How to Build a Non-Volatile Memory Database Management System

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

The difference in the performance characteristics of volatile (DRAM) and non-volatile storage devices (HDD/SSDs) influences the design of database management systems (DBMSs). The key assumption has always been that the latter is much slower than the former. This affects all aspects of a DBMS's runtime architecture. But the arrival of new non-volatile memory (NVM) storage that is almost as fast as DRAM with fine-grained read/writes invalidates these previous design choices.

In this tutorial, we provide an outline on how to build a new DBMS given the changes to hardware landscape due to NVM. We survey recent developments in this area, and discuss the lessons learned from prior research on designing NVM database systems. We highlight a set of open research problems, and present ideas for solving some of them.

1. INTRODUCTION

DBMSs have always dealt with the trade-offs between volatile and non-volatile storage devices. In order to retain modifications after a loss of power, the DBMS must write that data to a non-volatile device, such as a SSD or HDD. Such devices only support slow, bulk data transfers as blocks. Contrast this with volatile DRAM, where a DBMS can quickly read and write a single byte from these devices, but all data is lost once power is lost.

Non-volatile memory (NVM)[1] offers an intriguing blend of these two storage mediums. NVM is a broad class of technologies, including phase-change memory [67], memristors [73], and STT-MRAM [36], that provide low latency reads and writes on the same order of magnitude as DRAM, but with persistent writes and large storage capacity like a SSD [24].

Researchers have been discussing the possibility of building a DBMS for NVM for decades [31]. Although battery-backed DRAM has existed for some time, it has physical form, cost, and availability limitations that prevent it from being widely adopted. And until recently, it looked as if NVM would suffer the same fate. We contend, however, that there are three recent developments that make it look like we are finally at the point where NVM will become available. First is that the industry has agreed to standard definitions of NVM technologies and form factors [8, 10]. The second change is that both Linux [3, 1] and Microsoft [52, 48] have announced support for NVM in their respective kernels. Lastly, Intel has announced that there are new instructions coming in their next ISA update that are specifically for flushing data from CPU caches to NVM [69, 9].

All of these portend the soon arrival of NVM. But it is unclear at this point how to best leverage these new technologies in a DBMS. There are several aspects of NVM that make existing DBMS architectures inappropriate for them [33]. For example, disk-oriented DBMSs (e.g., Oracle RDBMS, IBM DB2, MySQL) are predicated on using block-oriented devices for durable storage that are slow at random access. As such, they maintain an in-memory cache for blocks of tuples and try to maximize the amount of sequential reads and writes to storage. In the case of memory-oriented DBMSs (e.g., VoltDB, MemSQL), they contain components for overcoming the volatility of DRAM. Such components may be unnecessary in a system with byte-addressable NVM with fast random access.

Tutorial Outline and Goals: This tutorial is a comprehensive discussion of how to design and build a DBMS for NVM. Our focus is on systems that contain a hybrid architecture comprised of DRAM, NVM, and HDD/SSDs (as opposed to one with only NVM [14]), as such platforms will likely be the most common in the near-term. The goal of our tutorial is to provide an overview of the major design decisions in a DBMS implementation that are affected by NVM. We will discuss issues related to both on-line transaction processing (OLTP) and on-line analytical processing (OLAP) systems, as well as hybrid (HTAP) DBMSs that seek to support both workloads in a single platform.

We will first present an overview of NVM technologies (Section 2), highlighting their differences both with each other and compared to DRAM/SSDs. We will then discuss the three areas of a DBMS architecture that are affected the most by NVM. First, we discuss the core components, such as memory management, that are shared throughout the DBMS (Section 3). We will then discuss how the DBMS builds off of these components to support storage and recovery mechanisms (Section 4). Then we describe how to build on top of this storage layer to perform query optimization and execution for NVM-resident databases (Section 5). Lastly, we conclude with a summary of the lessons that we have learned in building a DBMS for NVM.

