Leveraging Non-Volatile Memory for Instant Restarts of In-Memory Database Systems

David Schwalb, Martin Faust, Markus Dreseler, Pedro Flemming, Hasso Plattner
Hasso-Plattner-Institute, Potsdam, Germany

***Abstract*—Emerging non-volatile memory technologies (NVM) offer fast and byte-addressable access, allowing to rethink the durability mechanisms of in-memory databases. Hyrise-NV is a database storage engine that maintains table and index structures on NVM. Our architecture updates the database state and index structures transactionally consistent on NVM using multi-version data structures, allowing to instantly recover data-bases independent of their size.**

In this paper, we demonstrate the instant restart capabilities of Hyrise-NV, storing all data on non-volatile memory. Recovering a dataset of size 92.2 GB takes about 53 seconds using our log-based approach, whereas Hyrise-NV recovers in under one second.

I. Introduction

Traditionally, database systems had been designed with the fundamental assumption that data is stored on disks and paged into memory when required for query processing. Recently, several systems have emerged that use main memory as their primary location of data storage [1], [2], [3], [4], [5]. This allows in-memory database systems to be fast at runtime, but they still require the use of write-ahead logging and checkpointing on storage for durability and recovery. They suffer from startup and recovery times proportional to the database size as they need to load the entire dataset from storage into memory on startup. As an example, assume a large enterprise with a 5 TB dataset. An optimal loading process requires at least 55 minutes for the data transfer reaching a sustainable read bandwidth of 1.5 GB/s from storage. Additionally, recovery requires log-files to be replayed and index structures to be recreated, resulting in typical load times of multiple hours for large enterprise systems. Lazy data loading shifts the load cost to query processing, but makes query response times unpredictable. Such long recovery times are even problematic in replicated scenarios as the critical time during re-establishing redundancy after system updates and restarts should be minimized. Recent trends in memory hardware technology indicate that byte-addressable and non-volatile memory technologies (NVM) will soon be available on the memory bus [6], [7].

Contribution. This paper outlines a demonstration of the instant restart capabilities of Hyrise-NV. Hyrise-NV is a NVM-based storage engine that persists all table data and index structures directly on NVM and enables failure-atomic and durable updates on NVM using multi-version data structures. We compare our system with a log-based recovery approach and report restart times under 100ms.

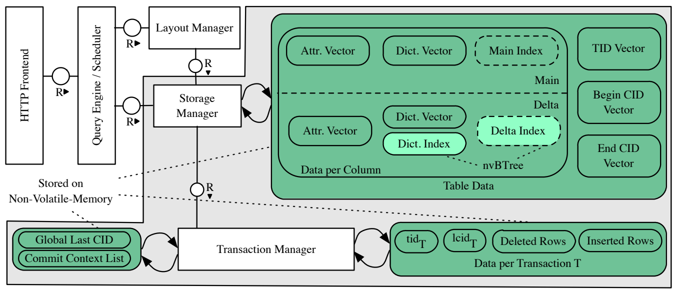


Fig. 1: Hyrise-NV architecture diagram: all objects that have their data stored on NVM are shown in green, dotted objects show optional index structures.

II. System Overview

Hyrise is an in-memory storage engine specifically targeted to mixed workload scenarios [1] and a balanced execution of both analytical and transactional workloads using task-based scheduling, while optimizing for the set-processing nature of business applications [8]. Hyrise-NV extends Hyrise and persists all table data and index structures directly on NVM as outlined by Figure 1. Hyrise-Log implements a traditional logging approach, where the data is periodically flushed to disk. Data modifications follow the insert-only approach [1] and Hyrise maintains separate index structures for the main and delta partition. To efficiently support the insertion of new values and range queries as commonly required in enterprise workloads, the delta index is implemented as a tree-based multi-map of values and positions [9]. Additionally, Hyrise supports multi-column indices [10]. Multi-version concurrency control is realized by storing begin and end timestamps for each row and filtering read-sets based on the transaction information. Updates are realized as an insert of the new version with an invalidation of the old version. Write-write conflicts are detected using a row-level locking mechanism based on atomic compare-and-swap operations, whereas the later transaction is aborted. Hyrise supports a lock-free commit mechanism using explicit commit dependencies.

