Glombiewski, Nikolaus; Götze, Philipp; Körber, Michael; Morgen, Andreas;
Seeger, Bernhard:

Designing an event store for a modern three-layer storage hierarchy

Original published in:	Datenbank-Spektrum. - Berlin : Springer. - 20 (2020), 3, p. 211-222.
Original published:	2020-10-16
ISSN:	1610-1995
DOI:	10.1007/s13222-020-00356-6
[*Visited:*	2020-11-09]

Datenbank Spektrum (2020) 20:211–222
https://doi.org/10.1007/s13222-020-00356-6

SCHWERPUNKTBEITRAG

Designing an Event Store for a Modern Three-layer Storage Hierarchy

Nikolaus Glombiewski[1] · Philipp Götze[2] · Michael Körber[1] · Andreas Morgen[1] · Bernhard Seeger[1]

Received: 7 June 2020 / Accepted: 5 September 2020 / Published online: 16 October 2020

Abstract
Event stores face the difficult challenge of continuously ingesting massive temporal data streams while satisfying demanding query and recovery requirements. Many of today's systems deal with multiple hardware-based trade-offs. For instance, long-term storage solutions balance keeping data in cheap secondary media (SSDs, HDDs) and performance-oriented main-memory caches. As an alternative, in-memory systems focus on performance, while sacrificing monetary costs, and, to some degree, recovery guarantees. The advent of persistent memory (PMem) led to a multitude of novel research proposals aiming to alleviate those trade-offs in various fields. So far, however, there is no proposal for a PMem-powered specialized event store.
Based on ChronicleDB, we will present several complementary approaches for a three-layer architecture featuring main memory, PMem, and secondary storage. We enhance some of ChronicleDB's components with PMem for better insertion and query performance as well as better recovery guarantees. At the same time, the three-layer architecture aims to keep the overall dollar cost of a system low. The limitations and opportunities of a PMem-enhanced event store serve as important groundwork for comprehensive system design exploiting a modern storage hierarchy.

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

1 Introduction and Motivation

Data stream applications are at the forefront of many of today's challenging processing use cases. In a broad sense, every source that continuously produces data is handled by streaming technology. Thus, use cases range from traditional scenarios such as infrastructure monitoring and log analysis to cutting edge technology like autonomous cars and the Internet of Things (IoT). An important sub-class of data streams that covers all four of those use cases is known as event streams.

In contrast to generic data items, an event represents an occurrence at a specific point in time, such as a measurement by a temperature sensor. Low latency analysis of event streams can be achieved through a plethora of powerful stream processing systems such as Apache Flink[1]. Besides, long-term storage for ad-hoc queries and compute-intensive analysis requires an efficient *event store*. Even though many modern key-value-based NoSQL systems can be used to meet the ever-increasing data ingestion demands, time series databases and specialized event stores can store and replay millions of temporal records per second with lower hardware requirements. However, as the amount of data is expected to grow continuously[2], pure event ingestion and queries on ever-expanding data sets will remain an important challenge for years to come. Thus, it is crucial to not only look at novel software solutions but consider those solutions in the presence of current and future hardware advancements that will help to alleviate some challenges.

✉ Nikolaus Glombiewski
glombien@informatik.uni-marburg.de

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

Michael Körber
koerberm@informatik.uni-marburg.de

Andreas Morgen
morgen@informatik.uni-marburg.de

Bernhard Seeger
seeger@informatik.uni-marburg.de

[1] University of Marburg, Marburg, Germany

[2] Technische Universität Ilmenau, Ilmenau, Germany

[1] https://flink.apache.org/.

[2] https://www.seagate.com/our-story/data-age-2025/.

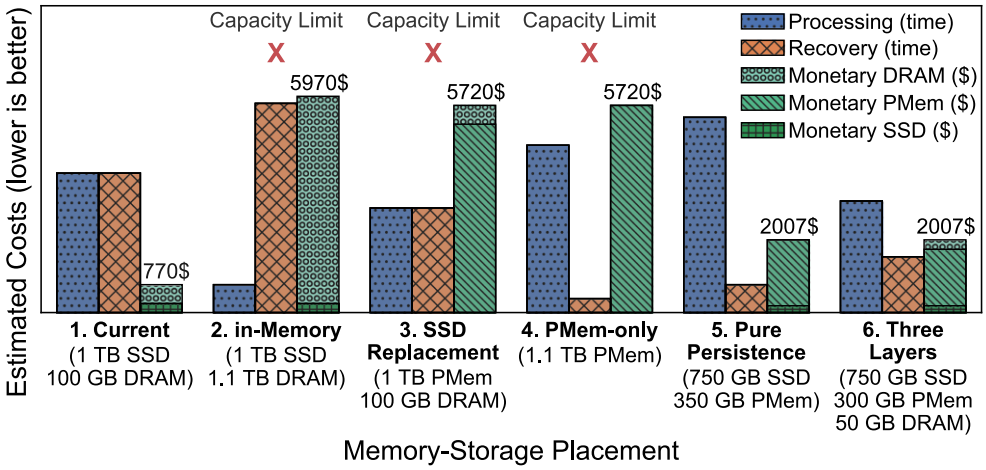

Fig. 1 Various costs depending on the data placement

In recent years, one of the most pioneering hardware advancements is the introduction of persistent memory (PMem). Unlike traditional secondary storage (SSDs, HDDs), PMem is byte-addressable and exhibits faster access times. Similar to the fairly modern approach of an in-memory database system – focusing on performance while sacrificing monetary and recovery costs – a straight forward solution to incorporate PMem into a current event store would be to use it as a direct replacement for SSDs or even DRAM. However, given the massive amount of continuously incoming data, these strategies are not universally advised and cost-effective. As an example, consider the use case and corresponding costs of several placement strategies in Fig. 1. It is based on an actual ratio taken from ChronicleDB consisting of 1 TB primary data and 100 GB reconstructable secondary data. The processing and recovery costs are estimates, while the monetary costs base on [9]. Given the capacity limits of our system (see Table 1), the before mentioned options (2–4) are excluded either way. The placement in our current system (1) is the most efficient option in monetary terms. Assuming that the following generations of PMem will become more affordable – as the history of flash and DRAM prices indicates [9] – a pure persistent system comprising PMem and flash (5) could become the most economical and ecological solution. Using the same argument, a three-layer system (6) is however the most promising solution providing the best balance across all metrics.

