Datenbank Spektrum (2018) 18:171–182
https://doi.org/10.1007/s13222-018-0301-1

SCHWERPUNKTBEITRAG

Data Management on Non-Volatile Memory: A Perspective

Philipp Götze[1] · Alexander van Renen[2] · Lucas Lersch[3,4] · Viktor Leis[2] · Ismail Oukid[4]

Received: 1 June 2018 / Accepted: 24 September 2018 / Published online: 5 October 2018

Abstract
The large performance gap between main memory and secondary storage accounts for many design decisions of traditional database systems. With the upcoming availability of Non-Volatile Memory (NVM), which has latencies in the same order of magnitude as DRAM, is byte-addressable and persistent, a completely new type of technology is added to the memory stack. This changes some basic assumptions such as slow storage, block granular access, and that sequential accesses are much faster than random accesses. New ideas are therefore needed to efficiently leverage NVM. Although several new approaches can be found in the literature, the exact role of NVM is not yet clear. In this paper, we survey recent work in this area and classify the existing approaches. We focus on two key challenges: (1) integration of NVM into the memory hierarchy and (2) the design of NVM-aware data structures. We contrast the different approaches, highlight their advantages and limitations, and make recommendations.

Keywords Non-volatile memory · Persistent memory · Data management · Databases

1 Introduction and Motivation

With manufacturers finally getting close to releasing Non-Volatile Memory (NVM)[1], it has gained a lot of popularity recently. This new storage technology is a hybrid between DRAM and flash (SSD): It combines byte-addressability and low read latencies (similar to those of DRAM) with the persistence and density of SSDs. In addition, the monetary cost per GB of NVM is predicted to be between that of DRAM and SSDs. Thus, it would be possible for NVM to replace one or both of these technologies. However, there are also drawbacks such as the limited write endurance and the higher write latency compared to DRAM as well as smaller capacities than disks. Therefore, NVM might not be the all-in-one solution, but instead a new layer in the memory hierarchy. In the first part of this work, we present possible server architectures for NVM together with their opportunities and limitations. After that, we survey software solutions for these architectures.

Traditional DBMSs are typically optimized to deal with the trade-offs of fast, volatile memory and slow, persistent storage. NVM disrupts this assumption and opens up new possibilities for many components of a database system. As highlighted in [62], a lot of potential is lost when simply treating NVM like DRAM or disk and reusing traditional abstractions. Hence, there is a need for new techniques to fully leverage NVM. The most obvious starting points are a faster recovery and the utilization of efficient random accesses [10, 14, 19, 35, 50, 63]. Much of the recent work examines primary data and index structures such as B+-trees [16, 20, 52, 61] or log-structured merge trees [27, 42, 43].

Our goal is to discuss recent work in the area of data management on NVM and to give a perspective for future research. We, therefore, focus on the storage component of data management systems, i.e., which data placement

[1] Also known as Persistent Memory (PM), Non-Volatile Random Access Memory (NVRAM), or Storage Class Memory (SCM).

✉ Philipp Götze
philipp.goetze@tu-ilmenau.de

✉ Alexander van Renen
renen@in.tum.de

✉ Lucas Lersch
lucas.lersch@sap.com

Viktor Leis
leis@in.tum.de

Ismail Oukid
ismail.oukid@sap.com

[1] Technische Universität Ilmenau, Ilmenau, Germany

[2] Technische Universität München, München, Germany

[3] TU Dresden, Dresden, Germany

[4] SAP SE, Walldorf, Germany

strategies would be possible and in which aspects of data management is NVM appropriate.

We first give an overview of NVM characteristics in Section 2. After that, Section 3 discusses how NVM hardware can best be added to current systems and then survey previously proposed storage engines in Section 4. Finally, we summarize our findings and give an outlook on future research directions in Section 5.

2 NVM Basics

2.1 History

In the early 70s, Leon Chua predicted the existence of the *memristor* [21], a fundamental circuit element capable of storing and retaining a state variable. In the late 2000s, a team at HP Labs discovered the memristor in the form of Resistive RAM (RRAM) [29, 60]. Since then, other memory technologies that exhibit the memristor property, i.e., the ability to store and retain a state variable, have been identified, e.g., Phase-Change Memory (PCM) [38] and Spin Transfer Torque Magnetic RAM (STT-MRAM) [33]. In the industry, these novel memory technologies are grouped under the umbrella term *Storage-Class Memory* (SCM) or Persistent Memory (PM). Here, we use the term Non-Volatile Memory (NVM).

After decades of research, the advent of NVM seems to be imminent. Intel has recently announced availability of its Optane DC Persistent Memory large-capacity DIMMs (Dual Inline Memory Module), based on a new NVM technology called 3D XPoint [3], for developers with a broad commercial availability for early 2019 [7]. Furthermore, there have been recent breakthroughs in the industry efforts to standardize different Non-Volatile DIMMs (NVDIMMs) form factors [1]. Moreover, Microsoft Windows [37] and Linux [8] have both announced support for NVDIMMs.

2.2 Properties

NVM promises to combine the low latency and byte-addressability of DRAM, with the density, non-volatility, and economic characteristics of traditional storage media. Regarding byte-addressability, even if both DRAM and NVM could be accessed one byte at a time, modern CPU architectures still access them in "blocks" referred to as cache-lines (typically 64 Bytes). However, a more important aspect of byte-addressability is the possibility of the device being directly accessed by the CPU through its caches, unlike most storage devices. Moreover, most NVM technologies exhibit asymmetric latencies, with writes being noticeably slower than reads. Table 1 summarizes the characteristics of three NVM candidates, RRAM, PCM and STT-MRAM, and compares them with those of DRAM. Like Flash Memory, NVM supports a limited number of writes; yet, from a material perspective, some NVM candidates promise to be as write-enduring as DRAM. We anticipate that the issue of NVM's limited write endurance will be addressed at the hardware level, such as already proposed by previous work [44, 57]. These candidates also promise to feature even lower latencies than DRAM. However, while its manufacturing technology matures, we expect the first few generations of NVM to exhibit higher latencies than DRAM, especially for writes. Given its non-volatility, idle NVM cells do not consume energy, contrary to DRAM cells that constantly consume energy to refresh their state. Consequently, NVM has the potential to reduce energy consumption, albeit, except for embedded systems where NVM chips are simple and have much lower power leakage than DRAM [34], this potential has yet to be shown for NVM DIMMs which embed complex logic and buffers to optimize for access latency and bandwidth. The aspect of economic costs combined with typical access characteristics was first discussed in the context of the five-minute rule [31], which states that disk pages accessed every five minutes or less should be cached. Thirty years later, this rule was revisited and also briefly considers upcoming NVM technologies, but with more focus on SSDs [12]. In terms of security, it is often imperative that only encrypted data becomes persistent. Therefore, Optane DC Persistent Memory, for example, offers encryption as a built-in hardware feature [7].