This tutorial differs from previous presentations on NVM [77] because we go beyond storage management and talk about the entire internal DBMS stack. We will couch all of the above topics in the context of the **Peloton** [4, 63] DBMS. Peloton is an open-source HTAP DBMS that we are building that is designed from the ground-up to use NVM. Our intended audience are developers, researchers, and practitioners with knowledge of DBMS internals. They do not need any in-depth background or experience with NVM.

[1] NVM is also referred to as *storage-class memory* or *persistent memory*.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05. . . $15.00
DOI: `http://dx.doi.org/10.1145/3035918.3054780`

	DRAM	PCM	RRAM	MRAM	SSD	HDD
Read latency	60 ns	50 ns	100 ns	20 ns	25 μs	10 ms
Write latency	60 ns	150 ns	100 ns	20 ns	300 μs	10 ms
Addressability	Byte	Byte	Byte	Byte	Block	Block
Persistent	No	Yes	Yes	Yes	Yes	Yes
Endurance	$>10^{16}$	10^{10}	10^{8}	10^{15}	10^{5}	$>10^{16}$

Table 1: Comparison of emerging NVM technologies with other storage technologies [26, 37, 66, 59]: phase-change memory (PCM) [67], memristors (RRAM) [73], and STT-MRAM (MRAM) [36].

2. BACKGROUND

There are essentially two types of DBMS architectures: disk-oriented and memory-oriented systems [33]. The former is exemplified by the first DBMSs from the 1970s, such as IBM's System R [16], where the system is predicated on the management of blocks of tuples on disk using an in-memory cache; the latter by IBM's IMS/VS Fast Path [43], where the system performs updates on in-memory data and relies on the disk to ensure durability. The need to ensure that all changes are durable has dominated the design of systems with both types of architectures. This has involved optimizing the layout of data for each storage layer depending on how fast it can perform random accesses. Further, the system needs to propagate updates that the transactions make on tuples stored in memory to an on-disk representation for durability.

NVM technologies, like phase-change memory [67] and memristors [73], remove these tuple transformation and propagation costs through byte-addressable loads and stores with low latency. This means that they can be used for efficient architectures that are used in memory-oriented DBMSs. But unlike DRAM, all writes to the NVM are potentially durable and therefore a DBMS can access the tuples directly in the NVM after a restart or crash without needing to reload the database first. As shown in Table 1, NVM differs from other storage technologies in the following ways:

- **Byte-Addressability:** NVM supports byte-addressable loads and stores unlike other non-volatile devices that only support slow, bulk data transfers as blocks.
- **High Write Throughput:** NVM delivers two orders of magnitude higher write throughput compared to SSD and HDD. More importantly, the gap between sequential and random write throughput of NVM is much smaller.
- **Read-Write Asymmetry:** In certain NVM technologies, writes take longer to complete compared to reads. Further, excessive writes to a single memory cell can destroy it.

Although the advantages of NVM are obvious, making full use of them in an DBMS is non-trivial. Previous comparisons of disk-oriented and memory-oriented DBMSs with NVM show that the two architectures achieve almost the same performance when using NVM because of the overhead of logging [33]. This is because current DBMSs assume that memory is volatile, and thus their architectures are predicated on making redundant copies of changes on durable storage. Thus, it is important to reexamine the design of different components of the database system to leverage the unique properties of NVM.

We now describe the three layers of a DBMS architecture that are affected by NVM. Figure 1 provides an overview of these layers. Table 2 presents a summary of prior research on the impact of NVM on different components of a DBMS. We first describe the interfaces for accessing NVM that are used across the entire DBMS. We then discuss how a DBMS manages data on a storage-hierarchy containing NVM, followed by how it executes queries and transactions on a NVM-resident database.

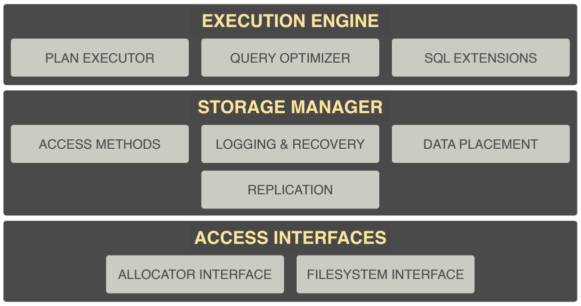

Figure 1: NVM-Aware Components – The main components of a DBMS that are affected by NVM.