Hence, instead of a direct replacement of traditional storage or main memory, we will examine PMem as a third layer in the storage hierarchy of an event store. The discussion will be framed as a case study for ChronicleDB [33, 34], a special-purpose database system for event streams. Even though we use ChronicleDB as an example, the lessons learned can be applied to both, more specialized (e.g., temporal indexing/storage design) as well as less specialized systems (e.g., ingestion/recovery). Our contributions are as follows:

- From the literature (§3), we identify those solutions that already utilize multiple storage layers and adapt existing insights for an event store.
- We will introduce PMem as a third layer into ChronicleDB's layout. We show various opportunities and limitations for using PMem in the respective components of an event store – such as ingestion, storage design, recovery, and index maintenance (§4).
- With the help of micro-benchmarks, we substantiate that our approaches can improve the general query performance, the recovery speed and guarantees, as well as flatten fluctuations in ingestion speed (§5).
- We summarize our findings and formulate future research directions (§6).

Table 1 Experimental setup

PROCESSOR	2 Intel® Xeon® Gold 5215, 10 cores / 20 threads each, max. 3.4 GHz
CACHES	32 KB L1d/L1i, 1024 KB L2, 13.75 MB LLC
MEMORY	2 × 6×32 GB DDR4 (2666 MT/s), 2 × 6×128 GB Intel® Optane™ DCPMM
STORAGE	1 TB Intel® SSD DC P4501 Series
OS & COMPILER	CentOS 7.8, Linux 5.6.11 kernel, OpenJDK 14.0.1

2 Background

In this section, we briefly describe our used memory and storage technologies focusing on PMem and which conclusions can already be drawn from their properties. Afterwards, we discuss design aspects of ChronicleDB, needed to follow the subsequent sections.

2.1 Persistent Memory

In the context of this paper, PMem stands primarily for Intel's Optane DC Persistent Memory Modules (DCPMM). The basic properties are byte-addressability, near-DRAM latency, and persistence. In particular with DCPMMs, some additional constraints must be taken into account to make the most efficient use of PMem (cf. [38]). First of all, the transfer size from the CPU to PMem, and also DRAM, is 64 bytes (a cache line). However, internally the hardware operates on 256-byte blocks, where a write-combining buffer is used to reduce write amplification. Consequently, data structures (i.e., their nodes/chunks) should be a multiple of 256 bytes in size and 64-byte aligned. Apart from the slightly worse latency, the bandwidth of PMem is also more limited compared to DRAM. Also, the read-write performance is asymmetric. Therefore, accesses to the device should be reduced – especially writes – to overcome these limitations. Another observation [8] is that despite the byte-addressability, there is still a relatively large discrepancy between sequential and random accesses.

To confirm these properties and to identify peculiarities of our system (see Table 1), we remeasured some performance indicators using Intel's *Memory Latency Checker* [12] for DRAM and PMem as well as *Flexible I/O Tester* [3] for the SSD/flash drive. The results are presented in Table 2. For the SSD, we worked on a 100 GB file and report the 99th percentile latency. We have experimented with varying block sizes as well as degrees of parallelism (i.e., threads and I/O depth) and report the best throughput. The measurements are largely in accordance with the specifications and other reports [9, 11, 18, 32, 38]. One anomaly, however, is the sequential write bandwidth for DRAM, which should be closer to the read speed. Nevertheless, compared to DRAM, PMem shows about a 2–4× higher latency and lower peak bandwidth. While the difference between sequential and random access for DRAM is hardly noticeable, it has a greater impact on PMem and even more on SSDs. The read-write asymmetry for PMem and SSDs is also clearly visible.

Table 2 Measured performance of memory/storage technologies within our server (see Table 1)

	DRAM	DCPMM	TLC Flash
Idle seq. read latency	81 ns	174 ns	14 μs
Idle rand. read latency	88 ns	325 ns	206 μs
Max. read bandwidth	85 GB/s	32 GB/s	3 GB/s
Max. write bandwidth	46 GB/s	13 GB/s	0.6 GB/s
Random reads	931 M/s	45 M/s	299 K/s
Random writes	703 M/s	30 M/s	61 K/s

2.2 The Event Store ChronicleDB

ChronicleDB is a database system specialized for storing and querying multi-variate event stream data. Event application scenarios usually involve a high amount of continuously arriving data in a short period of time. An example would be data generated by sensor streams. Historical analysis in these applications has to consider the huge amount of raw data in an efficient manner. To support those scenarios, ChronicleDB is designed around three requirements:

(*R*1) Ingestion of high input rate streams.
(*R*2) Fast stream replay and time travel operations.
(*R*3) Fast processing of point, range, and aggregation queries on secondary attributes.

While R1 is required to avoid load-shedding in cases of high volume input streams, R2 and R3 allow excellent query response times in a variety of use cases like post-mortem analysis of event stream queries, continuous batch processing for dashboard applications as well as to some degree serving traditional OLAP demands. The following will provide an overview of ChronicleDB's four core components and briefly describe their interaction in the system. Subsequent chapters will review relevant implementation details in the context of opportunities and limitations for utilizing persistent memory within the respective components.

Primary Index. ChronicleDB's target data model are multi-variate event streams, i.e., data consisting of multiple measurements per timestamps, with a fixed schema. Thus, a single primary index in ChronicleDB stores records called *events* of the form $e = (a_1, ..., a_n, ts)$ where a_i is a value from attribute domain A_i and ts is the event's timestamp from a temporal domain T. A potential infinite stream $E = \langle e_1, e_2, ... \rangle$ is ingested into the *Temporal Aggregated B$^+$-Tree* (TAB$^+$-Tree) index. Note that within E timestamps do not have to be unique. The overall index layout is presented in Fig. 2.

At its core, the index is an augmented B$^+$-Tree with T as its key domain and doubly-linked nodes on every level. This index design is important to fulfill (R2). To support fast ingestions (R1), ChronicleDB primarily adopts an append-only model, where the data log is also the database. This is reflected in the index design. As the default behavior, insertions are treated as a continuous bulk loading operation in a traditional B$^+$-Tree index. Under the assumption that the event stream E is in temporal order, a new event can be appended to the leaf node containing the most recent data. For the remainder of this work, this leaf node and its parents are referred to as the *right flank* of the TAB$^+$-Tree. To speed up ingestion in temporal order, the right flank is kept in DRAM at all times. Thus, without an additional log [33], the last leaf can be lost in a crash.

