



PDF Download
3774753.pdf
09 March 2026
Total Citations: 0
Total Downloads: 152

Latest updates: <https://dl.acm.org/doi/10.1145/3774753>

RESEARCH-ARTICLE

Update NDP: On Offloading Modifications to Smart Storage with Transactional Guarantees in Near-Data Processing DBMS

ARTHUR BERNHARDT, Reutlingen University, Reutlingen, Baden-Wurttemberg, Germany

SAJJAD TAMIMI, Technical University of Darmstadt, Darmstadt, Hessen, Germany

FLORIAN STOCK, Technical University of Darmstadt, Darmstadt, Hessen, Germany

ANDREAS KOCH, Technical University of Darmstadt, Darmstadt, Hessen, Germany

ILIA PETROV, Reutlingen University, Reutlingen, Baden-Wurttemberg, Germany

Open Access Support provided by:

Technical University of Darmstadt

Reutlingen University

Published: 06 March 2026
Online AM: 04 November 2025
Accepted: 06 October 2025
Revised: 12 June 2025
Received: 29 May 2024

[Citation in BibTeX format](#)

Update NDP: On Offloading Modifications to Smart Storage with Transactional Guarantees in Near-Data Processing DBMS

ARTHUR BERNHARDT, Data Management Lab, Reutlingen University, Reutlingen, Germany

SAJJAD TAMIMI, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Germany

FLORIAN STOCK, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Germany

ANDREAS KOCH, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Germany

ILIA PETROV, Data Management Lab, Reutlingen University, Reutlingen, Germany

The performance and scalability of modern data-intensive systems processing large datasets are limited by unnecessary data movement. Even though near-data processing (NDP) can provably reduce data transfers and increase performance, at present, NDP is utilized primarily in read-only settings. Near-data execution of data-intensive modification operations is currently infeasible due to the lack of transactional consistency and the absence of practicable low-latency synchronization mechanisms between the host database engine and the NDP-engine on smart storage.

In this article, we introduce *update NDP* as an approach to offloading modifications to computational storage with transactional guarantees in an NDP database system called neoDBMS. To ensure consistency, we introduce a low-latency shared lock table between the host and computational storage, based on novel *cache-coherent interconnects*. We also introduce a novel *locking protocol* that seamlessly integrates the shared lock table within the lock manager of the host NDP-engine. To handle failure recovery, while preserving high and robust performance, we introduce novel extended locking and logging mechanisms that allow the host and computational storage to perform useful work during log-movement. Our evaluation indicates that in-storage modifications in neoDBMS in mixed workload settings are $\geq 6.52\times$ faster than host-only executions and exhibit robust performance due to lower data movement and better resource utilization.

CCS Concepts: • **Information systems** → **DBMS engine architectures**; **Data locking**; **Transaction logging**; **Storage architectures**; **Storage management**;

Additional Key Words and Phrases: Database systems on smart storage, near-data processing

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – DFG grants *neoDBMS* and *neoDBMS.2* – 419942270.

Authors' Contact Information: Arthur Bernhardt, Data Management Lab, Reutlingen University, Reutlingen, Baden-Wuerttemberg, Germany; e-mail: Arthur.Bernhardt@reutlingen-university.de; Sajjad Tamimi, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Hessen, Germany; e-mail: tamimi@esa.tu-darmstadt.de; Florian Stock, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Hessen, Germany; e-mail: stock@esa.informatik.tu-darmstadt.de; Andreas Koch, Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Hessen, Germany; e-mail: koch@esa.tu-darmstadt.de; Ilia Petrov (corresponding author), Data Management Lab, Reutlingen University, Reutlingen, Baden-Wuerttemberg, Germany; e-mail: ilia.petrov@reutlingen-university.de.



This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 0362-5915/2026/03-ART11

<https://doi.org/10.1145/3774753>

ACM Reference Format:

Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Andreas Koch, and Ilia Petrov. 2026. Update NDP: On Offloading Modifications to Smart Storage with Transactional Guarantees in Near-Data Processing DBMS. *ACM Trans. Datab. Syst.* 51, 2, Article 11 (March 2026), 45 pages. <https://doi.org/10.1145/3774753>

1 Introduction

Motivation. Data modification operations on large datasets cause significant data movement. For one, modifications operate on combinations of hot and large cold data that typically reside on slow persistent storage [79]. For another, the data movement necessary to determine, the data items that meet the update conditions may be significant even if just a few of them get modified, not least due to the poor data locality in such datasets [41, 51]. Such data transfers are slow, are performed over energy-hungry interconnects, and tend to block processing until data becomes available. Consequently, data movement impairs performance and limits the scalability and resource efficiency of data-intensive systems.

Near-Data Processing (NDP) is a known paradigm [1, 18, 47, 77] that aim at offloading operations to a processing element near the physical data location [10], e.g., to smart storage, and can therefore provably reduce the impact of data transfers. Recently, multiple smart storage devices (also known as computational or intelligent storage) have emerged [33, 69, 70, 73, 97], which combine persistent memory and processing elements. Typically, such devices offer high device-internal bandwidth, parallelism, and low on-device latencies that benefit the offloaded operations. However, even though in-storage executions are seemingly beneficial for reading operations, the general case of in-situ modifications is infeasible to date.

Brief state-of-the-art overview. Despite all its advantages, NDP is utilized mainly in read-only settings and static datasets [3, 21, 51, 81]. This marks a severe limitation given the opportunity offered by modern DBMS designs [2, 4, 7, 8, 31, 34, 37, 48, 52, 54–56, 64, 67, 68, 75, 76, 78] that process concurrent OLTP-style modifications in-memory, while preserving the large cold dataset immutable to execute read-only analytical operations in-storage. Recently proposed techniques such as *snapshot-based, update-aware NDP* [14, 57, 89–91] mark a major step forward, in that they allow update transactions to be executed by a host-side engine, in parallel to read-only analytical NDP operations on the same dataset, which execute against a consistent on-device *snapshot*, with transactional guarantees.

Offloading modifications. There is a pressing need for offloading modification operations since they represent another major source of data movement. Firstly, modifications gain importance as recent studies indicate a significant shift towards write-intensive workloads in large-scale production systems [23, 98]. According to [98], more than 35% of all investigated workloads at Twitter/X are write-intensive (write/read ratio $\geq 30\%$), while 20% of the workloads have more writes than reads (write/read ratio $\geq 50\%$). A study from Facebook/Meta [23] indicates that modifications in persistent social graph workloads amount to approx. 15% to 30% (depending on the table), while the portion of modifications increases to 92% in machine-learning settings. Secondly, modifications may also regularly touch a large portion of the dataset [79], for privacy protection regulations such as GDPR, or the right-to-be-forgotten. Thirdly, workloads call for *offloading combinations of reading and modifying operations*. Noticeably, offloading individual operations does not always reduce data transfers overall. However, much potential is provided by offloading whole *sequences* of mixed operations as NDP-Pipelines [89], i.e., even if just a few data items get modified, the data movement necessary to determine them may be significant. This is especially relevant given the highly skewed distributions [13, 98] in large-scale systems.

Prior work [60, 71, 82] has already demonstrated the benefits of offloading modifications in some special cases. For instance, in an in-storage DBMS [71] in absence of concurrency or synchronization

with the host, or in out-of-place in-storage compaction for KV-stores [60, 82], where ad-hoc synchronization mechanisms are not required. Nonetheless, apart from such special cases, several problems preclude general offloading of modifications.

PROBLEM 1 (IN-STORAGE MODIFICATIONS NECESSITATE FINE-GRANULAR, LOW-LATENCY SYNCHRONIZATION AND INVALIDATION MECHANISMS). *To avoid conflicts with NDP updates, low-latency, invalidation and synchronization mechanisms between host and smart storage are necessary, yet infeasible over state-of-the-art PCIe. Consider, for instance, an NDP transaction (TX_{NDP}) offloading a modification operation to a record r to smart storage (Figure 1(A), 1(B)). An initial version of $r.v_0$ is present both in-memory and on-device. TX_{NDP} creates a new record version $r.v_1$, while at the same time, a host transaction TX_{HOST} creates its own $r.v_1$, as the host DBMS is unaware of the on-device modification. As a result of the write/write conflict, two version branches are anomalously created. Alternatively, (Figure 1(B)(B)) during TX_{NDP} commit processing, the version $r.v_0$ cached on the host should be invalidated, forcing TX_{HOST} to fetch $r.v_1$.*

Clearly, such conflicts can be mitigated by coarse-granular (e.g., full table) locks on the table containing r ahead of TX_{NDP} execution, yet this is impractical as it would severely limit concurrency. Alternatively, **Optimistic Concurrency Control (OCC)** seems applicable. Yet, it has been shown to lead to (a) starvation of read-intensive transactions where a writer wins over a reader [100], (b) resource losses as OCC aborts late at commit time [50], and (c) additional data transfers for the NDP read/write sets, and abort-handling under contention or mixed workloads. Lastly, fine-granular locking (i.e., record/tuple-level) offers a potential solution, yet it entails several severe drawbacks in external accelerator/smart storage settings. On the one hand, they result in small data transfers (a few cachelines) that are prohibitively slow over current PCIe interconnects [32, 63, 87], as these are bandwidth-optimized and aim for a transfer granularity of more than few KB. On the other hand, placing fine-granular locks mandates cache-coherency and ideally atomics, yet they are impractical since PCIe is non cache-coherent. As shown in prior work, implementing fine-grain cache-coherence over PCIe is infeasible [32] as it requires either intricate manual cacheline flushes [20], or OS page migrations [20]. Modern *disaggregated memory* settings aggravate matters as the pressure for low-latency cache-coherent interactions increases, if multiple smart devices participate in data processing operations.

PROBLEM 2 (LOW-OVERHEAD LOGGING IS NECESSARY TO ENSURE RECOVERY). *NDP modifications mandate novel logging approaches to ensure consistent rollback, as well as system and media recovery as smart storage devices may fail.*

This raises several questions. Firstly, what kind of logging is necessary, given the non-volatile nature of smart storage and where should the persistent log be kept? The currently prevailing combination of undo/redo logging and physiological approaches has been designed for two-tier storage systems with non-atomic writes. If naïvely applied to NDP, such approaches incur logging overhead and increase log-data movement. Therefore, one design aspect is the mitigation of such overheads, given that NDP modifications are performed within non-volatile smart storage and go to fresh storage regions, thus enabling write-atomicity.

Secondly, while keeping the log in smart storage is a natural approach, how can media-recovery be ensured if the smart device holding the log fails? In such cases, a device failure would cause both data and log loss, precluding media failure recovery. Transferring logs back and integrating them in the host log (and/or the archive log) is a plausible solution. On the downside, the data movement caused by such log-transfers and the long log-flushing delays would needlessly prolong lock-duration, causing lock contention, and unnecessarily increasing commit latency.

Update NDP. In this article, we present an approach called *update NDP* for offloading modification operations to smart storage. Update NDP is implemented in neoDBMS, which is an NDP-capable

DBMS [14, 89] (Section 2.3 provides a brief overview). We extend neoDBMS' architecture for offloading modifications to smart storage and executing them in-situ with transactional guarantees, propagating the freshest host-side modifications alongside the NDP invocation to smart storage, where the NDP modifications, potentially in combination with read-operations, are executed against a transactionally consistent snapshot constructed on-device. Noticeably, the NDP modifications execute in an autonomous and nearly interruption-free manner in-storage.

To address the synchronization and invalidation problem, we introduce a novel, low-latency **Shared Lock Table (SLT)** between the host-DBMS and smart storage that is based on novel **cache-coherent interconnects (CCI)** such as CCIX or CXL. In contrast to prior work [16], the SLT in neoDBMS allows for low-overhead tuple-level locking, high parallelism and is fully integrated into the lock-manager of neoDBMS with a novel locking protocol. Another major aspect is that the SLT is placed inside novel **cache-coherent SVM (ccSVM)** between host and external accelerators/smart storage nodes. With ccSVM novel *post-Moore* DBMS architectures become feasible, making NDP a first-class citizen in heterogeneous scale-up systems, and smart storage apt for cooperative execution.

The main idea behind handling NDP modifications is to store the newly created versions out-of-place, in exclusively allocated space and address ranges. Upon completion, the in-storage neoDBMS engine only transfers back to the host NDP-engine the changes to certain mapping tables, which are applied atomically at the host-side if the calling transaction commits. Thus, NDP modifications are performed in an *atomic manner*, loosely resembling shadow paging.

neoDBMS addresses the logging and recovery problem, based on a confounding observation. Since NDP modifications are atomic, neither undo-, nor redo-logging is necessary during the NDP operation, which is efficient and transfer-sparing. However, host-side logging pertains for the rest of the NDP transaction (host-side operations), if any. This way, transaction rollback and system recovery are always possible. Yet, in case of a media recovery upon smart storage device failure, the lost NDP modifications of committed transactions must be recovered. To this end, neoDBMS constructs a log covering the NDP modifications on device, which is then transferred to the host-DBMS and applied there as an overflow log. However, the resulting log-movement and log-flushing cause negative aspects, i.e., prolonged lock duration, lock contention, and ultimately lower throughput. To mitigate them, neoDBMS employs lock-violation techniques [36], allowing subsequent transactions to acquire locks conflicting with locks held by the NDP modification, at the cost of a commit dependency and as soon as the NDP transaction initiates its commit process. Therefore, such subsequent transactions can commence and perform useful work, while logs are transferred and flushed. What's more, such a technique is especially favorable for hot-tuples and contention hotspots between host and NDP transactions, as it makes it possible for hot-tuples to bounce between device and host, and between different NDP transactions on-device.

Introductory Experiment. We demonstrate the impact of *update NDP* in mixed-workload settings in a motivating experiment (Figure 1(A)) comparing traditional host-only executions under standard PostgreSQL to update NDP under neoDBMS. We select a modification-intensive workload, comprising a mix of write-heavy OLTP (YCSB-A [25], 50% read / 50% write) and an NDP-able update to stress effects of update NDP, host-shared locking, and logging. The workload is subdivided into three phases. In the first phase, both systems run a write-heavy host-side OLTP workload (YCSB-A). In the second MIXED phase, we inject an NDP-able update operation parallel to host-side OLTP, and switch back to pure host-side OLTP in the last phase.

In traditional PostgreSQL settings, the whole workload is executed by the host-DBMS engine and suffers from data movement and the resulting resource contention. In contrast, with update NDP, the modification is executed near-autonomously in-storage, while the host DBMS processes

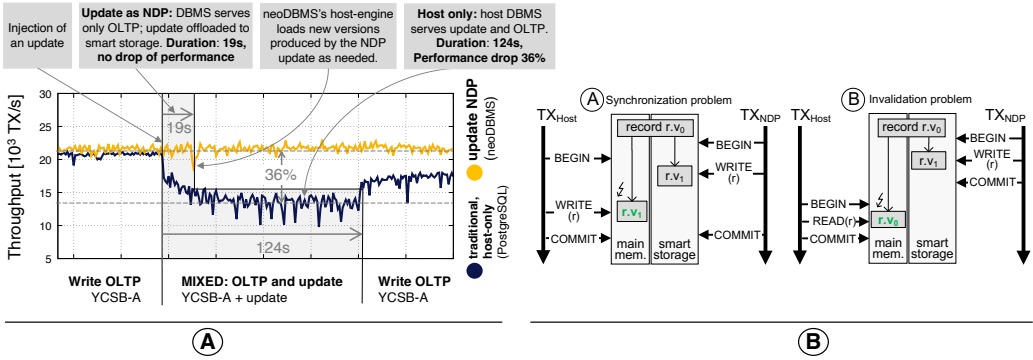


Fig. 1. (A) While traditional DBMS suffer from data movement during host-only processing of modifications, neoDBMS offloads modifications as NDP-operations to smart storage, where they are executed autonomously, i.e., without intervening host-processing and avoiding resource contention. Under mixed workloads neoDBMS executes on-device modifications faster than host-only ($\geq 6.52\times$, 19s vs. 124s), and with robust performance, while throughput drop during traditional host-only processing amounts to $\sim 36\%$. (B) Illustration of the synchronization and invalidation issues described by Problem 1.

the write-heavy OLTP workload in an interruption-free manner. neoDBMS performs the on-device modifications $\geq 6.52\times$ faster than host-only, i.e., 19s update NDP versus 124s host-only. The clear performance drop in traditional settings during the MIXED phase is due to excessive cold data transfers and the resource contention they cause, e.g., buffer pollution or lock contention. Noticeably, under update NDP the OLTP throughput remains robust, while the performance of the host-only execution drops by $\sim 36\%$.

Contributions. The article presents the following contributions:

- We introduce *update NDP* as a novel approach and DBMS architecture for executing modifications on smart storage. In neoDBMS modifications are executed in-storage with transactional guarantees and in a nearly autonomous manner, i.e., without interruptions and with negligible host/device interaction overhead.
- We introduce an integrated lock manager that combines a *SLT* with the host lock-table. The *SLT* relies on novel cache-coherent interconnects between the host and smart storage such as CCIX/CXL. *SLT* achieves low lock latencies of 80–400ns (host-side) and 750–800ns (device-side) depending on the level of contention.
- We introduce a novel integrated locking protocol that unifies host-side and *SLT*-locking. It ensures efficient synchronization in update NDP settings, without limiting the host parallelism.
- We introduce a novel logging approach for in-storage modifications in neoDBMS. It supports transaction rollback and system recovery seamlessly, while for media recovery we introduce an overflow-log mechanism. To mitigate the impact of log-movement and log-flushing on lock contention of hot-records, we use lock-violation techniques in the novel context of update NDP.
- NDP updates in neoDBMS are $\geq 6.52\times$ faster than host-only executions due to the reduction of data transfers and resource contention. Furthermore, update NDP yields robust performance.

Outline. We continue with a brief background on cache-coherent interconnects, followed by an introduction to DBMS design principles that enable the integration and execution of NDP, the foundation upon which neoDBMS extends. After that, we present the architecture of neoDBMS (Section 3) and provide an overview of the execution of updating NDP transactions (Section 4).

Novel host-shared locking techniques are introduced in Section 5, followed by our novel approach to NDP logging and system recovery (Section 6). We describe optimizations in Section 7, and discuss our experimental results following in Section 8. We address related work in Section 10 and conclude in Section 11.

2 Background

We start by introducing the challenges of extending single-host memory concepts across multiple devices over conventional interconnects and the difficulties in developing host-device synchronization mechanisms on top (Section 2.1). We then provide an introduction to novel cache-coherent interconnects and device-types that enable exposing and efficiently sharing memory between host and devices, facilitating synchronization primitives through cache-coherent transfers that neoDBMS' SLT builds upon (Section 2.2). In addition, we briefly review NDP-capable smart storage and DBMS architectures, focusing on the building blocks for database operation offloading. We explain how neoDBMS leverages snapshot-based NDP on multi-versioned storage (Section 2.3), that has been proven feasible for offloading read-only operations, which we extend in this work to also support offloading modification operations.

2.1 Shared Virtual Memory and Cache-Coherence

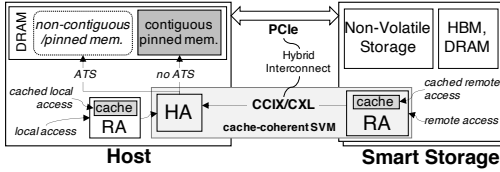
SVM extends the virtual memory concept beyond a single host to encompass an entire device group. With SVM, a host CPU has access to the entire memory hierarchy, however, with external hardware like accelerators or smart storage devices, extra care is needed to ensure *cache-coherency* for shared memory accesses of each party to the local and the remote party's memory. *Cache-coherency* for shared memory mandates that whenever the host or the smart storage device reads or modifies a shared memory location, the other party can always read the most recent value, whereas modifications are seen in the same order by all parties.