Table 1 Comparison of the characteristics of DRAM with those of three NVM candidates [45]

Parameter	DRAM	PCM	STT-MRAM	RRAM
Read Latency	50 ns	50 ns	10 ns	10 ns
Write Latency	50 ns	500 ns	50 ns	50 ns
Endurance	$>10^{15}$	10^8–10^9	$>10^{15}$	10^{11}
Density	Low	Medium	Low	High

2.3 NVM Access Model

The SNIA (Storage Networking Industry Association) recommends to manage NVM devices through a file system, similar to HDD and SSD [2]. Based on that, applications can access an NVM device via two different methods, as seen in Figure 1. In the first one, files on NVM can be accessed through standard system calls such as *open*, *read*, *write*, etc. This enables existing systems to take advantage of NVM performance to a certain extent, without requiring additional integration effort, since the file system interface is kept the same. Furthermore, file systems already provide most of the functionality required for managing a storage device: naming, corruption handling, persistent

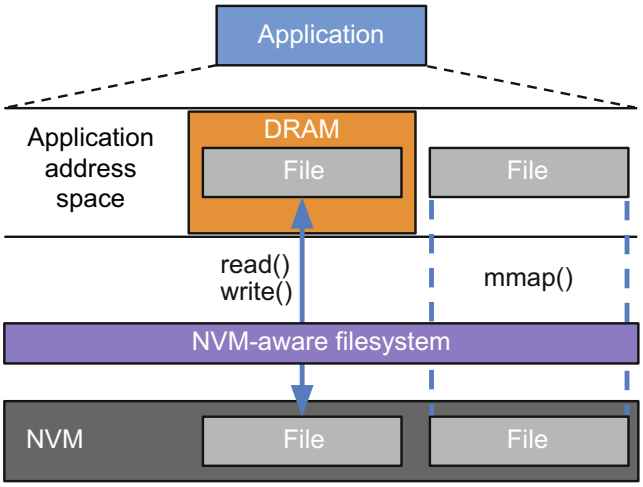


Fig. 1 Basic access methods to NVM device

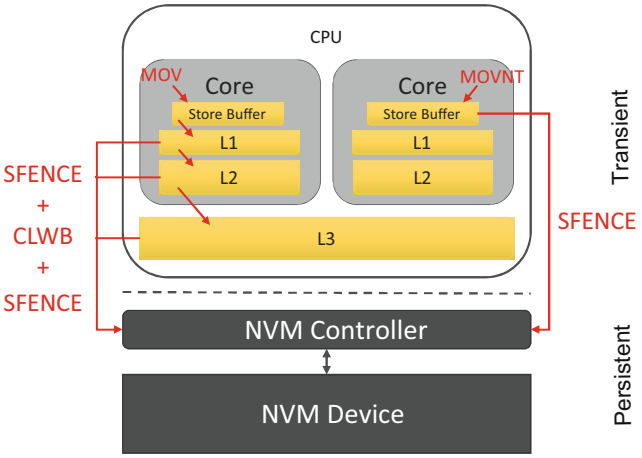


Fig. 2 Volatility chain in a 2-core x86-like processor

allocation, etc. Several general purpose NVM-aware file systems were proposed to better leverage NVM properties while maintaining the same interface to applications. Some examples of such file systems are: BPFS [23], PMFS [26], HiNFS [48], SCMFS [65], and NOVA [67].

In the second case, given NVM's byte-addressability and low latency, data in NVM can be accessed via load and store instructions through the CPU caches without buffering it in DRAM. Specialized applications like databases usually implement their own management of resources and might desire to leverage load/store semantics. To enable that, the file system must offer zero-copy (i.e., without any DRAM buffering) memory mapping of files residing on NVM to the application's virtual memory space. As an example, this feature is supported by ext4 [5] on Linux kernels 4.7 and above under the name *Direct Access* (DAX) [4]. Any load and store operation issued to an address inside the specified memory mapped region will access the NVM device directly. While this access method loses the benefits of transparent memory defragmentation and swapping enabled by the duality of device and DRAM pages, the virtual memory indirection still provides benefits such as: process isolation, position independent code/data, and memory sharing between processes. Without virtual memory, applications relying on these functionalities would have to be reconsidered.

2.4 NVM Programming Challenges

With the unique opportunities brought by NVM comes a set of novel programming challenges: (1) data consistency, (2) data recovery, (3) persistent memory leaks, (4) partial writes, and (5) persistent memory fragmentation. We briefly discuss these challenges in the following.

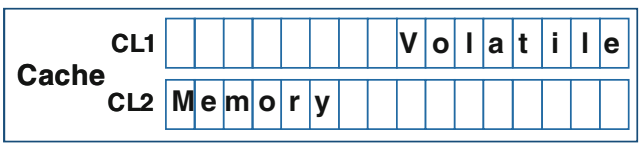
Data Consistency. NVM can be accessed directly with load and store semantics. The path from NVM to CPU registers is long and mostly volatile, as illustrated in Figure 2. It includes store buffers and CPU caches, over all of which software has little control. Additionally, modern CPUs implement complex out-of-order execution and either partial store ordering (Intel x86) or relaxed-memory ordering (ARM, IBM Power). Consequently, memory stores need to be explicitly ordered and persisted to ensure consistency. Current x86 CPUs provide the *clflush*, *mfence*, *sfence*, and non-temporal store instructions (*movnt*) to handle memory ordering and data durability. Additional instructions, namely *clflushopt* and *clwb*, have been announced for future platforms [6]. *clflush* evicts a cache line and writes it back to memory. It is a synchronous instruction and does not require a memory fence to be serialized. *sfence* is a memory barrier that serializes all pending stores, while *mfence* serializes both pending loads and stores. Non-temporal stores bypass the cache by writing to a special buffer, which is evicted either when it is full, or when an *sfence* is issued, as shown in Figure 2. *clflushopt* is the asynchronous version of *clflush*; it is not ordered with writes, which improves its throughput. Finally, *clwb* writes back a cache line to memory, but without evicting it from the CPU cache, thereby benefiting performance when data is accessed shortly after it is persisted. *clwb* executes asynchronously and is not ordered with writes. Both *clflushopt* and *clwb* require two *sfences* to be serialized, as illustrated in Figure 2; the first *sfence* ensures that the latest version of the data is flushed, while the second *sfence* ensures that the flushing instruction finishes executing.