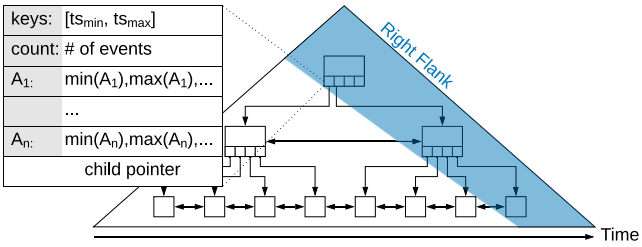

Fig. 2 Primary Index (TAB$^+$-Tree) Layout

Secondary Index. Besides the primary index, ChronicleDB allows adaptive and ad-hoc creation of two types of secondary indexes, which are referred to as *heavyweight* and *lightweight* indexes. Heavyweight indexes are traditional secondary index structures such as LSM [25] or COLA [4], built for one or more attributes of the stream. Leaf pages of heavyweight indexes refer to primary index pages with a record offset. Lightweight indexes are an adaptation of small materialized aggregates (SMAs) [24], which can be arbitrary aggregate functions on the event's data domain. In contrast to [24], those aggregates are stored within the primary index nodes of the TAB$^+$-Tree. Each child reference is associated with aggregated information of their respective nodes. In Fig. 2, an index entry consists of an overall event count aggregate and the minimum and maximum values for each attribute. The interleaved aggregates tremendously boost the query performance (R3) for simple queries (e.g., filter) and complex queries (e.g., pattern matching) alike. For instance, by intersecting a query range on a secondary attribute with its computed min/max range, large portions of the stream can be excluded from processing. Furthermore, temporal aggregation queries on those materialized aggregates can be answered in logarithmic time. Since lightweight index information is stored within the primary index, it is automatically persisted and does not require additional random I/O while querying a temporal region.

Storage Layout. To reduce the storage cost of massive amounts of data, compression is a necessity. Especially continuous sensor values, such as temperature and humidity measurements, feature a lot of similar values that can be easily compressed to reduce space requirements. ChronicleDB compresses each TAB$^+$-node with a configurable compression algorithm. This naturally results in various node sizes that cannot be mapped to fixed-size block addresses. Hence, some sort of address translation layer that maps logical node IDs to physical storage locations is required. This information needs to be stored on disk to avoid a full relation scan during recovery from a system failure. We refrained from storing the address translation information in a separate location but designed a storage layout that interleaves address translation information with the actual data, in order to enforce sequential access patterns on secondary storage[3]. The layout operates on data blocks of fixed size, where the size of such a block is a multiple of the *uncompressed* size of a TAB$^+$-node. For instance, in our previous work [34], we used 32 KiB blocks for uncompressed 8 KiB TAB$^+$-nodes. Each of these blocks either contains compressed TAB$^+$-nodes or the translation information for a set of TAB$^+$-nodes. Due to our goal of storing translation information interleaved with the data, the translation layer is organized as a tree structure, called *Address Translation Tree* (ATT). Each ATT leaf covers the same number of translations for a contiguous range of TAB$^+$-node IDs. Similarly, inner nodes cover a fixed range of node IDs and point to the corresponding address translation block in the next tree level. The most recent translations are kept in main memory and are written to secondary storage once the block becomes full (analogous to the *right flank* of the TAB$^+$-tree). Hence, this in-memory fraction needs to be recovered in case of a system failure.

Out-Of-Order Data. By itself, the TAB$^+$-Tree degenerates to the performance of a traditional B$^+$-Tree when data does not arrive in a temporal order. This type of data is referred to as *out-of-order (OOO) data*. Even though the system allows to switch the time domain from application time to system time in extreme cases of OOO data, most streaming data sets feature at least somewhat of a temporal order within data. For those scenarios, ChronicleDB indexes data on application time while deploying a three-step strategy as depicted in Fig. 3 to offset performance degradation.

First, OOO data is put into a dedicated OOO queue. This preserves the append-only nature. Second, nodes in the TAB$^+$-Tree can leave spare space to absorb OOO insertions without cascading node splits. For cheap spinning disks used to store large amounts of data, this has the additional benefit of preserving a sequential node layout. Third, whenever the OOO queue reaches a certain size, it is merged

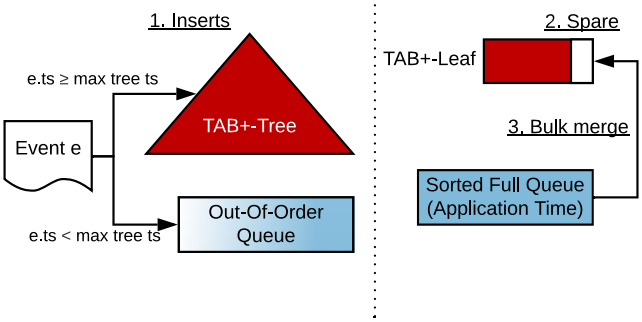

Fig. 3 Original OOO Handling

[3] Even though this aspect is less critical on SSDs than on HDDs, the performance of sequential access is still superior compared to random access [35].

into the primary index. This stabilizes query performance by adjusting lightweight indexes to more accurate values and merging logical temporal regions into close physical regions.

3 Related Work

Several of the concepts presented in this paper base on the insights of previous work. Particularly for event stores or time series databases – such as *Tidalrace* [14], *DataGarage* [23], *tsdb* [5], *Gorilla* [29], *InfluxDB* [10], etc. – there are, to our knowledge, no considerations for PMem yet. Therefore, here we elaborate on two existing directions for data management using multiple memory and storage technologies including PMem. The first direction comprises individual data structures that make use of at least two memory layers. The second part deals with more complex systems or storage engines that try to exploit the entire memory hierarchy.

3.1 Multi-Level Data Structures

The focus of most previous work for PMem-based data structures was on the B$^+$-Tree. Among these, however, only the *FPTree* [28] adopts a DRAM/PMem hybrid approach. They propose a persistent linked list of leaf nodes while keeping the inner nodes in DRAM, which are rebuilt upon recovery. Evaluations on real hardware have already shown that this division is definitely practical for hiding PMem's higher latency and achieving DRAM-like performance [18]. With the *LB$^+$-Tree* [22] the authors refine the concept for Intel's DCPMMs by utilizing multi-256-byte nodes and limiting the number of PMem line writes. In [13] it is shown that the *selective persistence* concept of the FPTree is similarly applicable to other data structures. The use case here is a persistent trie-like structure enhanced with various DRAM caching strategies. Instead of splitting up a single data structure, the authors of *HiKV* [37] opt for the use of two separate structures. Here, a partitioned hash index is kept in PMem for efficient single key-value operations while an additional volatile B$^+$-Tree enables faster scans. What is further proposed are multi-tier general-purpose buffer pools, which are supposed to exploit the respective properties of the diverse memory and storage technologies [2, 19, 26, 30, 31].

3.2 PMem-aware Storage Engines

Besides the individual data structures, there have been more extensive redesigns of storage engines including PMem. One of the first proposals considering a modern OLTP engine on future hardware was *FOEDUS* [16]. PMem is used here to store log entries and immutable snapshots. Another relatively early hybrid storage engine is *SOFORT* [27], designed to support both transactional and analytical workloads. It is organized column-wise. All primary data is directly stored and processed on PMem. Only secondary data such as indexes (e.g., dictionaries) are stored in DRAM to ensure near-instantaneous recovery. The authors of [7] also target analytical workloads but use a multi-dimensional clustering approach instead of secondary indexes to reduce writes to PMem. They distinguish between hot blocks which are placed in PMem and cold blocks moved to flash. Since the primary index can also be volatile, this results in a three-layer storage engine. The *SAP HANA* database has also been extended to support PMem [1]. Similar to SOFORT, the recent data resides in write-optimized deltas in DRAM which are periodically merged into the read-optimized main store in PMem.