However, several drawbacks exist. First, smart storage may operate on host memory addresses, yet this is cumbersome as the required *address translation* has to be ensured explicitly. Second, *cache-coherence* with the host system must be programmed manually over the state-of-the-art PCIe interconnects as they lack inherent support. However, SVM operations typically perform cacheline-sized accesses, which is impractical as PCIe requires much larger transfer sizes to reach its maximum throughput. For instance, fLink [26], reports transfer sizes of 128KB–256KB, necessary to reach maximum throughput, and utilizes multiple DMA engines. First benchmarks of commercial PCIe Gen5 SSDs indicate bandwidth saturation with 64KB–128KB transfer sizes [24].¹ Moreover, the long PCIe latencies for such small messages make automated cache-coherence protocols over PCIe impractical [32, 63]. In disaggregated memory settings and heterogeneous systems, the concept of SVM becomes even more difficult. One solution is offered by novel cache-coherent interconnects, which we overview next.

2.2 Cache-Coherent Interconnects (CCI): CXL, CCIX

Over the last decade several proposals for CCI emerged, e.g., Gen-Z, Open-CAPI, CCIX, and the **Compute Express Link (CXL)**, all of which eventually subsumed under CXL. Among others, CCI aim to enable efficient communication and data sharing between processors and accelerators/smart storage in heterogeneous environments. Broad CXL hardware support of the upcoming industry standard is still unavailable. We elaborate more on our system choice after a short introduction to both CCI technologies.

¹With PCIe 6.1, a new Flit-mode is introduced, addressing the need for faster messages and a better bandwidth utilization with smaller transfer granularities.



(A) smart storage devices in a direct-attached CCIX topology.

		CCIX (cache-coherent accesses)				PCIe
		cached remote access	remote acces	ATS + remote access	local access	
Read	Cacheline (64 B)	100 ns	699 ns	2454 ns	60 ns	1050ns
	Page Size (8KB)	1820 ns	3409 ns	4662 ns	-	2270ns
Write	Cacheline (64 B)	80 ns	686 ns	2418 ns	60 ns	530ns
	Page Size (8KB)	1920 ns	3018 ns	7466 ns	-	1290ns

(B) CCIX versus PCIe access latencies [87].

Fig. 2. CCIX memory sharing with smart storage.

2.2.1 *Cache-Coherent Interconnect for Accelerators (CCIX)* resembles CXL as an advanced I/O interconnect that enables cache-coherent data sharing across heterogeneous devices (e.g., CPUs and accelerators) that supports signaling rates between 16-25 GT/s per link. It defines a set of new *CCIX protocol and link layers* on top of PCIe [42]. As a result, applications on top of CCIX can autonomously move data between the host-CPU and the accelerator/smart storage caches/memory, building the foundation for *ccSVM*. CCIX enables data sharing with shared memory and virtual addresses, which allows for *pointer-based* data structures to be exchanged transparently without tedious pointer-swizzling, serialization/deserialization, and intricate DMA. Virtual address resolution is performed by *CCIX ATS* (Address-Translation Service, Figure 2(A)). In this article, we employ the simplest *direct-attached* CCIX topology [42] (Figure 2(A)), directly connecting the host CPU and smart storage. Even this topology allows (a) the host and smart storage to share memory, achieving *larger-than-host-memories*; and (b) if the host shares memory, *ccSVM* enables partitioning across multiple devices.

CCIX architecture. CCIX introduces different types of hardware *agents* (Figure 2(A)) that represent hardware modules to realize different topologies. For simplicity, we only consider the **Home** and **Request Agents (RA)** in a directly attached topology. Figure 2(A) shows the host sharing memory with a smart storage device or an accelerator. **Home Agents (HA)** will be placed on the node (host or smart storage) that shares its memory. The HA is responsible for managing memory access to a pre-defined shared address range. It passively manages coherence to the shared address range by sending snoop messages to involved RA whenever the state of a cacheline changes.

Furthermore, several *RA* will be present on all nodes (smart storage, accelerators, or the host itself) accessing the shared memory. The RAs actively initiate local and remote read/write accesses to HAs. Moreover, RAs may perform local *caching* to speed up accesses. For instance, local *RA-caches* allow making portions of the shared memory (here host memory) available on accelerators as local memory for faster accesses, i.e., *cached remote accesses*. Alternatively, an access to a new location results in an RA local cache-miss and thus in a *remote access*. In our example (Figure 2(A)), the host contains both a HA for managing coherence over the shared memory address range and an RA for locally accessing that shared memory, while the accelerator accommodates just an RA, as its own memory is not shared. Therefore, the host-DBMS will be performing *local accesses* or *cached local accesses*. Depending on whether the host or smart storage shares an address range, CCIX allows different placements of the HA² and the RAs, which is an important design decision in disaggregated memory settings. Placing the HA on smart storage (if it shares an address range) speeds up on-device accesses and reduces access times for NDP operations. This is the right design decision if the workload is device-centric and involves high-contention update-intensive NDP operations. Placing HA on the host reduces the host access contention and improves performance in host-centric workloads.

²Systems may have multiple HAs for different memories/storage.

CCIX Address-Translation Service (ATS) enables transparent virtual address resolution and pointer-based accesses. It allows an RA, e.g., on a smart storage/accelerator, to access any location of the host memory, while CCIX performs virtual address translation at the cost of multiple roundtrips and TLB accesses. The ATS also comes with its own ATS cache (*ATC Translation Lookaside Buffer Cache*), besides the RA cache. The flexibility of automated ATS comes at a price: according to Ref. [87] a full remote access with ATS (Figure 2(B)) costs $\sim 2400ns$. However, with *contiguous and pinned* shared memory ranges and if all accesses are constrained within the shared address range, the ATS overhead can be avoided, reducing the remote access costs to $\sim 680ns$, as the RA ATS can cache address tables in its ATC.

CCIX Atomics. On top of cache-coherence, CCIX introduces support for *atomics*, which is especially important for the programming model on top of ccSVM. Currently supported are: *AtomicLoad*, *AtomicStore*, *AtomicCompare*, or *AtomicSwap* on up to 128b of data.

2.2.2 Compute Express Link (CXL) treats the host and accelerator as peers. In contrast to CCIX, CXL has a CPU-centric, asymmetric view. Technically, CXL creates a cache-coherent framework over the physical layer of PCIe and reaches 32 GT/s with PCIe 5.0. CXL defines three different protocols: *CXL.io*, *CXL.mem*, and *CXL.cache*. The I/O protocol (*CXL.io*) aims at I/O operations with the non-coherent load/store semantics of PCIe and at simple control. It is used for device discovery, status reporting, or address-translation (ATS). *CXL.io* is mandatory for each CXL device. The memory protocol (*CXL.mem*) enables the host CPU to access remote memories in a NUMA-style fashion. *CXL.cache* allows the device and host to cache-coherently access each other's memory, using the widespread MESI coherency protocol on 64B cache lines. *CXL.cache* operates solely on physical host-addresses, therefore, CXL devices need to implement TLB, but can leverage PCIe's ATS and *CXL.io*. To this end, ATS is extended for *CXL.io* or *CXL.cache*.

Coherence Management. *CXL.mem* enables CXL devices to expose their device-local memory to the host, which is referred to as **Host-managed Device Memory (HDM)**. As a result, *CXL.mem* enables (a) a memory expander mode for larger-than-memory systems, and allows for (b) exposing device-local memory to the host. This raises the question of coherence management.

The coherence in the former expander mode (called Host-managed coherence or *HDM-H*) is solely managed by the CPUs, and there is no device \leftrightarrow host cache-coherence. Thus, such CXL device settings are unsuitable for ccSVM. However, if *CXL.cache* and *CXL.mem* are implemented, the latter mode with so-called Device-managed coherence (*HDM-D*) becomes possible. *HDM-D* allows exposure of device-local memory, and cache-coherency between device and host. While *HDM-H* settings employ no coherence protocol, *HDM-D* device-settings include cache state and snooping attributes in their messages, allowing to change cache-states of the host using *CXL.cache* interaction-roundtrips.

CXL Device Types. CXL categorizes devices into three types, depending on their capabilities and protocol support. In Figure 3 we highlight their applicability for ccSVM. A *CXL Type 1* device, e.g., a SmartNIC, implements *CXL.io* and *CXL.cache*. They do not expose their local storage/memory for cache-coherent interactions, however, they can cache-coherently interact with shared host memory, exposed over *CXL.cache*. *Type 2* devices implement all three protocols and encompass computational storage, GPGPUs, or FPGA systems. *Type 2* devices can coherently expose parts or all device memory to the host and other CXL devices (*HDM-H* & *HDM-D*). *Type 3* are typically memory expansion systems that implement *CXL.io* and *CXL.mem*, limiting cache-coherency solely to the host (*HDM-H*). Update NDP targets *Type 2* devices, while some of the proposed techniques are also applicable to *Type 1*, e.g., for in-network processing.

First CXL *Type 3* devices where cache-coherence is managed by the host CPU (*HDM-H*) were evaluated and published [59, 83]. Numbers reported range from 155 ns [59], utilizing PCIe 5, DDR5,

	CXL.io	CXL.mem	CXL.cache	
Coherence model		HDM-H host-managed coherence	HDM-D device-managed coherence	Targeted by Update NDP
Type 1 - device	✓		✓	Current hardware availability and research investigation
Type 2 - device	✓	✓	✓	
Type 3 - device	✓	✓		

Fig. 3. Only CXL Type 3 devices [59, 83], where cache-coherence is managed by the host CPU (HDM-H), were available ahead of 2024. While Type 3 devices remain prominent, there is a gradual emergence of host machines, devices, and design tools/IP blocks supporting CXL.cache since early 2024. Update NDP targets mainly Type 2 devices and is applicable to Type 1 devices, both of which are only slowly becoming available at the time of writing.

and an ASIC implementation in a directly attached topology, in memory-pooling settings, i.e., cache misses result in CXL requests without page-faults or DMAs. An evaluation of the latest hardware supporting CXL .mem (Type 3) shows latencies ranging from 170 ns for an ASIC-based implementation and up to 300 ns for an FPGA-based CXL solution [83].

Along the same lines, a recent evaluation of commercially available CXL Type 2 devices [45] report up to 67%(LLC-miss) - 96%(LLC-hit) higher latency for device to host accesses on real CXL-IP compared with emulated NUMA setups. In addition, their results show up to 40% higher latency for additional cache-coherence roundtrips during write contention compared with CXL Type 3 devices and HDM-H settings. Notably, they demonstrate that lower latency can be achieved if the device accesses the host memory compared with the host accessing the device memory. This aligns with our design choice in neoDBMS to share host memory with the device to enable ccSVM.

2.2.3 CCIX vs. CXL. The research described in this article utilizes CCIX [42] to attach the FPGA accelerator to the host system in a cache-coherent manner, also allowing the FPGA and the host CPU to share a common virtual address space. We call this operating mode ccSVM.

While CCIX itself has become deprecated, host machines, devices, and design tools/IP blocks supporting CXL .cache are only very slowly becoming available since early 2024. Similarly, the ARM N1SDP platform used as a host in the evaluation was the *only* commercially available server that has *actually* been qualified for FPGA-CPU ccSVM operation using CCIX. Thus, for demonstrating the ideas in actual hardware instead of just simulation, our technology choices were extremely limited. At the time of performing the research, neither CXL-enabled FPGA boards, nor CXL-qualified host systems actually supporting CXL Type 2 interaction were commercially available.

Insight: CXL .cache and HDM-D will add extra latency to the above numbers, due to additional cache-coherence roundtrips mandated by HDM-D. As of today, it is only possible to speculatively extrapolate these extra costs, yet we expect the above latencies to double. Therefore, a CXL Type 2 device using faster PCIe 5.0 will outperform CCIX, targeting PCIe 3.0/4.0. Consequently, the approaches proposed in this article are expected to maintain their validity and improve with CXL.

2.3 Overview of NDP DBMS Design Principles

Offloading DBMS operations to NDP-capable storage incurs challenges, particularly when it comes to ensuring data consistency and synchronization between the host DBMS and attached NDP devices. This section delves into key design considerations and provides background on NDP-DBMS architectures. We also summarize techniques already introduced by neoDBMS as they form a major portion of the prior work [14, 16, 89]. In addition, the hardware design principles and NDP architectures of smart storage devices utilized by neoDBMS are described in detail in Ref. [85].

2.3.1 Challenges. For successfully offloading DBMS operations, it is mandatory that the NDP-capable computational storage device obtains all necessary data *before* starting execution, or is able to identify and obtain them efficiently *during* execution. This is challenging for traditional DBMS designs, as modification operations often scatter their updates across the entire database address space. Beyond actual records, modifications spill across various auxiliary data structures, such as status and mapping tables, e.g., logical-to-physical address mapping ($L2P_{map}$). Prior to execution, these changes or updates need to be propagated to the device to ensure consistent NDP executions.

Another challenge is how to handle updates that are propagated to storage during the NDP execution itself. Consider, for example, offloading an analytical workload with potentially long execution times. Tuples, as well as mapping structures, might change multiple times *during* execution, resulting in inconsistencies. Effective strategies for managing and incorporating these propagated updates are crucial for maintaining data integrity and consistency throughout NDP.

2.3.2 Snapshot-Based NDP Execution. MVCC databases utilize out-of-place updates and the creation of new record versions, making them suitable for snapshot-based NDP execution [14]. This addresses both challenges discussed earlier: (a) the transactional snapshot specifies the data necessary on-device prior to NDP execution, and (b) even during consistent generation and propagation of new record versions, an NDP execution can determine visible and relevant tuples of its own snapshot. This enables an intervention-free execution of read-only NDP [89], allowing multiple NDP operations to execute concurrently on their own snapshot.

This opens the question of efficiently creating a snapshot for NDP execution. One approach is to compute and construct a snapshot on the DBMS first and subsequently replicate it on device, resulting in a *single-versioned storage* [17]. Afterwards, NDP operations operate directly on all stored data, as they belong to that single snapshot. While this approach eliminates the need for snapshot construction on the NDP device itself, careful scheduling of transactions and a batch-wise execution of NDP operations becomes necessary. Notably, concurrent modifications between DBMS and NDP devices are prohibited to maintain the integrity of the on-device snapshot during execution.

An alternative *multi-versioned storage* approach [14, 89–91] is introduced by nKV or neoDBMS, allowing concurrent NDP executions to operate on their own snapshot, as multiple versions are simultaneously present on-device. This approach was later also adopted by AIDE [57], however, they face the challenge of identifying visible record versions during NDP execution. Both neoDBMS and AIDE employ version indexes to efficiently iterate all versions. However, AIDE excludes snapshot construction and visibility checking from NDP processing and instead relies on a pre-screening step, to filter out invisible versions at the host-side prior NDP invocation before passing the whole filtered index to the NDP device. While AIDE's [57] approach eliminates the impact of concurrently propagated updates on the NDP execution, it increases data movement and write-amplification as each NDP invocation requires its own filtered version index, even after small updates, limiting the effective utilization of on-device memory- and cache-hierarchies (which are flexible and configurable in FPGA-based systems) to accelerate frequent index accesses.

In neoDBMS, visibility checking and snapshot creation is part of the NDP execution on device. The version index (VID-Mapping, Figure 5) always points to the latest version of a tuple, and gets regularly updated on-device, especially prior to NDP invocation. This may affect other concurrently active NDP executions, as the in-situ snapshot construction may require the traversal of the version-chain (version index represents the entry point of a version-chain that points to the newest tuple version), to find visible data belonging to the current snapshot. However, neoDBMS handles snapshot creation efficiently on-device [14], especially on byte-addressable NVM-based storage.

Surprisingly, although the approach [14] of continuously updating the version index has been designed for *read-only* NDP operations, it also proves beneficial to offloading *modifications* in

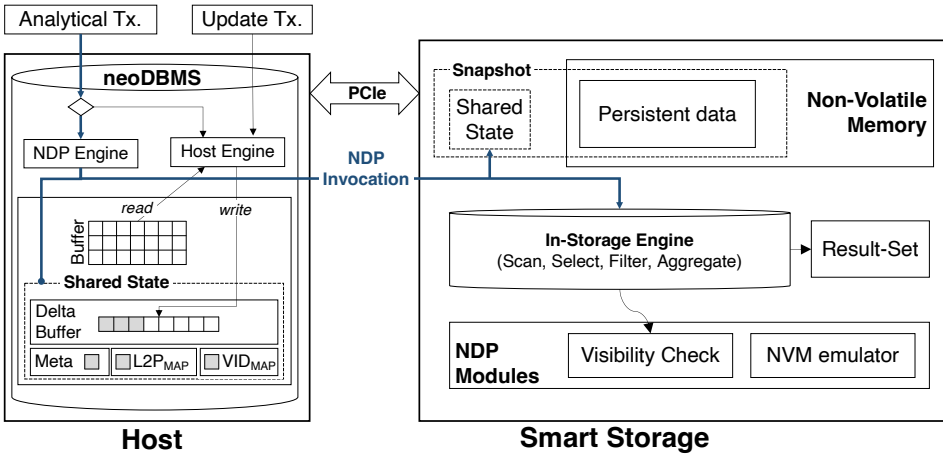


Fig. 4. Prior, update-aware NDP architecture of neoDBMS, capable of offloading read-only NDP operations, in the presence of concurrent host-side modifications. These are accumulated in the shared-state and the delta-buffer and transferred regularly, but always along with each NDP invocation. Once present on-device, the shared state together with the large cold data, contributes to the construction of a transitionally consistent snapshot of the whole dataset, against which the NDP operations are executed.

neoDBMS. We discover that it enables the detection of potential update and write-write conflicts during update NDP. In addition, snapshot-based NDP on multi-versioned storage has been feasible across different architectures such as PostgreSQL (relational data) and MySQL (key-value-store), highlighting its broad applicability to other systems that neoDBMS builds upon.

2.3.3 Shared-State between Host and NDP Devices. neoDBMS incrementally accumulates the modifications to DB-objects and mapping tables and places them in shadow data structures that are collectively referred to as *shared-state* [89] (Figure 4). As a result, the original data is left unmodified in the large DBMS memory. The shared-state is small and configurable in the range of a few hundred KB to a few MB at most and can be propagated at low overhead. Noticeably, the shared-state is the only delta between the *working set* in the buffer of the host DBMS that holds the most recent data, and the much larger but colder and *complete* dataset on smart storage. The shared-state is propagated to smart storage in two distinct modes [89]. Under the *flush & append* mode, the shared-state gets regularly propagated whenever it reaches a predefined size. Versions from still active or aborted transactions are removed, it is compacted and prepared for merge in the persistent storage. This allows for *lightweight checkpointing* and reduces the garbage collection effort. The second mode is *pass along & cache* [89]: at the time of an NDP invocation, the shared-state is snapshotted and, together with the list of transactions currently in-flight, propagated as part of the NDP invocation. The state is merely *cached* on-device for the duration of the call, and released/garbage-collected upon its completion. Thus, after propagation, the smart storage holds all data to construct a transactionally consistent *on-device snapshot* of the DBMS.

The *Delta-Buffer* (Figure 4) is a key element of the shared-state [89]. It accumulates the versions newly created by active transactions, while predecessor versions remain in the normal buffer in memory. Both can be accessed by concurrent transactions. As soon as the *Delta-Buffer* size reaches a threshold it gets propagated as part of the *shared-state* to smart storage together with incremental changes to the address-mapping table ($L2P_{map}$) or the VID table (VID_{map}), data dictionary definitions and so on.

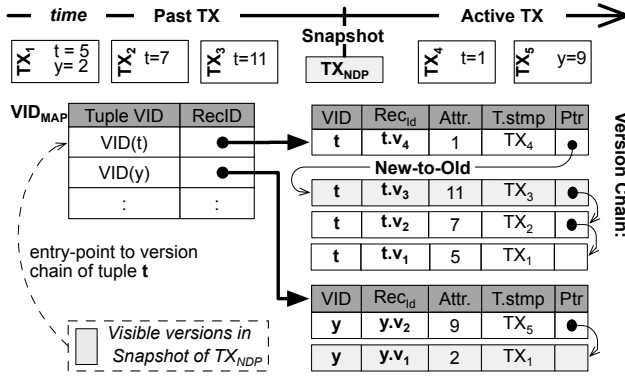


Fig. 5. Version organization in neoDBMS. Version-records have a backwards reference to the predecessor, forming a singly-linked list (version chain) in a new-to-old manner. The VID_{map} stores the references to the newest tuple versions, serving as an entry-point to the version chain of a tuple. Versions are invalidated implicitly by the existence of a new successor version.