Data Recovery. When a program restarts, it loses its previous virtual address space, including its memory mappings, invalidating any stored virtual pointers. Given that NVM is addressed using virtual pointers (just like DRAM), there is

a need to devise ways of discovering and recovering data stored in NVM. Using a file system on top of NVM provides a way of discovering data after a restart. Reads and writes to a file created and memory mapped by an NVM-aware file system are made with direct load and store instructions. Hence, NVM-aware file systems should not have a negative performance impact on the application's performance. Therefore, a potential solution would be to create one file per data object. However, database systems can have millions of objects of different sizes, ranging from a few bytes to several gigabytes, making this solution unrealistic. Therefore, as an alternative, NVM-aware allocators [18, 47, 53, 59] have been introduced which use offsets instead of virtual pointers to support restarts and re-mappings of the address space.

Non-Volatile Memory Leaks. Memory leaks pose a greater problem with persistent memory than with volatile memory: they are *persistent*. Additionally, NVM faces a new class of memory leaks resulting from software or power failures. To illustrate this problem, consider the example of a linked list. If a crash occurs during an append operation after a new node was allocated but before it was linked to the tail node, the NVM allocator will remember the allocation while the data structure will not, leading to a persistent memory leak, as depicted below:

Partial Writes. We define a *p-atomic* store as one that is retired in a single CPU cycle, i.e., a store that is immune to partial writes. Current x86 CPUs support only 8-byte p-atomic stores; larger write operations are prone to partial writes since the CPU can speculatively evict a cache line at any moment. For example, suppose we want to write the string below to NVM:



If a failure occurs during the write operation, assuming we write 8-bytes at a time, the corresponding NVM string location might be in one of the following states: (1) “”; (2) “Volatile”; (3) “\0\0\0\0\0\0\0\0Memory”; (4) “Volatile Memory”. Cases 2 and 3 can result, for instance, from the writing thread being descheduled. Meanwhile, the CPU might speculatively flush the first or the second cache line due to a set conflict. A failure at this time would corrupt the string in NVM. A common way of addressing this problem is to use flags that can be written p-atomically to indicate whether a larger write operation has completed.

Non-Volatile Memory Fragmentation. While a restart remains a valid, but last-resort way of defragmenting volatile memory, it is not effective in the case of NVM – objects stored in NVM survive restarts. This is a similar problem to that of file systems. However, file system defragmentation solutions cannot be applied to NVM because file systems have an additional indirection layer: they use virtual memory mappings and buffer pages in DRAM, which enables them to transparently move physical pages around to defragment memory. In contrast, NVM mappings give direct physical memory access to the application layer without buffering data in DRAM. Hence, NVM cannot be transparently moved as it is bound to its memory mapping.

From the above challenges, we conclude that there is a need for novel programming models for NVM, which must provide data discovery and recovery mechanisms, a data consistency model, and prevent failure-induced memory leaks. A recent tutorial elaborated in detail on existing techniques to fulfill each one of these requirements [49]. These techniques are usually provided in libraries that serve as an interface between NVM and the programmer. Among many libraries, Intel's Persistent Memory Development Kit (PMDK) [9] is emerging as a common standard.

3 NVM in the Hardware Landscape

NAND (flash) SSDs were introduced in the early 90s and by the beginning of the 2000s they were already common in servers and data centers. While they differ from classical HDDs in their internal operation, the interface exposed to the user application is very similar, i.e., a block-based interface. Therefore, whenever talking to the storage device, the application must convert data from its logical domain (e.g., an object) residing in DRAM to a serialized format to be stored in persistent devices. Due to the common interface, better performance, and initially higher cost, SSDs have naturally found their place between DRAM and HDD in the modern storage hierarchy.

Not only systems rely on SSDs to implement a persistent cache to HDDs, but also hybrid devices were common even in a consumer-level, due to the initial high costs of SSDs. Nowadays SSDs have evolved to a point where they offer a much higher performance, improved reliability, and lower cost compared to the first models released. It is therefore not unrealistic that SSDs will largely replace HDDs in the near future – with the exception of archival data.

With this brief history of SSDs in mind and watching all the advancements and buzz going on around upcoming NVM technologies, one is to wonder if there is really any

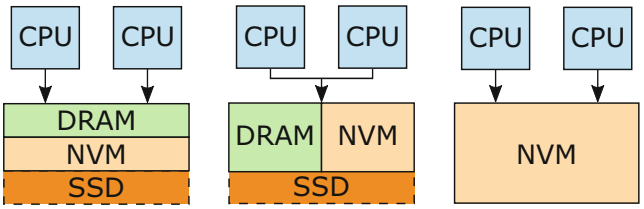


Fig. 3 Placement strategies for NVM in the hardware landscape. *Dotted lines* denote optional component. From *left* to *right*: NVM below DRAM, NVM side-by-side with DRAM, NVM-only

reason for excitement or if history is going to simply repeat itself. While bandwidth, latency, reliability, and density of upcoming NVM devices promise to be much better than modern top-tier SSDs, these aspects have little impact on the way we design system architectures when compared to the aspect of *byte-addressability*. In this survey, whenever we refer to byte-addressability the focus is the direct access aspect, rather than the granularity in which the access occurs. This single characteristic enables a much higher degree of flexibility regarding the placement of NVM devices in the modern storage hierarchy.

In the following subsections, three strategies for integrating NVM devices into modern and future hardware landscapes are categorized, as seen in Fig. 3 and similarly discussed in [54]. While approaches can be easily combined and extended, we consider that these basic categories offer a common ground to initially classify existing systems.

3.1 NVM below DRAM

A first option for placing NVM devices in the storage hierarchy is to follow the intuitive approach and use them as middle-tier between DRAM and SSD. Such a design is justified by the fact that the expected initial NVM performance and price falls exactly in between the gap of these devices. This allows systems with NVM to implement a persistent caching for SSDs [27, 43].