RESEARCH AREA	COMPONENTS
Access Interfaces	**Allocator Interface** [70, 79, 59, 29, 78, 62, 14] **Filesystem Interface** [37, 5, 21, 30, 48, 1]
Storage Manager	**Access Methods** [61, 81, 55, 25, 28, 75] **Logging and Recovery** [15, 50, 14, 12, 64, 40, 62, 80] **Data Placement** [49, 51, 57] **Replication** [68, 82, 10, 15]
Execution Engine	**Plan Executor** [27, 76, 42, 20, 22] **Query Optimizer** [19, 65] **SQL Extensions** [11, 7]

Table 2: Classification of NVM research – A summary of prior research on the impact of NVM on different components of a DBMS.

3. ACCESS INTERFACES

In this section, we describe the two interfaces that a DBMS uses for accessing NVM. The first interface is through byte-level memory allocation and the second is through a filesystem API. This discussion is in the context of how the latest versions of the Linux and Windows OSs support NVM [3, 1, 52, 48].

3.1 Allocator Interface

An NVM-aware memory allocator allows the DBMS to allocate chunks of memory through a persistent memory programming library [6]. Such an allocator differs from a volatile memory allocator in three ways [70, 79, 17, 29, 78].

The first difference is that it provides a *durability* mechanism to ensure that modifications to data stored on NVM are persisted [59]. This is necessary because the changes made by a transaction to a location on NVM may still reside in volatile processor caches when the transaction commits. If a power failure happens before the changes reach NVM, then these changes are lost. The allocator exposes a special API call to provide this durability. Internally, the allocator first writes out the cache lines containing the data from any level of the cache hierarchy to NVM using `CLWB`, the optimized cache flushing instruction that is part of the newly proposed NVM-related instruction set extensions [9][2]. Then, it issues a `SFENCE`

[2] The `CLWB` instruction writes back a cache line to NVM similar to the `CLFLUSH` instruction. But it differs in two ways: (1) it is weakly-ordered and thus perform better than the strongly-ordered `CLFLUSH`, and (2) it retains a copy of the line in the cache hierarchy in exclusive state, thereby reducing the possibility of cache misses during subsequent accesses. In contrast, the `CLFLUSH` instruction always invalidates the cache line, which means that the DBMS has to retrieve the data again from NVM.

instruction to ensure that the stores are ordered ahead of subsequent instructions. At this point, the stores may reside in the memory controller's write-pending queue (WPQ). In case of a power failure or shutdown, the *asynchronous DRAM refresh* (ADR) instructions of newer NVM platforms automatically flushes the WPQ [69], thus ensuring that the data is durable.

The second variation is that the allocator provides a *naming* mechanism for allocations so that pointers to memory locations are valid even after the system restarts [62, 17]. The allocator ensures that the virtual memory addresses assigned to a memory-mapped region never change. With this mechanism, a pointer to a NVM location is mapped to the same virtual location after the OS or DBMS restarts. We refer to these pointers as *non-volatile pointers*. They provide the foundation for building crash-consistent data structures [14].

Lastly, the allocator ensures the *atomicity* of all the memory allocations so that after a system failure all memory regions are either allocated or available for use [70]. Allocator guarantees that there are no torn data writes, dangling references, and persistent memory leaks by decoupling memory allocations into two steps: (1) *reserve* and (2) *activate*. After the *reserve* step, the DBMS can use the reserved memory region for storing ephemeral data. In case of a failure, however, the allocator reclaims this memory region. To ensure that it owns a memory region even after a failure, the DBMS must request the allocator to separately *activate* the memory region by updating the meta-data associated with that region. Under this two-step process, the DBMS first initializes the contents of a memory region after the *reserve* step but before it activates it.

