Abstract—Multi-versioning and MVCC are the foundations of many modern DBMSs. Under mixed workloads and large datasets, the creation of the transactional snapshot can become very expensive, as long-running analytical transactions may request old versions, residing on cold storage, for reasons of transactional consistency. Furthermore, analytical queries operate on cold data, stored on slow persistent storage. Due to the poor data locality, snapshot creation may cause massive data transfers and thus lower performance. Given the current trend towards computational storage and near-data processing, it has become viable to perform such operations in-storage to reduce data transfers and improve scalability. neoDBMS is a DBMS designed for near-data processing and computational storage. In this paper, we demonstrate how neoDBMS performs snapshot computation in-situ. We showcase different interactive scenarios, where neoDBMS outperforms PostgreSQL 12 by up to 5×.

Index Terms—ndp, smart storage, disaggregated memory

I. INTRODUCTION

The foundation of many modern DBMS architectures is multi-versioning. It fits current workloads, such as Hybrid Transactional and Analytical Processing (HTAP), that combine long-running analytical queries (OLAP) with frequent and low-latency update transactions (OLTP) on the same dataset and DBMS [7]. Multi-versioning offers good performance, as readers never block writers. It also matches key properties of modern hardware, in terms of parallelism, data placement, or copy-on-write.

Background. With multi-versioning and MVCC, multiple versions of each data item (i.e. tuple) may physically co-exist. For consistent execution, every transaction operates against a snapshot of the database that comprises all currently visible tuple-versions. An update transaction (i.e. \( TX_{U1} \), Fig. 1) produces a new version (i.e. \( t_v1 \), Fig. 1) of the data item (i.e. tuple \( t \)) and invalidates the predecessor version (i.e. \( t_v0 \)). All versions of a tuple form a version-chain, for which many organizations exist [13]. Timestamps determine, which of the existing tuple-versions is visible to a transaction. Read transactions (i.e. \( Q_R \) in \( TX_R \)) operate on the latest committed version visible to them (i.e. \( t_v0 \)) and never stall.

Motivation. Snapshot creation and visibility checking may become very expensive with long version chains that occur naturally under mixed workloads. Intermediate versions along the chain cannot always be garbage collected and snowball to cold storage. In large industrial HTAP settings, the number of active versions can be as high as several hundred millions [6]. Long analytical queries that process cold data must perform visibility checking, which is slow and I/O intensive.

NDP and Computational Storage. Near-Data Processing makes it possible for the storage to perform visibility checking in-situ, i.e. as close as possible to the physical data location of the cold data. NDP has become viable, as hardware manufacturers can economically fabricate and package combined storage and compute elements in the same storage device (computational storage). In addition, with semiconductor storage, the device-internal bandwidth, parallelism, and latencies are much better than external ones.

In this paper, we introduce neoDBMS, which is a DBMS designed for NDP and computational storage. In particular, we demonstrate how neoDBMS performs version visibility checks in-situ. The core insight is that computational storage can quickly return the tuple version visible to a certain transaction, thus creating the transactional snapshot efficiently in-situ. It can be consumed by both the host-DBMS, or a follow-up in-situ NDP-operation. Our contributions are: (i) we demonstrate how neoDBMS performs version visibility checks in-situ; (ii) neoDBMS outperforms the baseline PostgreSQL 12 by up to 5×; (ii) the process is performed in-situ without any interaction with the host as an intervention-free NDP-operation; (iii) neoDBMS creates the NDP snapshot with transactional guarantees; (iv) neoDBMS interprets the data layout in-situ (DBMS pages and the version records) to extract the
transaction timestamps with the help of in-situ format parsers and layout accessors; (v) in doing so neoDBMS relies on the byte-addressable nature of non-volatile memory (NVM) storage to reduce read-amplification.

II. ARCHITECTURE OF NEODBMS

We now provide a brief overview of neoDBMS. It is based on PostgreSQL, however it employs different version organization, native storage management, and NDP handling.

Version organization. neoDBMS introduces a different version organization and invalidation model [2] to allow for append behavior and version placement. neoDBMS organizes version records as a singly-linked list in a new-to-old (N2O) manner (Fig. 3), where every successor version has a reference to the predecessor. PostgreSQL relies on an old-to-new (O2N) organization. Furthermore, neoDBMS introduces a different version invalidation model, where each version contains the timestamp of the creating transaction (one-point invalidation), and gets implicitly invalidated by the existence of a successor version. PostgreSQL places the timestamps of both the creating and the invalidating transactions on the version record, causing in-place updates. To mark the entry-point of a chain, neoDBMS introduces a VID Map containing the RecordID of the latest version for each tuple. As a result, neoDBMS optimizes for append-based storage (physically disjoint) and OLTP, while PostgreSQL has the better organization for OLAP and speedy visibility checks.