Another representative from the industry is Facebook, who propose the key-value store *MyNVM* [6] to reduce the DRAM footprint and consequently the total cost of ownership. In this work, PMem is used as a second-level block cache whereas the database and logs stay on flash. Also based on *RocksDB*, in *NVMRocks* [21] the LSM-Tree placement is revised. The authors propose two possible adaptions. At first, they simply replace flash with PMem and remove unnecessary components. The second approach also considers moving *MemTables* to PMem to avoid logging. This is further enhanced with a multi-tier read cache. *NoveLSM* [15] is another approach extending the LSM-Tree design. Instead of moving all *MemTables* from DRAM to PMem, the authors propose to have a larger additional persistent replication. As soon as the volatile part is full and compaction to an *SSTable* starts, the PMem replication is used for concurrent queries. A recent proposal for a modern key-value store is *RStore* [20]. It can be summarized as log-structured storage plus index. The actual data resides in PMem as append-only blocks whereas the index is volatile and rebuilt during recovery. Furthermore, they use a small recovery log per partition and some other auxiliary structures in PMem.

With few exceptions, hardly any approach exploits all three layers (DRAM, PMem, and flash). Furthermore, to our knowledge, PMem has never been considered in the context of event stores and streaming. These are the challenges we address with this paper.

4 Concepts and Approaches

In the following, we discuss the opportunities and limitations of using persistent memory in ChronicleDB. We will focus on three components of the system: (i) the TAB$^+$-Tree, (ii) the physical storage layout, and (iii) handling of out-of-order data.

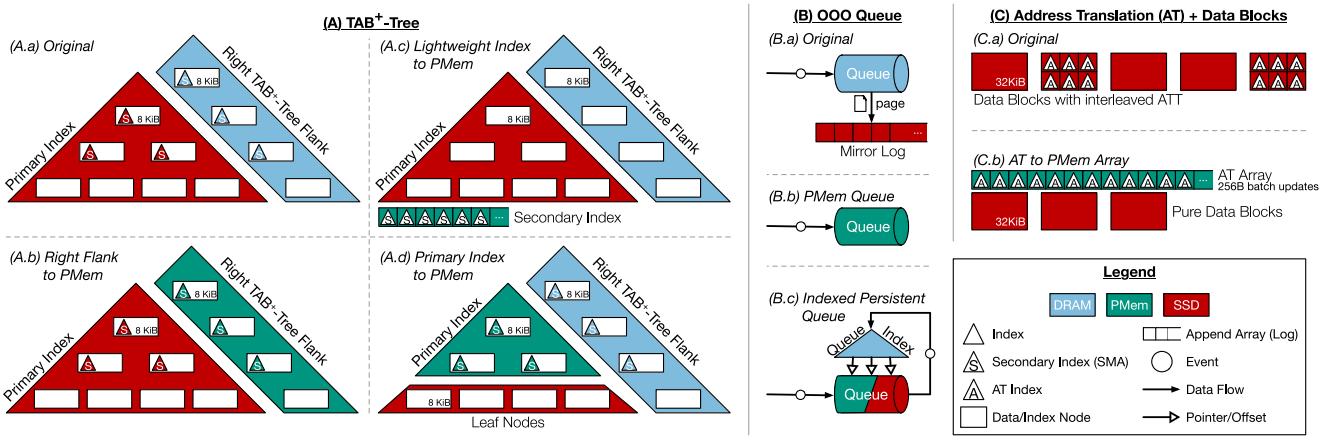

Fig. 4 Overview of approaches applied to ChronicleDB

4.1 TAB$^+$-Tree

ChronicleDB's main goal is to efficiently store and index high volume event streams. It is designed for a two-layer storage architecture consisting of main memory and secondary storage (SSDs/HDDs). Hence, the architecture of the primary index is a result of seeking a balance between durability and insert/query performance under this setting (Fig. 4 A.a). By introducing PMem as a connection between main memory and secondary storage, the primary index design needs to be reconsidered to take advantage of byte-addressable persistent storage. In the following, we present three approaches for introducing PMem into ChronicleDB's primary index. First, we move the *right flank* from DRAM to PMem which fully eliminates the need for recovering the flank after a system failure (Fig. 4 A.b). Then, we separate lightweight index information from the inner index nodes by storing the computed aggregates on PMem (Fig. 4 A.c). Finally, we move all index nodes from flash to persistent memory (Fig. 4 A.d).

Right Flank. Keeping the *right flank* of the TAB$^+$-Tree in DRAM is a crucial aspect of ChronicleDB's insert performance. However, if the most recent data is lost in case of a system failure, the flank needs to be recovered. By managing the *right flank* on PMem, we can avoid data loss and improve recovery time, since the most recent state of the flank is available at any time – in particular after a restart of the system. In the following, we discuss the changes required to manage the *right flank* on persistent memory.

The in-memory layout of pages is byte-oriented so that they can be written to external storage without further serialization. Every incoming event is directly converted into the corresponding binary representation and attached to the current leaf page. If there is insufficient space, the current leaf is written to secondary storage, a new empty leaf page is allocated and the event is appended to that new page. The same mechanisms are applied to index nodes. Hence, the major challenge when moving the right flank to PMem is to ensure that the binary representation of the corresponding pages is in a consistent state at any time – especially after a system failure. We ensure this by forcing a write-back of modified memory regions periodically as follows.

Each page consists of a header and a data region. The header holds the sibling link information and the number of stored events, while the data region contains event data. When reusing a page after it was written to secondary storage it is sufficient to reset and flush only the header region to PMem since the event count determines the valid data region. However, to guarantee that a newly appended event is persisted, both the header and the affected area of the data region need to be flushed to PMem. Since events are small in general, this leads to performance degradation when executed after every append operation due to write amplification and CPU cache misses. To tackle this problem, we introduce batch flushing which persists appended events in configurable batches (i.e., trades durability for performance). We discuss the performance/durability trade-off in Sect. 5.2.

Aggregates. As detailed in Sect. 2.2, every index (inner) node of the TAB$^+$-Tree holds for each child-reference a configurable set of aggregates. Those aggregates summarize the data stored in the corresponding sub-tree and are utilized to boost the performance filter and temporal aggregation queries. However, the more aggregates are stored the smaller the fan-out of index nodes. For instance, with a page size of 8 KiB the fan-out decreases from 459 to 43 when lightweight indexing (via aggregates sum, min, max) is applied to six 64-bit floating-point values. To alleviate this drawback of lightweight indexing, we moved the index information (i.e., the aggregates) to persistent memory.