3.2 Filesystem Interface

A DBMS can also access NVM through a special filesystem that is optimized for non-volatile memory [37, 5, 21, 30]. This interface allows it to use the POSIX filesystem interface to read/write data to files (e.g., logs and auxiliary files). The filesystem relies on write-ahead logging to preserve meta-data consistency and uses shadow paging only for data. Normally, in a block-oriented filesystem (e.g., EXT4) that is designed for disks, file I/O requires two copies; one involving the block device and another involving the user buffer. The efficiency of the I/O stack within the OS is not critical when it is hidden by the disk latency. However, NVM is byte-addressable and supports I/O in the sub-microsecond range.

To cope with these shorter NVM latencies, Microsoft and Linux are adding support for direct access storage (DAX) in Windows Server 2016 [48] and Linux 4.7 [1], respectively. With DAX, a DBMS directly allocates and uses NVM without an intervening filesystem. This requires only one copy between the file and the user buffers, thus improving the file I/O performance by an order of magnitude compared to block-oriented filesystems.

4. STORAGE MANAGER

With the access interfaces that we described above, the next step is to implement the DBMS's storage manager.

4.1 Access Methods

Given the read-write asymmetry in NVM, it is important to redesign the persistent data structures that are used as access methods in a DBMS so that they perform fewer writes to NVM [61, 81, 55, 25, 28]. In a persistent NVM-aware B+tree index, the foremost change is to keep the entries in the leaf node unsorted so that the tree performs fewer writes and cache line flushes when it is mutated [54, 74]. Unsorted key-value entries in the leaf node require an expensive linear scan. This operation is sped up by hashing the keys, and using the hashes as a filter to avoid comparing the keys [61].

Another design optimization is to selectively enforce persistency where the tree only persists its leaf nodes and reconstructs its inner nodes during recovery [81]. In a storage hierarchy containing DRAM and NVM, the tree can persist the leaf nodes on NVM and maintain the inner nodes on DRAM. During recovery from a system failure, the tree rebuilds all the inner nodes that it placed in DRAM. Although this approach increases the recovery latency of the tree, the associated improvements in search and update operations during regular processing justify it [61].

A DBMS can also leverage the asymmetric I/O property by temporarily relaxing the balance of the B+tree [75]. Such imbalance (potentially) causes extra reads to access the tree but reduces the number of writes. This is a good trade-off for NVM because reads are less expensive than the writes and reduces wear-down of the storage device. By periodically re-balancing the tree and reducing the number of writes, an NVM-aware B+tree outperforms the regular B+tree across different workloads. We note that other data structures used as access methods in a DBMS, such as hash tables, must also be redesigned for NVM.

4.2 Logging & Recovery

The write-ahead logging (WAL) protocol supports efficient transaction processing when memory is volatile and durable storage cannot support fast random writes [58, 39, 33]. But this assumption causes unnecessary performance degradations in a DBMS with NVM storage [14]. Consider a transaction that inserts a tuple into a table. A DBMS first records the tuple's contents in the log, and it later propagates the change to the database. With NVM, a DBMS can employ a logging protocol that avoids this unnecessary data duplication. The reason why NVM enables a better logging protocol than WAL is two-fold. The write throughput of NVM is more than an order of magnitude higher than that of an SSD or HDD. Further, the gap between sequential and random write throughput of NVM is smaller than that in SSD and HDD. Hence, a DBMS can flush changes directly to the database in NVM during regular transaction processing [15, 14, 12, 64, 40, 62, 80].

For example, when a transaction inserts a tuple into a table, the DBMS records the tuple's contents in the database even before it writes any associated meta-data in the log. Thus, the log is always slightly behind the contents of the database, and this technique is referred to as write-behind logging (WBL) [15]. The DBMS can still restore the database to the correct state after a restart by keeping track of the transactions active in the current group commit interval.

The WBL recovery algorithm comprises of only an analysis phase. During the analysis phase, the DBMS scans the log backward till the most recent checkpoint log record to determine the transactions active in the failed group commit interval. There is no need for a redo phase because all the modifications of committed transactions are already present in the database. WBL also does not require an WAL-style undo phase. Instead, the DBMS only tracks the transactions active in the current group commit interval as determined by the analysis phase, so that it can ignore the effects of the associated uncommitted transactions [15, 44, 46]. In case of a transaction failure, the transaction manager rolls back any dirty changes flushed to NVM using the meta-data that it records in the dirty tuple table.