2.3.4 Version-Organization, Multi-Versioning and Version-Index. Next, we provide a brief overview of the version organization and invalidation model of neoDBMS in parallel to PostgreSQL. neoDBMS organizes version records as a singly-linked list in a *new-to-old* (N2O) manner [14, 89], where every successor version has backwards reference to the predecessor [35], while PostgreSQL employs an *old-to-new* (O2N) organization [94]. Furthermore, neoDBMS introduces a one-point *version invalidation* model [14, 89], where each version contains only the timestamp of the creating transaction (omitting the invalidation timestamp), and is implicitly invalidated by the existence of a successor version. Modification operations in PostgreSQL cause in-place updates, as the timestamps of both creating and invalidating transactions are placed on the version record. The singly-linked version organization raises the issue of identifying the head of each version chain. To efficiently traverse all versions, neoDBMS utilizes a version index called VID_{map} (Figure 5), which stores the RecordID of the most recent version for every tuple.

All version records in a chain have the same *Virtual ID* (VID, e.g., t or y in Figure 5) as they belong to the same tuple. These are utilized by the version visibility check to construct a transactionally consistent snapshot. For instance, to construct the snapshot between TX_3 and TX_4 (Figure 5), the version chain is traversed to determine the first visible version of each tuple to a transaction, e.g., $t.v_3$ and $y.v_1$ to a transaction starting at the time of the snapshot.

In brief, neoDBMS optimizes for OLTP and NDP by ensuring a logical append (physically disjoint) storage. It speeds up visibility-checking, especially for fresh data, yet retrieving older versions may be slow. PostgreSQL has the better organization for long-running operations in HTAP as the oldest version is directly accessible.

2.3.5 On-Device Snapshot Construction. To create a snapshot on-device, the visibility check must determine the latest version record of a version chain, committed prior to the start of the NDP transaction. To this end, neoDBMS takes the snapshot information from the NDP invocation, i.e., the transaction ID (TxID) of the calling NDP transaction, and the delta-buffer and latest modifications to the $L2P_{map}$ and the VID_{map} from shared-state (Section 3.1) [14].

Given the N2O version organization in neoDBMS, the visibility check (Figure 6) is performed by traversing the records in a version-chain backwards, and comparing their creation timestamps against the TxID. The visibility task on each processing element (PE) extracts the entry-point RecordID of each entry (8B transfer) in the PE's VID_{map} partition and resolves it on-device. This

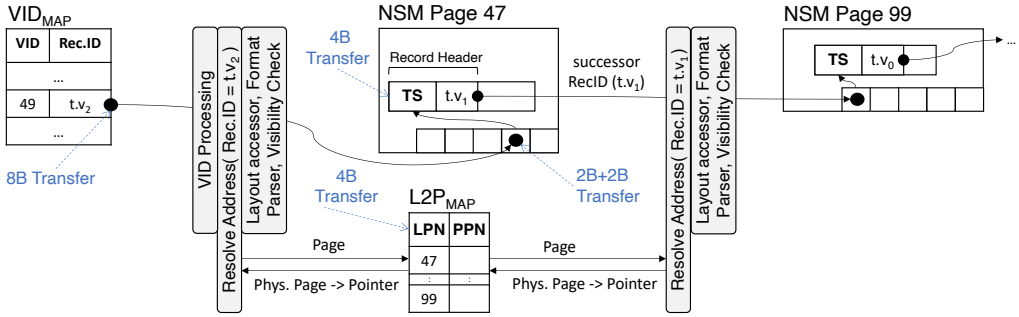


Fig. 6. On-device visibility checking in neoDBMS [14]. Layout accessors and format parsers are deployed for binary data navigation and interpretation. Logical to physical address resolution is performed directly on-device, utilizing $L2P_{map}$. neoDBMS may exploit the byte-addressability of the underlying non-volatile memory, reducing read-amplification. The shaded areas are executed on the individual PEs.

RecordID resolution is based on the $L2P_{map}$ (4B transfer) and yields the physical page pointer and a slot offset. It is then passed to the *layout accessor* on the PE, which retrieves the corresponding NSM page slot (4B transfer). In a successive 4B transfer, the *layout accessor* retrieves the record header and passes it to the format parser to retrieve the transaction timestamp and compare it to TxID. The invalidation timestamp is available from the predecessor (N2O org.) that has already been processed. The visibility decision is taken based on both timestamps. All visible versions are passed on to the NDP operation or pipeline.

The on-device snapshot construction and visibility checking may cause significant on-device data movement with page-granular I/O. To this end, neoDBMS may leverage the byte-addressability of the underlying NVM and minimize read-amplification (described in Section 7.1 in more details).

2.3.6 neoDBMS Shared-Locking Table. To the best of our knowledge, we introduced the first design of a SLT between a host DBMS (neoDBMS) and an NDP-capable smart storage device over a CCIX in our prior work [16]. The initial proposal describes a basic SLT design *without* full DBMS integration. The SLT [16] offers tuple-level locking and yields low-latencies for small random accesses, while ensuring cache-coherence and mitigating the overhead of virtual address translation. At the core of the SLT is a hash table with a hash function that uses a combination of the VID and tuple version number as a key. Each bucket represents a small queue, optimized for cacheline-sized accesses (64B), storing the timestamps of transactions wanting to acquire a lock on that tuple. The first slot in this queue is reserved for the transaction currently holding the lock, while the remaining seven slots represent pending lock requests. When a lock is released, the entire queue shifts to pass the lock to the next transaction. However, if the queue is already full, transactions (be them host or NDP) need to wait to acquire a free slot. This is a major disadvantage that limits the level of parallelism under contention. To ensure that locks are placed and released without race conditions, the system relies on atomic **compare and swap (CAS)** operations, which are supported by both the hardware module for tuple locking on smart storage and CCIX itself.

Comparison. Although our initial SLT design proposed in [16] characterizes locking latencies over CCI, it comes with certain limitations. First and foremost, it lacks full integration in a DBMS. It leaves the question of a combined locking protocol and integrated lock management open. Likewise, logging and recovery aspects are not discussed. Consequently, [16] does not include an evaluation of actual modifying NDP transactions. Secondly, the initial SLT design is limited to only eight concurrent lock-requests per queue. As a result, only eight transactions (be them host or NDP) can

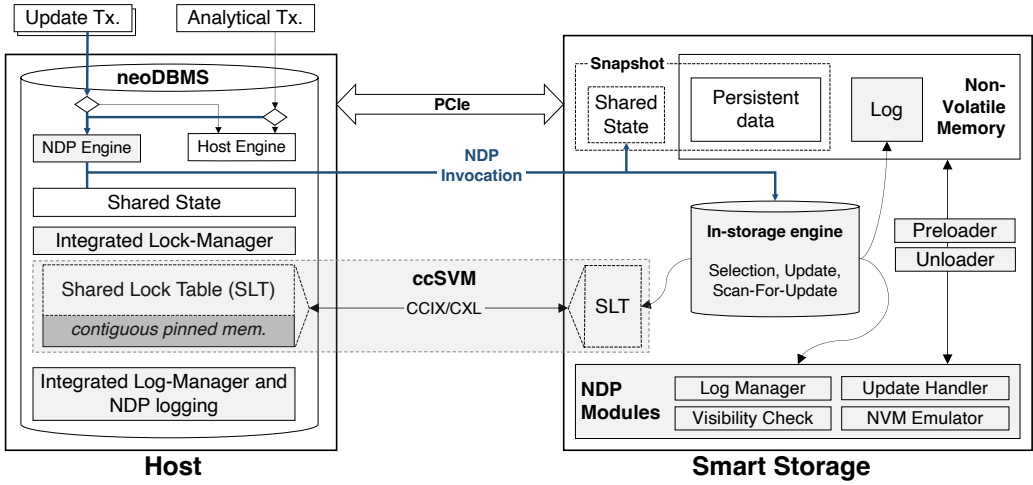


Fig. 7. Architecture of neoDBMS with extensions for update NDP represented by the shaded areas.

request a lock on a tuple, which is insufficient in contended settings or given the possible level of host parallelism. The SLT [16] relies on 128-bit atomics for both the host and the device, which could potentially become limiting factors for systems only supporting 64-bit atomics. Finally, [16] limits the level of parallelism as transactions processing contended tuples need to wait. To this end, we present an integrated host-shared locking protocol and an integrated lock-table design (Section 5) that addresses the above issues in neoDBMS.

3 Architecture of neoDBMS

In this section we overview the proposed update NDP architecture (Figure 7) and elaborate on its components.

3.1 Architectural DBMS Extensions for Update-Handling

neoDBMS, introduces a novel NDP engine handling NDP transactions, NDP space management, and NDP-inocations (Figure 7). NDP operations from updating NDP-transactions, are pushed down alongside the shared-state to smart storage, where an in-storage engine prepares, schedules, and distributes the execution to available **processing elements (PEs)** on-device. Multiple NDP invocations can run concurrently, each operating on its own snapshot with transactional guarantees. Furthermore, the NDP engine manages the commit and abort handling of NDP transactions.

neoDBMS also introduces an Integrated Lock-Manager that extends the host-locking and synchronization mechanisms with our novel SLT to coordinate concurrent accesses to tuples by host and NDP transactions (Figure 7). The SLT is allocated in ccSVM and therefore shared between the host and smart storage devices. To this end, neoDBMS allocates contiguous pinned memory during startup and propagates the memory address range to the in-storage engine, allowing the configuration of ccSVM, triggering ATC address caching (only once), and initializing the SLT on-device.

Novel NDP logging mechanisms are seamlessly integrated into the smart storage and DBMS, supporting transaction rollback and system recovery, as well as incorporating mechanisms for media recovery by introducing an overflow-log. To address the potential impact of log movement and flushing on lock contention for hot records and overall system performance, we implement lock-violation techniques in the context of update NDP.

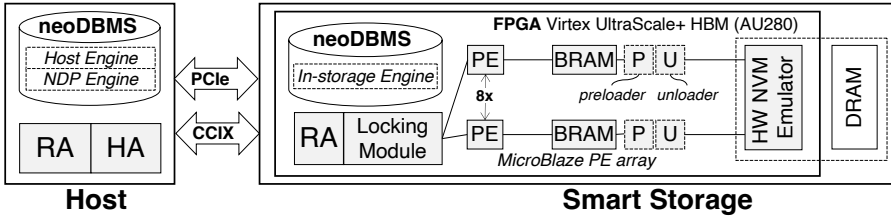


Fig. 8. Hardware architecture of neoDBMS. The host-side NDP engine offloads NDP operations to the in-storage engine, which schedules the workload across an array of processing elements (PE) on smart storage. Each PE can access its own fast scratch-pad memory (BRAM) and utilize preloader and unloader modules to asynchronously read and write data from NVM.

neoDBMS employs NDP operations, visibility-checker, layout accessors, and format parsers as small pre-compiled binaries that are loaded at the point of NDP invocation into individual PE. A key intuition is that the in-storage engine and NDP operations run as flexible software tasks, easily programmable in C. We call this approach *Software-NDP*. An array of up to 8 PEs can be used for distributing and executing operations (Figure 8) in the current hardware design. All PEs contain a MicroBlaze [65] soft-core, running at 180MHz, and have access to the on-board FPGA memory. Another key intuition is that on-device FPGAs have customizable memory hierarchies that can be efficiently utilized for NDP. Thus, fast on-FPGA BRAM memory (1 clock cycle) is attached to each PE as a local scratchpad. Each BRAM is attached to its own *preloader* and *unloader* HW-module, allowing byte-addressable DMA transfers to DRAM. The SLT is accessed via a dedicated HW locking module that translates lock requests into CCI transfers. To match the access characteristics of NVM on top of DRAM, neoDBMS employs the hardware NVM emulator from [86] in front of the memory controller.

The proposed design does not assume any specific type of processing elements for its implementation and is not limited to MicroBlazes or any other particular architecture. The used MicroBlaze PEs are off-the-shelf and ready-to-use synthesizable processor cores. However, they do not support any synchronization mechanisms over CCI. Furthermore, they do not include a DMA engine, making larger transfers from/to NVM slow and compute-intensive. Our HW-modules (SLT locking module, preloader and unloader) address these issues and are designed to be flexible and adaptable, allowing for the future use of RISC-V or ARM cores, or even fully custom FPGA logic operators.

neoDBMS introduces the concept of a novel *hybrid interconnect* to smart storage and disaggregated memory that combines PCIe for high-bandwidth I/O and CCI for low-latency cache-coherence in a unified design for the first time. Our unified hardware design switches between PCIe DMA [96] and CCIX Agents as they utilize the same PCIe lanes. The hybrid interconnect aligns the design to CXL, where the large PCIe transfers are performed over CXL .io, whereas cache-coherency is ensured over CXL .cache.

4 Execution of Updating Transactions

Before proceeding with our locking protocol and lock-table design, we now briefly overview the main steps in the execution of NDP modifications and NDP transactions in neoDBMS.

Transactions that contain NDP operations are called *NDP transaction*. neoDBMS combines a *host execution engine* and an *NDP engine* (Figure 7). The former handles transactions and operations scheduled for host execution, while the latter tackles NDP executions, NDP space management, NDP invocations, and commit and abort handling.

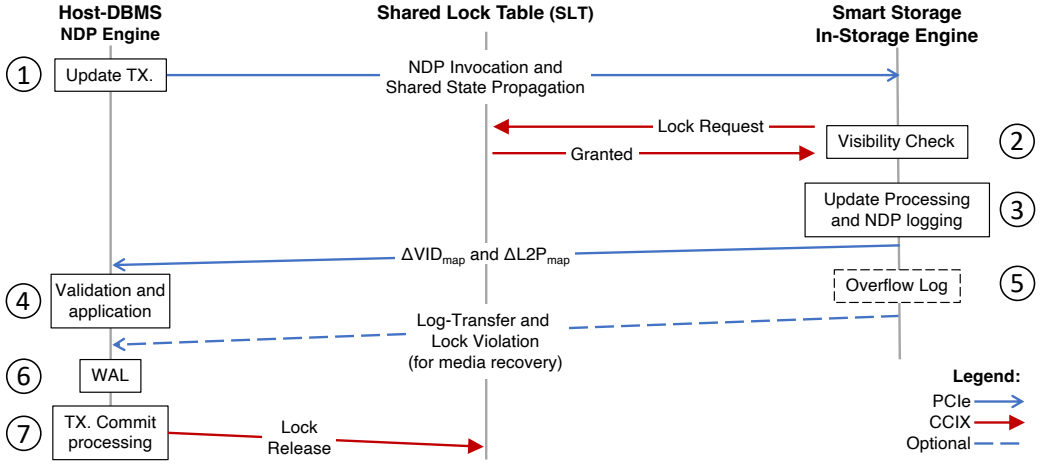


Fig. 9. The execution of an NDP modification operation.

4.1 NDP Transaction Management

4.1.1 NDP Offloading Decisions. A naïve approach, funneling all operations to smart storage, may not be suitable for all types of workloads. In OLTP, for example, frequent low-latency transactions operating on the newest data mandate caching it in the fast host memory buffer. Whereas, OLAP style transactions result in a lot of data-transfers from storage, which causes buffer pollution and impedes OLTP performance. By processing fast OLTP updates on the host and offloading OLAP and larger modifications to smart storage, we can maintain high OLTP performance, while reducing data movement and accelerating OLAP queries, modifications, or mixed workloads.

However, query plan optimization is non-trivial, and accounting for the different compute and I/O characteristics on host systems and smart storage further complicates the decision process to determining, which operations from what queries or transactions to offload. Given the ad hoc nature of queries, such choices must be automated during query optimization. To enable automated offloading decisions, we employ an NDP plan optimizer that takes the initial host-only query execution plan and determines, which operations or whole sub-plans (NDP pipelines) are better suited for offloading to the NDP engine, by comparing the relative costs. The optimizer is based on hybridNDP’s [53] decision cost-model. It incorporates a HW-model that provides configurable device characteristics, which we adjust to match our hardware architecture. neoDBMS extends the model, by injecting the relative modification costs based on the estimated selectivities. This way, modifications with low costs for determining the qualifying tuples (e.g., matching index and low estimated selectivity) that update just a few tuples get assigned to the host-engine, while large(bulk)-updates or modifications with high costs to determine the qualifying-tuples get assigned to the NDP-engine.

4.1.2 NDP Invocation and Shared State Propagation. Before the NDP-engine offloads any NDP operation or pipeline, neoDBMS *prepares* the NDP execution (Figure 9.①). First, it acquires the transaction context, which comprises the unique transaction ID (TxID) of the NDP transaction and the list of all concurrent transactions, both of which are necessary for in-storage execution and on-device snapshot creation. Next, an NDP invocation is constructed, which comprises the transaction context, the involved DB objects, and the operations to be executed. During the NDP invocation *pushdown*, neoDBMS snapshots the current shared-state and propagates it alongside the call (Section 3.1).

During the preparation, neoDBMS also *estimates* and allocates the storage and on-device resources necessary for the invocation. The estimation is performed based on the expected selectivity and calculated as part of the query optimization step (Section 4.1.1), the necessary compute resources or cache/FPGA-BRAM, and the current device load. Storage space is provided as new DB pages, reserved and/or allocated for the pending NDP invocation. The NDP space management in neoDBMS is performed solely by the host-side DBMS engine, and therefore, if the NDP execution runs out of storage, it requests additional pages from the DBMS at the cost of an extra roundtrip. We measured additional roundtrip costs of 2–10ms, depending on the number of invoked PEs. Noticeably, neoDBMS configures and allocates on-device resources for each NDP call.

Once the NDP invocation is performed, the worker threads of the NDP engine get suspended and sleep-wait for the response of the in-storage engine (Section 4.1.5). The NDP engine can perform further invocations if new NDP transactions arrive. Noticeably, *parallel NDP invocations execute against their own on-device snapshots with transactional guarantees*.

4.1.3 In-Storage Engine on Smart Storage. NDP transaction processing on smart storage begins with the arrival of the NDP invocation command (Figure 9.②). The *in-storage engine initializes* the execution state by evaluating the NDP command to determine the operations to be executed and the involved DB objects. A log-entry comprising all NDP invocation parameters is created first, marking the start of the NDP execution. Moreover, required HW modules (i.e., ccSVM, preloader, and unloader) are configured and activated. Next, the *in-storage engine* schedules the workload across the estimated number of PEs specified in the invocation. This is done by partitioning the on-device VID_{map} , as well as distributing the pre-allocated storage uniformly over all PEs.

Once the initialization completes, the in-storage engine launches the NDP modification as partitioned *NDP jobs* across the number of PEs specified in the invocation. The PEs run generic binaries, which are instrumented by the NDP jobs with the NDP (pipeline) operations to be executed and parameterized with the NDP invocation settings. The current neoDBMS design only supports scans, selections, projections, and updates as NDP operations.

The NDP jobs start by performing a visibility check for each VID_{map} entry (tuple's entry-point in the version-chain) in their partition (Figure 9.②). Visible versions are passed through a filtering operation while invisible ones are disregarded, i.e., deleted tuples and their tombstone records, or versions created after invocation. To verify if a tuple meets the update conditions, the whole version-record must be fetched from NVM storage. Tuples satisfying the filter condition are passed to NDP update processing.

4.1.4 In-Storage Update Handling. Before any modification to tuples is permitted, a tuple-lock must be acquired (Figure 9.③). Each PE, as well as the host, has access to neoDBMS's SLT. We elaborate on the SLT and our locking protocol in Section 5.

After successfully acquiring SLT locks, the in-storage modifications are performed, resulting in the creation of new version-records that are placed in a fast (BRAM-based, Figure 8) scratchpad memory, the size of a DB page. Once it gets full, it is flushed and persisted at the physical location pointed to by the $L2P_{map}$ and the PE initializes a new DB page from the pre-allocated storage assigned at NDP invocation time. All map changes (ΔVID_{map} , $\Delta L2P_{map}$) during the NDP operation are continuously logged and persisted as part of the BRAM page flushing process. A ΔVID_{map} entry is small and contains the VID, the RecID of the predecessor version, and the RecID of the newly created version. Noticeably, modifications to the mappings are not immediately applied, instead they are logged and kept solely on device, which is critical for the validation of potential abort steps to follow.