Two other alternatives are: statically placing data in NVM or SSD based on the application logic, or allow data to dynamically move between these devices based on a certain policy. Both approaches differ from caching by the fact that only a single persistent copy of data exists, either in NVM or SSD. However, based on the assumption that the initial cost per GB of SSDs will remain much lower than that of NVM, saving space on SSD is probably not advantageous if that would introduce non-trivial complexity to the data placement algorithm.

Regarding the interaction with DRAM, in this placement strategy, NVM is seen as a storage device. Similar to SSD, data is always copied from NVM to DRAM before being read or written by the CPU, but possibly in a much smaller granularity (one or a few cache-lines). Therefore, well-known techniques for defining page sizes can still be applied to optimize data movement between devices [17]. Such an approach allows the developer to control when data is made persistent, e.g., via buffer pools, independent of the operating system or the hardware, which significantly reduces the implementation complexity required for guaranteeing consistency.

According to [25], NVM can also act as an anti-cache [24], with data being moved from DRAM to the lower NVM layer when the amount of data exceeds a certain threshold. Such an anti-cache is based on the ideas that exactly one copy of a data element exists at any time and that no synchronization mechanisms are required between a cached and a major copy.

3.2 NVM side-by-side with DRAM

With NVM being directly accessed by the CPU through the memory bus, leveraging the opportunity of low latency direct access, while non-trivial, has potential to improve the performance of systems. On the one hand, systems with high performance requirements, like OLTP, will still run mainly on faster DRAM. On the other hand, in cost-efficient scenarios one might decide for slightly lower performance if that means considerably lower hardware costs. Nevertheless, assuming that the latency of DRAM will still be lower than NVM, a careful placement of data is required to achieve the best of both worlds. In that regard, the placement of data in either DRAM or NVM could be static or dynamic.

A static placement of data implies that system developers or users explicitly choose the most appropriate device. Examples of that are hybrid data structures, such as B+-Trees placing the leaf nodes in NVM and inner nodes in DRAM [52]. Different node sizes can be employed to better leverage the characteristics of either DRAM or NVM and optimize data movement to CPU caches. This approach offers benefits such as low DRAM consumption (since inner nodes are only a small fraction of the total number of nodes) and recovery simplicity (as leaf nodes do not have to be recovered). Another example is the proposal to extend the SQL standard to expose the underlying hardware landscape and allow the user to explicitly specify if a table, column, or partition should be DRAM-based or NVM-based [13].

Statically placing data, while simpler, has the limitation of not being able to adapt to changes in the workload and hardware. Abstractions well known to database systems, such as buffer policies, can be used to dynamically decide where data is placed. For example, a leaf-node of a B+-Tree being frequently accessed could be placed in DRAM to leverage its lower latency and higher bandwidth. Cost functions to evaluate and predict access patterns should be used to decide how and when movement of data is advantageous.

One important thing to consider in such a scenario is that, while data in HDD/SSD must always be loaded to DRAM, in a side-by-side scenario data in NVM can either be read directly or transferred to DRAM prior to the access. As an example, sequential scans could access data directly in NVM, thereby avoiding trashing the DRAM buffer and hiding higher NVM latencies through hardware pre-fetching.

3.3 NVM-only

A third scenario extrapolates the expectations of upcoming technologies and considers that NVM can be used as universal memory, eliminating the boundaries between volatile and persistent devices. In such case, NVM devices would have to provide performance characteristics equal or better than other volatile memory technologies. While this scenario might be unrealistic in the short and midterm, thinking about the design of systems in such a context raises interesting discussions.

Algorithms for storage and recovery of databases having non-volatile main memory have already been discussed for a long time [10]. More recently the discussion was reignited [14] in view of modern technologies announced by hardware vendors [3]. While the concept of non-volatile and byte-addressable main-memory is common to most works, algorithms vary depending on the assumptions made about the characteristics of the underlying hardware, such as write endurance, latency, and persistence guarantees.

In an even more extreme scenario, CPU caches and registers could also be made persistent to a point where volatility is completely eliminated from the hardware system. CPUs would require additional support for dealing with such an architecture in the case of power failures. It is not intuitive, however, to think about systems for a landscape where volatility is completely eliminated. Even if data is always persistent and available, modern systems still present a clear separation between data that makes sense only during runtime and data that has to be persistent. As an example, consider that persisting the state of a reference counter or of a latch would require the application to guarantee that a deadlock never occurs since the state cannot be reset by rebooting. While challenging, designing a system that fully leverages persistency in a consistent way could result in a never-stopping/no-recovery system with significantly improved availability.

4 NVM-aware Data Structures and Engines

In this section, we evaluate storage engines for NVM-based hardware architectures, as introduced in Section 3. We define a storage engine as the part of a data management system that provides transactional semantics on top of the underlying hardware; i.e., the atomicity, consistency and durability aspects of the ACID principle. We focus on tree-like structures due to their ubiquity in database systems [22].

In the following, we select representative existing techniques and categorize them depending on how they use NVM into two groups: First, **NVM-direct engines** (Section 4.1), which use NVM as their primary working memory and build on the introduced NVM-only and NVM + DRAM hardware architectures. They perform updates directly on NVM, without buffering in DRAM. Thus, making all changes immediately persistent, which is a double-edged sword: On the one hand it allows for instant restarts and a single layer architecture, but on the other hand, poses significant challenges in ensuring failure atomicity.

The second group (Section 4.2) introduces **buffered engines** that avoid this issue by updating data in a DRAM buffer before writing it back to NVM. In addition, they are able to bundle multiple writes together, thus reducing the wear and amount of high latency writes. The downside is that they need to perform additional work to manage the DRAM buffer.

4.1 NVM-direct Engines

Storage engines of this category place and work on data directly in NVM. Thus, they essentially use NVM as a traditional random access memory (i.e., DRAM), which is also durable. At a first glance, working directly and exclusively on NVM is very promising:

Instant Restart. On block-based devices, it is not possible to update individual data items (hundreds of Bytes), but only entire blocks (several thousand Bytes). Therefore, it is infeasible to write back individual data records. Instead, these random writes are buffered and a write-ahead log [46] is used to ensure durability. During restart, the log is used to redo changes that have not been written back (recovery phase).

