = 150

Similarly, for a simpler analysis, let the algorithms available at join node B be restricted to the following:

Hash Join (B_{HJ}): LC = 100, WC = 200

Nested-loop Join (B_{NLJ}): LC = 250, WC = 100

Under these conditions, the search space of the current optimizer will consist of six (2×3) plans. On calculating the overall costs of each of these plans, we get the following numbers (the information in brackets indicating the combination of execution algorithms):

Plan P$_1$ (A_{HJ}, B_{HJ}): LC = 200, WC = 500

Plan P$_2$ (A_{MJ}, B_{HJ}): LC = 250, WC = 400

Plan P$_3$ (A_{NLJ}, B_{HJ}): LC = 300, WC = 350

Plan P$_4$ (A_{HJ}, B_{NLJ}): LC = 350, WC = 400

Plan P$_5$ (A_{MJ}, B_{NLJ}): LC = 400, WC = 300

Plan P$_6$ (A_{NLJ}, B_{NLJ}): LC = 450, WC = 250

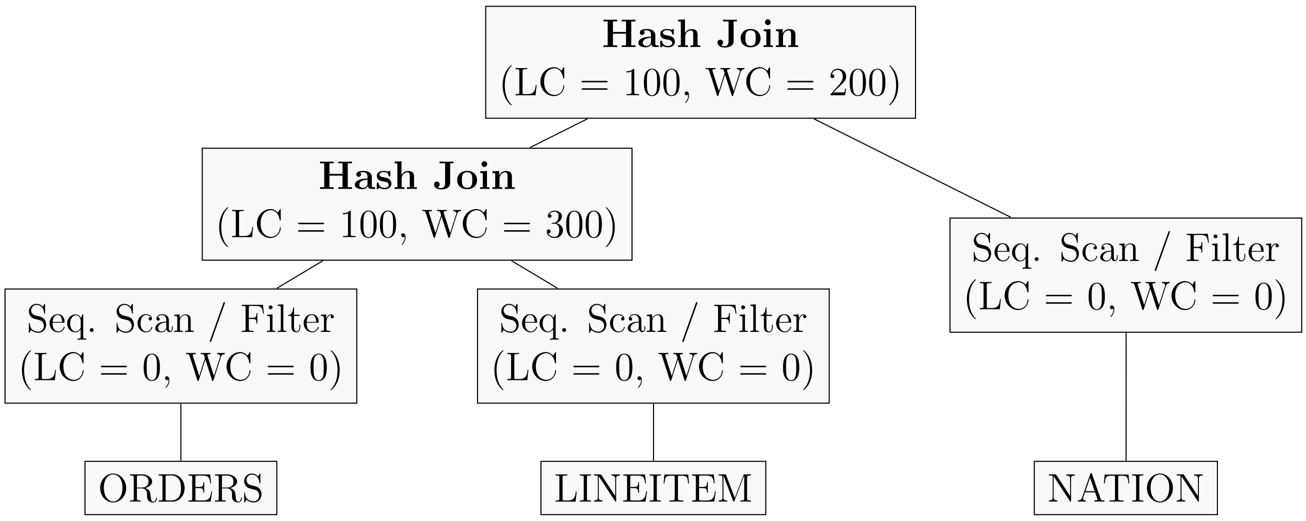


Figure 6.8: LC-optimal plan (P_1)

The LC-optimal plan for this query having the least LC value of 200 is shown in Figure 6.8. Setting $C_o = 200$ and assuming $\lambda = 1$, we are now interested in finding the minimum writes plan having LC bounded by $(1 + \lambda)C_o = 400$. In the next sections, we discuss the final plans returned by the different algorithms.

An enumeration plan selection algorithm, i.e. the one that sequentially tries out each algorithm for every operator node, will return P_5 (displayed in Figure 6.9) having WC value of 300 as the final plan. Note that although plan P_6 has the lowest WC value of 250, it doesn't qualify as a candidate plan by virtue of having LC of 450, which is greater than the bound of 400.