We expect that the first NVM products will initially be more expensive than current technologies, and thus using less storage means a lower procurement cost. As WBL only records minimal meta-data in the log, it shrinks the storage footprint of the DBMS on NVM, thereby improving its storage utilization in comparison to the WAL protocol.

Another design choice is for the DBMS to use NVM only for storing the log and manage the database still on disk [50]. This is a more cost-effective solution, as the cost of NVM devices are expected to be higher than that of disk. But this approach only leverages the low-latency sequential writes of NVM, and does not exploit its ability to efficiently support random writes and fine-grained data access.

4.3 Data Placement

In a two-tier storage hierarchy with DRAM and NVM, the DBMS places objects that incur frequent row buffer misses on DRAM, and stores those that do not on NVM [51, 57]. We note that nearly all the loads and stores are directed to less than 1% of the objects [49]. These include the hot tuples in the database, hot tuples in intermediate results, and global variables. The OS policies for page migration, unfortunately, do not work well for database systems [72]. This is because the OS keeps track of page access statistics at the granularity of virtual memory pages. Migrating such pages may lead to sub-optimal data placement decisions when DBMS objects with different access patterns are placed on the same page [49]. Thus, it is always better if the DBMS manages data placement and migration.

In a three-tier storage hierarchy with DRAM, NVM, and SSDs, the DBMS ensures that the "hot" data resides on DRAM. It then migrates the "cold" data to the NVM and SSD layers over time as the data ages and is likely to be updated. The latency of a transaction that accesses a cold tuple will be higher in a three-tier storage hierarchy. This is because NVM supports faster reads than SSD. During update operations, however, the DBMS quickly writes to the log and database on NVM. Eventually, the DBMS migrates the cold data to SSD.

We note that WBL can be used even in such a three-tier storage hierarchy. In this case, the DBMS uses a SSD to store the less frequently accessed tuples in the database. It stores the log and the more frequently accessed tuples on NVM. As bulk of the data is stored on SSD, the DBMS only requires a less expensive NVM device with smaller storage capacity. We note that the impact of dynamic data placement on a three-tier storage hierarchy containing NVM must be explored in future work.

A related line of research focuses on the problem of evicting cold data to disk in main memory DBMSs [56]. The *anti-caching* architecture for H-Store [2] moves cold tuples from DRAM to a disk-resident hash table [34, 71]. To evict data, the DBMS invokes a special system transaction that blocks other transactions while it combines the coldest tuples into a block and writes them out to the anti-cache. When a transaction attempts to access an evicted tuple, the DBMS aborts the transaction and then asynchronously fetches the requested tuple in a separate thread.

Microsoft's Project Siberia [38, 35] for Hekaton takes a different approach. It identifies what tuples to evict in a background thread that analyzes the DBMS's logs, thereby avoiding the overhead of having to maintain a tracking data structure that is updated during execution [53]. The cold tuples are moved to secondary storage using a special migration transaction that is composed of insert and delete operations. An evicted tuple is merged back into memory only when it is updated by a transaction, otherwise it is stored in a private cache and then released after that transaction terminates.

4.4 Replication

With the WBL logging protocol described in Section 4.2, the DBMS can recover from system and transaction failures. However, it cannot cope up with media failures or corrupted data. This is because it relies on the integrity of durable data structures (e.g., the log) during recovery. These failures are instead overcome through replication, wherein the DBMS propagates changes made by transactions to multiple servers [68, 45]. When the *primary* server incurs a media failure, replication ensures that there is no data loss since the *secondary* servers can be configured to maintain a transactionally consistent copy of the database.

The round trip latency between the primary and secondary server is on average a couple of orders of magnitude higher than the durable write latency of NVM. The networking cost is, thus, the major performance bottleneck in replication. A faster replication standard, such as the NVMe over Fabrics [10], is required for efficient transaction processing in a replicated environment containing NVM [82]. With this technology, the additional latency between a local and remote NVM device is expected to be less than a few microseconds. As every write must be replicated in most usage models, we expect a logging scheme designed for NVM to outperform WAL in this replicated environment because it incurs fewer writes.