Upon completion of all NDP jobs, the in-storage engine determines the sizes and addresses of the on-device logs and map changes, and returns them back to the host engine as optimized PCIe DMA transfers.

4.1.5 NDP Update Validation and Application. Depending on the return status of the in-storage engine, the host-side NDP engine takes over and either commits or aborts the NDP transaction or retries the NDP invocation. Upon *success*, the ΔVID_{map} is validated to detect potential update conflicts (Figure 9.④). In absence of conflicts, the NDP engine applies the $\Delta L2P_{map}$ and ΔVID_{map} to their host-side counterparts. Since NDP transactions only use new and exclusively reserved DB pages, the $\Delta L2P_{map}$ can be safely applied since no other transactions can access them. Afterwards, entries in the ΔVID_{map} are applied atomically, whereas locks are still held. At this stage, VID_{map} points to the new on-device tuple version-records, whose RecordIDs can be resolved through the $L2P_{map}$ and loaded from smart storage, if necessary.

4.1.6 Logging. In-storage modifications go to the storage space specifically allocated for the NDP invocation, leaving old versions intact. This mode is fast, ensures recovery from system failures, and is favorable to applications that can tolerate windows of media non-recoverability. However, systems that require protection against media/device failures mandate transferring the NDP-logs to the host. To this end, neoDBMS introduces an overflow-log (Figure 9.⑤) mechanism that integrates into the host-side WAL (Figure 9.⑥) and is described in more details in Section 6.

However, log movement entails significant *drawbacks* as it lies on the critical commit path, and leads to delays in the commit processing, lock contention, and ultimately lower throughput. To this end, neoDBMS extends its locking protocol with lock-violation techniques. They enable subsequent transactions to acquire, but also violate locks held by NDP transactions to interleave NDP log movement with useful work. The extended locking protocol is discussed in Section 5.4.

4.1.7 NDP Transaction Commit Processing. In neoDBMS, both host and NDP transactions can only be committed at the host side, against the host-DBMS (Figure 9.⑦). All unused pre-allocated pages are marked free and can be reused in the next call, while on-device logs are asynchronously transferred and integrated with the host log (Section 6). Finally, all acquired SLT and HLT locks are released, waking up all waiting transactions.

4.1.8 NDP Transaction Abort Handling. If an abort condition occurs during NDP invocation, the in-storage engine will initiate an abort routine. This is the case when, in a *First-Updater-Wins* [12, 19] setting, a host TX that started after the NDP transaction manages to lock a tuple, update it and commit, before the NDP update can lock it on-device.

During *abort handling*, all PE jobs from the current NDP invocation get notified. Furthermore, the in-storage engine notifies the NDP engine of the abort, eliminating the need to transfer $\Delta L2P_{map}$ and ΔVID_{map} , as well as the NDP logs. No changes need to be applied to the host-side maps, and therefore, no invalidation takes place. The NDP modifications made on-device are discarded, to restore the state prior to NDP invocation. This is done by releasing and reclaiming the storage space that exclusively allocated and assigned to the NDP invocation.

Although an NDP invocation may be successful, subsequent follow-up operations at the host (if any) might encounter failures. In such instances, the host NDP engine executes a standard rollback, reverting modifications made on the host side by replaying undo-records from the host-log, while the NDP part is rolled back as described above: all changes to $\Delta L2P_{map}$ and ΔVID_{map} are discarded, and exclusively allocated space is released and reclaimed.

5 Locking and Shared Lock Table

We now describe the SLT and its integration in smart storage and in the host DBMS, as well as lock management. We begin by highlighting the organization of the SLT within neoDBMS. We then describe our integrated (DBMS-SLT) lock-table design and a novel integrated locking protocol, followed by an extended locking approach that allows the interleaving log-movement and flushing with useful work.

5.1 SLT Organization in neoDBMS

The SLT is shared between the host and smart storage devices. It is accessed by the DBMS through simple software interfaces, whereas the smart storage uses a dedicated hardware module designed to handle frequent lock requests, including hashing and status monitoring, which in turn are translated into CCI requests. The SLT has a *ccSVM-aware* design that facilitates on-device virtual address translation and mitigates CCI address-translation (ATS) overhead (Section 2.2.1). Firstly, the SLT is allocated on the host-side in physically contiguous and pinned memory, resulting in cached local ccSVM accesses, minimal overhead, and reduced lock latencies for all host-transactions. Therefore, enabling the high computational capabilities and parallelism of the host to handle the more frequent lock requests efficiently. Secondly, as the smart storage is slower relative to the host, it will access ccSVM less frequently and mitigate CCI address-translation by buffering its address range in the CCI address caches (e.g., *CCIX ATC*). Lastly, by deliberately designing a small SLT, it aids in caching ccSVM remote accesses and reducing cache misses.

The SLT is organized as a hash table. Each SLT entry (bucket) represents a 16B fixed-sized queue that provides eight or sixteen lock slots, with slot sizes of 2B or 1B. These sizes are aligned to the 16B-sized *CCIX* and *ARM atomics*. However, they are also aligned to the 8B x86 atomics possibly with a 1B slot-size. Each lock slot stores either (a) a host-shared lock with a generic id ($TxID_{HOST}$); or (b) a currently assigned NDP JobID/ $TxID_{NDP}$; or is (c) left empty as a free lock queue slot. All lock requests for a tuple are managed within a single SLT entry/bucket, which is organized as a queue. The SLT employs a hash function using a combination of the DB-object number and the tuple's VID.

The head-slot of an SLT entry represents the transaction currently *holding* the lock, while the other slots hold the $TxIDs$ of transactions waiting to *acquire* the lock and therefore represent pending lock requests. Whenever a lock is released, the entire queue is shifted by one slot to pass the lock to the next transaction. Race-free placement and the release of locks is ensured by atomic CAS operations, supported by the HW module and *CCIX* itself. The last slot in the queue is always reserved for host transaction locks ($TxID_{HOST}$), see Sect. 5.2 for details.

Thus, given a 2B SLT slot-size, our SLT design supports: (a) $2^{16} - 2$ concurrent NDP transactions without host contention; and (b) max. 4/7 concurrent NDP transactions accessing the same tuple (Figure 11). Alternatively, a 1B slot-size yields: (a) $2^8 - 2$ concurrent NDP transactions without contention; and (b) max. 7/15 concurrent NDP transactions accessing the same tuple. This suffices given the tradeoff between their long-running nature, the weak smart storage PEs, and the fast execution requirement. One important aspect of the SLT design is that it does not limit the number of concurrent host transactions.

5.2 Integrated Lock-Table Design (ILT)

neoDBMS introduces an ILT design that combines the SLT and the **host lock-table (HLT)** as a logical chain (Figure 10). The SLT represents the logical *head*, whereas some of its entries may correspond to the sequence of an unlimited number of entries in the HLT, which is the logical *tail*. The host lock-manager operates on both SLT and HLT, while the in-storage engine only uses the SLT. This design allows neoDBMS to tackle several problems. On the one hand, we must account

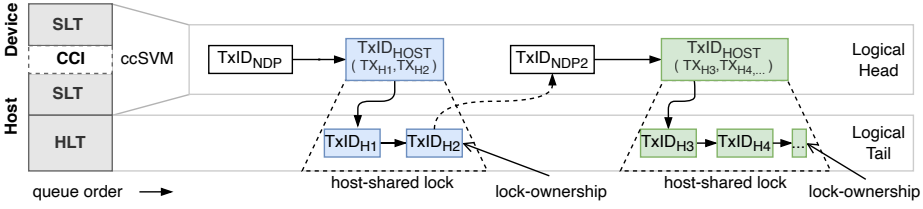


Fig. 10. SLT as head and HLT as tail forming a logical chain. Host transactions can share and reuse a generic SLT host lock ($\text{TxID}_{\text{HOST}}$) as long as no concurrent NDP operation requests the same lock, to avoid limiting host parallelism. The last host transaction able to reuse a host-shared lock claims ownership and is responsible for releasing the lock in the SLT. The figure depicts the lock table queue for transactions seeking to acquire a lock on a tuple t in the following order: Tx_{NDP} , Tx_{H1} (host), Tx_{H2} (host), Tx_{NDP2} , Tx_{H3} (host), Tx_{H4} (host).

for the much higher host parallelism and allow for a large number of host-locks, while keeping the size of a lock queue small. On the other hand, the lock manager of neoDBMS must minimize the overhead of tracking the relationship between SLT locks and the large number of host transactions. A naïve design, placing all pending host lock requests for a tuple t in the SLT, would rapidly deplete the few (7/15) available SLT slots, limiting host parallelism and causing incoming host transactions to abort. Another advantage of the design is its adaptability to other locking mechanisms. Since HLT remains unchanged and SLT is merely an extension in the logical chain, HLT’s functionality remains consistent across locking mechanisms.

Host-Shared Locks. To address both issues, neoDBMS introduces the concept of a host-shared lock ($\text{TxID}_{\text{HOST}}$). Any NDP operation seeking to update a tuple t requests an SLT lock and the in-storage engine enqueues its TxID_{NDP} .³ Concurrent host transactions seeking to modify the same tuple enqueue their lock requests in the HLT, but most importantly, they also place a host-shared lock request (i.e., $\text{TxID}_{\text{HOST}}$) in the SLT, if it does not already exist. Thus, all host transactions share the same host-shared lock $\text{TxID}_{\text{HOST}}$, as long as no concurrent update NDP operation requests the same lock.

Consider the example shown in Figure 10. Initially, an NDP transaction Tx_{NDP} requests and acquires an exclusive lock on t . Since operations of NDP transactions are slow relative to the frequent low-latency host transactions, multiple host lock requests for t will arrive while Tx_{NDP} is active. These host transactions (e.g., Tx_{H1} , Tx_{H2}) will share the same $\text{TxID}_{\text{HOST}}$ and utilize a single SLT lock. The actual host locks and, more importantly, their order are managed by the HLT. If now another on-device operation from a second NDP transaction Tx_{NDP2} requests a lock on t , $\text{TxID}_{\text{HOST}}$ cannot be shared among follow-up host transactions (e.g., Tx_{H3} , Tx_{H4} , etc.) anymore. To account for such cases, another host-shared lock (a second $\text{TxID}_{\text{HOST}}$) is enqueued that corresponds to the respective HLT entries for Tx_{H3} , Tx_{H4} , and so on.

Insight: The use of a host-shared lock ($\text{TxID}_{\text{HOST}}$) that is *shared* among all pending host transactions, instead of the specific host TxID is a key design decision in neoDBMS. Firstly, it allows neoDBMS to avoid tracking the relation between host transactions and SLT-locks. Secondly, it prevents limiting the number of concurrent host transactions, at the acceptable cost of two host-side lock-table lookups: one in the SLT, and potentially a second one in the HLT. Lastly, it results in a significant reduction in lock- and SLT-sizes.

³We speak of TxIDs , yet physically the in-storage engine enqueues the JobID instead of the TxID_{NDP} , since the use of the 8B TxID_{NDP} is infeasible, because of the 2B/1B SLT slot size limitation. However, the conceptual difference between both is negligible, since a JobID corresponds to exactly one TxID_{NDP} .

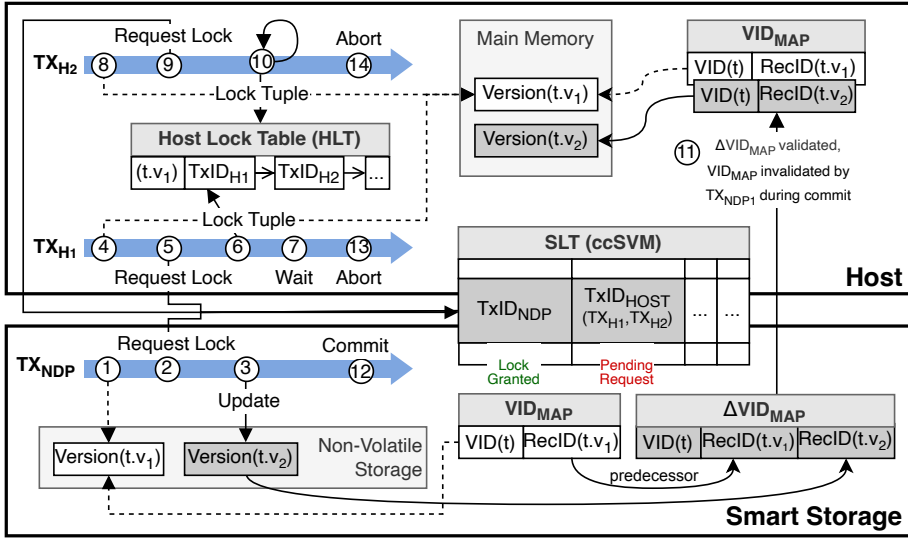


Fig. 11. Concurrent modifications of NDP and host transactions synchronize by requesting an SLT lock before proceeding. Race-free lock placement is ensured through atomic CAS operations and cache-coherent synchronization over CCI. ΔVID_{map} is transferred back to the host where it is applied to the host map, thus invalidating old host-side versions.

Lock Ownership. Releasing a host-shared lock poses another problem, as it is unclear when to remove the $TxID_{HOST}$ from the SLT or which host-transaction will acquire the lock next. To this end neoDBMS introduces the concept of *lock ownership*. If a $TxID_{HOST}$ is shared among multiple host transactions, the ownership of the current lock is passed on to the next host transaction as soon as its lock request is placed in the HLT. Each host-shared lock is managed by the HLT, which also forms a queue of host transactions waiting to acquire a lock on t . The ownership of the host-shared lock is always claimed by the last host transaction, corresponding to that $TxID_{HOST}$. In turn, releasing a host-shared lock is only permitted for transactions with ownership (i.e., the last entry in the HLT queue for the given host-shared lock). Concurrent update NDP lock requests break the reuse chain of host-shared locks. That is, upon arrival of a new NDP transaction requesting to lock t , a new $TxID_{HOST}$ slot will be appended in the queue (Figure 10, second $TxID_{HOST}$), with a new lock-owner to ensure that preceding host-shared locks are released first according to their order.

5.3 Integrated Locking Protocol

Consider a TX_{NDP} seeking to update tuple t (Figure 11.①). TX_{NDP} requests an SLT lock before proceeding with the update (Figure 11.②). The lock queue for tuple t is obtained via an atomic load operation of the SLT entry (single cacheline access) [6]. As the head-slot of the SLT entry is still empty (i.e., no TX holding the lock), TX_{NDP} inserts its $TxID_{NDP}/JobID$ at the head-slot and updates the SLT entry via an atomic CAS operation, which is coherently synchronized with the host over CCI. On success, the lock is immediately acquired and the update commences (Figure 11.③). While, initially the entry-point of tuple t in the VID_{map} points to the physical address of the latest committed version t, v_1 (Figure 11.①), the update creates a new version t, v_2 , and inserts a ΔVID_{map} entry for the modification.

Now consider a concurrent host transaction TX_{H1} that reads tuple t (no lock required) and then attempts to update it (Figure 11.④). As the host VID_{map} still points to t, v_1 , TX_{H1} is unaware of

the TX_{NDP} changes. TX_{H1} requests an SLT lock (Figure 11.⑤), which is not granted since the head-slot is already taken. Instead, TX_{H1} is enqueued into the next free slot by placing a host-shared lock ($TxID_{HOST}$) there, in lockstep placing another lock into the HLT (Figure 11.⑥). Furthermore, TX_{H1} also obtains the lock ownership, as it is the first host lock on tuple t . At this point TX_{H1} registers to be notified about the commit or abort of TX_{NDP} (preceding SLT lock), and sleep-waits (Figure 11.⑦), thus avoiding continuous queue status-polling.

To illustrate the utility of lock ownership and the generic $TxID_{HOST}$ lock, consider yet another host transaction TX_{H2} that likewise reads and then attempts to modify t (Figure 11.⑧). Its SLT lock request (Figure 11.⑨) fails again, but instead of enqueueing another lock request, TX_{H2} reuses $TxID_{HOST}$ by taking lock ownership and preventing previous owners from releasing the SLT lock. In a lockstep, TX_{H2} attempts to acquire the HLT lock (Figure 11.⑩) and blocks, as it is already granted to TX_{H1} .

Commit processing. Now, let us consider the commit process of TX_{NDP} (Figure 11.⑫). NDP transactions are initiated by the host-side DBMS and also commit there. Upon the completion of the NDP call, the ΔVID_{map} and $\Delta L2P_{map}$ are first transferred from device while data pages containing updated tuples remain on storage.

Next, an important validation step takes place (Figure 11.⑪), that ensures that conflicts with prior lock-acquisitions are detected, by comparing the predecessor of $RecID(t.v2)$ in the ΔVID_{map} ($RecID(t.v1)$) against the current host VID_{map} entry ($RecID(t.v1)$). Clearly, in absence of conflicts, both should match. Such conflicts can occur, for example, if a host transaction TX_2 (not depicted in Figure 11) that started after TX_{NDP} , manages to update a tuple t and commit, before the TX_{NDP} is able to lock t . Thus TX_2 will create $t.v2$ and update the host VID_{map} to $RecID(t.v2)$. Since the shared-state for TX_{NDP} is already propagated, but does not include the changes of TX_2 , and TX_2 's lock on t is already released, TX_{NDP} can lock t and create an anomalous $t.v2$. In absence of conflicts commit processing commences, the ΔVID_{map} and $\Delta L2P_{map}$ are applied *atomically* to the host maps, thus invalidating the old host-side version RecordIDs (Figure 11.⑪). Furthermore, a commit log record is placed in the host log, which eventually gets flushed. At this point the SLT locks held by TX_{NDP} are released.

Subsequently, TX_{H1} is notified and wakes up, as it now obtains both SLT and HLT locks. Before updating t , TX_{H1} verifies the validity of its initial read access by comparing the RecordID (Figure 11.④), against the current VID_{map} value. It detects a mismatch and initiates an abort, as TX_{NDP} has already invalidated TX_{H1} version of t . TX_{H1} does not release the $TxID_{HOST}$ locks in the SLT as it lacks ownership. However, all granted HLT tuple locks are released, resulting in the notification of TX_{H2} , which continues and detects the same mismatch. The abort handler of TX_{H2} releases the HLT locks and the $TxID_{HOST}$ lock, as it is now its owner. As the host NDP-engine is now aware of the device modifications created in-situ, any follow-up host transactions accessing t will look up its VID that now refers to $t.v2$, which is loaded from smart storage.

5.4 Extended Locking Protocol.

5.4.1 Lock Contention and Log-Movement. To ensure media recovery, NDP modifications necessitate transferring the log that protects them from smart storage failure, as well as integrating and flushing it to the main DBMS log. (We describe logging and recovery in more detail in Section 6.) However, log-movement and log-flushing are I/O intensive and slow. They pose a major *challenge* since they lie on the critical commit path, i.e., the commit of a transaction cannot be acknowledged and its locks cannot be released before its commit log-record is written (hardened) to persistent storage, which results in commit-delays, lock-contention, and ultimately throughput drops. This effect is especially prominent in update NDP settings, as potentially many fast-paced host

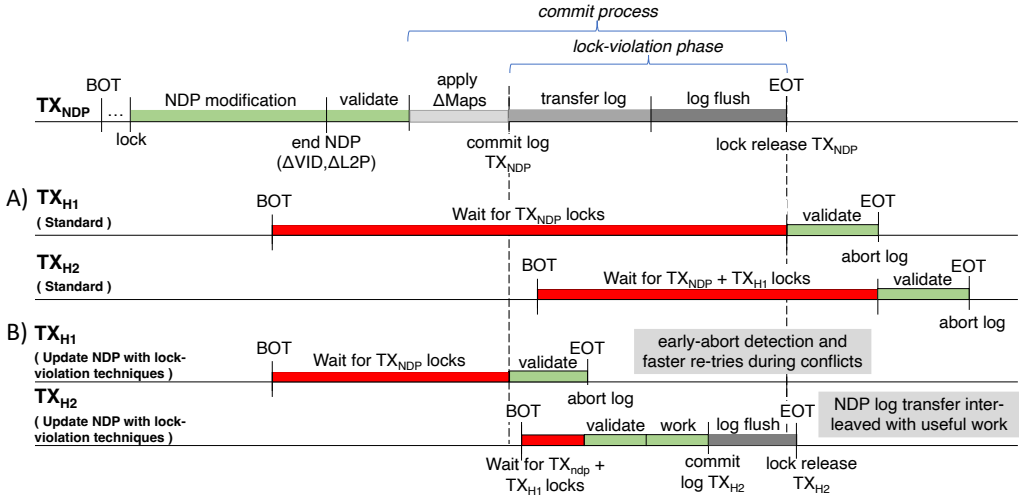


Fig. 12. Naively transferring on-device logs after an NDP modification (so-called overflow log) to ensure media recovery, needlessly prolongs lock-duration and increases contention for hot-tuples (A). By applying controlled lock violation techniques, useful work and log-movement can be overlaid, reducing lock duration and decreasing contention (B).

transactions cannot acquire locks on NDP-modified records and perform useful work in parallel to log-movement. Furthermore, there is a pronounced impact on access hotspots since hot-records cannot bounce between the host and smart storage, even though the NDP-modification is complete, because of the prolonged lock-duration caused by log-movement.