Aggregates are computed in a bottom-up fashion during the insertion of events. Whenever a leaf node becomes full and is written to secondary storage, the aggregates for all events within this leaf are computed in batch and propa-

gated to the parent level. The parent node attaches this information to the regular index entry (key, child-pointer). For an index node, the handling is similar: If it becomes full, the aggregates of its index entries are merged and propagated up the tree. Note that this requires aggregate functions to be decomposable as in [36]. By keeping aggregates within the index nodes, storing and accessing them incurs only very little cost. To be competitive, the overhead of managing the aggregates on PMem must be kept as small as possible. Thus, we modeled the aggregate store on persistent memory as a flat array. Each slot of the array holds the aggregated values of one node of the primary index. Because the TAB$^+$-Tree assigns consecutive node IDs starting at 0, accessing the aggregates of a page with ID i translates to a lookup of slot i in the PMem array.

The increased fan-out of inner nodes reduces the tree height and, thus, is beneficial for queries on the time-domain of events. However, compared to standard ChronicleDB, query processing now incurs access to secondary storage (index nodes) and persistent memory (aggregates). We will discuss this performance trade-off in detail in Sect. 5.

Index Nodes. To tackle the double-access problem introduced by outsourcing aggregates to persistent memory, our next approach stores only leaf nodes on secondary storage and manages the index nodes of the tree on persistent memory. As a result, all index navigation and aggregate access are handled without accessing secondary storage. This approach requires only minimal modification of ChronicleDB's insert mechanisms. Similar to aggregates, the required persistent memory space is managed as a flat array. The capacity of one array slot matches the configured page size (typically 4 KiB or 8 KiB), and each slot contains one index node. Due to two independent storage locations, we also require two ID sequences. One for leaf nodes and one for index nodes. Hence, to uniquely identify a page an additional bit of information is required to determine whether the given ID refers to a leaf or an index page. The benefit of using two independent ID sequences is that we achieve a sequential write pattern for both, secondary storage and persistent memory, which in both cases improves write-throughput (cf. Sect. 2.1).

Compared to the aggregate-only solution, this approach requires additional space on PMem (20 bytes page header, 16 bytes for key and child ID per entry). However, it fully excludes secondary storage when accessing inner nodes, and thus achieves better query performance compared to the aggregate-only solution.

4.2 Storage Layout

Utilizing interleaved tree-based address translation in ChronicleDB's storage layer leads to a good balance between insert, query, and recovery performance. However, for access to TAB$^+$-Nodes whose translation does not reside in main memory, the performance of address lookups is far from optimal. Consider an ATT of height three with 32 KiB block sizes. Assuming the root to be in main memory still two random reads with 32 KiB each are required to resolve the node's address.

By moving the address translation to byte-addressable persistent memory, updating address translations does not interfere with data-writes anymore. Thus, we switch from the tree-based approach to a flat lookup-table that is updated synchronously as pages are written. Similar to managing the aggregates of the TAB$^+$-Tree on persistent memory, we manage the translation as a flat array such that slot i of the array contains translation information for the node with ID i (Fig. 4 C.b). Moreover, by flushing the data after each update, recovery could be fully avoided. However, the size of translation information for a single node is only 8 bytes. Hence, forcing each update immediately would lead to performance degradation due to write amplification. To resolve this issue, we batch updates to match the 256-byte write granularity before forcing them to be persisted. Since at most 32 translations need to be restored, the recovery time is reduced drastically compared to our tree-based approach. Besides recovery and lookup performance, inserts also benefit from an address translation on persistent memory. While the majority of inserts resulted in writing 8 bytes to main memory, some inserts suffered from writing translation blocks to secondary storage. For a tree of height h, an insert can trigger h writes of 32 KiB to secondary storage in the worst case. With persistent memory, worst-case insert performance improves due to continuous flushes of small batches.

4.3 Out-of-Order Data

The current OOO strategy tries to keep a balance between query performance, recovery guarantees, and insertion performance. However, there is an inherent trade-off for all three factors. For good query performance, the queue is kept in main memory such that any query has fast access to data. To avoid data loss, large queues spanning multiple page sizes, are backed by a traditional secondary storage medium. Full pages are written out to the storage medium in an efficient append-only manner. This solution favors query performance, but the dual maintenance of the queue drains main memory resources. At the same time, a full page of data can still be lost. Recovery alternatives, such as writing each record individually, would incur the same insert problem the queue is designed to resolve. The following will explore how an additional PMem layer can improve upon these deficiencies.

Persistent Memory Queues. The tension between query and recovery requirements can be attributed to the lack of persistence in main memory and inadequate access granularity in secondary storage. PMem characteristics inherently address both issues. At its core, our new approach redirects each incoming OOO event to a persistent memory queue. The queue is stored in system-time order, i.e., events are inserted in a fast append-only manner. For a batch of OOO events, this also utilizes faster sequential writes (cf. Table 2). Since OOO events are expected to be rare, each insertion is followed by a force command. Thus, the queue is fully persisted and no data is lost in case of a crash. This improves upon the original recovery guarantees, and consequently, there is no additional layer on secondary storage.

Indexing. Standard ChronicleDB keeps a main memory copy of the OOO queue for two reasons. First, it allows direct access to single elements of the queue. Without the main-memory copy, accessing events on inherently unordered data would fetch a page from secondary storage for each event in the worst case. Second, the main memory copy is ordered by application time. This enables efficient processing of queries on the time domain as well as fast bulk merging into the primary index. PMem is byte-addressable and, compared to flash storage, offers excellent random access performance. This automatically resolves the first reason for a copy. For efficient application time access, we replace the full queue copy with a lightweight in-memory application time index (see Fig. 4 B.c). The index refers to the byte-offset within persistent memory. This reduces the main memory footprint. Furthermore, random access for fetching events in application time order is expected to be less of a problem on PMem than on flash storage. In the future, this strategy can be expanded for more elaborate merge solutions, such as merging only certain ranges.

5 Micro-Benchmarks

In our experiments, we focus on micro-benchmarks on the approaches presented in the previous section. All of them are integrated into ChronicleDB and evaluated on a server equipped with real PMem hardware. Our aim is not to compare with other event stores, but to show opportunities in such a three-layer system. Thus, we reveal which approaches prove to be useful in practice and where fine-tuning is still necessary.

5.1 Experimental Setup

The used machine is a dual-socket Intel Xeon Gold 5215 server as outlined in Table 1. Six DCPMMs are grouped into a single region and namespace per socket. They are accessed via an *ext4* file system and mounted with the `DAX` option. The operating mode of the modules is set to `App Direct`.