6.3.1 Heuristic-Propagation Algorithm

Let $\lambda_l = \lambda = 1$. The local LC-optimal sub-plan cost at join node A is $c_o = 100$. Although both A_{MJ} and A_{NLJ} qualify at this node for having LC within $(1 + \lambda_l)c_o = 200$, the local pruning at each node dictates that we propagate just A_{NLJ} since it has the minimum WC. Thus, plans P_2 and P_5 comprising of A_{MJ} are automatically pruned from the set of candidates.

Coming to the root node – join node B – we find that among the plans P_3, P_4 and P_6, plan

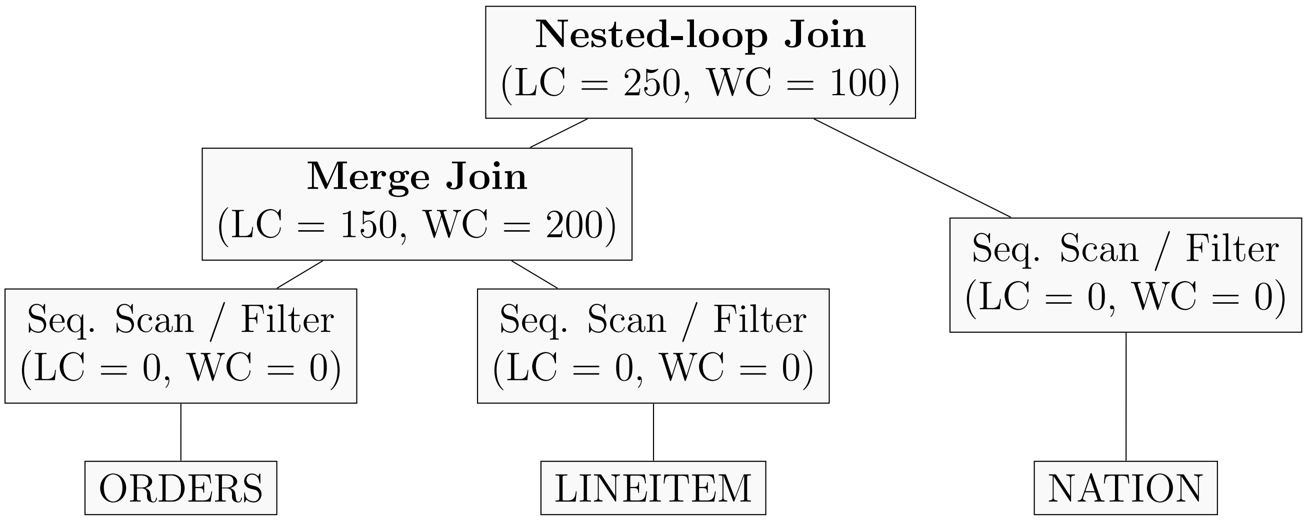


Figure 6.9: Enumeration Algorithm plan (P_5)