An illustrative example is shown in Figure 12(A). An NDP transaction TX_{NDP} updates a tuple t among others, while subsequent host transactions TX_{H1} and TX_{H2} seek to update t as well. Under standard approaches (Figure 12(A) TX_{NDP} will hold all its locks during the slow log-movement and flushing phase. As a result, TX_{H1} and TX_{H2} that wait to update the same tuples must abort past the successful end of TX_{NDP} .

5.4.2 Lock Violation Techniques. In pursuit of a general solution, update NDP employs a locking technique inspired by **controlled lock-violation (CLV)** [36]. The core idea is to interleave log-movement and -flushing with useful work by allowing subsequent transactions to acquire locks, violating the locks still held by the NDP transaction until its end, as soon as the NDP transaction initiates commit-processing and allocates its commit-record in the log buffer. (The NDP transactions hold all the locks until their end.) To preserve correctness, lock-violation introduces a commit-dependency, i.e., the new transactions that acquire the conflicting locks can perform useful work, but are only allowed to commit *after* and *if* the NDP transaction completes successfully, and therefore they form a commit-chain. On the downside, in case of a system failure, all these dependent transactions will be forced to rollback during recovery.

The lock-violation techniques introduced in Update NDP (Figure 12(B)) allow host transactions TX_{H1} and TX_{H2} to proceed as soon as the commit-record of TX_{NDP} is allocated in the log. However, TX_{H1} will abort early since t has been updated by TX_{NDP} and the validation phase will fail. Interestingly, TX_{H2} can proceed, violate TX_{NDP} 's locks, and perform useful work during the log-movement and flushing phase of TX_{NDP} . Furthermore, not only will transactions such as TX_{NDP} and TX_{H2} commit close to each other, but also their commit log-records will be flushed in the same I/O.

This extended locking technique is especially favorable for hot-tuples and skewed data-access distributions, i.e., when contention hotspots between NDP and host transactions occur. It makes it possible for hot-tuples to bounce between device and host, but also between different NDP transactions on-device.

5.4.3 Extended Integrated Locking Protocol. We now describe how the locking protocol introduced in Section 5.3 must be extended for lock violation.

Protocol description. Consider TX_{NDP} , aiming to update some tuple t (Figure 12(B)). Similar to the basic locking protocol (Section 5.3), before modifying the version $t.v_1$, which is the current entry-point in the VID_{map} , an SLT lock is acquired for t . Next, TX_{NDP} updates $t.v_1$ and creates a new version, $t.v_2$. Furthermore, the VIDs of both $t.v_2$ and its predecessor $t.v_1$ are stored as a new entry in the ΔVID_{map} . Pre-allocated pages reserved for the *overflow log* are used to log the changes to $t.v_1$. After NDP completion, the $\Delta L2P_{map}$ and ΔVID_{map} are transferred back to the host, and the predecessor VID of $t.v_2$ undergoes validation against the local VID_{map} to identify concurrent host changes during NDP modification.

Upon successful validation and in absence of subsequent post-NDP operations, TX_{NDP} can begin commit processing. At this point, TX_{NDP} cannot abort anymore under normal circumstances. Consequently, under the extended locking protocol TX_{NDP} applies the $\Delta Maps$, allocates a slot in the commit log, and updates its transaction state, permitting lock violation. This newly introduced state lets TX_{NDP} behave as if it were already committed. However, all transactions waiting to modify t get notified and are allowed to acquire locks and proceed with modifications (while TX_{NDP} continues with commit processing and awaits the completion of the device-log transfer and flush), but must enter a commit-dependency with TX_{NDP} , i.e., their commit will not be acknowledged until the EOT of TX_{NDP} . Upon completion, TX_{NDP} releases all SLT locks and proceeds to commit, subsequently updating the transaction state to *committed* and logging the **end-of-transaction (EOT)** log record.

Illustrative Example. Consider a concurrent host transaction TX_{H1} (Figure 12(B)), aiming to modify t , that starts before the lock violation phase of TX_{NDP} . At this point, TX_{H1} will only read $t.v_1$, as Snapshot Isolation mandates that only versions committed prior to TX_{H1} are visible to it, regardless of whether TX_{NDP} has already applied its $\Delta Maps$. TX_{H1} will request an SLT lock on t , which will not be granted, as TX_{NDP} still holds the lock. Instead, the NDP-engine queues up the lock request and TX_{H1} enters a waiting state. When TX_{NDP} enters the lock violation phase, TX_{H1} is notified and wakes up to acquire the HLT tuple lock on t . TX_{H1} violates the SLT lock still held by TX_{NDP} , which is now explicitly permitted. Before updating t , TX_{H1} must certify that the tuple version, read prior to locking still matches the entry point of t in the VID_{map} . However, TX_{H1} must abort, as $t.v_1$ no longer corresponds to the current entry in the VID_{map} for t , since it was updated and therefore invalidated by TX_{NDP} during the ΔVID_{map} application. Subsequently, TX_{H1} releases its SLT and HLT locks and aborts.

Consider now yet another concurrent host transaction TX_{H2} (Figure 12(B)), aiming to modify t , which this time starts after TX_{NDP} has and entered the lock violation phase and has applied its ΔVID_{map} . Therefore, TX_{H2} can already access $t.v_2$. It requests an SLT lock on t , which is now granted as TX_{NDP} allows SLT lock violation. In a lockstep, TX_{H2} also acquires a HLT tuple lock on t (already released during the abort of TX_{H1}) and successfully certifies that the previously accessed tuple version of t ($t.v_2$) still matches the entry point of t in the VID_{map} . Subsequently, TX_{H2} proceeds with the update to create $t.v_3$, and then initiates commit processing, allocating a commit log-record. However, the commit is not acknowledged, due to the commit dependency to TX_{NDP} . Once TX_{NDP} completes, TX_{H2} is notified and resumes commit processing, releases locks and completes.

Benefits for concurrent NDP transactions. The proposed lock-violation techniques are also favorable to concurrent NDP transactions and can be realized through minor extensions. Once TX_{NDP} applies

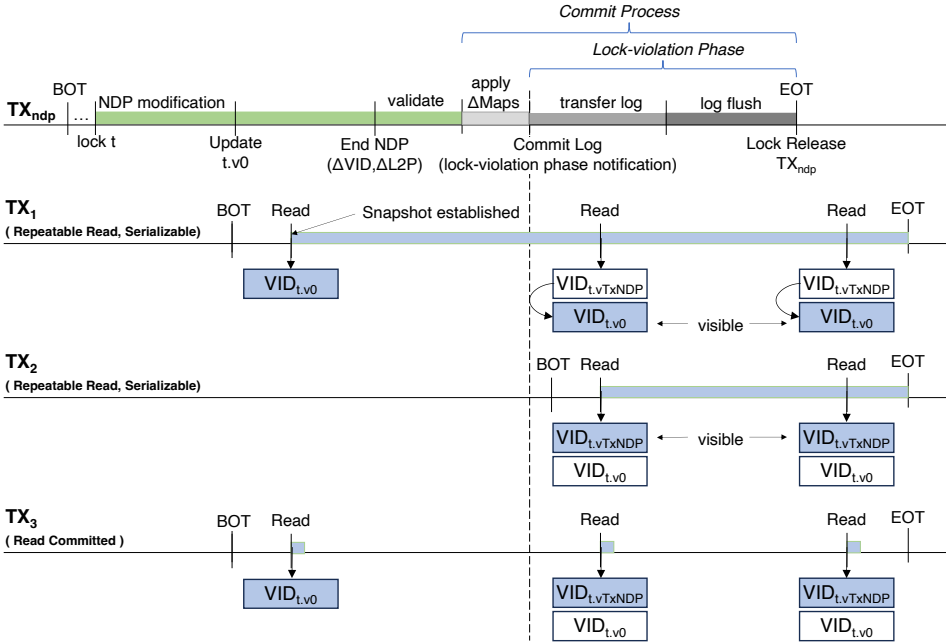


Fig. 13. Snapshot isolation and different isolation levels in neoDBMS during lock-violation.

its ΔVID_{map} and $\Delta L2P_{map}$ on the host, they integrate into the shared state, which is propagated and cached on device during subsequent NDP invocations along with the lock-violation state of TX_{NDP} . As a result, a subsequent NDP operation can compute a fresher snapshot, where changes made by TX_{NDP} are visible and identify tuple versions that are solely visible due to lock-violation. Noticeably, when one of these tuple versions is utilized during NDP processing, the in-storage engine will notify the host NDP engine about the discovered commit dependencies and provide a list of transactions to construct the commit-chain. Furthermore, if this NDP operation requests an SLT lock and the current lock-holding transaction permits lock-violation, while being the predecessor in the SLT queue, it can continue processing, violating the lock.

5.4.4 Lock-Violation and Transaction Isolation Levels. Lock-violation techniques prove to be well-suited for snapshot isolation within neoDBMS. Transactions with stronger isolation levels such as REPEATABLE READ or SERIALIZABLE ensure consistent reads within a transaction, as the snapshot is established by the initial read and remains unchanged during subsequent operations (Figure 13). Consequently, a transaction that establishes a snapshot before the lock-violation phase, remains oblivious to changes in it during execution. However, these transactions may still violate the SLT locks of transactions permitting lock-violation, and reassess whether they need to abort due to conflicts. This mechanism allows for *early-abort* and faster retries without waiting for NDP log-processing to complete. However, if such a transaction (REPEATABLE READ or SERIALIZABLE) establishes its snapshot after lock-violation is permitted, all its changes become visible and other transactions can perform useful work instead of waiting. Accesses to tuples solely visible due to the lock-violation state result in a commit dependency, enforced upon any kind of modification. That said, neoDBMS relies on PostgreSQL's **Serializable Snapshot Isolation (SSI)** [74] implementation to detect conflicts necessary for serializability.

Weaker Isolation Levels. Transactions operating with weaker isolation levels, such as READ COMMITTED, may establish a transactional snapshot before or after gaining lock-violation

permission (Figure 13). In both cases, lock-violation is permitted, at the cost of commit-dependencies. However, in the former case, tuple versions may already be invalidated, due to the “older” snapshot, which results in *early-aborts* during modifications. In the latter case, access to tuples visible due to lock-violation forces transactions to delay their completion until all commit-dependencies are resolved first.

5.4.5 Discussion. Interestingly, *the modifications of NDP operations in neoDBMS are atomic.* For one, modifications yield a new version record that is placed in exclusively allocated page ranges, leaving the old version unchanged. For another upon completion the in-storage engine returns the $\Delta L2P_{map}$ and the ΔVID_{map} (Section 4), which are applied *atomically* at the host-side. As a result, NDP modifications are either successfully persisted and exposed to the host as a whole, or, in case of abort or a failure, the host NDP-engine can always resort to the state prior to the NDP invocation, and simply discard all modifications. As a result, neither partial NDP modifications, nor their partial application are possible in neoDBMS.

While this delays their visibility to the host, it also has implications on logging. The primary drawback lies in the long log-flush and transfer times, leading to commit delays and prolonged lock contention. To this end, update NDP introduces lock-violation techniques to interleave useful work with NDP Log transfers, which minimizes the lock contention and removes the impact on host performance. Furthermore, lock-violation is favorable for hot-tuples and skewed data-access distributions, i.e., when hotspots between NDP and host transactions occur. It makes it possible for hot-tuples to bounce between device and host, but also between different NDP transactions on-device.

6 Logging and Recovery

NDP modifications mandate novel logging approaches to ensure consistent rollback and recovery from system failures, as well as media recovery as smart storage devices may fail. We now describe how *update NDP* addresses these questions in neoDBMS with two alternative approaches.

We set off by reiterating that NDP modifications are atomic under update NDP and neoDBMS and therefore partial NDP modifications or their application are impossible. Somewhat confoundingly, as a result *NDP modifications necessitate neither undo-, nor redo-logging for transaction rollback or system recovery* (except for the host part).

6.1 Basic Logging, Rollback, and Recovery

Based on the above observation, we first introduce a basic logging approach, which is simple and fast, but ensures *only* transaction rollback and recovery from system failures.

neoDBMS preserves and marginally extends host-side **write-ahead logging (WAL)** for NDP transactions (Figure 14(A)). At the start of an NDP call, the NDP engine creates a log-record of a new type – *Start-NDP* containing all call parameters. The **log-sequence number (LSN)** of that record is written both in the host-log and is propagated alongside the NDP call.

During the NDP call and while performing the NDP modifications, the in-storage engine gathers the changes to the on-device tables (address mapping table $L2P_{map}$ and the VID table VID_{map}) in the $\Delta L2P_{map}$ and ΔVID_{map} . Upon the successful completion of the NDP invocation, both are then returned to the host NDP engine with few DMA transfers with low overhead. At the same time, an *End-NDP* log-record, that contains both $\Delta Maps$ and references the $LSN(Start-NDP)$, is written in the host-log. Please remember that the $\Delta L2P_{map}$ and ΔVID_{map} are applied atomically to the main maps only during commit processing (Section 5.3), i.e., before the commit log record is placed in the log buffer.

Recovery from system failures. During the *redo* phase the log is replayed in a forward manner. Since the smart storage is not corrupted, the NDP transactions are committed, and the pages

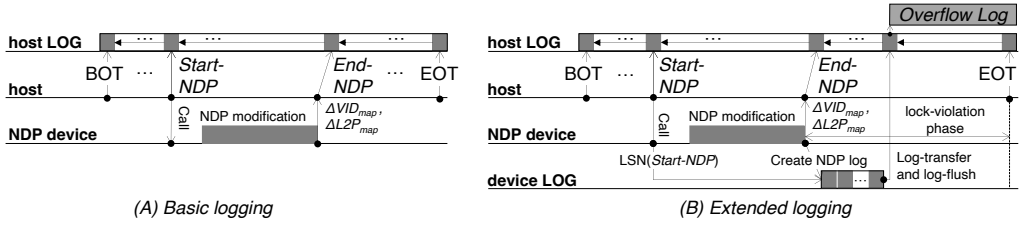


Fig. 14. Host- and on-device logging in neoDBMS.

containing the NDP modifications are intact, only the $\Delta L2P_{map}$ and ΔVID_{map} are applied to the main maps to point to the correct pages and version records.

Summary. The basic approach is simple as it requires marginal changes to the logging and recovery management. It is also fast and incurs low overhead since, as we will show later on, the $\Delta L2P_{map}$ and ΔVID_{map} are small (as they only contribute $<0.7\%$ of the total I/O as we will later show in Figure 2) and can be transferred efficiently over the hybrid interface with a few large DMAs to utilize the PCIe bandwidth, while gathering the $\Delta Maps$ is efficient. Its main disadvantage is that it fails to ensure media recovery. It is, however, applicable to systems that ensure media recovery otherwise or to applications that can tolerate windows of media non-recoverability.

6.2 Extended Techniques and Recovery from Smart Storage Device Failures

To ensure *media recovery* upon a smart storage device failure, neoDBMS extends the logging and the commit process (Figure 14). Firstly, upon the completion of an NDP invocation, an NDP redo-log is asynchronously constructed on-device. To this end, the in-storage engine relies on the new pages and the $\Delta Maps$. The log is then sent asynchronously back to the host (Figure 14(B)), prior to the commit of the NDP transaction. Secondly, it is then stored in an *overflow log-file*, referenced by the *End-NDP* log-record, thus avoiding expensive log-merges. This separation allows more flexibility in log design and physical formats, enabling the development of more portable NDP-log readers and writers independent of DBMS specifics. The process has low overhead as it uses few large DMA data transfers over the bandwidth-optimized hybrid-interface (PCIe). However, it increases the log size. In the event of a *device failure*, the host NDP engine applies the overflow log as part of the redo-phase to *redo* the modifications of committed NDP transaction.

7 Optimizations

7.1 Custom Data Preloader and Unloader

The on-device visibility checking and layout access described in Sections 2.3.5 rely on block transfers. However, both types of on-device accesses can be optimized to utilize the byte-addressability of the underlying NVM storage. In-situ, this is possible since PEs can operate on physically persistent pointers and transfer precisely the byte-sequences mandated by the data layout. For instance, to check a version record (Figure 6), neoDBMS must read just 20B in five transfers, which significantly reduces the read-amplification compared with page-sized transfers. However, the high NVM access latencies [30, 44, 72, 88] relative to DRAM pose a challenge, as they additionally slow down the already slow on-device PEs (e.g., 180 MHz).

To this end, neoDBMS introduces a novel *custom byte-level preloader* [15] that facilitates interleaving compute and data transfers in-situ, allowing the PE to make the most of low compute power. The byte-level preloader enables asynchronous NDP-application-defined transfers from

```

1:      // Activate the Preloader Module
2:      preloader_init( );
3:      // Flush cached modifications and invalidate cache range.
4:      preloader_invalidate_cache( );
5:      // Push async DMA jobs to preloader
6:      preloader_set(source_addr_nvm_1, sink_addr_buffer_1, length);
7:      preloader_set(source_addr_nvm_2, sink_addr_buffer_2, length);
8:      /* Do useful work ... */
9:      // If not already preloaded, wait for DMA jobs to finish
10:     preloader_wait( );

```

Fig. 15. Example code snippet demonstrating the use of the byte-level preloader of NDP.

arbitrary addresses into PE’s BRAM (Figure 8), while the PE performs useful work. An example use of the preloader is shown in Figure 15.

Persisting in-storage modifications suffers the inverse problem, as the PE is slowed down by expensive flushes from BRAM to NVM. Yet, update processing and flush transfers can be interleaved. To this end, neoDBMS employs a page-level *unloader* [15] HW module that relies on BRAM double buffering and asynchronous DMA transfers to NVM. It behaves similarly to the preloader in the code snippet.

Optimizing I/O and compute is essential for efficient NDP, as we will demonstrate shortly, and both HW modules in combination with a fast scratchpad memory (BRAM) address the lack of I/O and caching capabilities of our employed Microblaze PEs. The utilization of preloading and unloading on our smart storage device are therefore integral parts of our NDP operation programming and compilation model.

7.2 In-Storage Select-for-Update

Long-running modifying operations, in combination with write-heavy OLTP workloads, have a tendency to abort due to the high update contention. To prevent unnecessary retries and reduce the abort overhead, transactions typically employ a `SELECT FOR UPDATE` technique to lock all update candidates in advance. Since this can lead to heavy data transfers, neoDBMS allows the on-device execution of NDP pipelines, combining `SELECT FOR UPDATE` and NDP update. An in-situ `SELECT FOR UPDATE` first locks qualifying tuples and constructs a list of their physical visible version-record addresses. The update operation can be safely performed at a later stage in the NDP pipeline without the risk of aborting or the overhead of a repeated visibility check.

Notably, this optimization does not require additional DB application changes. `SELECT FOR UPDATE` is a standard technique across most DBMS for transaction programming and allows neoDBMS to build upon existing statements in the code of a transaction. During NDP query optimization (Section 4.1.1), neoDBMS assign such operations to the NDP-engine and its own NDP mechanisms for offloading, and to leverage them during NDP.

8 Experimental Evaluation

We now describe our experimental setup followed by an examination of individual components through micro benchmarking and an evaluation of the overall system performance using end-to-end benchmarks.