5. EXECUTION ENGINE

We now describe the changes necessitated by the advent of NVM in different components of the execution engine. This the part of the DBMS that is responsible for generating query plans and executing them on the database.

5.1 Plan Executor

The algorithms backing the relational operators in main-memory DBMSs are designed to have low computational complexity and to exploit the processor caches in modern multi-core chips [23, 41, 47, 13]. We now need to redesign these algorithms to also reduce the number of writes to durable storage, given the read-write asymmetry and limited write-endurance of NVM [27, 76, 42, 20, 22].

The cache-friendly implementation of the hash-join algorithm partitions the input tables so that every pair of partitions can fit within the CPU caches. Unfortunately, the partitioning phase necessitates writing out the entire tables back on NVM in their partitioned form [27]. One can avoid this by keeping track of *virtual partitions* of the tables that only contain the identifiers of the tuples that belong to a given partition, and accesses the records in place during the join phase. In this manner, virtual partitioning avoids data copying to reduce the number of writes at the expense of additional reads.

For sorting NVM-resident data, the DBMS can use a hybrid write-limited sorting algorithm called *segment sort* [76, 22]. This algorithm sorts a fraction of the input using the write-intensive faster external merge-sort algorithm, and the remaining fraction using the write-limited slower selection sort algorithm. The selection sort algorithm involves multiple read passes over the input, and writes each element of the input only once at its final location. The DBMS uses the fraction as a knob for constraining the write-intensiveness of the algorithm with respect to its symmetric-I/O counterpart at the cost of lower performance.

Like with sorting, there are also join algorithms that are designed for asymmetric NVM storage. The *segmented Grace hash-join*, which unlike the regular Grace join, materializes only a fraction of the input partitions, and continuously iterates over the rest of the input to process the remaining partitions [76]. The associated read-amplification does not hurt performance given the read-write asymmetry. We note that it is important that write-limited algorithms converge to the I/O-minimal behavior of their counterparts that are designed for symmetric I/O at lower write-intensity levels.

5.2 Query Optimizer

Cost-based query optimizers in modern DBMSs are designed to take into consideration the gap between sequential and random I/O costs of durable storage devices. They, however, do not account for read/write asymmetry exhibited by NVM while performing sequential and random I/O [19, 65]. The optimizer must, therefore, differentiate between reads and writes, and take into consideration the convergence of sequential and random accesses in NVM.

The table and index scan algorithms retrieve the tuples satisfying a given predicate by scanning through the associated table and index respectively. The original cost functions of these algorithms in the optimizer only distinguish between sequential and random accesses. We adapt these functions to indicate that these accesses are reads. Similarly, when the result of a sub-tree in an execution plan is needed multiple times by the associated parent node, the DBMS materializes it. The cost function for this materialization operation should indicate that the associated accesses are writes.

We adapt the cost function of a join algorithm by considering the two phases of the algorithm. The function should account for writing and reading all the data one time each. All the reads during the join phase tend to be sequential, while the writes in the partitioning phase are random. We note that these adapted cost functions still do not account for the byte-addressability of NVM [77].

5.3 SQL Extensions

The DBMS contains certain SQL extensions to allow the user to control data placement on NVM [11, 7]. For instance, the user can indicate that certain performance-critical tables and materialized views should reside on NVM using the ON_NVM attribute. When this attribute is specified for a tablespace, the DBMS creates all the tables and materialized views within this tablespace on NVM.

```
ALTER TABLESPACE nvm_table_space DEFAULT ON_NVM;
```

By default, the DBMS stores all the columns in a table tagged with the ON_NVM attribute on NVM. However, the user can choose to store only a subset of the columns on NVM if desired. For instance, the following SQL statement excludes the ORDER_TAX column in the ORDERS table from being stored on NVM.

```
ALTER TABLE orders ON_NVM EXCLUDE(order_tax);
```