Architecture. Based on its version model, all new version records in neoDBMS are placed in a small delta-buffer (Fig. 2). It comprises a set of pre-reserved logical database pages, which are flushed to storage once they get full. For NDP execution, the current state of the delta buffer is transferred and cached on-storage, and every execution is initiated.

Native Computational Storage. neoDBMS, like NoFTL [3] and qKV [10], relies on native storage (Fig. 2) and eliminates intermediary layers (e.g. file systems) along the critical I/O path, and operates directly on NVM storage. neoDBMS can therefore control the physical placement of DB pages, which is critical for utilizing the on-device I/O and compute parallelism. neoDBMS places pages of a DB-object on physical entities chips/channels (using regions [4]) to utilize the on-device parallelism and reduce the processing latency. Parallelism is essential for scaling the number of in-situ processing elements (PE), as they can utilize the bandwidth of the independent on-device channels. Another property of native storage is its ability to reduce read- and write-amplification, which in [9], [10] refers to operating at page-granularity. In neoDBMS we extend it to exploit the byte-addressability provided by the underlying NVM storage.

Storage Processing Elements. neoDBMS can utilize up to 16 RISC-V soft-cores as PEs for intervention-free, in-situ visibility checks, and follow-up in-situ NDP-operations. These are located on the FPGA chip of the computational storage and have direct access to NVM. The RISC-V PEs can be programmed in C, for easy development and dynamic deployment. When an NDP-operation is invoked, the purpose-built binary file is determined and loaded into the PEs ahead of the execution. The current HW design is efficient, as the PE array and the NDP architecture require only approx. 10% of the FPGA resources (Table II).

In-situ Data Interpretation. Being able to interpret the data format (pages, records) and navigate over the layout (e.g., NSM pages) is crucial for in-situ data processing. To this end, neoDBMS employs layout accessors and format parsers, like [11]. Whenever new DB-objects are created, the page and record formats are extracted and used for the configuration of format parsers and layout accessors. These pre-compiled and optimized RISC-V binaries, specific for a data layout and adapted to the requirements of a NDP-operation, are stored for later use (Fig 4). Hence, the small PE resource footprint and better FPGA utilization (Table II).

NDP Interface. Interaction with the NDP storage employs a native storage interface [3] with commands like PAGE_READ or PAGE_WRITE. The low-level device interaction is performed with TaPaCo [5], which provides a fast layer for the integration of FPGA-based accelerators. Based on the execution plan created by PostgreSQL, neoDBMS first identifies operations supported by the computational storage and invokes an CALL_NDP() command with the physical address of VID Map and delta-buffer (DB-Object specific), timestamp of the calling transaction, operation details like involved columns, result handling behavior, and other parameters.

In-situ Snapshot Computation. To perform in-situ snapshot creation, the NDP invocation takes the transaction timestamp of the calling transaction (TxID) as well as the delta-buffer and latest modifications to the logical-to-physical address mapping (L2P Map) and the VID Map (Fig. 3). Given the N2O version organization in neoDBMS (Fig. 3), the visibility

![Fig. 2. Architecture of neoDBMS](image-url)
check is performed by traversing the linked version-chains backwards, and extracting and comparing the creation timestamps against the $T_{xID}$. To compute the snapshot for the second chunk splits the $VID_{Map}$ into equal sized partitions, and assigns a partition to each of the PEs. Next, the second chunk spawns independent visibility check tasks on each RISC-V processor, calling the respective binary. Next, the PEs process the $VID_{Map}$ entries in order. The visibility task on each PE extracts the entry point $RecordID$ of each entry and resolves it in-situ. This $RecordID$ resolution is based on the $L2P_{Map}$ and yields the physical page pointer and a slot offset. It is then passed to the layout accessor on the PE, which retrieves the slot. In a follow-up transfer, the layout accessor retrieves the record header and passes it to the format parser to retrieve the transaction timestamp and compare it to $T_{xID}$. The invalidation timestamp is available from the predecessor that has already been processed. Based on both, the visibility check can be performed.