The byte-addressability of NVM makes it feasible to persist individual records immediately. Thus, eliminating the need for a DRAM-resident buffer and making it possible to maintain an up-to-date state on durable storage (NVM). The advantage is two-fold: (1) data is only written once (no log) and (2) restart times are greatly reduced (no recovery phase).

Single Layer. Around 40% of the work done by a traditional database system is caused by logging and buffer management [32]. An NVM-direct engine can immediately avoid this due to its one layer architecture.

NVM Capacity. Due to the high density of NVM (compared to DRAM), NVM-direct engines can handle much larger workloads than main memory engines.

However, there is no free lunch and NVM-direct engines also pose new challenges. In the remainder of this section, we summarize the existing literature on this.

4.1.1 Failure Atomicity

NVM-direct engines are able to immediately persist updates and thereby avoid costly restart procedures. However, there is a flip side to this coin: Even intermediate and potentially inconsistent changes to a data structure might be persisted because it is not possible to prevent writes from being propagated to NVM (cf. Section 2). Therefore, it is necessary that data structures are always in a consistent state or there is additional information (e.g., logs) that can be used to restore them. This property is called failure atomicity.

Conceptually, failure atomicity can either be accomplished with in-place updates and logging or with shadowing [10]. However, for efficiency reasons, multiple techniques have been proposed to implement and (sometimes) combine these concepts:

Atomics. Atomic operations (e.g., compare-and-swap) serve as a building block for all following techniques. The NV-Tree [68] uses them to build unsorted B-Tree nodes: The data is first written into an empty slot and flushed to NVM. Afterwards, a flag is atomically swapped to indicate that the new slot is valid and occupied. To reduce the search overhead (linear scan), the FPTree [52] adds a fingerprinting array which uses hashing to avoid costly scans. Additionally, this technique has been used to construct a radix tree directly on NVM called WORT [39]. Note, that failure atomicity does not imply thread safety: Two threads could concurrently write to an unoccupied slot, thus corrupting each other's data.

Multi-Word Atomics. For performance reasons, B-Tree nodes often use sorted arrays, which, however, cannot be updated atomically due to the limited size of atomic operations on modern CPUs (8 Byte). To circumvent this, the PMwCAS [64] operation has been introduced. It is a software abstraction that provides a multi-word compare-and-swap operation with failure atomicity. It buffers changes and then atomically applies these via a descriptor (cf. [64]) in a thread-safe fashion. The operator successfully hides the complexity of dealing with NVM and has been used to construct a high performance, latch free B-Tree [16].

Indirection & Shadowing. The PMwCAS is widely applicable, however more problem specific approaches can outperform it by handling failure atomicity and thread safety manually. One way to achieve this is by moving larger data chunks (like the sorted array) out of the node and addressing via a pointer instead. In case of an update, a copy of the array is created and updated. Once the array is flushed to NVM, the pointer to it is atomically swapped. Lee et al. [39] use this technique to re-design the adaptive radix tree (ART) [40] for NVM to create the two persistent ART variants WOART and ART+CoW. The latter does not only move individual parts out of a node, but always copies and swaps the entire node (also known as shadowing or copy-on-write).

Invalidation. However, moving data out of the node causes additional cache misses (pointer chasing) and thereby might reduce performance. The wB-Tree [20] avoids this by keeping the slots unsorted but adding an additional array to store the slot indexes in sorted order. Whenever this array is updated a flag is set first to indicate that an update is in progress. If this flag is set during restart, the index array can be reconstructed.

Reconstruct. Simply detecting and then reconstructing corrupted data does not solve the failure atomicity problem. However, it provides a simple and efficient way to handle secondary data and has been driven further in the NV-Tree [68] and FPTree [52]: In a B-Tree all primary data is stored on the leaf pages, inner nodes do not contain additional information, but are merely a way to access leaf nodes faster. Hence, inner nodes can simply be reconstructed during recovery and do not need to be failure atomic.

Write-Ahead Logging. A more traditional way to ensure failure atomicity is by using a write-ahead log: Before any data is updated on the node itself, a log record is created and persisted. The failure atomicity of the log itself (append only) can easily be ensured by using the previously introduced atomics. During restart the log can be used to recover corrupted nodes. Unlike logging in traditional systems, this log can be transaction local and discarded once the actual data has been flushed. Due to the complexity of writing atomic data structures that also provide failure atomicity, some systems (wB-Tree [20] and FPTree [52]) fall back to logging for difficult operations (e.g., node splits). Other approaches [28] rely solely on undo-logs. They can avoid expensive NVM writes by using multi-dimensional clustering instead of multiple secondary indexes.

Write-Behind Logging. Contrary to the traditional ARIES-style write-ahead logging, [15] propose write-behind logging to better leverage the byte-addressability of NVM. Whenever a tuple is updated, a new version is created and written to NVM. At commit time, all changed tuples are persisted. After that, a log entry is written to set the new

versions valid. This can greatly reduce the amount of data being written compared to write-ahead logging, where each change is potentially recorded twice: (1) in the log and (2) in the data page. However, it incurs many random writes which is slow and wastes NVM endurance.

4.1.2 Hybrid Data Structures

As laid out in Section 3, NVM will likely not replace DRAM, but exist alongside to it and have a higher latency and lower bandwidth. Therefore, some of the NVM direct systems additionally utilize DRAM to improve their performance. The most prominent example is the FPTree [52], which is a B+-Tree that places its inner nodes in DRAM while keeping the leaf nodes on NVM. In case of a restart, the inner nodes can be reconstructed using the information from the leaves. A further example is HiKV [66], which implements an NVM-resident hash table and keeps a B+-Tree in DRAM to support range scans. The failure atomicity of the hash table is maintained with atomic operations and the B+-Tree can be reconstructed upon restart. These designs reduce the implementation effort of complicated operations (node splits) by moving them onto DRAM. Thus, trading performance for restart time.

4.1.3 Media Failure

A storage engine does not only need to handle system failures (e.g., power loss) but also media failures (e.g., broken NVM DIMMs):

Logging & Snapshots. Traditional storage engines write a log and take periodic snapshots of the database. In case of a failure, the latest snapshot is retrieved and the log is used to replay all changes, thus bringing the snapshot up to the most recent state. For failures of individual DIMMs, a technique called *instant recovery* [30] can be used to restore individual pages (or in this case a set of pages) "on-demand".