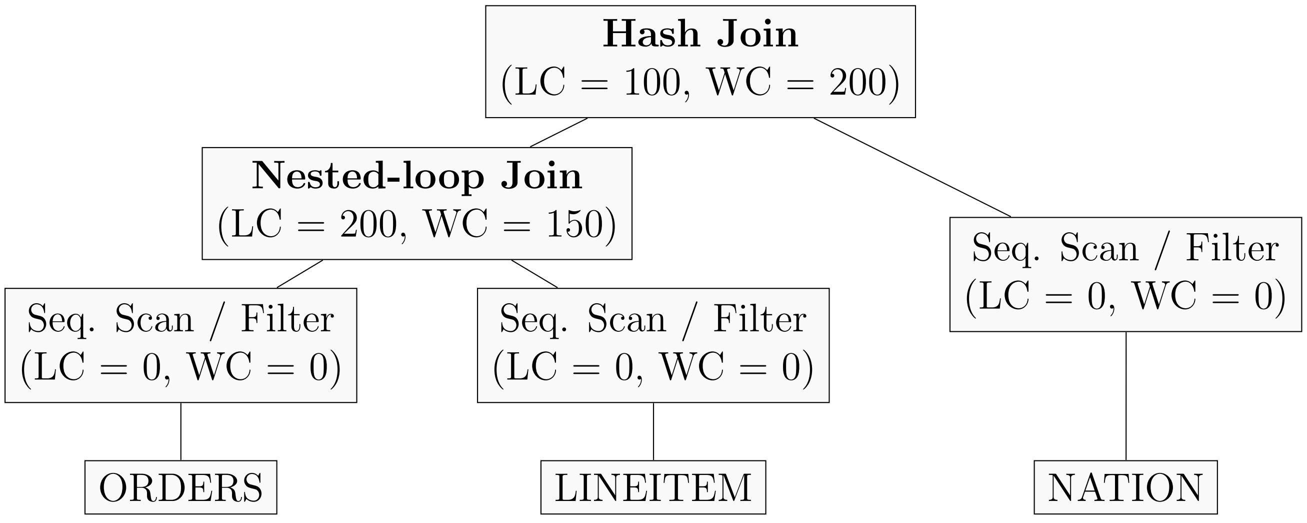


Figure 6.10: Heuristic-Propagation Algorithm plan (P_3)

P_3 having WC = 350 qualifies to be the least WC plan; plan P_6 being discarded for reasons described earlier. This plan is shown in Figure 6.10 .

6.3.2 Greedy Algorithm

In this case, we order the available operator algorithms at each plan node in increasing order of their LC. We take the value of 300 (corresponding to the highest WC among all operator algorithms) as the common large positive constant described in Section 6.2, and subtract the writes of all operator algorithms from it to obtain their corresponding NWC values. Thus the final pair of values for each of these algorithms are as follows:

Join Node A:

Hash Join (A_{HJ}): LC = 100, NWC = 0

Merge Join (A_{MJ}): LC = 150, NWC = 100

Nested-loop Join (A_{NLJ}): LC = 200, NWC = 150

Join node B:

Hash Join (B_{HJ}): LC = 100, NWC = 100

Nested-loop Join (B_{NLJ}): LC = 250, NWC = 200

For all operator algorithms corresponding to both A and B join nodes, none of the conditions of LP-domination (as discussed in Section 6.2) holds, as can be seen from Figure 6.11. The LP-undominated sets R_i are therefore the same as the original sets N_i. Consequently, the sequence of operator algorithms at each of these nodes ordered by their non-decreasing LC is:

Join Node A: A_{HJ}, A_{MJ}, A_{NLJ}

Join Node B: B_{HJ}, B_{NLJ}

On starting with choosing the least LC algorithm at both nodes, i.e. A_{HJ} at A and B_{HJ} at B, we get LC_{sum} and NWC_{sum} values as:

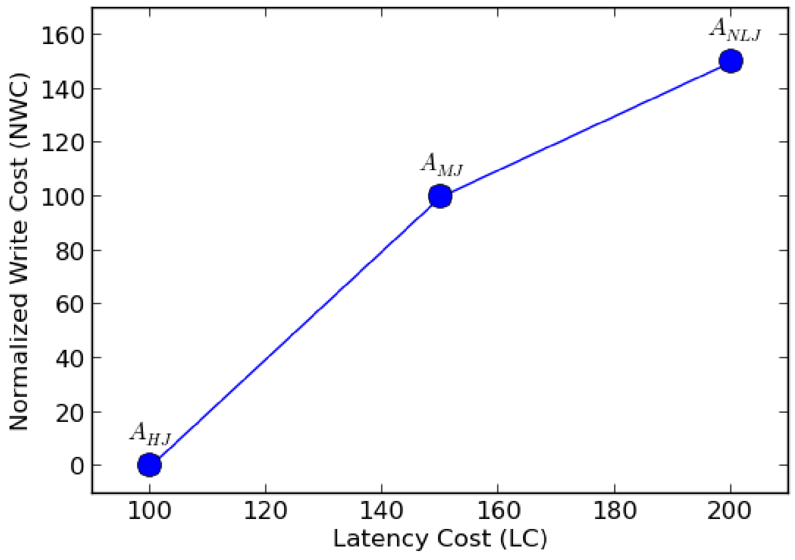
$$LC_{sum} = LC(A_{HJ}) + LC(B_{HJ}) = 100 + 100 = 200$$