Hyrise-Log implements logging, checkpointing and recovery mechanisms that are used as a baseline implementation to compare against Hyrise-NV. The ARIES style logging mechanism [11] leverages the applied dictionary compression and only writes redo information to the log as no undo information is required due to the lack of in-place updates [9]. New log entries are buffered in a ring-buffer and written to disk using a group commit mechanism [9], [11]. Checkpoints create a consistent snapshot of the database as a binary dump on disk in order to speed up recovery by allowing to directly load the checkpoint from disk without the need of expensive

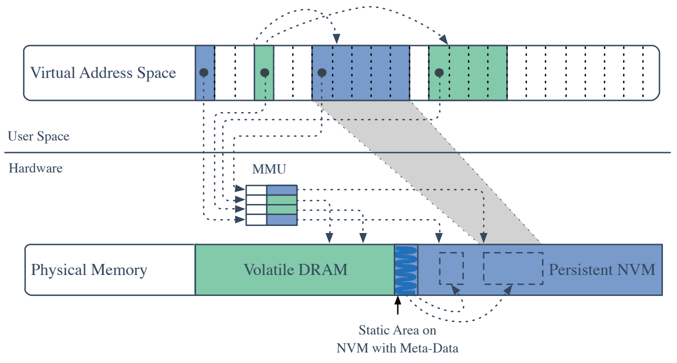


Fig. 2: System with hybrid configuration of DRAM and NVM. Mapping through hardware MMU on page level, resulting in virtual address space of mixed volatile and persistent memory.

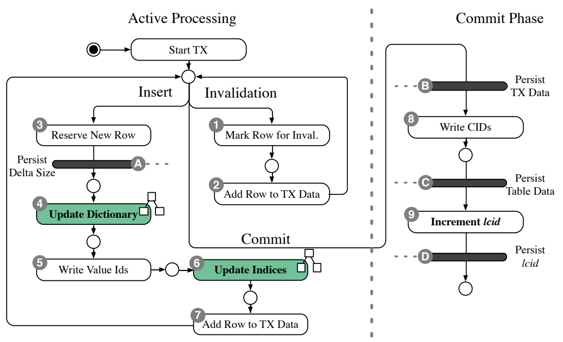


Fig. 3: Additional barriers are required during transaction processing on NVM to explicitly flush data from caches and to guarantee write ordering.

delta log replays. In contrast to disk-based database systems where a buffer manager only needs to flush all dirty pages in order to create a snapshot, Hyrise-Log supports checkpoints by persisting the complete delta partitions of all tables plus begin and end timestamps for main and delta to disk.

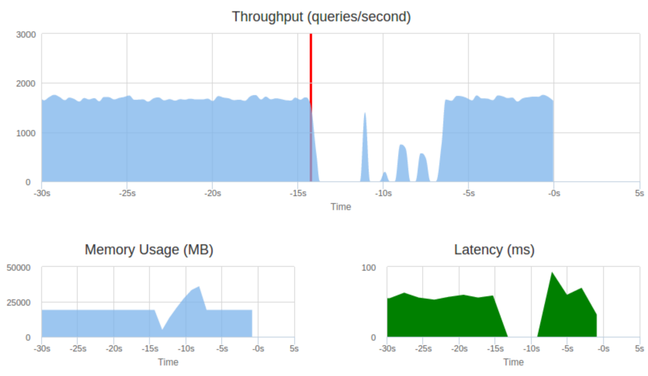
III. Adapting Hyrise to NVM

This section presents Hyrise-NV and focuses on changes required to the architecture of Hyrise in order to leverage NVM as the primary persistence domain. Hyrise-NV stores the complete database state on NVM, including all table data and index data structures, in order to enable instant restarts of the system. We show how using NVM as the primary persistence domain does not interfere with the ACID criteria and how to ensure that changes reach NVM in an atomic and durable way. We envision future system architectures to support mapping both volatile and non-volatile main memory into the virtual address space. Consequently, it will be the responsibility of the application to track and recover allocation on non-volatile memory and to differentiate between the volatile and non-volatile world, as outlined in Figure 2. Dynamic memory allocation on NVM requires carefully designed allocators that are failure-atomic to avoid inconsistent states and permanent memory leaks. Hyrise-NV uses a custom memory allocator for NVM that allows to differentiate between volatile and non-volatile memory regions and to implement application specific recovery strategies to find and restore objects from NVM [12].