6. LESSONS LEARNED

The advent of NVM invalidates the long-held assumption that durable storage is several orders of magnitude slower relative to the CPU. The shrinking I/O gap makes it challenging for the DBMS to saturate these devices even with an efficient storage manager [60, 32, 18]. Hence, it is important to reexamine the design choices made in different components of the DBMS to leverage the raw device performance differential. Our own experience with redesigning the logging and recovery protocols for NVM has shown that it is useful to reflect on the changes required for a storage hierarchy comprising of only NVM, which is an interesting point in the design space [14, 15]. We have highlighted only a subset of open research problems. Given that the performance of the storage layer has improved by several orders of magnitude over a short period of time, we anticipate high-impact research in this space.

7. ACKNOWLEDGEMENTS

This work was supported (in part) by the Intel Science and Technology Center for Big Data, the U.S. National Science Foundation (CCF-1438955), and the Samsung Ph.D. Fellowship Program.

8. BIOGRAPHIES

Joy Arulraj is a Ph.D. candidate at Carnegie Mellon University. His research focuses on the design and implementation of non-volatile memory database management systems. He interned at the Microsoft Research Database Group in 2016. He is a recipient of the 2016 Samsung Ph.D. Fellowship.

Andrew Pavlo is an Assistant Professor of Databaseology in the Computer Science Department at Carnegie Mellon University. At CMU, he is a member of the Database Group and the Parallel Data Laboratory. His work is also in collaboration with the Intel Science and Technology Center for Big Data.

References

[1] Direct access for files (DAX). https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[2] H-Store. http://hstore.cs.brown.edu.

[3] LIBNVDIMM: Non-volatile devices. https://www.kernel.org/doc/Documentation/nvdimm/nvdimm.txt.

[4] Peloton Database Management System. http://pelotondb.org.

[5] Persistent memory file system (PMFS). https://github.com/linux-pmfs/pmfs.

[6] Persistent memory programming library. http://pmem.io/.

[7] SAP HANA administration guide: Load/unload a column table into/from memory. http://help.sap.com/saphelp_hanaplatform/helpdata/en/c1/33165bbb57101493c5fb19b5b8607f/content.htm.

[8] JEDEC announces support for NVDIMM hybrid memory modules. https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules, 2015.

[9] Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf, 2016.

[10] NVM Express over Fabrics specification. http://www.nvmexpress.org/specifications, 2016.

[11] Oracle In-Memory Database White Paper. http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html, 2016.

[12] R. Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. IWDM, pages 269–285, 1989.

[13] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[14] J. Arulraj, A. Pavlo, and S. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[15] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. In *VLDB*, 2017.

[16] M. M. Astrahan and *et al.* System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[17] A. Badam and V. S. Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *NSDI*, pages 211–224, 2011.

[18] P. Bailis, C. Fournier, J. Arulraj, and A. Pavlo. Research for Practice: Distributed consensus and implications of NVM on database management systems. volume 14 of *Queue*, July 2016.

[19] D. Bausch, I. Petrov, and A. Buchmann. Making cost-based query optimization asymmetry-aware. In *DaMoN*, pages 24–32. ACM, 2012.

[20] N. Ben-David and *et al.* Parallel algorithms for asymmetric read-write costs. In *SPAA*, 2016.

[21] K. Bhandari and *et al.* Implications of CPU caching on byte-addressable non-volatile memory programming. 2012.

[22] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *SPAA*, pages 1–12, 2015.

[23] P. Boncz and *et al.* Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[24] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. Res. Dev.*, 52(4):449–464, July 2008.

[25] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 2015.

[26] F. Chen, M. P. Mesnier, and S. Hahn. A protected block device for persistent memory. In *MSST*, pages 1–12, 2014.

[27] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31, 2011.

[28] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

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

[30] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, pages 133–146, 2009.

[31] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for Safe RAM. VLDB, pages 327–335, 1989.

[32] J. Corbet. LFCS: Preparing linux for nonvolatile memory devices. LWN, April 2013.