$$NWC_{sum} = NWC(A_{HJ}) + NWC(B_{HJ}) = 0 + 100 = 100$$

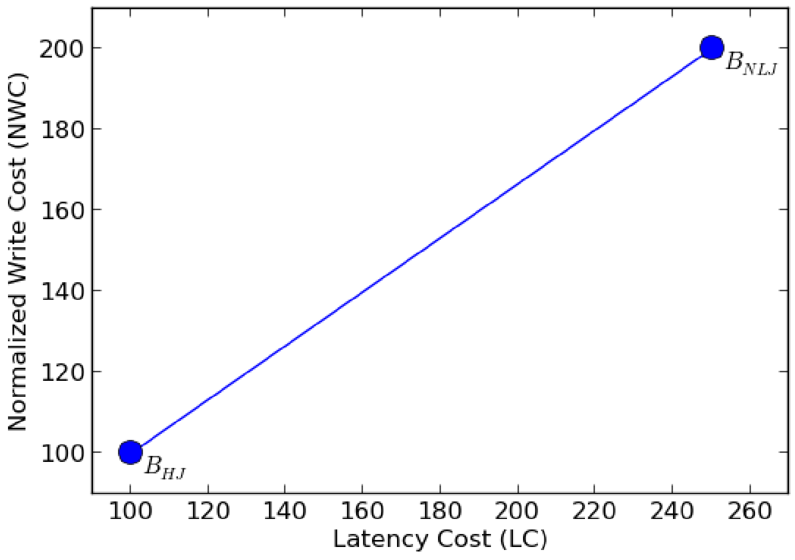
Further, the σ values for the operator algorithms available at both these nodes, calculated using step 4 of Algorithm 4, are:

$$\sigma_{12} = \frac{NWC(A_{MJ}) - NWC(A_{HJ})}{LC(A_{MJ}) - LC(A_{HJ})} = \frac{100 - 0}{150 - 100} = 2$$

$$\sigma_{13} = \frac{NWC(A_{NLJ}) - NWC(A_{MJ})}{LC(A_{NLJ}) - LC(A_{MJ})} = \frac{150 - 100}{200 - 150} = 1$$



(a) Join Node A



(b) Join Node B

Figure 6.11: Plot of algorithm costs

$$\sigma_{22} = \frac{NWC(B_{NLJ}) - NWC(B_{HJ})}{LC(B_{NLJ}) - LC(B_{HJ})} = \frac{200 - 100}{250 - 100} = 0.67$$

The non-increasing order of slopes is therefore $\{\sigma_{12}, \sigma_{13}, \sigma_{22}\}$.

Since the LC_{sum} is less than the allowable limit of 400, we replace the algorithm corresponding to the first σ value. Starting with σ_{12} and replacing the values of A_{HJ} with A_{MJ} in both LC_{sum} and NWC_{sum} we get:

$$LC_{sum} = LC_{sum} - LC(A_{HJ}) + LC(A_{MJ}) = 200 - 100 + 150 = 250$$

$$NWC_{sum} = NWC_{sum} - NWC(A_{HJ}) + NWC(A_{MJ}) = 100 - 0 + 100 = 200$$

The above step of operator algorithm replacement at the node that corresponds to the successive σ value in the sequence is repeated until LC_{sum} equals 400 or the next replacement exceeds it. We find that this condition happens at σ_{13} when the subsequent replacement of B_{HJ} with B_{NLJ} would have exceeded 400, as shown in the following:

Before replacement:

$$LC_{sum} = LC(A_{NLJ}) + LC(B_{HJ}) = 200 + 100 = 300$$

After replacement:

$$LC_{sum} = LC_{sum} - LC(B_{HJ}) + LC(B_{NLJ}) = 300 - 100 + 250 = 450$$

Thus, according to Algorithm 4, the optimal plan consists of a weighted combination of B_{HJ} and B_{NLJ} algorithms at join node B. The respective weights are:

$$w_{B_{NLJ}} = \frac{c - LC_{sum}}{LC(B_{HJ}) - LC(B_{NLJ})} = \frac{400 - 300}{250 - 100} = 2/3$$

$$w_{B_{HJ}} = 1 - w_{B_{NLJ}} = 1/3$$

Hence, the overall optimal plan has the following values of the costs:

$$\text{LC} = LC(A_{NLJ}) + \frac{1}{3} \times LC(B_{HJ}) + \frac{2}{3} \times LC(B_{NLJ}) = 200 + \frac{100 + 2 \times 250}{3} = 400$$

$$\text{WC} = WC(A_{NLJ}) + \frac{1}{3} \times WC(B_{HJ}) + \frac{2}{3} \times WC(B_{NLJ}) = 150 + \frac{200 + 2 \times 100}{3} = 283.3$$