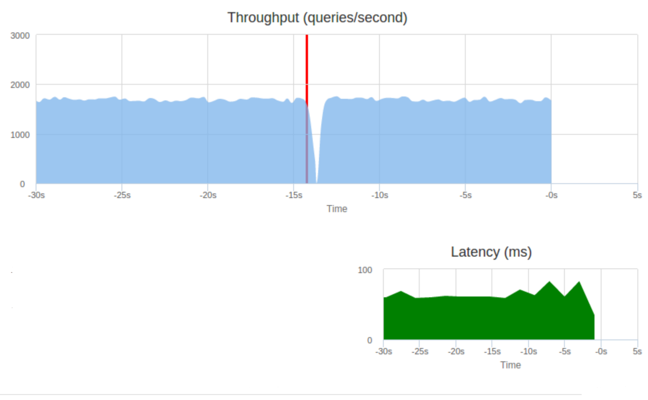
To guarantee the consistency during updates of all data structures, a careful system design is necessary, with explicit barriers in the system to guarantee the write order on NVM, as well as mechanisms to provide atomicity and durability for transaction management on NVM. Columns, dictionaries and indices are designed to be append-only and do not execute in-place updates of values. This insert-only approach allows to avoid in-place updates of rows and to ensure atomicity of transactions by using the multi-version concurrency control. Building on this, a transaction can work on its private data space on NVM until processing has finished and all changes are made visible atomically by incrementing the last visible commit id on NVM.

Figure 3 outlines the steps during transaction processing in our system and the explicitly needed barriers to guarantee consistency on NVM. A barrier consists of *clflush* instructions to write all modified cache lines to NVM and a *sfence* instruction to enforce their completion. To ensure the consistency on NVM, Hyrise-NV require four barriers A, B, C and D during the processing of transactions: *Barrier A* ensures that the size of the delta partition is flushed to NVM before the transaction proceeds to populate the reserved rows. The barrier is required to avoid index structures referencing rows that do not yet exist. When a transaction enters the commit phase, *barrier B* ensures that the transaction context is persisted on NVM, containing a list of all inserted and updated rows. This is required in order to repair in-flight transactions in case of failures. *Barrier C* persists all table data and flushes all dirty changes on attribute vectors and transaction data. Therefore, it is guaranteed that all transaction changes are persisted before incrementing the last visible commit id. Incrementing the last visible commit id makes all changes of a committing transaction visible for subsequent transactions. The final *barrier D* ensures that the last visible commit id is written to NVM before returning a transaction as committed to the client. The combination of append-only updates, explicit barriers and consistent tree structures enables the system to guarantee a consistent state on NVM at any time. Storing indices on NVM requires special handling to ensure failure atomicity. Hyrise maintains separate index structures for the main and delta partitions. Indices on the main partition leverage its read-only nature to reduce the storage footprint by creating an immutable structure for the mapping of values to positions during the merge process. In contrast to the main index, the delta index needs to efficiently handle newly inserted values. It is implemented as a tree-based multi-map of values (K) and positions (P), to efficiently support range queries as commonly required in enterprise workloads [8].

When adapting the index data structures, we also face the challenge to ensure data consistency across failures. Indices for the main partition are updated during the merge process using a shadow-copy mechanism [8] and do not need to be treated differently using NVM. However, indices on the delta partition are frequently updated and need to be stored on NVM in order to avoid expensive rebuilds on restart or recovery. Hyrise-NV uses a tree-based index structure called nvBTree to support persistent indices for the delta. nvBTree is designed to be stored on NVM and implements a multi-map of values and positions into the delta partition. It extends an existing B^+-tree implementation by adding multi-versioning



(a) Log-based approach.



(b) NVM-based approach.