Replication. Alternatively, one could run additional backup instances of the storage engine, which perform the same work, thus serving as a replica. In case of a media failure, one of the standby servers can take over.

There has been little work on this topic and it would be interesting to see how the introduced storage engines can be extended to cope with this problem.

4.2 Buffered Engines

As shown in Section 4.1, performing updates directly on NVM requires careful engineering. In addition, the need to access NVM reduces performance compared to a main memory database engine as DRAM cannot be fully utilized. Therefore, we are considering alternative systems in this section that use a DRAM-resident buffer to perform updates. Thus solving the problem of leaking inconsistent state to durable storage. We will first look at ways to extend traditional (buffer-managed) engines and then investigate some novel engine designs.

4.2.1 Buffer-managed Engines

The most straightforward approach is to replace the SSD-based storage with NVM. This greatly improves the speed of page transfers between memory and storage, the logging latency (and therefore commit latency) and the restart time. From an engineering point of view, it is simple to realize and should mainly affect the storage layer of a well-designed database engine.

One of these systems is FOEDUS [36]. It uses fixed-size pages to transfer data from NVM to DRAM, where it is processed. An asynchronous process is used to merge changes back into the persistent NVM state using the logging information. It performs well as long as the hot data fits into DRAM but suffers once NVM is frequently accessed because it needs to transfer entire pages, unlike the NVM direct systems of Section 4.1, which can transfer individual tuples.

This issue is solved by a technique called cache-line-grained loading [58]: The idea is to keep a bit set for each DRAM-resident page indicating which cache lines have already been loaded. This way, only cache lines which are actually used are loaded. Thereby, the system can compete with NVM direct systems for NVM-sized workloads and even outperform these depending on the buffer pool size and the skewness of the workload.

Traditionally, buffer management is thought to be a heavy-weight component. However, recent work has shown that buffer-manged systems can compete with main memory engines [41] when designed under the assumption that most data fits in DRAM. Utilizing such lightweight buffer management techniques, a buffer-managed system can outperform NVM direct systems, as long as data fits into DRAM and (in the case of HyMem) stay competitive for larger workloads.

Additionally, the page-oriented memory layout of buffer-managed systems can be utilized to re-add an SSD without significant overheads. Thus ending up with a three-layer system (DRAM, NVM, SSD) that performs well for small workloads, but is still capable of using the high capacities of SSDs (or even HDDs). Compared to buffer-managed systems (DRAM+SSD), the additional NVM layer servers as a cache in SSD-sized workloads and provides a significant speedup.

In summary, lightweight buffer management in combination with cache-line-grained loading can be utilized to

extend a traditional buffer-managed engine to build a system that is more efficient or at least competitive for various workload sizes.

Buffer management has also been investigated in the context of log-structured merge-trees (LSMs) [55] on NVM. The LSMs are append-only systems and implement separated buffers for writing (MemTable) and reading (block cache). Due to its append-only nature, guaranteeing the failure atomicity of LSMs on NVM is less complex than in update-in-place systems. Furthermore, a block-cache policy has also been proposed to dynamically identify hot blocks and move them to DRAM, while still allowing cold blocks to be directly read from NVM without any additional transfer overhead [42].

4.2.2 Novel Engines

Besides an NVM-aware buffer manager, more radical designs have also been proposed. First of which is the SOFORT database engine [50, 51]. It uses a copy-on-write architecture where all primary data is stored and processed directly on NVM. Thus, it is possible to almost immediately restart after a crash or planned shutdown. It also eliminates the warm-up phase because no data needs to be re-loaded. The user can decide on the placement of secondary data (e.g., indexes). In addition, it allows placing secondary data (e.g., indexes) on DRAM, thus trading throughput for restart time.

The SAP HANA database has already been extend-ed to integrate NVM [11]. HANA's delta-merge architecture is well suited for NVM: All recently changed data resides in a small, write-optimized delta store. Periodically, the delta is merged into the large, read-optimized store. In the NVM adoption, the main store is placed on NVM while the delta store is kept in DRAM. This greatly extends the storage capacity and allows for faster restarts. However, point lookups in the main store suffer from the high NVM latency.

Another approach is the Peloton [56] in-memory database engine. While mainly designed as a fast self-driving DBMS, it also utilizes NVM with write-behind logging [15]. The idea is to write all changed tuples in-place to NVM at commit time. Once persisted, a single log entry is written to set the tuples atomically to a valid state.

5 Summary & Research Directions

We now summarize our findings from the previous sections to give an overview of representative existing approaches in the field of NVM-aware data structures. In addition, we formulate general recommendations and problems that have not yet been sufficiently discussed in the literature. Table 2 gives a brief overview of existing NVM-based data structures proposed in the literature. It covers the approaches applied to achieve failure atomicity (cf. Section 4) and concurrency, how the NVM properties were exploited (cf. Section 2), as well as data placement (cf. Section 3).

Failure Atomicity. In Section 4 we have already discussed the various techniques for achieving failure atomicity. We have seen, that all of them come with their advantages and disadvantages and thus there is no ultimate method. Libraries such as PMDK [9] may be able to bypass programming complexity. However, in some cases they unnecessarily double the costs of NVM writes due to logging. This is why most of the approaches currently try to control failure atomicity and recovery on their own. In the long term, we think new hardware features or specialized libraries will decrease the costs and complexity for guaranteeing failure atomicity, offering the best of both worlds: good performance and easy programming interface.

Concurrency. Only some of the approaches considered supporting concurrency. Others further assume that this is controlled in higher levels of a DBMS. Therefore concurrency is not dicussed extensively in the previous sections. We could envisage a combination of hardware and software-based concurrency schemes. After a limited number of retries, a hardware transaction could fall back to the programmer-defined approach as demonstrated with the FP-Tree [52]. Apart from PMwCAS [16] there is no true "compare, swap, and persist" operation, so one has to fall back on locks for the software-based transaction. However, it is not recommended to use standard locks on a persistent medium, as this can lead to persistent deadlocks or corrupted regions.