8.1 Experimental Setup

The experimental evaluation is performed on an ARM Neoverse **N1 System Development Platform (N1-SDP)**[5] as host, equipped with 4× ARM N1-CPU cores operating at 2.6GHz and a total

Table 1. CoreMark [29] Scores [Iterations/Sec]

CPU	1 Core	4 Cores	Max. Perf.	Memory
Neoverse N1 (2.6 GHz)	20 668	79 417	79 417 (4 Cores)	Stack
MicroBlaze 64-bit (180 MHz)	374 (1 PE)	1 496 (4 PEs)	2 992 (8 PEs)	Stack
Intel Xeon Silver 4110 (2.1 GHz - 3.0 GHz)	16 046	64 116	207 630 (16 Cores)	Stack

The 4-core N1 reaches a max. performance of 79 417 [it./sec.], while the 8-core array of weak MicroBlaze PEs offers only 2 992 [it./sec.]. For comparison, we also provide the numbers for an enterprise-class Intel Xeon Silver CPU, equipped with 16 cores.

of 16GB RAM. The N1SDP is based on the same Neoverse N1 architecture that is used in Google’s TAU T2a cloud platforms. Both Amazon’s own Graviton CPUs and Ampere Altra CPUs employed in Microsoft’s Azure cloud further highlight the trend to ARM-based processors in data centers. A CCIX-capable Xilinx Alveo U280 FPGA (AU280) board serves as smart/computational storage device and is connected via CCIX-enabled PCIe Gen3 x16 to the host. Both the host and the device support CCIX, with the N1-SDP being configured as CCIX-HA (Home Agent) and the AU280 as CCIX-RA (Request Agent). In order to interface with the device, neoDBMS relies on TaPaSCo [39] for ccSVM interactions, PCIe transfers, and NDP invocations.

8.2 Micro-Benchmarks

8.2.1 Processor Performance Evaluation. To compare the processing elements (CPU, PEs) employed in our system, we utilize EEMBC’s CoreMark[29] benchmark, which is designed to operate across a spectrum of devices, from micro-controllers to multi-core processors. CoreMark provides a standardized approach to measuring CPU performance through the execution of a set of small yet representative tasks, including matrix multiplication, linked list manipulation and traversal, state-machine parsing, and CRC, among others. The benchmark’s portability enables its deployment on various processor architectures, facilitating a comparison between the MicroBlaze cores on smart storage and the ARM cores on the N1-SDP. Furthermore, we provide the score of an enterprise-class Intel CPU for comparison. The summarized results are presented in Table 1.

Insight: Compared with the host-side ARM cores, the on-device MicroBlaze PE-array is more than 26× slower. It is, therefore, valid to assume low on-device compute performance and relatively weak PEs. In contrast, the ARM cores in the N1-SDP exhibit performance that is comparable to enterprise-grade x86 CPUs. Although neoDBMS relies on *Software NDP* and emphasizes its benefits of fast compilation and loading of binaries (compared with FPGA hardware-reconfiguration), and versatile exchange of NDP functionality and operations, significant speedups are achieved with the weak MicroBlaze PE-array. We expect even greater speedups to be achieved by incorporating more hardwired FPGA functionality.

8.2.2 NVM Emulation. To emulate the access characteristics of NVM, in particular Intel Optane DC, we employ an open-source NVM emulator (NVMulator [86]), and calibrate and configure it based on latency sets from [30, 44, 72, 88]. We consider NVM in *memory mode* and distinguish a *best-*, *middle-ground*, and a *worst-case*, investigating the impact of those latency-sets on NDP updates. In addition, to validate the preloader and unloader impact in mitigating higher-latency I/O, we executed a full table update *without* locking (Figure 16) on a YCSB dataset (SF 6000).

Insight: Surprisingly, the on-device *preloader* and *unloader* effectively mask the NVM latencies by interleaving computation and data transfers, resulting in stable execution times across different read/write latency configurations (Figure 16).

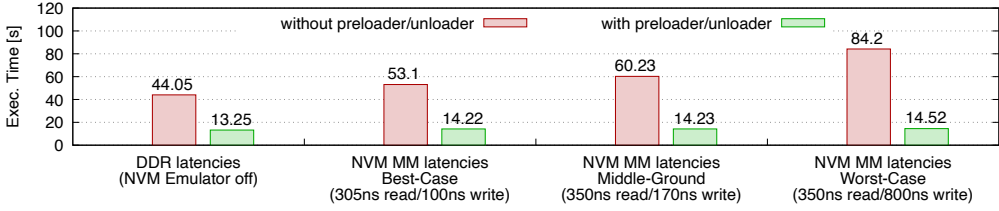


Fig. 16. On-device *preloader* and *unloader* effectively mask NVM latencies resulting in stable performance across various settings.

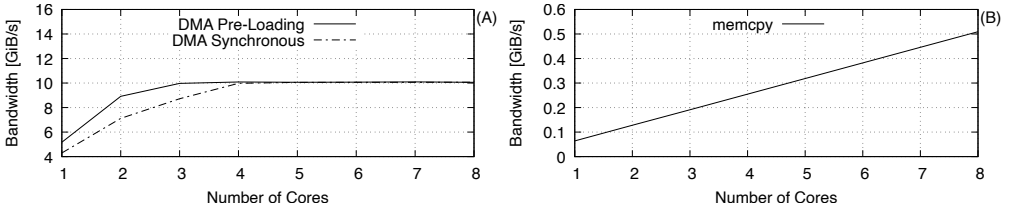


Fig. 17. MicroBlaze NVM bandwidth usage with increasing core count. Comparing hardware sync/async DMA transfers (A) to synchronous software memcopy (B).

8.2.3 MicroBlaze NVM Bandwidth Utilization. Allocating additional cores to NDP operations can enhance both the compute and memory bandwidth capabilities. However, the weak MicroBlaze cores easily become *compute-bound* even in data-intensive settings, which limits on-device bandwidth utilization. To this end, we introduced preloader and unloader modules in neoDBMS, that *interleave compute and I/O* for NDP operations.

We now investigate their effect by measuring the maximum achievable bandwidth with increasing core/PE count. As a baseline, we report the MicroBlaze *memcopy* performance. The NVM emulator is configured with Middle-Ground latencies (350ns read/170ns write).

Insight: Software-NDP is slowed down by primitives such as *memcopy*. General hardware primitives such as *preloader/unloader* leverage the flexible on-device (FPGA) memory hierarchy and reduce the computational-boundness of the weak PEs and increase the bandwidth utilization. Our results (Figure 17), demonstrate the advantages of sync/async DMA transfers over regular MicroBlaze memcopy operations.

8.2.4 Read-Only NDP Performance. We evaluate read-only NDP performance by offloading full-table scans to the smart storage device, comparing neoDBMS against PostgreSQL on YCSB-generated data (SF 6000 if not stated otherwise). We vary the impact of data-transfers across three scenarios: (a) single-attribute aggregation (single value as result); (b) varying the projectivity (increasing number of attributes per result-set); and (c) fix projection-size of five attributes but varying the selectivity. Our results (Figure 18) demonstrate that offloading size-reducing operations to smart storage significantly improves the performance by reducing data transfers between DBMS and smart storage.

8.3 System-Level Benchmarks

Throughout the evaluation, we use a set of different YCSB workloads [25] and a scale factor of 6000 if not stated otherwise. The dataset size is 6.5GiB (initial)/14GiB(total). The HW NVM emulator was configured with middle-ground latencies: 350ns read/170ns write for cacheline-sized accesses

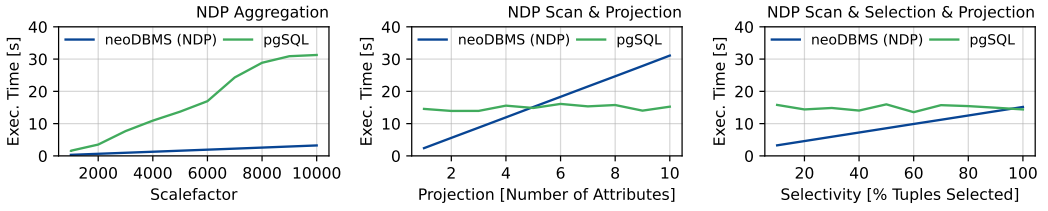


Fig. 18. Comparing read-only NDP scan performance: (left) aggregations; (mid) varying projectivity; (right) varying selectivity. Offloading size-reducing read-only operations to smart storage can greatly improve performance.

(Figure 16(A) [88]). Our main baseline is PostgreSQL v.12 (pgSQL), which relies on a traditional file-system (*ext4*) stack with block-device storage (*Block/BLK*).

8.3.1 Experiment 1: Update NDP Improves Overall System Performance. We open the evaluation with a general experiment, demonstrating the end-to-end effect of in-storage modifications on system performance and showing that *update NDP* significantly reduces data movement and buffer contention. To this end, both neoDBMS and pgSQL execute a mixed workload: starting with a read-only OLTP workload (YCSB-C [25]), during which we inject an update with varying selectivities, and return to read-only OLTP. We vary the selectivity over the primary key of the YCSB table. pgSQL therefore utilizes an index scan to reduce the effort, while neoDBMS traverses the whole VID_{map} , as it currently does not use indices on-device.

Figure 19(A) shows the end-to-end execution time for the update with different selectivities ranging from 0% to 100%. neoDBMS consistently outperforms pgSQL for selectivities >0%: from 1.68 \times at 10% selectivity up to $\geq 4.67\times$ at 100% selectivity. Execution time for 0% selectivity represents the full scan performance of neoDBMS, which is comparable (0.94 \times) to the index scan of pgSQL and its efficient detection of empty ranges facilitated by the B-tree index, which is currently not used on-device in neoDBMS. Noticeably, the impact of selectivity, which strongly correlates with data-movement, is much less prominent in neoDBMS, since data-transfers are performed only on-device.

In Figure 19(C) we show the I/O throughput over time. For orientation, an overview of the measured I/O paths is provided in Figure 19(B). Executing large modifications (100% selectivity) under mixed workloads solely on the host (pgSQL) results in significantly inferior performance, due to the increased write I/O and buffer contention (Figure 19(C)). neoDBMS reduces the I/O and thus data-movement by executing the update on-device. Afterwards, pages containing new tuple versions and $\Delta VID_{map}/\Delta L2P_{map}$ are transferred back if requested by the workload, which is recognizable as a small read spike past the end of the NDP execution (Figure 19(C), 19(D)).

The on-device bandwidth utilization during update NDP execution is shown in Figure 19.D. During scan and filter operations (read-only phase to determine visible tuple-versions qualifying for the update), neoDBMS is able to utilize approx. 92 MiB/s read bandwidth. Noticeably, during this phase update NDP mainly performs byte-sized transfers for visibility-checking and predicate evaluation. With selectivities >0%, whole records and their pages get preloaded, which leads to the sudden increase of bandwidth utilization. Furthermore, as the update intensity increases with rising selectivity, neoDBMS is able to utilize up to 490 MiB/s read and 530 MiB/s write bandwidth. At this stage, the weak MicroBlazes start to become compute bound, not able to fully utilize the 10 GiB/s on-device NVM-bandwidth (Figure 17).

Insight: neoDBMS enables transactionally consistent in-storage modifications and demonstrates robust and $\geq 4.67\times$ better overall performance. Leveraging snapshot based NDP and multi-versioned

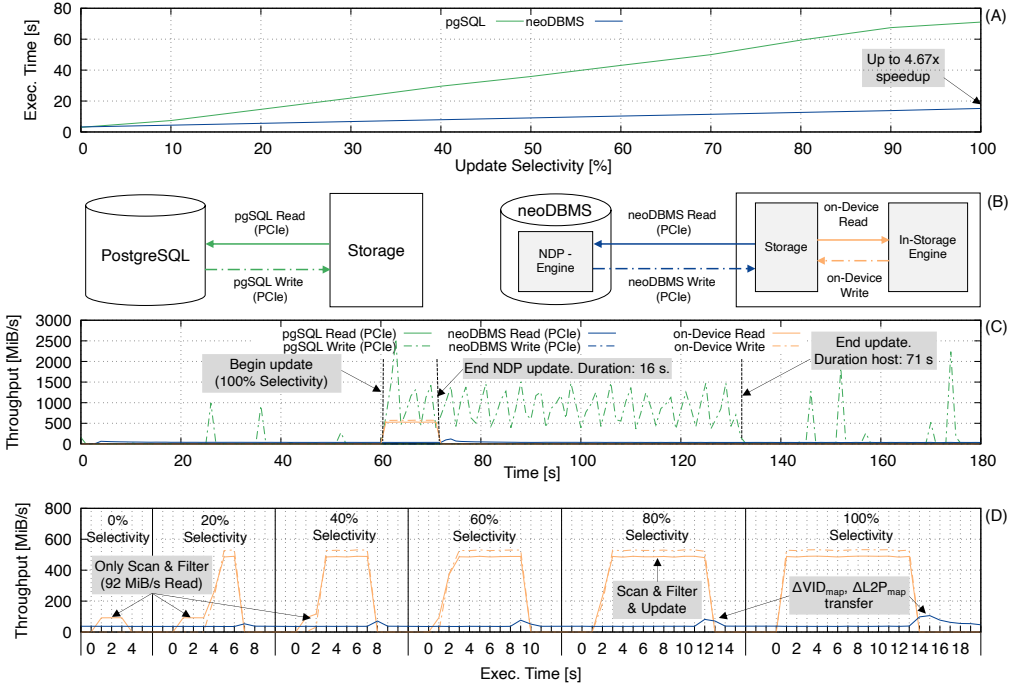


Fig. 19. Mixed workload performance: update NDP is fast and robust, while host-only executions yield notable drops. (A) Impact of update selectivity on execution time. (B) Overview of measured I/O paths in traditional(left) and neoDBMS setting (right); these I/O paths, denoted by the arrows, correspond to the legend entries in sub-figures C and D below. (C) I/O utilization during mixed workload. (D) On-device I/O utilization during update NDP execution.

Table 2. Overhead of Shared Lock-Table and Δ Maps Transfer

I/O	MiB	Ratio
Read	6307	47.48%
Write	6748	51.49%
ΔVID_{map} $\Delta L2P_{map}$	92	0.69%
SLT Locking	46	0.34%

UpdateNDP	Time
Device w/o setting locks	13.0s
Device with setting locks	13.9s
Locking Overhead	6.4%
Host w/o lock release	1.4s
Host with lock release	1.5s
Release Overhead	4.4%

storage allows update NDP to execute nearly intervention-free, except for the low-overhead SLT interactions. Noticeably, selectivity has much lower impact on system performance with update NDP, as the necessary data-movement is performed on-device.

8.3.2 Experiment 2: SLT and Delta-Maps Transfer have Low Impact on Performance and Data-Movement. The prior experiments make combined use of the SLT and Δ Maps transfer (ΔVID_{map} and $\Delta L2P_{map}$). To evaluate their individual performance impact, we now investigate the data transfer occurring on-device and compare the execution times of NDP updates with and without SLT locking (Table 2) at 100% selectivity. The SLT footprint is small by design and thus contributes just 0.34% to the overall data transfers during the update. The size of Δ Maps is very small, amounting to just

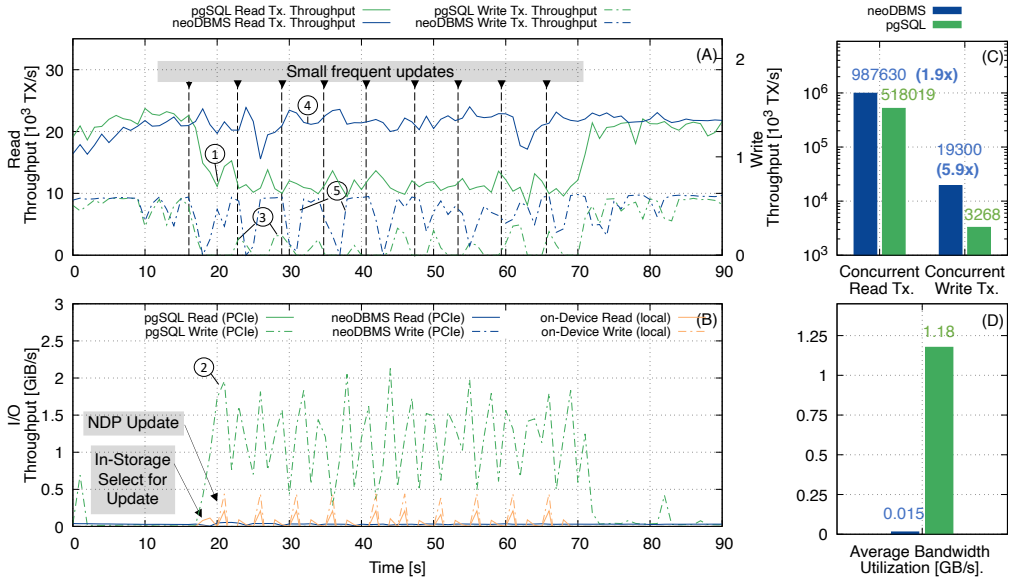


Fig. 20. Effect of frequent updates on performance: with update NDP, neoDBMS maintains a stable read throughput, as updates are offloaded as NDP operations, reducing I/O and buffer contention, while write throughput recovers quickly upon commit, due to shorter lock duration. PostgreSQL suffers a throughput drop due to data movement resource and lock contention. During the mixed workload neoDBMS improves concurrent read transactions by 1.9 \times and write transactions by 5.9 \times .

92 MiB and contributing 0.69% to the overall transfers. Although fine-grain locking with the SLT introduces overheads on the device (setting locks), as well as on the host (releasing locks), both are very low with 0.9s (6.4%) and 0.1s (4.4%) respectively.

Insight: By combining high-bandwidth PCIe and low-latency CCI in a single hybrid interconnect, neoDBMS can leverage both properties for efficient larger Δ Maps and result-set transfers and low-overhead synchronization between host and smart storage devices utilizing ccSVM-based SLT.

8.3.3 Experiment 3: Shared Locking Behavior under Contention. We now investigate the impact of lock contention on update NDP. Long running NDP updates have higher probability of aborting, since fast-paced concurrent host-transactions may update qualifying tuples and commit first. Such aborts result in discarding already performed useful NDP update work (Section 7.2). While this can be avoided by locking qualifying tuples in advance at the host-side, it causes significant data movement. neoDBMS mitigates such overheads by an in-situ SELECT FOR UPDATE.

To evaluate this claim, we execute a mixed workload, starting with a read/write OLTP workload (YCSB-A) and injecting frequent, but small updates (10% selectivity), distributed over the whole dataset (Figure 20(A)). A transfer-cautious execution is feasible with fine-grain synchronization mechanisms, like the proposed SLT. As demonstrated in the previous experiment, injecting the updates impacts heavily pgSQL's read throughput (Figure 20(A) ①). The read/write OLTP workload (YCSB-A, 50% read/ 50% write) increases the I/O and buffer contention, which manifests itself as increased pgSQL writes (Figure 20(B) ②). Furthermore, pgSQL holds locks longer, which impedes the throughput of writing transactions and concurrency (Figure 20(A) ③). neoDBMS keeps the read transaction load stable (Figure 20(A) ④), as updates are offloaded as NDP operations, reducing I/O and buffer contention. The write throughput recovers quickly upon NDP transaction commit,

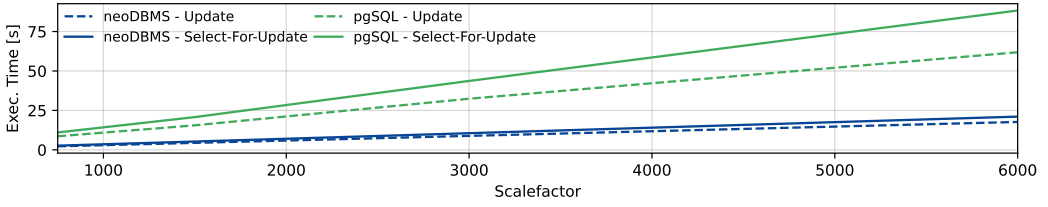


Fig. 21. Host-side and NDP SELECT FOR UPDATE techniques against normal update execution. neoDBMS reduces the overhead of SELECT FOR UPDATE by locking qualifying tuples in-situ, reducing data transfers.

as locks that update NDP holds shorter are released (Figure 20(A) ⑤). During the mixed workload neoDBMS’s SLT locking is able to improve concurrent read transactions by 1.9× and write transactions by 5.9× (Figure 20(C)). In comparison, full-table locks prior NDP invocation would block all concurrent write transactions, including modifying NDP transactions, even in lock-contention-free scenarios. neoDBMS also manages to reduce the average host-storage bandwidth utilization by 80× (Figure 20(D)) by performing the updates on-device and reducing buffer contention, resulting in less page evictions.