Clearly, the WC of this plan (displayed in Figure 6.12) is the best among all the plans returned by the plan selection algorithms discussed so far. Therefore, the greedy algorithm has the potential to find out a plan – by exploring all possible weighted combinations of execution algorithms at each operator node – that can be far superior to the plans in the limited search space of either the enumeration algorithm or the heuristic-propagation algorithm. Moreover, this exploration is conducted in an efficient manner, finishing in a time that is polynomial in the total number of available operator algorithms.

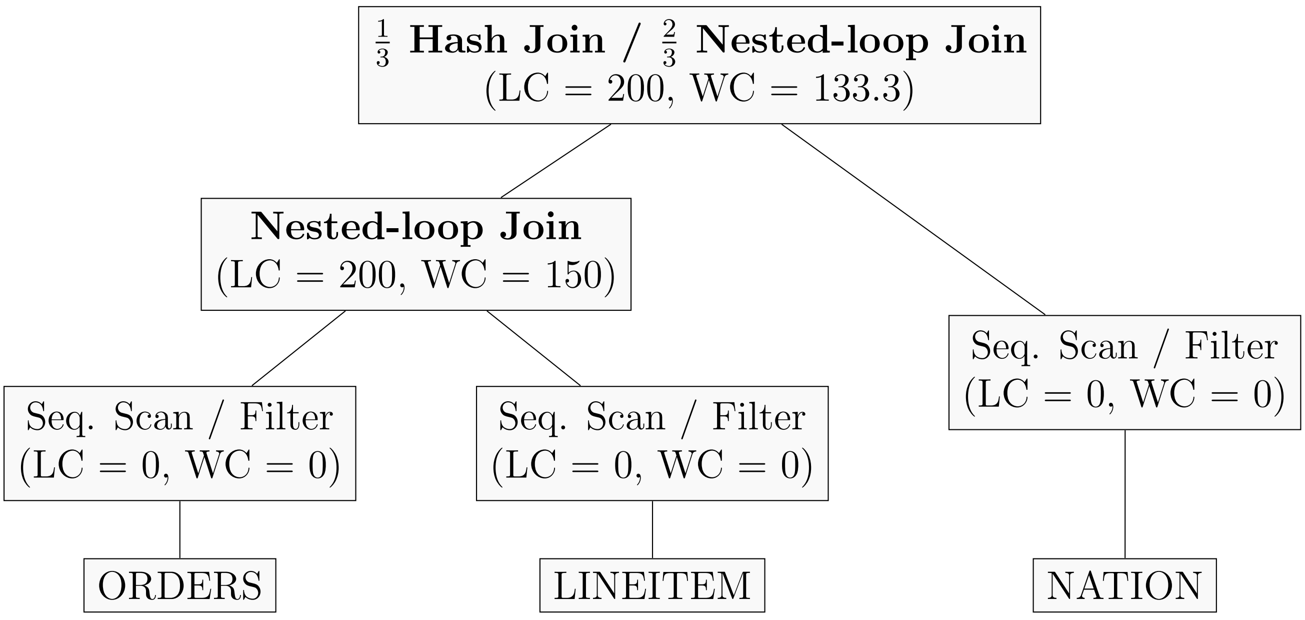


Figure 6.12: Greedy Algorithm Plan

Notwithstanding its advantages, the greedy algorithm comes with some major limitations that make its adoption at the current state infeasible.

First, the algorithm works with the assumption that the join order of the WC-optimal plan is same as that of the LC-optimal plan, which may not always hold true. Handling multiple join orders would require us to run a copy of the greedy algorithm for each possible join order. However, this can be expensive for queries containing a large number of join relations, due to an exponential number of such orders.

Second, it assumes that the costs associated with each operator algorithm is fixed and is independent of choices at other nodes. This assumption doesn't hold in case of interesting orders, for in such cases, choosing an operator algorithm that outputs the tuples in some interesting order at one particular node might serve to reduce the cost of another operator algorithm at a later node.