[33] J. DeBrabant, J. Arulraj, and *et al.* A prolegomenon on OLTP database systems for non-volatile memory. In *ADMS@VLDB*, 2014.

[34] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, Sept. 2013.

[35] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, 2013.

[36] A. Driskill-Smith. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[37] S. R. Dulloor, S. K. Kumar, A. Keshavamurthy, P. Lantz, D. Subbareddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[38] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

[39] M. Franklin. Concurrency Control and Recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997.

[40] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: Reducing transaction logging overhead with PCM. In *CIKM*, 2011.

[41] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE TKDE*, pages 509–516, Dec. 1992.

[42] V. Garg, A. Singh, and J. R. Haritsa. On improving write performance in PCM databases. Technical report, TR-2015-01, IISc, 2015.

[43] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS Fast Path. Technical report, Tandem, 1985.

[44] G. Graefe and *et al.* Instant recovery with write-ahead logging: Page repair, system restart, and media restore. *Synthesis Lectures on Data Management*, 2015.

[45] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Record*, 25(2):173–182, 1996.

[46] T. Härder, C. Sauer, G. Graefe, and W. Guy. Instant recovery with write-ahead logging. *Datenbank-Spektrum*, pages 235–239, 2015.

[47] S. Harizopoulos and *et al.* Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.

[48] R. Harris. Windows leaps into the NVM revolution. `http://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/`, Apr. 2016.

[49] A. Hassan and *et al.* Energy-efficient in-memory data stores on hybrid memory hierarchies. In *DaMoN*, page 1. ACM, 2015.

[50] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, pages 389–400, Dec. 2014.

[51] H. Kim and *et al.* Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *FAST*, 2014.

[52] T. Klima. Using non-volatile memory (NVDIMM-N) as byte-addressable storage in windows server 2016. `https://channel9.msdn.com/events/build/2016/p470`, 2016.

[53] J. J. Levandoski, P.-Å. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[54] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. ICDE, pages 302–313, 2013.

[55] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang. NVM Duet: Unified working memory and persistent store architecture. *SIGARCH Computer Architecture News*, 42(1):455–470, 2014.

[56] L. Ma and *et al.* Larger-than-memory data management on modern storage hardware for in-memory oltp database systems. In *DaMoN*, 2016.

[57] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A case for efficient hardware/software cooperative management of storage and memory. 2013.

[58] C. Mohan and *et al.* ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[59] I. Moraru and *et al.* Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS*, 2013.

[60] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. Non-volatile Storage: Implications of the datacenter's shifting center. volume 13 of *Queue*, January 2016.

[61] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *SIGMOD*, pages 371–386, 2016.

[62] I. Oukid and *et al.* SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. DaMoN, 2014.

[63] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, et al. Self-driving database management systems. In *CIDR*, 2017.

[64] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[65] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be ssd-aware? 2011.

[66] T. Perez and C. Rose. Non-volatile memory: Emerging technologies and their impact on memory systems. *PURCS Technical Report*, 2010.

[67] S. Raoux and *et al.* Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[68] D. Roberts. *Efficient Data Center Architectures Using Non-Volatile Memory and Reliability Techniques*. PhD thesis, University of Michigan, 2011.

[69] A. Rudoff. Deprecating the PCOMMIT instruction. `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`, 2016.

[70] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm malloc: Memory allocation for NVRAM. In *ADMS*, pages 61–72, 2015.

[71] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *DaMon*, 2013.

[72] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[73] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, (7191):80–83, 2008.

[74] S. Venkataraman and *et al.* Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, 2011.

[75] S. D. Viglas. Adapting the b+-tree for asymmetric i/o. In *ADBIS*, pages 399–412, 2012.

[76] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[77] S. D. Viglas. Data management in non-volatile memory. SIGMOD, pages 1707–1711, 2015.

[78] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS*, 2011.

[79] C. Wang and *et al.* NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines. In *IPDPS*, pages 957–968. IEEE, 2012.

[80] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[81] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: reducing consistency cost for NVM-based single level systems. In *FAST 15*, pages 167–181, 2015.

[82] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.