Insight: neoDBMS enables efficient concurrent host/device processing in heterogeneous systems while preserving transactional consistency. Notably, its fine-grain SLT incurs low overhead even under high contention. By reducing resource contention and data transfers, neoDBMS maintains higher concurrent reads and accelerates modifications on smart storage via compute and I/O interleaving, shortening lock durations and enabling greater write concurrency.

8.3.4 Experiment 4: In-Storage SELECT FOR UPDATE. We evaluate the impact of applying SELECT FOR UPDATE techniques in isolation. neoDBMS enables the offloading of SELECT FOR UPDATE as NDP-operations, allowing the locking of qualifying tuples in-situ without the need to transfer the data to the host first. To evaluate the performance benefits and overheads of utilizing SELECT FOR UPDATE, we conducted a comparison between host-side and in-storage execution of normal updates that lock tuples during execution against locking all qualifying tuples prior execution with SELECT FOR UPDATE. We update the whole dataset while varying the scale factor from 750 to 6000. Our experimental results (Figure 21) demonstrate a significant reduction of the SELECT FOR UPDATE overhead from 1.43× on the host down to 1.19× in neoDBMS and yielding a 4.2× speedup by executing the SELECT FOR UPDATE and update operation on-device.

Insight: While SELECT FOR UPDATE mitigates the risk of aborts during long-running transactions, it can cause a lot of data movement to identify all qualifying tuples prior execution, resulting in resource contention and buffer pollution that might force the eviction of data needed for the execution itself. neoDBMS mitigates this by performing SELECT FOR UPDATE on device and utilizes the ΔVID_{map} to store successful locks as logical pointers to tuples, only requiring the reevaluation and snapshot creation of tuples with unsuccessful locks prior execution.

8.3.5 Experiment 5: Comparison of Locking Approaches. Coarse-granular locking and OCC techniques can mitigate update conflicts in NDP transactions, besides the proposed, fine-granular SLT locking. To compare their performance (Figure 22(A)), we execute an in-situ update with varying selectivity in isolation, without concurrent workloads on the host.

As expected, locking entire DB-objects prior to updates performs best, as it requires the least effort to manage locks and detect conflicts. However, it severely limits concurrency. SLT follows closely as it takes full advantage of the CCI characteristics, enabling fine-grain locking with low overhead. Despite not having to manage locks, OCC’s verification phase results in worse performance compared with SLT, which under contention limits its applicability.

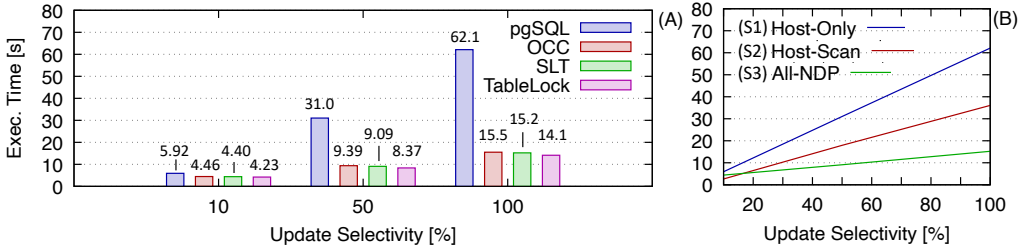


Fig. 22. (A) Comparison of locking approaches: fine-grain and low-latency SLT locking is feasible and a better solution in terms of concurrency; (B) NDP pipelines: offloading whole sequences of mixed operations reduces I/O and roundtrips resulting in improved NDP efficiency.

Insight: Fine-grain SLT locking is feasible using CCI low-latency, cache-coherent, and atomic transfers, achieving performance comparable to full-table locking or OCC. Unlike coarse-grained table locks or lock-free OCC which incur negligible or no lock management, SLT maintains low overhead despite individually locking all tuples, offering a better solution in terms of concurrency under contention (Figure 20(C)).

8.3.6 *Experiment 6: NDP Operation Pipelines Improve Performance.* We now investigate the effect of offloading sequences of NDP operations as NDP-pipelines, since offloading individual update operations does not always reduce data movement. Furthermore, combining SELECT FOR UPDATE with a successive update is another candidate for NDP pipelining. To demonstrate the benefits, we investigate the execution of the update `UPDATE usrtbl SET f=? WHERE ycsb_k=?` in three different stages: (S1) executes the selection/scan and the update on the host; (S2) executes the scan on the host but updates the result-set on-device; (S3) both are executed as an NDP pipeline in-storage. There is no concurrent host load. Our results (Figure 22(B)) show that S3 speeds up execution for update selectivities >17%. The scans in S1 and S2 benefit from the data being already in memory and from index lookups, while S3 scans the whole dataset. However, S2 has the disadvantage that intermediate results must be pushed down together with the NDP invocation.

Insight: Allowing the flexible assignment of individual transaction operations to either the NDP or host engine allows neoDBMS to further optimize I/O and round-trips to improve NDP efficiency, enabling future exploration of query plan optimizations.

8.3.7 *Experiment 7: Abort and Rollback Behavior.* Under contention, long-running modifying transactions have a higher probability of aborting, which increases the pressure on garbage collection and successive visibility checks. In this experiment, we investigate the *abort handling performance* of update NDP with fine-grain SLT locking and with OCC under mixed workloads. We show that SLT yields better performance and a higher long-term impact (Figure 23(A)). To this end, we define different *abort timepoints* within a transaction ranging from *early abort*, where 10% of the data is processed until abort, to *late abort*, where the last tuple provokes the abort. SLT allows for early-stop-conditions that signal the abort to all involved PEs, resulting in smaller $\Delta Maps$ and *clean* version-chains. Conversely, the OCC verification phase occurs after the update, always yielding a late abort (Figure 23(A)). This phase also results in increased *host-side processing* costs (Figure 23(A)), depending on the time elapsed until collision detection. Consequently, new versions are marked aborted but still appended to the version chain, increasing the pressure on visibility checking, as aborted versions have to be accessed first to determine the predecessor version. This effect is clearly visible in Figure 23(B). Here, we execute YCSB-A on the host and attempt multiple NDP updates without SELECT FOR UPDATE, which abort due to the high write contention. The use

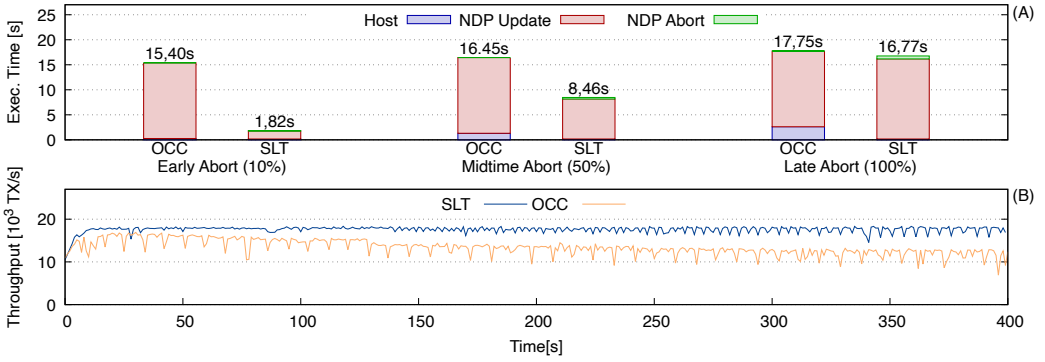


Fig. 23. Abort performance under write contention. (A) The OCC verification phase occurs after the NDP update, resulting in long abort times. SLT allows for the detection of early-stop conditions during NDP update. (B) New versions already appended to the version chain are marked aborted during OCC abort, increasing the pressure on visibility checking and garbage collection, impacting throughput over time. SLT aborts simply discard all changes, resulting in clean aborts and robust performance.

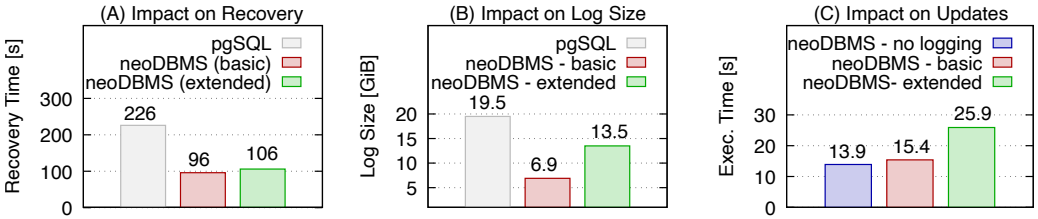


Fig. 24. Comparing neoDBMS's integrated logging approaches and their impact on recovery time (A), log size (B), and exec. time (C).

of SLT allows neoDBMS to simply discard $\Delta L2P_{map}$ and ΔVID_{map} changes, resulting in a robust system throughput.

Insight: Compared with isolated NDP executions or data snapshots, neoDBMS allows new data or modifications to continuously funnel through to the smart storage and are visible to all PEs. In combination with cache-coherently shared SLT-locks allow NDP transactions that are update-aware to concurrent host and NDP modifications and able to immediately react upon collision detection. This is beneficial under high contention and allows neoDBMS to mitigate expensive abort overheads like in OCC. On-device ΔMap maintenance and shared-state propagation yield fast rollbacks, clean aborts, and ultimately robust performance over time.

8.3.8 Experiment 8: Recovery and Logging Performance. NDP modifications mandate logging mechanisms that seamlessly integrate into smart storage and DBMS to ensure consistent rollback and recovery from system and device failures. We evaluate neoDBMS's integrated log-manager which supports three modes: no logging, basic logging (Section 6.1), which seamlessly integrates into the DBMS and update NDP by transferring and storing minimal logs, only covering system failures, as well as our extended logging techniques (Section 6.2) that ensure recovery in cases of device failure by asynchronously transferring NDP-logs back to the host and storing them in an overflow log-file.

First, we evaluate neoDBMS's recovery time (Figure 24(A)) and log size (Figure 24(B)) and compare them against PostgreSQL. During device failure recovery, both systems replay 6M tuple insertions

of YCSB data (SF 6000) followed by 6M updates resulting from an additional full-table update, modifying an attribute of each tuple (YCSB field1). In neoDBMS, inserts are handled via standard PostgreSQL WAL recovery, while updates leverage NDP log writer/reader (Custom WAL resource manager in PostgreSQL) due to being offloaded as update NDP operations. Our results show that neoDBMS's extended logging approach improves recovery performance by 2 \times . For comparison, the *basic* approach performs even better due to efficiently replaying $\Delta Maps$ stored in the NDP redo-log, which however is only sufficient for system failure recovery. The extended approach incurs a 10% overhead over *basic* due to additionally replaying the overflow log-file. Notably, this approach avoids additional DBMS-side allocations by directly transferring pages to smart storage through the initial replay of $\Delta L2P_{map}$, thereby minimizing memory contention during recovery. Standard PostgreSQL performs worst, since it replays updates tuple-by-tuple, rebuilding all data pages from scratch. We observe high memory consumption during recovery and page flushes impacting PostgreSQL's recovery performance. Additionally, updates in PostgreSQL are logged as its tuple-level deltas and require the predecessor versions during redo, making memory contention worse. Even with lower contention (e.g., YCSB SF100) neoDBMS still delivers 24%-43% faster recovery. Surprisingly and despite PostgreSQL's delta logging during updates, its overall log size is larger with 19.5 GiB compared with neoDBMS's extended approach with 13.5 GiB, which includes all pages created during update NDP, $\Delta Maps$, and logs created on the host side representing the inserts.

Next, we evaluate the impact of neoDBMS's logging mechanisms on update NDP execution (Figure 24(C)). We measure their NDP-log creation overhead during the full-table update described above. Our results show that basic logging incurs minimal overhead compared with no logging, as $\Delta L2P_{map}$ and ΔVID_{map} are small (Table 2) and storing them in the host-log prior commit is cheap. Both $\Delta Maps$ are necessary for the validation/invalidation of NDP modifications on the DBMS side (Figure 11). Combining their use for logging minimizes additional data transfers or log creation overheads. In isolation, the extended logging approach does however impact performance by 1.68 \times (Figure 24(C)). Beyond the $\Delta L2P_{map}$ and ΔVID_{map} , the NDP redo-log which relies on the new pages constructed on-device needs to be transferred back to the host, resulting in noticeable log-transfer and log-flushing overheads (Figure 24(B)). However, these disadvantages and the prolonged commit latency can be mitigated by Lock-Violation techniques as the following experiment 9 shows.

Insight: Our basic logging approach is well-suited for applications tolerating media non-recoverability windows and system-recovery with alternative mechanisms, leveraging on-device persistence to minimize log transfers and log sizes, providing high NDP update performance with minimal overhead compared with no logging. However, ensuring continuous device failure recovery requires persisting NDP-logs on secondary storage prior commit that neoDBMS addresses through its extended logging approach, efficiently transferring and integrating NDP-logs into the host logging infrastructure and persistence, achieving competitive recovery performance and log sizes. While our current design prioritizes compute efficiency under weak PEs, further optimizations must balance I/O against computational overheads. These overheads are problematic as prolonged commits extend lock duration, impacting concurrency. To address this challenge, we introduced lock-violation techniques that we evaluate next.

8.3.9 Experiment 9: Lock-Violation Techniques. The transfer of the overflow-log is essential for media failure recovery. Nevertheless, log-transfers and log-flushing incur delays in the commit-phase of NDP transactions, leading to prolonged lock duration, lock contention and lower concurrency. Lock-violation techniques are employed to allow host-side and NDP transactions waiting for a lock to proceed and perform useful work, interleaving overflow log-movement.

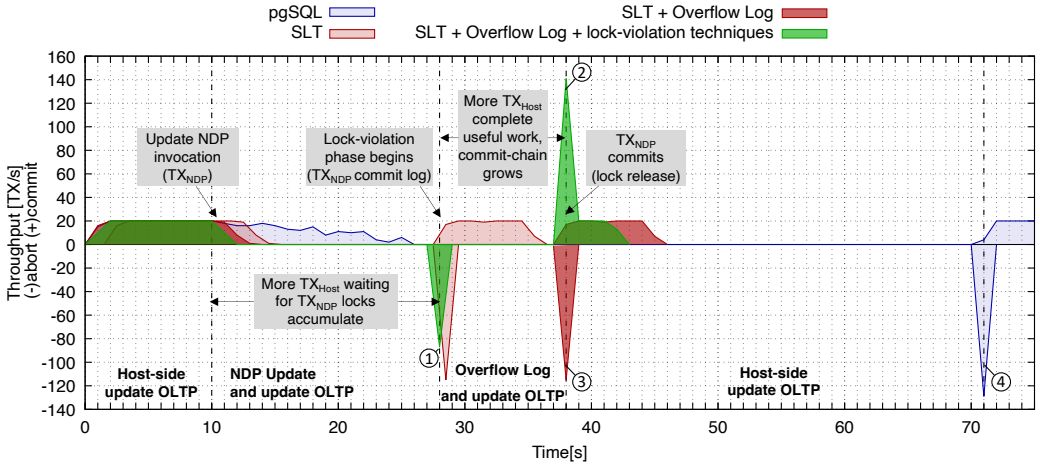


Fig. 25. Lock-violation techniques allow for more TX. to complete useful work and detect early aborts during the NDP commit phase while interleaving log-movement and flushing. ① Accumulated waiting TX_{Host} violate TX_{NDP} locks, validation resulting in early-abort (abort spike). ② TX_{NDP} commits and so do all TX_{Host} in lock-violation commit-chain (commit spike). ③ Without lock-violation, TX_{Host} face prolonged lock-durations, delaying validation, and resulting in late-aborts only. ④ pgSQL with a significantly longer update-phase and late-aborts of accumulated waiting TX_{Host} .

The impact and efficiency of lock-violation in a concurrent host-NDP write setting is evaluated in Figure 25. We employ a YCSB dataset, with an initial size of SF 4000. On the host side, we execute an update-intensive OLTP workload at a constant transaction admission rate of 20 TX/s, keeping the impact on resource contention as low as possible. Next, we inject a large update operation (100% selectivity) in parallel to OLTP, using NDP SELECT FOR UPDATE ahead of the actual modification. PostgreSQL serves as a baseline for the host-only update execution, while neoDBMS executes the update on-device. We compare neoDBMS under three different scenarios: (i) basic *SLT* as a baseline without media recovery; (ii) *SLT with overflow-log* as a baseline that ensures media recovery but it suffers log-movement; and (iii) *SLT with lock-violation and overflow-log*, which bridges the performance gap between the prior two extremes. Measured execution times and the commit/abort rates show the overhead of log-movement and its impact on performance.

During the first phase (Figure 25), the OLTP workload maintains a constant transaction admission rate of 20 new TX/s by design. In the second phase, starting at 10 seconds, we inject an NDP transaction, comprising only the update. With the initiation of the in-situ SELECT FOR UPDATE strategy, all tuples are gradually locked, leading to a decreased throughput as fewer transactions are able to commit at the host side, as they are forced to wait for the lock release and the completion of the NDP update transaction. Naturally, the host-side workload comes to a halt as soon as the NDP update acquires all the locks (approx. in the span of the 12–15 second-marks). Consequently, an increasing number of transactions waiting for locks accumulate, which grows larger the longer the duration of the NDP update. Please notice that although the host-only execution under PostgreSQL (blue curve in Figure 25) follows the same pattern, it is much slower, with its second phase beginning at approx. the 27 second-mark.

In the third phase and upon the completion of on-device update processing, the changes in $\Delta L2P_{map}$ and ΔVID_{map} are transferred back to the host for validation and application. Upon successful validation, the host NDP-engine initiates commit-processing (since the NDP transaction contains no further operations) and places a commit-record in the log buffer. At this point,

lock-violation permits subsequent transactions to violate or ignore lock conflicts with locks held by the NDP transaction. As a result, the host-side transactions, waiting for locks get notified, wake up, and reevaluate the state of the respective tuples, recognizing that they have been altered. Consequently, they are forced to abort, as serializability cannot be guaranteed anymore. This causes the notable abort spike occurring at the 28s mark (Figure 25 ①).

Under lock-violation both the modifying host-side transactions and potential parallel NDP transactions engage in productive work, while the overflow log transfer and log-flushing take place. Although they can execute their operations during this phase, they are not permitted to commit and wait until the NDP update transaction commits first. This results in the creation of a lock-violation commit-chain, which grows longer, the longer the commit phase, overflow log-movement take. As soon as the commit processing successfully completes, the lock-violation commit-chain is released, leading to a substantial spike in commit activity at the 38s mark (Figure 25 ②), upon which the host-side OLTP recovers back to stable 20 TX/s. In contrast, locks are held for an extended duration during the overflow-log processing in neoDBMS without lock-violation. This leads to a significant buildup of waiting transactions that perform no useful work, resulting only in a surge in aborts at the 38s mark (Figure 25 ③).

PostgreSQL exhibits inferior update performance, due to higher resource contention and data-movement, as prior experiments have shown. This leads to a substantially prolonged SELECT FOR UPDATE phase, achieving full tuple locking only by the 27 second-mark (Figure 25). The update phase is also considerably longer, concluding with a commit at the 71 second-mark (Figure 25 ④). Again, waiting transactions are forced to abort, evident as abort spike at the commit.

Insight: Under Update NDP, lock-violation allows for interleaving useful work with log-transfer and flushing. Firstly, lock-violation reduces the overhead for media recovery to almost zero, yet the performance is as high as basic logging, i.e., without media recovery. Secondly, lock-violation allows hot tuples to bounce back to a host transaction, or be processed by other NDP operations during the commit process of a modifying NDP transaction (lock-violation phase), which is especially relevant for skewed workloads.