Third, the implementation of the greedy plan selection algorithm requires a considerable redesign to a database optimizer which uses a DP-based approach for selecting a plan, such as PostgreSQL. Additionally, changes are also required to the executor to support weighted execution of multiple operator algorithms for a given operator node in a query plan tree.

In essence, the greedy approach serves to provide a framework that lays the foundation for building future PCM and other similar NVM-based multi-objective query optimizers.

Chapter 7

Conclusion

Designing database query execution algorithms for PCM platforms requires a change in perspective from the traditional assumptions of symmetric read and write overheads. It now opens up a new area of execution algorithm design wherein the algorithm needs to be cognizant about, not just the number of memory operations, but the kinds of operations as well. Prior techniques such as cache-conscious processing have the potential to find renewed applications in this area. At the same time, custom asymmetric I/O design decisions – such as trading writes for reads – can go a long way in determining not just the performance profiles of database engines, but also the lifetime of the underlying PCM hardware.

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 these algorithms, our aim was to facilitate a quick an easy transition of database engines to a PCM-based hardware. Further, each of these algorithms was accompanied by the corresponding write estimator that was shown to predict writes with a reasonable degree of accuracy.

Using a state-of-the-art simulator after addition of PCM support, we performed detailed experimentation on *complete* TPC-H benchmark queries; something that has not been undertaken in any prior literature in this area. 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.

Furthermore, we introduced a novel metric of write cost that plays a critical role in the choice of algorithms on PCM hardware. We presented two algorithms to integrate this metric in the query optimizer – one heuristic and another based on greedy solution to the linear multiple choice knapsack problem. The heuristic technique is easy to implement but suffers from a high probability of providing grossly sub-optimal query plans. The greedy algorithm based approach, on the other hand, provides an optimal plan but only within the search space of the join order corresponding the latency cost optimal plan. Moreover, it is also incapable of handling interesting orders, besides warranting considerable changes to the database engine.

In this manner, we incorporated 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.

For our future work, we would like to accomplish the following:

1. Improve our write estimators to additionally predict the *wear distribution* of algorithms. The current estimators are restricted to providing a count of the total number of writes for each operator execution algorithm. However, it might be preferable to choose an algorithm that incurs a higher number of writes in a uniformly distributed manner, over the one that leads to a lower number of writes but with a skewness in their distribution. The improved estimators would thus enable the optimizers to make these decisions in an informed manner.

2. Extend the greedy plan selection algorithm presented in Section 6.2 to cover all possible plans in the search space. This requires handling plans for two scenarios – diverse join orders as well as interesting order. Using this, we hope to provide provable performance guarantees for the WC-optimal plan returned by the optimizer for any type of query.

3. Implement this plan selection algorithm in the PostgreSQL optimizer and experimentally

compare its performance with the native version, in order to obtain actual improvement numbers.

4. Design improved PCM-conscious query execution algorithms suitable for multi-query workloads. Parallel queries, especially with large memory footprints, can exacerbate the problem of limited DRAM availability, by mutual eviction of intermediate data. Such a scenario offers new challenges for ensuring minimal writes despite these evictions.

5. Explore the possibility of dividing the write traffic to frequently written data structures to separate memory locations. This can be particularly useful in cases such as aggregate functions which keep updating a limited set of aggregate variables. In such cases, these aggregate variables can be divided into multiple *sub-aggregates*, which can be eventually combined to provide the overall result.

6. Investigate other areas of PCM-conscious database design, such as ensuring ACID semantics during query execution on PCM, and leveraging the non-volatility of PCM to reduce logging requirements.

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

Bibliography

[1] http://www.postgresql.org. 45

[2] http://www.tpc.org/tpch. 45, 63

[3] Phase Change Memory. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009. xiii, 5

[4] Samsung ships industry's first multi-chip package with a PRAM chip for handsets. http://www.samsung.com/us/aboutsamsung/news/newsIrRead.do?news_ctgry=irnewsrelease&news_seq=18828, 2010. 2

[5] Micron announces availability of Phase Change Memory for mobile devices. http://investors.micron.com/releasedetail.cfm?ReleaseID=692563, 2012. 1