ChronicleDB is entirely written in Java. We used the JDK 14 extensions[4] for accessing persistent memory via Java's `ByteBuffer` interface. We configured a tree-node size of 8 KiB and used the LZ4 algorithm to compress the nodes. Furthermore, each node maintains a spare space of 10% for absorbing OOO events. The block-size of the storage layer was set to 32 KiB.

To avoid performance degradation due to loading the input data from disk, we generated events in main-memory and fed them to ChronicleDB within the same Java process. In particular, we used two synthetically generated event streams. The first stream (`Stock`) was taken from [39]. It simulates a stock ticker with four attributes: sequence number, symbol, price, and volume. Including the timestamp, events are 28 bytes in size. If not stated otherwise, this stream was used in the experiments. The second stream (`Sine`) comprises events with six 64-bit floating-point attributes. Thus, the size of an event is 56 bytes including the 8-byte timestamp value. For the i^{th} event, the corresponding attribute values are generated as $\sin(\frac{i \bmod 1M}{1M} \cdot 2\pi)$. This allows us to control the selectivity of filter conditions on secondary attributes. By default, both data sources use increasing timestamps (i.e., $e_i.t = i$) and consist of 100M events.

5.2 TAB$^+$-Tree

Right flank. To showcase the impact of storing the right flank in `PMem`, we measured the wall-clock time for inserting `Stock`. The results are depicted in Fig. 5 for increasing flush batch sizes. When flushing every event, the insertion rate drops to approx. 50% of the `DRAM` implementation while still allowing for a respectable 2.1M insertions

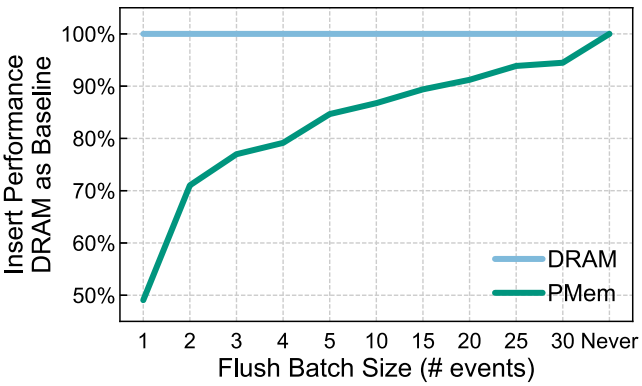

Fig. 5 Insert performance when managing TAB$^+$-Tree's right flank on PMem compared to main memory as a function of the flush batch size

[4] https://openjdk.java.net/jeps/352.

per second. However, with a batch size of 25 events, we achieve 95% of DRAM performance and reduce data loss by at least 91% (losing less than 25 instead of 291 events). Additionally, PMem reduced the recovery time after a system failure from 40 ms to below 1 ms for a tree of height 3.

Aggregates/Index nodes. Next, we discuss the impact of storing lightweight index information (i.e., aggregates) and inner index nodes on persistent memory. For aggregates on PMem, we implemented two variants. The first variant stores aggregates densely, while the second variant aligns them on 64-byte boundaries for cache efficiency. We inserted the events of Sine into the TAB$^+$-Tree and compare the insertion wall-clock time as well as the average response time for a variety of queries. The results are summarized in Fig. 6 with the original ChronicleDB serving as a baseline. Insert performance is barely affected by any of the approaches. This is expected since writing leaf nodes is the dominant cost factor of insertion. Furthermore, aligning aggregates on cache-line boundaries had no visible effect on query performance. This could be explained by the fact that not all the PMem bandwidth is utilized in this setup. For application time point queries, the higher fan-out achieved when storing aggregates on PMem reduces execution time by approx. 15%. However, for temporal aggregation queries covering a variety of time ranges the double access (index node + aggregate) introduces a performance penalty of 10% to 25%. Finally, when using lightweight indexing for filter-queries on a secondary attribute with a selectivity of 0.1%, the benefit of a higher fan-out is partly eliminated by the extra access to PMem to read aggregates (approx. 15% improvement). In contrast, storing entire index nodes on PMem results in a reasonable performance boost for all query types (35%–40%), because index navigation and aggregate access do not incur reads on secondary storage.

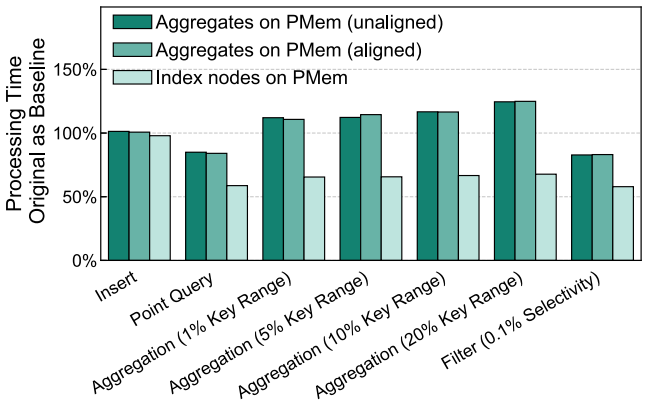

Fig. 6 Comparison of insert and query performance for TAB$^+$-Tree aggregates/index nodes managed on flash storage and persistent memory

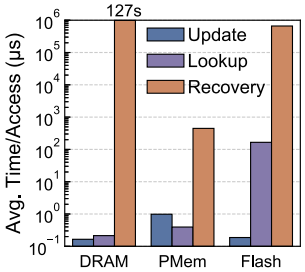

Fig. 7 Comparison of update, lookup, and recovery performance for address translation managed in DRAM, on PMem, and flash

5.3 Address Translation

For our address translation benchmark, we measured the component in isolation and implemented a pure in-memory version (DRAM) used in our comparison. Fig. 7 shows the average time of a sequential update, a random lookup, and the total recovery time. Each of those measurements is based on the wall-clock time of 10M operations. Even though DRAM exhibits excellent update and lookup performance, it requires a full file scan upon recovery making it infeasible for production use cases. Compared to Flash, the PMem solution offers superior lookup and recovery time. However, sequential updates on Flash are 5x faster compared to PMem. The reason for this is that the flash-based ATT maintains its right-flank in main memory. In summary, even though the very small update/lookup times are hardly noticeable outside of isolated benchmarks, PMem can bridge the gap between significant Flash recovery times and DRAM access times while simplifying the implementation complexity due to its flat-array structure.

5.4 Out-of-Order Handling

For the OOO benchmarks, we compare a DRAM red-black application time tree with persistent solutions (PMem, Flash with 8-KiB pages). For the latter, we differentiate between a pure solution and one with an additional in-memory application time index (Indexed). The following set of experiments is based on a queue size of 100 MiB (approx. 3.5M events).