NVM Utilization. Basically, there are three aspects to be considered when utilizing NVM: reducing writes or general access to NVM, saving space, and fast recovery. The corresponding column in the table states how this is achieved. One of the most common optimization steps, at least for trees, is to leave the nodes unsorted to avoid unnecessary writes caused by shifting records. An obvious drawback of this is the impact on sorted-scans, which are an important use case in databases. For recovery, it seems to be the trade-off between fast restart but general higher latencies and rebuilding secondary data with faster general access. So far there is little consideration for the limited capacity of NVM compared to SSDs or HDDs. The saving of NVM capacities, however, also depends on the general data placement.

Data Placement. The prevailing architectures assumed in the literature are the following: NVM beside DRAM and NVM below DRAM (in case of caching). As already mentioned above, NVM is not deemed as a replacement, but is integrated into the architecture. Regarding the data place-

Table 2 Overview of existing NVM-based data structures

	Failure Atomicity	Concurrency	NVM Utilization	Placement
CDDS-Tree [61]	– versioning + atomics – shadowing (split,merge)	– only multiple-reader, single-writer model	– re-using dead versions	NVM-only
NV-Tree [68]	– append-only + atomics – reconstruction	– latches on modification – less locks due to append-only character	– unsorted leafs – cache-optimized format – selective consistency	NVM-only
wB-Tree [20]	– indirection + invalidation – redo-only logging (split,merge)	N/A	– unsorted nodes – binary search by sorted slot array	NVM-only
FP-Tree [52]	– indirection + invalidation – lightweight logging	– hardware-based + fine-grained locks	– reduced footprint – unsorted leafs – hashing – group allocation	NVM + DRAM
WO[A]RT/ ART+CoW [39]	– indirection + atomics – shadowing + atomics	N/A	– avoid key comparisons and tree rebalancing	NVM-only
HiKV [66]	– shadowing + atomics	– hardware-based for tree – locking within hash partitions	– avoids costly structure operations	NVM + DRAM
BzTree [16]	– persistent multi-word CaS operation (PMwCAS)	– PMwCAS + spinning – optimistic protocol (insert)	– near-instantaneous recovery	NVM-only
BDCC+ NVM [28]	– undo logging through PMDK	N/A	– clustering instead of additional indexes – unsorted blocks – instant recovery	NVM + SSD

ment, however, many of the presented approaches considered single-level systems instead of thinking about hybrid structures to overcome the limits of each technology. Nevertheless, we expect that in the future there will be more hybrid data structures that dynamically place data on DRAM, NVM, and disk, given their properties and application requirements. The actual placement also depends on the cost of NVM. If it is much cheaper than DRAM, one would limit the amount of data in DRAM and store the majority on NVM. However, if NVM becomes comparatively expensive, one will rely more on DRAM and SSD.

In conclusion, first efforts to port existing structures to NVM unveiled unprecedented programming challenges, making the gains vs. development tradeoff non-trivial. Hence, it is important to become aware of the new properties of the technology and to exploit them efficiently. Particularly in the areas of failure atomicity, concurrency, and data placement, new challenges arose which still need to be resolved in the future. In order to support a transactional system in principle, there seems to be no alternative to certain primitives, such as logging, shadowing, or locking. In the end, the actual role and usage of NVM still remains open and can ultimately only be decided when real hardware is available.

Acknowledgements This work was partially funded by the German Research Foundation (DFG) in the context of the projects "Transactional Stream Processing on Non-Volatile Memory" (SA 782/28) and "Interactive Big Data Exploration on Modern Hardware" (KE401/22) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037). Additionally, Alexander van Renen is supported by Fujitsu Laboratories.

References

1. JEDEC (2015) JEDEC announces support for NVDIMM hybrid memory modules. https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules. Accessed 25 May 2018
2. SNIA (2017) NVM programming model V1.2. Tech. rep. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf. Accessed 28 June 2018
3. Micron (2018) 3D XPoint technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology. Accessed 28 May 2018
4. Kernel (2018) Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt. Accessed 28 May 2018
5. Kernel (2018) Ext4 file system. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. Accessed 25 May 2018
6. Intel (2018) Intel 64 and IA-32 architectures software developer manuals. http://software.intel.com/en-us/intel-isa-extensions. Accessed 28 May 2018
7. Mott N (2018) Intel announces Optane DC persistent memory is sampling now, with broad availability in 2019. https://www.tomshardware.com/news/intel-announces-optane-dc-persistent-memory,37145.html. Accessed 27 June 2018
8. Kernel (2018) Linux NVDIMM library. https://www.kernel.org/doc/Documentation/nvdimm/nvdimm.txt. Accessed 25 May 2018
9. Intel (2018) Persistent Memory Development Kit. http://pmem.io/pmdk/. accessed: May 30, 2018
10. Agrawal R, Jagadish HV (1989) Recovery algorithms for database machines with nonvolatile main memory. IWDM '89, pp 269–285