[6] IBM demonstrates next-gen phase-change memory that's up to 275 times faster than your SSD. http://www.extremetech.com/extreme/182096-ibm-demonstrates-next-gen-phase-change-memory, 2014. 1

[7] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. Making cost-based query optimization asymmetry-aware. In *Proceedings of the 8th International Workshop on Data Management on New Hardware (DaMon)*, 2012. 17, 18

[8] Geoffrey W. Burr, Bülent N. Kurdi, J. Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, 2008. 5

[9] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, 2015. 13

[10] S Chaudhuri and V Narasayya. Program for TPC-D data generation with skew. Technical report, `ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew`, 2012. 49

[11] Shimin Chen and Qin Jin. Persistent B$^+$-trees in non-volatile main memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, 2015. 15

[12] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011. xi, xiii, 2, 3, 4, 5, 6, 14, 15

[13] Ping Chi, Wang-Chien Lee, and Yuan Xie. Making B$^+$-tree efficient in pcm-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISPLED)*, 2014. 15

[14] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO)*, 2009. 5, 10

[15] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. 13

[16] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009. 12

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, third edition, 2009. 26

[18] Jaeyoung Do and Jignesh M Patel. Join processing for flash SSDs: remembering past lessons. In *Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMon)*, 2009. 17

[19] AP Ferreira, Miao Zhou, S Bock, B Childers, R Melhem, and D Mosse. Increasing PCM main memory lifetime. In *Proceedings of the 13th Conference on Design, Automation and Test in Europe (DATE)*, 2010. 7, 10, 40, 41

[20] Jingtong Hu, Chun Jason Xue, Qingfeng Zhuge, Wei-Che Tseng, and Edwin H-M Sha. Write activity reduction on non-volatile main memories for embedded chip multiprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3), 2013. 11

[21] Weiwei Hu, Guoliang Li, Jiacai Ni, Dalie Sun, and Kian-Lee Tan. B^p-Tree: A predictive B^+-Tree for reducing writes on phase change memory. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(10), 2014. 15

[22] Yiming Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18(6), 2008. 5

[23] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. NVRAM-aware logging in transaction systems. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014. 14

[24] Per-Ake Larson. Grouping and duplicate elimination: Benefits of early aggregation. *Microsoft Technical Report*, 1997. 30

[25] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009. xiii, 1, 5, 9

[26] Daniel Myers. On the use of NAND flash memory in high-performance relational databases. Master's thesis, 2007. 17

[27] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012. 14

[28] Karl-Dietrich Neubert. The flashsort1 algorithm. `http://www.drdobbs.com/database/the-flashsort1-algorithm/184410496`, 1998. 21

[29] JH Oh, Jae Hyo Park, YS Lim, HS Lim, YT Oh, Jong Soo Kim, JM Shin, Young Jun Song, KC Ryoo, DW Lim, et al. Full integration of highly manufacturable 512mb pram based on 90nm technology. In *Proceedings of the 2006 International Electron Devices Meeting (IEDM)*, 2006. 4

[30] David Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2), 1995. 65, 66

[31] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO)*, 2009. 9, 10, 44

[32] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009. xi, xiii, 1, 4, 5, 6, 10

[33] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, 2011. 12

[34] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Yi-Chou Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, Hsiang-Lan Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5), 2008. 4

[35] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191), 2008. 5

[36] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A Shah, Janet L Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 28th ACM SIGMOD International Conference on Management of Data*, pages 59–72, 2009. 2, 17, 18

[37] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012. 2, 37

[38] Meduri Venkata Vamsikrishna, Zhan Su, and Kian-Lee Tan. A Write Efficient PCM-Aware Sort. In *Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA)*, 2012. 16

[39] Stratis D Viglas. Adapting the B$^+$-tree for asymmetric I/O. In *Proceedings of the 16th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, 2012. 15

[40] Stratis D Viglas. Write-limited sorts and joins for persistent memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014. 2, 5, 16

[41] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011. 13

[42] Sebastian Wild and Markus E Nebel. Average case analysis of java 7's dual pivot quicksort. In *Proceedings of the 20th European Symposium on Algorithms (ESA)*, 2012. 21

[43] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007. 5, 7, 9, 40, 42

[44] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009. 11

[45] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, 2013. 12

[46] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009. 1, 9