First, Fig. 8 illustrates results for the OOO component in isolation while comparing implementations w.r.t. write, recovery, and query performance in milliseconds. Comparing write performance, both PMem solutions cannot compete with DRAM, because every event is flushed directly. However, they outperform both flash variants by 3× (pure) and 2× (Indexed), respectively. During recovery, only the persistent Indexed variants have to perform work besides opening the log. In this case, PMem and Flash exhibit similar performance, because the recovery time is dominated by re-building the index while the cost of scanning data is negligible. For query performance, we executed application time point queries and measure the average query

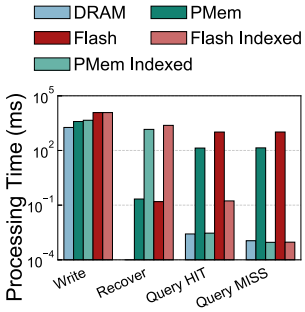

Fig. 8 Comparison of insert, recovery, and query performance of various OOO queue approaches

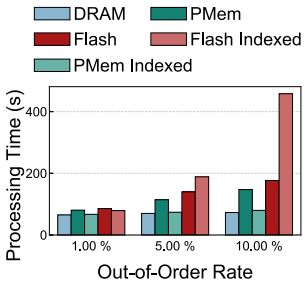

Fig. 9 Inserting 100M events with occasional OOO occurrences, and merges of OOO events into the primary index

time while distinguishing between HIT (query has a result) and MISS (query has no result) queries. For MISS-queries, all `Indexed` variants exhibit the same performance, since only the index has to be considered. However, for HIT-queries, `Flash Indexed` suffers from reading a full page from the log for each query. In contrast, `PMem Indexed` can directly access the corresponding event, and thus is as quick as `DRAM`.

Second, we discuss the overall impact of the queue implementations within ChronicleDB. For this purpose, we generate various degrees of OOO data for the `Stock` data source. In particular, for an OOO fraction of x%, (100 – x)% of events are inserted in application time order. From the remaining x% events, 10% are randomly uniformly distributed over the application time span. The other 90% are equally distributed over 10,000 equi-distant temporally close batches. This workload represents occasional OOO data coupled with short bursts. Fig. 9 shows the total processing time in seconds for inserting data sets with 1%, 5%, and 10% OOO fractions into ChronicleDB. Due to the limited capacity of 100 MiB, queue merges impact the write performance. As a result of unsorted data, non-indexed variants cannot take advantage of bulk-merging. `Flash Indexed` performs worst for 5% and 10% because reading data in application-time order from the log incurs random access to pages. As expected, `PMem Indexed` does not suffer from this drawback, and thus achieves the best of both worlds.

6 Summary & Research Directions

To store massive amounts of continuous temporal data, event stores have to provide fast ingestion speeds with adequate query performance. Most proposed generic storage systems that can also be used for events utilize single- or two-layer approaches, featuring a combination of main memory, secondary storage media, and, since recently, persistent memory. Using ChronicleDB as an example, we proposed several solutions for the first PMem-enhanced event storage system that utilizes all three layers for an overall cheap yet efficient system. We summarize the lessons learned from adapting ChronicleDB components to PMem as follows.

(LL1) Even though PMem can be used to improve the number of events recoverable upon a system crash, it is important to keep the **characteristics of the component** in mind. For frequent updates, as exhibited by the right flank, batching needs to be applied to meet insertion performance. A similar lesson can be applied to LSM stores. However, for infrequent updates, such as the OOO queue, there are more relaxed requirements and also opportunities to **flatten performance fluctuations**. This practically solves the primary challenge of OOO management. These insights can also be transferred to other buffering or index maintenance techniques.

(LL2) **Access patterns still matter.** As seen with our lightweight indexing proposal, sacrificing spatial locality for storing lightweight index information on a faster medium is not beneficial in all cases. Thus, correlated index structures for event streams need to be considered even for byte-addressable PMem.

(LL3) Although components like the address translation layer have little impact on the overall performance, the simplified design leads to **lower code complexity**. Additionally, there are great recovery benefits even for seemingly non-performance critical components.

Based on those lessons, there are a plethora of research directions to explore besides applying those insights to more specialized or more generic systems.

(RD1) As a direct continuation of this work, more sophisticated **out-of-order merge strategies** can be examined to further speed up event ingestion. One example would be merging only certain OOO regions and utilizing PMem for efficient pinpoint free space management.

RD2 In a similar vein, data streams that provide frequent updates instead of inserts such as spatio-temporal moving objects, can be an interesting direction to explore **update characteristics of PMem**.

RD3 Following *LL1* and *LL2*, there are several other ways to make even **finer use of PMem**, such as optimizing the node size, the alignment, as well as the internal node organization and the associated access patterns (cf. [8]).

RD4 Finally, while this work focused on the storage system itself, **query processing** mechanisms such as efficient event replay remain an ongoing challenge. In that regard, a three-layer caching mechanism as an extension of the LeanStore [17], as well as adaptive bi-temporal index building on PMem is yet to be explored.