Fig. 4: Screenshots of our tool showing query throughput, memory usage and latency of two running Hyrise instances over time. The red marker indicates a simulated system crash. (a) shows the restart of our log-based approach, whereas (b) shows the recovery of our NVM-based approach.

to make consistent and atomic updates on NVM, requiring no additional logs [13]. The tree allows for parallel read operations, but requires exclusive write locking. It is designed to support failure-atomic inserts directly on NVM by enforcing the write order and using multi-versioning.

Having the complete database state persisted on NVM, Hyrise-NV is able to recover using this information in case of failures without requiring write-ahead logs. If the system crashes, the information persisted on NVM is the only version of the data and needs to allow recovery to the latest consistent database state. The recovery process works by (i) re-initializing the system with persisted data structures, (ii) repairing inconsistent states of in-flight transactions and (iii) recovering index structures. The state of the transaction manager, meta-data and table data are reinitialized in the first step. The system might crash while a transaction T was in progress and has written its commit id to the data, but before it could increment the last visible commit id. During recovery, T needs to be reverted to avoid that new transactions will make the partial changes of T visible. To avoid scanning all tables during recovery to find changes of in-flight transactions that have to be reverted, the transaction context of committing transactions is persisted when entering the commit phase. Therefore, the recovery can easily traverse all in-flight transactions and revert potential changes by iterating through the lists of inserted and deleted rows. Technically, the presented database recovery mechanism depends on the number of tables and their columns, the number of index structures with partial updates, and the number of in-flight transactions in the system. Practically, the recovery mechanism minimizes the necessary work during recovery and system restarts, allowing to recover databases independent of their number of rows. In practice, we achieved sub-second recovery times even for large data sets over hundreds of gigabytes.

IV. Demonstration

The demonstration shows a comparison between our proposed system Hyrise-NV and Hyrise-Log on the same 92 GB dataset. We built a tool that allows to start both instances in parallel on two different servers, load a TPC-C data set and execute the TPC-C query set against both instances in parallel. The tool allows to monitor query throughput and latencies. Additionally, we can simulate a failure that leads to a crash of both instances at the same time. The goal of this set-up is to show throughput and recovery performance of both approaches and allow an intuitive comparison in a running environment. The demonstration clearly shows how Hyrise-NV can restart and recover from the crash with all data on NVM in less than 100 milliseconds. In contrast, the log based approach needs to replay its latest log files and reload all data into memory before executing queries again, leading to an outage of almost one minute before reaching its maximum throughput again. Figures 4a and 4b show screenshots from our demonstration with a dataset of 18.4 GB. We monitor multiple statistics of the two Hyrise instances. The main statistic is the query throughput. We measure the number of TPC-C transactions per second, as well as the DRAM memory usage.

Figure 4a shows a screenshot from our monitoring tool where we monitor a Hyrise-Log instance. The instance has loaded data of 18.4 GB. The top graph shows the query throughput, whereas the bottom right graph pictures average transaction latencies. The red marker indicates a simulated system crash. We can see the throughput dropping to zero and only recovering after around 15 seconds. Inside the lower left chart we can see the memory usage of Hyrise-Log. We can see the memory usage growing linearly when recovering the data. During restart, Hyrise-Log builds several auxiliary data structures while loading the data from a PCIe attached enterprise-grade SSD. This is the reason why the memory usage grows larger than the data size during recovery. After finishing recovery, the memory usage drops back to the actual data size when these auxiliary data structures are removed. In Figure 4b, we can see the same demonstration for Hyrise-NV. Again, at the red marker a restart of Hyrise has been triggered. Here we can see that the throughput only drops for an instant as recovery takes only a few hundred milliseconds. The memory usage graph shows a DRAM footprint of 84 MB as the major parts of the data structures are allocated on NVM. In Table I,

TABLE I: Recovery Times

Data Size	Hyrise-Log	Hyrise-NV
3.1 GB	1.01 s	0.02 s
5.1 GB	1.79 s	0.03 s
9.3 GB	2.65 s	0.03 s
18.4 GB	5.31 s	0.06 s
36.9 GB	13.52 s	0.12 s
92.2 GB	53.22 s	0.29 s

we list detailed recovery times for different sizes of datasets. We report a recovery time of Hyrise-NV that is more than 100 times smaller compared to Hyrise-Log for data sets larger than 30 GB.