11. Andrei M, Lemke C et al (2017) SAP HANA adoption of non-volatile memory. Proceedings VLDB Endowment 10(12):1754–1765
12. Appuswamy R, Borovica R et al (2017) The five minute rule thirty years later and its impact on the storage hierarchy. ADMS @ VLDB '17.
13. Arulraj J, Pavlo A (2017) How to build a non-volatile memory database management system. SIGMOD '17, pp 1753–1758
14. Arulraj J, Pavlo A, Dulloor S (2015) Let's talk about storage & recovery methods for non-volatile memory database systems. SIGMOD '15, pp 707–722
15. Arulraj J, Perron M, Pavlo A (2016) Write-behind logging. Proceedings VLDB Endowment 10(4):337–348
16. Arulraj J, Levandoski JJ et al (2018) BzTree: a high-performance latch-free range index for non-volatile memory. Proceedings VLDB Endowment 11(5):553–565
17. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Inform 1:173–189
18. Bhandari K, Chakrabarti DR, Boehm H (2016) Makalu: fast recoverable allocation of non-volatile memory, pp 677–694
19. Chatzistergiou A, Cintra M, Viglas SD (2015) REWIND: recovery write-ahead system for in-memory non-volatile data-structures. Proceedings VLDB Endowment 8(5):497–508
20. Chen S, Jin Q (2015) Persistent B+-trees in non- volatile main memory. Proceedings VLDB Endowment 8(7):786–797
21. Chua L (1971) Memristor – the missing circuit element. IEEE Trans Circuit Theory 18(5):507–519
22. Comer D (1979) Ubiquitous b-tree. ACM Comput Surv 11(2): 121–137
23. Condit J, Nightingale EB et al (2009) Better I/O through byte-addressable, persistent memory. ACM, SOSP '09, pp 133–146
24. DeBrabant J, Pavlo A et al (2013) Anti-caching: a new approach to database management system architecture. Proceedings VLDB Endowment 6(14):1942–1953
25. DeBrabant J, Arulraj J et al (2014) A prolegomenon on OLTP database systems for non-volatile memory. ADMS @ VLDB '14, pp 57–63
26. Dulloor SR, Kumar S et al (2014) System software for persistent memory. ACM, EuroSys '14, pp 15:1–15:15
27. Eisenman A, Gardner D et al (2018) Reducing DRAM footprint with NVM in Facebook. ACM, EuroSys '18, pp 42:1–42:13
28. Götze P, Baumann S, Sattler KU (2018) An NVM-aware storage layout for analytical workloads. HardBD & Active @ ICDE '18.
29. Govoreanu B, Kar G et al (2011) 10×10nm2 Hf/HfOx crossbar resistive RAM with excellent performance, reliability and low-energy operation. IEEE, IEDM, pp 31.36.31–31.36.34
30. Graefe G, Guy W, Sauer C (2016) Instant recovery with write-ahead logging: page repair, system restart, media restore, and system failover, 2nd edn. Synthesis lectures on data management. Morgan & Claypool, San Rafael
31. Gray J, Putzolu GR (1987) The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. SIGMOD '87, pp 395–398
32. Harizopoulos S, Abadi DJ et al (2008) OLTP through the looking glass, and what we found there. SIGMOD '08, pp 981–992
33. Hosomi M, Yamagishi H et al (2005) A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM. IEEE, IEDM, pp 459–462
34. Hu J, Xue CJ et al (2011) Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. IEEE, DATE, pp 1–6
35. Huang J, Schwan K, Qureshi MK (2014) NVRAM- aware logging in transaction systems. Proceedings VLDB Endowment 8(4):389–400
36. Kimura H (2015) FOEDUS: OLTP engine for a thousand cores and NVRAM. In: SIGMOD, pp 691–706
37. Klima T (2016) Using non-volatile memory (NVDIMM-N) as byte-addressable storage in windows server 2016. https://channel9.msdn.com/events/build/2016/p470. Accessed 25 May 2018
38. Lee BC, Zhou P et al (2010) Phase-change technology and the future of main memory. IEEE Micro 30(1):131–141
39. Lee SK, Lim KH et al (2017) WORT: write optimal radix tree for persistent memory storage systems. FAST '17, pp 257–270
40. Leis V, Kemper A, Neumann T (2013) The adaptive radix tree: aRTful indexing for main-memory databases. ICDE '13, pp 38–49
41. Leis V, Haubenschild M et al (2018) Leanstore: in-memory data management beyond main memory. ICDE '18.
42. Lersch L, Oukid I et al (2017) Rethinking DRAM caching for LSMs in an NVRAM environment. ADBIS '17, pp 326–340
43. Li J, Pavlo A, Dong S (2017) NVMRocks: RocksDB on non-volatile memory systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/. Accessed 27 May 2018
44. Liu Q, Varman P (2017) Ouroboros wear-leveling: a two-level hierarchical wear-leveling model for NVRAM. MSST '17.
45. Mittal S, Vetter JS (2016) A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans Parallel Distrib Syst 27(5):1537–1550
46. Mohan C, Haderle DJ et al (1992) ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database Syst 17(1):94–162
47. Moraru I, Andersen DG et al (2013) Consistent, durable, and safe memory management for byte-addressable non volatile main memory, pp 1:1–1:17
48. Ou J, Shu J, Lu Y (2016) A high performance file system for non-volatile main memory. EuroSys '16, ACM, pp 12:1–12:16
49. Oukid I, Lehner W (2017) Data structure engineering for byte-addressable non-volatile memory. ACM, SIGMOD '17, pp 1759–1764
50. Oukid I, Booss D et al (2014) SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. DaMoN '14, pp 8:1–8:7
51. Oukid I, Lehner W et al (2015) Instant recovery for main memory databases. CIDR '15.
52. Oukid I, Lasperas J et al (2016) FPTree: a hybrid SCM- DRAM persistent and concurrent B-tree for storage class memory. SIGMOD '16, pp 371–386
53. Oukid I, Booss D et al (2017) Memory management techniques for large-scale persistent-main-memory systems. Proceedings VLDB Endowment 10(11):1166–1177
54. Oukid I, Kettler R, Willhalm T (2017) Storage class memory and databases: opportunities and challenges. it Inf Technol 59(3): 109–115
55. O'Neil P, Cheng E et al (1996) The log-structured merge-tree (LSM-tree). Acta Inform 33(4):351–385
56. Pavlo A, Angulo G et al (2017) Self-driving database management systems. CIDR '17.
57. Qureshi MK, Karidis J et al (2009) Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. ACM, MICRO 42, pp 14–23
58. van Renen A, Leis V et al (2018) Managing non-volatile memory in database systems. SIGMOD '18, pp 1541–1555
59. Schwalb D, Berning T et al (2015) nvm_malloc: memory allocation for NVRAM. ADMS @ VLDB '15, pp 61–72
60. Strukov DB, Snider GS et al (2008) The missing memristor found. Nature 453(7191):80–83
61. Venkataraman S, Tolia N et al (2011) Consistent and durable data structures for non-volatile byte- addressable memory. FAST '11, pp 61–75
62. Viglas SD (2015) Data management in non-volatile memory. SIGMOD '15, pp 1707–1711

63. Wang T, Johnson R (2014) Scalable logging through emerging non-volatile memory. Proceedings VLDB Endowment 7(10):865–876
64. Wang T, Levandoski J, Larson PA (2017) Easy lock-free indexing in non-volatile memory. Technical report, Microsoft research
65. Wu X, Reddy A (2011) SCMFS: a file system for storage class memory. ACM, SC '11, pp 39:1–39:11
66. Xia F, Jiang D et al (2017) HiKV: a hybrid index key-value store for DRAM-NVM memory systems. USENIX ATC '17, pp 349–362
67. Xu J, Swanson S (2016) NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. FAST '16, pp 323–338
68. Yang J, Wei Q et al (2015) NV-tree: reducing consistency cost for NVM-based single level systems. FAST '15, pp 167–181