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 "High-performance event processing on modern hardware: bridging the gap between low-latency and high throughput" (SE553/9) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037). This work has been co-funded by the LOEWE initiative (Hesse, Germany) within the emergenCITY centre.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Andrei M, Lemke C, Radestock G, Schulze R, Thiel C, Blanco R, Meghlan A, Sharique M, Seifert S, Vishnoi S, Booss D, Peh T, Schreter I, Thesing W, Wagle M, Willhalm T (2017) SAP HANA Adoption of Non-Volatile Memory. Proc Vldb Endow 10(12):1754–1765
2. Arulraj J, Pavlo A, Malladi KT (2019) Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. CoRR abs/1901.10938. http://arxiv.org/abs/1901.10938. Accessed 25 Sep 2020
3. Axboe J (2020) Flexible I/O Tester. https://github.com/axboe/fio, version 3.7. Accessed 25 Sep 2020
4. Bender MA, Farach-Colton M, Fineman JT, Fogel YR, Kuszmaul BC, Nelson J (2007) Cache-Oblivious Streaming B-trees. In: SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9–11, 2007, pp 81–92, https://doi.org/10.1145/1248377.1248393
5. Deri L, Mainardi S, Fusco F (2012) tsdb: A Compressed Database for Time Series. In: Traffic Monitoring and Analysis—4th International Workshop, TMA 2012, Vienna, Austria, March 12, 2012. Proceedings, pp 143–156, https://doi.org/10.1007/978-3-642-28534-9_16
6. Eisenman A, Gardner D, AbdelRahman I, Axboe J, Dong S, Hazelwood KM, Petersen C, Cidon A, Katti S (2018) Reducing DRAM Footprint with NVM in Facebook. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23–26, 2018, pp 42:1–42:13, https://doi.org/10.1145/3190508.3190524
7. Götze P, Baumann S, Sattler K (2018) An NVM-Aware Storage Layout for Analytical Workloads. In: 34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16–20, 2018, pp 110–115, https://doi.org/10.1109/ICDEW.2018.00025
8. Götze P, Tharanatha AK, Sattler KU (2020) Data Structure Primitives on Persistent Memory: An Evaluation. In: International Workshop on Data Management on New Hardware (DAMON'20), June 15, 2020, Portland, OR, USA, https://doi.org/10.1145/3399666.3399900
9. Haas G, Haubenschild M, Leis V (2020) Exploiting Directly-Attached NVMe Arrays in DBMS. In: CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings, http://cidrdb.org/cidr2020/papers/p16-haas-cidr20.pdf. Accessed 25 Sep 2020
10. InfluxData Inc (2020) InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData. https://www.influxdata.com/. Accessed 25 Sep 2020
11. Intel Corporation (2019) Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf, chapter 11 – Intel® Optane™ DC Persistent Memory. Accessed 25 Sep 2020
12. Intel Corporation (2020) Intel® Memory Latency Checker v3.8. https://software.intel.com/en-us/articles/intelr-memory-latency-checker. Accessed 25 Sep 2020
13. Jibril MA, Götze P, Broneske D, Sattler K (2020) Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. In: 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), pp 115–120, https://doi.org/10.1109/ICDEW49219.2020.00010
14. Johnson T, Shkapenyuk V (2015) Data Stream Warehousing In Tidalrace. In: CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4–7, 2015, Online Proceedings, http://cidrdb.org/cidr2015/Papers/CIDR15_Paper4.pdf. Accessed 25 Sep 2020
15. Kannan S, Bhat N, Gavrilovska A, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2018) Redesigning LSMs for Nonvolatile Memory with NoveLSM. In: 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11–13, 2018, pp 993–1005, https://www.usenix.org/conference/atc18/presentation/kannan. Accessed 25 Sep 2020
16. Kimura H (2015) FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31–June 4, 2015, pp 691–706, https://doi.org/10.1145/2723372.2746480
17. Leis V, Haubenschild M, Kemper A, Neumann T (2018) LeanStore: In-Memory Data Management beyond Main Memory. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16–19, 2018, IEEE Computer Society, pp 185–196, https://doi.org/10.1109/ICDE.2018.00026
18. Lersch L, Hao X, Oukid I, Wang T, Willhalm T (2019) Evaluating persistent memory range indexes. Proc Vldb Endow 13(4):574–587

19. Lersch L, Lehner W, Oukid I (2019) Persistent Buffer Management with Optimistic Consistency. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019, pp 14:1–14:3, https://doi.org/10.1145/3329785.3329931
20. Lersch L, Schreter I, Oukid I, Lehner W (2020) Enabling Low Tail Latency on Multicore Key-Value Stores. Proc Vldb Endow 13(7):1091–1104
21. Li J, Pavlo A, Dong S (2017) NVMRocks: RocksDB on Non-Volatile Memory Systems. https://web.archive.org/web/20200217045318/istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems/. Accessed 25 Sep 2020
22. Liu J, Chen S, Wang L (2020) LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. Proc Vldb Endow 13(7):1078–1090
23. Loboz C, Smyl S, Nath S (2010) DataGarage: Warehousing Massive Performance Data on Commodity Servers. Proc Vldb Endow 3(2):1447–1458
24. Moerkotte G (1998) Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In: VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24–27, 1998, New York City, New York, USA, pp 476–487, http://www.vldb.org/conf/1998/p476.pdf. Accessed 25 Sep 2020
25. O'Neil PE, Cheng E, Gawlick D, O'Neil EJ (1996) The Log-Structured Merge-Tree (LSM-Tree). Acta Inf 33(4):351–385
26. Ou Y, Chen L, Xu J, Härder T (2014) Wear-Aware Algorithms for PCM-Based Database Buffer Pools. In: Web-Age Information Management – WAIM 2014 International Workshops: BigEM, HardBD, DaNoS, HRSUNE, BIDASYS, Macau, China, June 16–18, 2014 Revised Selected Papers, pp 165–176, https://doi.org/10.1007/978-3-319-11538-2_16
27. Oukid I, Booss D, Lehner W, Bumbulis P, Willhalm T (2014) SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In: Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014, pp 8:1–8:7, https://doi.org/10.1145/2619228.2619236
28. Oukid I, Lasperas J, Nica A, Willhalm T, Lehner W (2016) FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26–July 01, 2016, pp 371–386, https://doi.org/10.1145/2882903.2915251
29. Pelkonen T, Franklin S, Cavallaro P, Huang Q, Meza J, Teller J, Veeraraghavan K (2015) Gorilla: A Fast, Scalable, In-Memory Time Series Database. Proc Vldb Endow 8(12):1816–1827
30. Pelley S, Wenisch TF, Gold BT, Bridge B (2013) Storage Management in the NVRAM Era. Proc Vldb Endow 7(2):121–132
31. van Renen A, Leis V, Kemper A, Neumann T, Hashida T, Oe K, Doi Y, Harada L, Sato M (2018) Managing Non-Volatile Memory in Database Systems. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10–15, 2018, pp 1541–1555, https://doi.org/10.1145/3183713.3196897
32. van Renen A, Vogel L, Leis V, Neumann T, Kemper A (2019) Persistent Memory I/O Primitives. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019, pp 12:1–12:7, https://doi.org/10.1145/3329785.3329930
33. Seidemann M, Seeger B (2017) ChronicleDB: A High-Performance Event Store. In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21–24, 2017, pp 144–155, https://doi.org/10.5441/002/edbt.2017.14
34. Seidemann M, Glombiewski N, Körber M, Seeger B (2019) ChronicleDB: A High-Performance Event Store. ACM Trans Database Syst 44(4):13:1–13:45, https://doi.org/10.1145/3342357
35. Stoica R, Athanassoulis M, Johnson R, Ailamaki A (2009) Evaluating and Repairing Write Performance on Flash Devices. In: Proceedings of the Fifth International Workshop on Data Management on New Hardware, Association for Computing Machinery, New York, NY, USA, DaMoN '09, p 9–14, https://doi.org/10.1145/1565694.1565697
36. Tangwongsan K, Hirzel M, Schneider S, Wu KL (2015) General incremental sliding-window aggregation. Proc Vldb Endow 8(7):702–713
37. Xia F, Jiang D, Xiong J, Sun N (2017) HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In: 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12–14, 2017, pp 349–362, https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia. Accessed 25 Sep 2020
38. Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S (2020) An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In: 18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24–27, 2020, pp 169–182, https://www.usenix.org/conference/fast20/presentation/yang. Accessed 25 Sep 2020
39. Zhang H, Diao Y, Immerman N (2014) On complexity and optimization of expensive queries in complex event processing. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, SIGMOD '14, p 217–228, https://doi.org/10.1145/2588555.2593671