V. Related Work

The emerging NVM technologies have been subject to related research, which evaluates the technology in the context of database systems. Chen and Jin [14] present a PCM-friendly B^+-Tree structure with unsorted leaf nodes and Yang et al. [15] propose a consistent and cache-optimized B^+-Tree structure with reduced CPU cacheline flushes. In contrast, we evaluate our proposed tree structure in the context of database systems and using a hardware-based emulation approach. Oukid et al. [16] present work similar to our approach, describing a prototypical storage engine that leverages full capabilities of NVM by removing the traditional log and updating the persisted data in-place. Their results are similar to ours and we validate them using an additional method to simulate NVM latencies by overriding the serial presence detect. Pelley et al. [22] reconsider storage management with NVM to optimize recovery performance and forward-processing throughput. In contrast to our approach, they focus on disk-based databases and use NVM to store the log file instead of using a NVM-only architecture. Venkataraman et al. [13] present data structures designed for byte addressable NVM. They eliminate the need for a write-ahead log by providing consistent updates of durable data structures. We build on those mechanisms, integrating and evaluating them in an in-memory database system. Additionally, multiple systems and techniques have been proposed to integrate NVM into existing systems and simplify its usage, e.g. Mnemosyne [17] and NV-Heaps [18] to provide transactional semantics using software transactional memory or specialized file systems for NVM like PMFS [19].

VI. Conclusion and Future Work

In this paper, we presented a tool to demonstrate the instant restart capabilities of Hyrise-NV, storing all data on non-volatile memory. Recovering a dataset of size 92 GB takes about 53 seconds using our log-based approach, whereas Hyrise-NV recovers under one second. The instant restart capabilities do not come for free and require to persist the complete data set on NVM, leading to significant performance overheads [20]. We expect future hardware systems to introduce new instructions that allow for a more optimized flushing of caches and ordering of writes so that the overhead can be reduced, e.g. a new write-back instruction promises to write back data without cache invalidations. Future work includes research on optimized data structures for NVM, thus reducing the need to flush caches and to enforce barriers through CLFLUSH and SFENCE instructions. Additionally, future evaluations using real NVM hardware are planned as well as more sophisticated considerations to ensure the correctness and consistency of the durable data on NVM.

References

[1] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise: a main memory hybrid storage engine," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 105–116, 2010.

[2] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, and Y. Zhang, "H-store: A High-Performance, Distributed Main Memory Transaction Processing System," *VLDB*, 2008.

[3] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots," *ICDE*, 2011.

[4] H. Plattner, M. Faust, D. Schwalb, S. Mller, J. Wust, and M. Uflacker, "The Impact of Columnar In-Memory Databases on Enterprise Systems," *VLDB*, 2014.

[5] V. Raman *et al.*, "DB2 with BLU Acceleration: So Much More than Just a Column Store," *VLDB*, 2013.

[6] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[7] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[8] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier, "Fast updates on read-optimized databases using multi-core cpus," *VLDB*, 2011.

[9] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner, "Efficient transaction processing for hyrise in mixed workload environments," in *IMDM*, 2014.

[10] M. Faust, D. Schwalb, J. Krüger, and H. Plattner, "Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance," *ADMS*, 2012.

[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *TODS*, 1998.

[12] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory Allocation for NVRAM," *ADMS*, 2015.

[13] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory," *FAST*, 2011.

[14] S. Chen and Q. Jin, "Persistent B+-Trees in Non-Volatile Main Memory," *VLDB*, 2015.

[15] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems," *FAST*, 2015.

[16] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis, "Instant Recovery for Main-Memory Databases," *CIDR*, 2015.

[17] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGPLAN*, 2011.

[18] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," *SIGPLAN*, 2011.

[19] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," *EuroSys*, 2014.

[20] D. Schwalb *et al.*, "Hyrise-NV: Instant Recovery and Startup for In-Memory Databases using Non-Volatile Memory," *DASFAA*, 2016.