We observe that the current benchmark suite [27] is suboptimal for evaluating lock-violation techniques in concurrent host-NDP executions. While sustaining high throughput, it enforces strict sequential execution of transactions within a worker thread. Limitations: As a result, under high lock-contention prolonged workload stalls may occur throughout the lock duration. We mitigate the impact of such stalls and sustain transaction throughput, by continuously spawning a new worker as soon as one stalls. Yet, this approach introduces significant overhead on both the DBMS and the benchmark due to lack of database connection pooling or reuse, limiting our sustained throughput to 20 TX/s. These limitations prevent us from investigating the impact of log-movement and lock-violation under higher concurrent transactional throughput. We expect that under such conditions the advantages of our lock-violation techniques would be much more pronounced.

9 Discussion

The use of NDP in DBMS opens up the discussion on aspects such as bandwidth development or operator pushdown in disaggregated cloud settings. Firstly, NDP aims at reducing data movement, however this raises the question of the *impact of data transfers given the increasing transfer bandwidths of PCIe*. The bandwidth of PCIe has been doubling with each next generation (from approx. 16 GB/s for PCIe 3 x16 to approx. 128 GB/s for PCIe 6 x16). However, on-device bandwidth also increases in lockstep, as it is a function of the steadily increasing storage density as well as the number of chips and channels on a smart storage device. In fact, while the PCIe bandwidth has been increasing approx. 7×/year (PCIe gen. 1, 2007 through PCIe gen. 6, 2022) the channel bandwidth on ONFI flash chips has increased 237×/year (ONFI v1, 2007 and ONFI v5.1, 2022). As a result,

Table 3. Comparison of NDP Frameworks and Their Supported NDP Capabilities

NDP Frameworks	Functionality	Remarks
IBEX [93], BlueDBM [66], JAFAR [9], ISP [51], YourSQL [46], BlockNDP [11], Biscuit [38], PolarDB [21]	read-only	offloading size-reducing read-only operations
nKV [90, 91], AIDE [57]	read-only, update-aware	concurrent host-side modifications and transactional guarantees
Caribou [43], Willow [80], PapyrusKV [49], Kanzi [40]	update and read	single key-value pairs, no transactional support
neoDBMS	update and read	transactional guarantees, fine-granular synchronization, NDP logging and recovery, relational

enterprise-class smart storage devices will have sufficient on-device bandwidth. In addition, as data volumes also increase at super-linear rates [84], data movement also increases. Even with increasing interconnect (PCIe) bandwidth, data must be transferred and processed on the host in traditional architectures. As a result, significant resource contention that impacts performance, scalability, as we show in our Experiments 1, 2 and 3. As a result we expect that, growing interconnect (PCIe) transfer rates do not outweigh the advantages achieved through NDP.

The second important aspect is UpdateNDP’s advantage over approaches for operator pushdown in disaggregated cloud systems. Systems like PushdownDB [101], Taurus [61], Teleport [102] or FlexPushdownDB [99] have demonstrated the advantages of operator pushdown to the memory or the storage in a disaggregated system. PushdownDB, FlexPushdownDB rely on the S3 Select service for operation pushdown to the storage layer and thus saving data movement and network bandwidth, which however has been deprecated. Offloading to storage offers even further advantages in terms of bandwidth savings, resource efficiency and scalability. Furthermore, the systems mentioned above can only pushdown read-only analytical operations. neoDBMS can in addition offload update operations and save the data movement for determining the qualifying tuples.

10 Related Work

With the advances in the semiconductor industry Smart SSDs [28, 80] were proposed, combining storage technologies (Flash, NVM) and reconfigurable PE. Since then, a variety of specific databases and generic NDP frameworks have been introduced (Table 3). Systems like IBEX [92, 93], BlueDBM [66], JAFAR [9, 95], ISP [51], YourSQL [46], BlockNDP [11], Biscuit [38], and PolarDB [21, 22] demonstrate successfully that offloading size-reducing operations like filters and even non-size-reducing operations like JOINS can be accelerated using NDP. Yet, they are read-only. X-Engine [41] proposed an optimized storage engine for transactional processing, utilizing FPGA for accelerated in-storage compaction. NDP systems targeting key-value stores like Caribou [43], Willow [80], PapyrusKV [49], and Kanzi [40] support NDP modifications but operate on single key-value pairs. Additionally, transaction handling built on top of these systems needs to be implemented first. nKV [90, 91] can execute read-only NDP transactions with transactional guarantees. Recently, AIDE [57] also adopted the design principles behind neoDBMS and nKV. NDP in cloud settings has been addressed by Taurus [62] and PolarDB Serverless [21], which pushdown data processing operations to data/storage nodes with compute capabilities.

Prior work [60, 71, 82] has demonstrated the benefits of offloading modifications, even though these cover special cases. For instance, [71] presents a vision for a DBMS engine residing completely on the computational storage device. As SQL or JDBC over PCIe are envisaged as an interface to smart storage. While [71] does not explicitly demonstrate offloading modifications, they may be feasible in cases where no transactional conflicts exist. That said [71] aims at batch updates in an exclusive mode and does not address ad-hoc modifications, synchronization mechanisms, locking techniques, or recovery approaches. Furthermore, [60, 82] present approaches to offloading

the compaction operations in key-value stores to computational storage. Given the architecture of such KV-engines (e.g., LevelDB, RocksDB) the offloaded compaction is performed out-of-place in pre-defined address ranges. Therefore, such approaches do not require and therefore do not consider ad-hoc synchronization mechanisms.

Ref. [58] presents an approach for DBMS log-shipping and replication that is implemented near-data support and aims to tackle log-movement. In contrast to X-SSD [58], which tackles log-movement across storage devices, update NDP and neoDBMS aim at addressing device-host log-log movement and interleaving it with useful work to achieve robust throughput and avoid lock contention.

To the best of our knowledge, neoDBMS is the first system to demonstrate transactional consistency using fine-granular locking on smart storage. Umbra [68] and neoDBMS follow a similar design philosophy, optimizing buffer utilization for efficient host-side hot-data processing and high-performance storage, yet without NDP in Umbra.

11 Conclusions

In this article, we introduce *update NDP* as a novel approach for executing modifications and NDP-pipelines on smart storage with transactional guarantees. We introduce a CCI-based *SLT*, based on novel *ccSVM* and hybrid interconnect, which is fast, economical, and applicable to future smart storage NDP and heterogeneous systems. We expect that our results will be applicable to upcoming CXL hardware with the deprecation of CCIX. Update NDP in neoDBMS outperforms host-only executions in PostgreSQL $\geq 6.52\times$. Notably, neoDBMS performs transfer-cautious and intervention-free NDP. Update NDP improves CPU utilization and memory contention. The proposed SLT has very low overhead, even under contention. neoDBMS employs an easy-to-develop and deploy *Software-NDP*, which is noteworthy in light of the $\geq 6.52\times$ speedup. With novel *custom preloader* and *unloader*, NDP leverages the properties of NVM, interleaving data movement and compute.

References

- [1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active disks: Programming model, algorithms and evaluation. In *Proc. ASPLOS (1998)*.
- [2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O. In *Proc. SIGMOD (2014)*.
- [3] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. 2020. Tackling Hardware/Software co-design from a database perspective. In *Proc. CIDR (2020)*.
- [4] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *Proc. (CIDR'17)*.
- [5] ARM. 2023. Neoverse N1 SDP - technical specifications. Retrieved from <https://developer.arm.com/ToolsandSoftware/NeoverseN1SDP>
- [6] ARM Cortex-A Series Programmer's Guide for ARMv8-A. 2023. The ARMv8 instruction sets. Retrieved from <https://developer.arm.com/documentation/den0024/a/An-Introduction-to-the-ARMv8-Instruction-Sets/The-ARMv8-instruction-sets/Addressing>
- [7] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE TKDE* 30, 4 (2018), 689–702.
- [8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*. 583–598.
- [9] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. 2015. JAFAR : Near-data processing for databases. In *Proc. SIGMOD'15*.
- [10] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a MICRO-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.
- [11] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. 2020. BlockNDP: Block-storage near data processing. In *Proc. Middlew. (2020)*.
- [12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10.

- [13] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosf, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib caching engine: Design and experiences at scale. In *Proc. OSDI (2020)*.
- [14] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Carsten Heinz, Christian Knoedler Tobias Vinçon, Andreas Koch, and Ilia Petrov. 2022. neoDBMS: In-situ snapshots for multi-version DBMS on native computational storage. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3170–3173.
- [15] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Andreas Koch, and Ilia Petrov. 2025. PUL: Pre-load in software for caches wouldn't always play along. In *Proc. ADBIS'25*. 44–59.
- [16] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Andreas Koch, Tobias Vincon, and Ilia Petrov. 2022. Cache-coherent shared locking for transactionally consistent updates in near-data processing DBMS on smart storage. In *Proc. EDBT (2022)*.
- [17] Nils Boesch and Carsten Binnig. 2022. GaccO - A GPU-accelerated OLTP DBMS. In *Proc. SIGMOD (2022)*.
- [18] Haran Boral and David J. DeWitt. 1989. Parallel architectures for database systems. In *Database Machines*. A. R. Hurson, L. L. Miller, and S. H. Pakzad (Eds.). Springer Berlin Heidelberg, Chapter Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, 11–28.
- [19] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages.
- [20] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proc. ASPLOS (2021)*.
- [21] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *Proc. FAST (2020)*.
- [22] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB serverless: A cloud native database for disaggregated data centers. In *Proc. SIGMOD (2021)*.
- [23] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB key-value workloads at facebook. In *Proc. FAST (2020)*.
- [24] Marco Chiappetta and Zak Killian. 2023. Phison E26 SSD preview: PCIe 5 storage breaks out for 2023. Retrieved from <https://hothardware.com/reviews/phison-e26-pcie-5-nvme-ssd-preview>
- [25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC (2010)*.
- [26] David de la Chevalerie, Jens Korinth, and Andreas Koch. 2016. fLink: A lightweight high-performance open-source PCI express Gen3 interface for reconfigurable accelerators. *SIGARCH Comput. Archit. News* 43, 4 (April 2016), 34–39.
- [27] Djelle Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288.
- [28] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 1221–1230.
- [29] EEMBC. 2023. CoreMark - EEMBC embedded microprocessor benchmark consortium. Retrieved from <https://www.eembc.org/coremark/>
- [30] Sean Eilert, Mark Leinwander, and Giuseppe Crisenza. 2009. Phase change memory: A new memory enables new memory usage models. In *2009 IEEE International Memory Workshop*.
- [31] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA database – An architecture overview. *IEEE Data Eng. Bull.* 35 (03 2012), 28–33.
- [32] Victor Garcia-Flores, Eduard Ayguade, and Antonio J. Peña. 2017. Efficient data sharing on heterogeneous systems. In *ICPP (2017)*.
- [33] Pordesign GmbH. 2023. Prodesign HAWK versal VC1902 acceleration card. Retrieved from <https://www.prodesign-fpga-acceleration.com/prodcard>
- [34] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. In *Proc. VLDB Endow. (2015)*.
- [35] Robert Gottstein, Ilia Petrov, and et al. 2017. SIAS-Chains: Snapshot isolation append storage chains. In *ADMS@VLDB (2017)*.
- [36] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proc. SIGMOD (2013)*.

- [37] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE-A main memory hybrid storage engine. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 105–116.
- [38] Boncheol Gu, Andre S. Yoon, and et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proc. ISCA (2016)*.
- [39] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo open-source toolflow. *JSPS* 93, 5 (May 2021), 545–563.
- [40] Masoud Hemmatpour, Mohammad Sadoghi, and et al. 2016. Kanzi: A distributed, In-memory key-value store. In *Proc. Middlew. (2016)*.
- [41] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proc. SIGMOD (2019)*.
- [42] CCIX Consortium Inc. 2016. An introduction to CCIX - white paper. Retrieved from <https://www.ccixconsortium.com/wp-content/uploads/>
- [43] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent distributed storage. In *Proc. VLDB (2017)*.
- [44] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic performance measurements of the intel optane DC persistent memory module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714. <http://arxiv.org/abs/1903.05714>
- [45] Houxiang Ji, Srikanth Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. 2024. Demystifying a CXL type-2 device: A heterogeneous cooperative computing perspective. In *Proc. MICRO'24*. DOI: [10.1109/MICRO61859.2024.00110](https://doi.org/10.1109/MICRO61859.2024.00110)
- [46] Insoon Jo, Duck-ho Bae, and et al. 2016. YourSQL : A high-performance database system leveraging In-storage computing. In *Proc. VLDB (2016)*.
- [47] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISks). *SIGMOD Rec.* 27, 3 (Sept. 1998), 42–52.
- [48] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE (2011)*.
- [49] Jungwon Kim and et al. 2017. PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures. In *Proc. SC (2017)*.
- [50] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proc. SIGMOD (2016)*.
- [51] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. *Inf. Sci.* 327, C (Jan. 2016), 183–200.
- [52] Hideaki Kimura, Alkis Simitis, and Kevin Wilkinson. 2017. Janus: Transactional processing of navigational and analytical graph queries on many-core servers. In *Proc. CIDR (2017)*.
- [53] Christian Knödler, Naeem Ramzan, and Ilia Petrov. 2025. hybridNDP: Dynamic operation offloading and cooperative query execution in smart storage settings. In *Proc. EDBT'25*. DOI: [10.48786/EDBT.2025.62](https://doi.org/10.48786/EDBT.2025.62)
- [54] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck Hua Lee, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zait. 2015. Oracle database in-memory: A dual format in-memory database. *Proc. ICDE*. 1253–1258.
- [55] Per Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. In *Proc. VLDB Endow. (2015)*.
- [56] Juchang Lee, Wook Shin Han, Hyoung Jun Na, Chang Gyoo Park, Kyu Hwan Kim, Deok Hoe Kim, Joo Yeon Lee, Sang Kyun Cha, and Seung Hyun Moon. 2018. Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases. *VLDB J.* 27, 3 (June 2018), 421–444.
- [57] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. 2023. Deploying computational storage for HTAP DBMSs takes more than just computation offloading. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1480–1493.
- [58] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A storage system with native support for database logging and replication. In *SIGMOD'22*.
- [59] H. Li et al. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proc. ASPLOS*.
- [60] Minje Lim, Jeeyoon Jung, and Dongkun Shin. 2021. LSM-tree compaction acceleration using in-storage processing. In *Proc. ICCE-Asia (2021)*.
- [61] Shu Lin, Arunprasad P. Marathe, Per-Åke Larson, Chong Chen, Calvin Sun, Paul Lee, Weidong Yu, Jianwei Li, Juncai Meng, Roulin Lin, Xiaoyang Chenxi, and Qingping Zhuxii. 2022. Near data processing in taurus database. In *Proc. ICDE*. DOI: [10.1109/ICDE53745.2022.00170](https://doi.org/10.1109/ICDE53745.2022.00170)

- [62] Shu Lin, Arunprasad P. Marathe, Per-Åke Larson, Chong Chen, Calvin Sun, Paul Lee, Weidong Yu, Jianwei Li, Juncui Meng, Roulin Lin, Xiaoyang Chenxi, and Qingping Zhuxii. 2022. Near data processing in taurus database. In *ICDE*.
- [63] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proc. SIGMOD (2020)*.
- [64] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proc. SIGMOD (2017)*.
- [65] MicroBlaze, Xilinx Inc. 2022. MicroBlaze soft processor core. Retrieved from <https://www.xilinx.com/products/design-tools/microblaze.html>
- [66] Sang-woo Jun Ming, Arvind, and et al. 2015. BlueDBM: An appliance for big data analytics. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 1–13.
- [67] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. ScyPer: A hybrid OLTP&OLAP distributed main memory database system for scalable real-time analytics. In *Datenbanksysteme für Business, Technologie und Web (BTW) 2014*. Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen (Eds.). Gesellschaft für Informatik e.V.
- [68] Thomas Neumann and Michael J Freitag. 2020. Umbra: A disk-based system with in-memory performance. In *CIDR (2020)*.
- [69] OpenSSD. 2023. DAISY plus openSSD FPGA platform. Retrieved from <https://www.crz-tech.com/crz/article/DaisyPlus/>
- [70] OpenSSD Project 2019. *COSMOS Project Documentation*. OpenSSD Project. Retrieved from http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources
- [71] Jong-Hyeok Park, Soyeon Choi, Gihwan Oh, and Sang-Won Lee. 2021. SaS: SSD as SQL database system. *Proc. VLDB Endow.* 14, 9 (May 2021), 1481–1488.
- [72] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the intel optane byte-addressable NVM. In *Proc. MEMSYS (2019)*.
- [73] Rekha Pitchumani and Yang-Suk Kee. 2020. Hybrid data reliability for emerging key-value storage devices. In *Proc. FAST (FAST'20)*. 309–322.
- [74] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in postgresSQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1850–1861.
- [75] Vijayshankar Raman, Gopi Attaluri, and Ronald Barber. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proc. VLDB (2013)*.
- [76] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *Proc. SIGMOD (2020)*.
- [77] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active storage for large-scale data mining and multimedia. In *Proc. VLDB*.
- [78] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A real-time OLTP and OLAP system. In *Proc. EDBT (2018)*.
- [79] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Letha: A tunable delete-aware LSM engine. In *Proc. SIGMOD (2020)*.
- [80] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 67–80.
- [81] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. DoppioDB: A hardware accelerated database. In *Proc. SIGMOD (2017)*.
- [82] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. 2023. An asynchronous compaction acceleration scheme for near-data processing-enabled LSM-tree-based KV stores. *ACM Trans. Embed. Comput. Syst.* 23, 6, Article 93 (Sept. 2024), 33 pages.
- [83] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL memory with genuine CXL-ready systems and devices. *IEEE MICRO (2023)*.
- [84] Alexander Szalay and Jim Gray. 2006. 2020 computing: Science in an exponential world. *Nature* 440, 7083 (March 2006), 413–414.
- [85] Sajjad Tamimi, Arthur Bernhardt, Florian Stock, Ilia Petrov, and Andreas Koch. 2024. DANSEN: Database acceleration on native computational storage by exploiting NDP. *ACM TRETS* 18, 1, Article 4 (Dec. 2024), 33 pages.
- [86] Sajjad Tamimi, Florian Stock, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. 2023. NVMulator: A configurable open-source non-volatile memory emulator for FPGAs. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing.
- [87] Sajjad Tamimi, Florian Stock, Andreas Koch, Arthur Bernhardt, and Ilia Petrov. 2022. An evaluation of using CCIX for cache-coherent host-FPGA interfacing. In *IEEE FCCM (2022)*.

- [88] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *Proc. DaMoN (2019)*.
- [89] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under HTAP workloads. *Proc. VLDB Endow.* 15, 10 (June 2022), 1991–2004.
- [90] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2020. nKV: Near-data processing with KV-stores on native computational storage. In *Proc. DaMoN (2020)*.
- [91] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Christian Riegger, Sergey Hardock, Christian Knoedler, Florian Stock, Leonardo Solis-Vasquez, Sajjad Tamimi, Andreas Koch, and Ilia Petrov. 2020. nKV in action: Accelerating KV-stores on native computational storage with near-data processing. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2981–2984.
- [92] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibox: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974.
- [93] Louis Woods, J. Teubner, and G. Alonso. 2013. Less watts, More performance: An intelligent storage engine for data appliances. In *Proc. SIGMOD (2013)*.
- [94] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.
- [95] Sam Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. 2015. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN'15)*. Article 2, 10 pages.
- [96] Xilinx. [n. d.]. DMA/Bridge subsystem for PCI express product guide (PG195) - Xilinx. Retrieved from <https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Introduction>
- [97] Xilinx. 2021. SmartSSD Computational storage drive. Retrieved from <https://www.xilinx.com/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf>
- [98] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. OSDI(2020)*.
- [99] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulmaga, and Michael Stonebraker. 2021. FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS. *Proc. VLDB Endow.* 14, 11 (July 2021), 2101–2113.
- [100] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220.
- [101] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulmaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *Proc. ICDE*.
- [102] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing data-intensive systems in disaggregated data centers with TELEPORT. In *Proc. SIGMOD (SIGMOD'22)*.

Received 29 May 2024; revised 12 June 2025; accepted 6 October 2025