**Byte-addressability and NDP.** Both the in-situ visibility check and the layout accessors utilize the byte-addressability of the underlying NVM storage. This is possible within the storage device, since PEs can operate on physical persistent pointers, address byte-locations, and to transfer byte-sequences according to the data formats. Consider, for instance the transfer size depicted in Fig. 4. To check a version record, $neoDBMS$ must read just 20B in five 32B transfers. In this way, read amplification is reduced, while I/O parallelism and on-device bandwidth are efficiently utilized. At the same time, all PEs are kept busy, performing useful work, and the scalability grows almost linearly for up to 12 cores (Table I).

<table>
<thead>
<tr>
<th>Cores</th>
<th>4B transfers [IOPS-10(^3)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.0</td>
</tr>
<tr>
<td>4</td>
<td>27.0</td>
</tr>
<tr>
<td>8</td>
<td>53.5</td>
</tr>
<tr>
<td>12</td>
<td>77.5</td>
</tr>
<tr>
<td>16</td>
<td>83.2</td>
</tr>
</tbody>
</table>

### III. DEMONSTRATION WALK-THROUGH

**Demo Setup.** The demo setup comprises the ARM Neoverse N1 Platform as host with 4 ARM-CPU’s operating at 2.6GHz and 3GB RAM. Connected via PCIe Gen3 x16 is a Xilinx Alveo U280 FPGA board using 2GB DDR4 memory, which serves as computational storage. We slow down the DRAM accesses to emulate NVM latencies. The TaPaSCo-Framework [5] is utilized to create the FPGA design in advance and to manage and control multiple RISC-V PEs at runtime. We benchmark the default PostgreSQL12 implementation, using either a Samsung NVMe SSD 970 EVO 500GB, or the U280 via a EMMC block device driver, both as “dumb storage”, against NDP on the U280 in $neoDBMS$. We keep the datasets relatively small, to allow for shorter run-times and better interaction with the audience.

**Baselines.** We demonstrate $neoDBMS$ under a mixed workload on the TPC-C $Orderline$ table and showcase the impact on snapshot creation by varying the number of active versions in a chain, and setting the snapshot marker to an arbitrary version along the chain. The performance of $neoDBMS$ is compared against PostgreSQL12 on top of ext4 on “dumb storage”. Here, both the NVMe SSD and the U280 in block-mode yield very similar DBMS performance for the benchmark.

**Scenario 1: Snapshot creation with increasing data volume.** The demonstration begins by allowing the audience to pick an initial number of tuples for the $Orderline$ table. Each tuple is created in a single initial physical version. Afterwards, the audience is invited to select a simple aggregate function
(e.g., MAX, SUM) as an NDP-operation to be executed in-situ on top of the transactional snapshot of the single-version dataset. The process is repeated for different numbers of PEs and dataset sizes. The objective is to establish a performance baseline (Fig. 6).

**Observation:** Although parallel scans yield a performance improvement, neoDBMS outperforms both baselines.

**U280-No Parallel Worker**

**NVMe-No Parallel Worker**

**neoDBMS**

**Observation:** Although parallel scans yield a performance improvement, neoDBMS outperforms both baselines.

Fig. 6. Performance on varying dataset sizes (Scenario 1).

**Scenario 2: Increasing number of active versions.** Now the audience can start varying the update frequency of each record, and hence the number of active versions. To this end, the audience uses the workload composer to create a series of updating (commit) and reading transactions (that remain active to keep the versions alive). The snapshot marker is set so that the latest version in the chain is indeed visible. The objective is to demonstrate the effect of the in-situ snapshot creation and neoDBMS’s new-to-old organization for OLTP-style NDP-operations on the latest version of a long chain.

**Observation:** neoDBMS offers constant executions for OLTP-style NDP-operations that require the latest version.

Fig. 7. Snapshot on the first version of a chain (Scenario2).

**Scenario 3: NDP Snapshot in time on long version chains.**

In this final scenario, we let the audience pick a point-in-time, and create an in-situ transactional snapshot for an NDP-operation injected into it. To this end, the audience can compose a workload as shown for the previous scenarios. We then execute a series of updating transactions (that commit) and snapshot markers, which start transactions that remain active. We encourage the audience to set snapshot markers in different positions of the version chain. To this end, the audience can compose an additional workload that will be appended after the snapshot marker. In this manner, we demonstrate the impact of short-running transactions on long-running transactions, as more old versions need to be requested. As shown in Fig. 8, in-situ snapshot creation in neoDBMS outperforms the baselines for different snapshots on different chain positions.

**Observation:** NDP snapshots in neoDBMS are beneficial for HTAP workloads, where operations on cold data are executed at different points in time with transactional guarantees.

IV. RELATED WORK

The concept of NDP is based on well-known techniques, such as *database machines* or *Active Disks*. With widespread use of semiconductor storage, FPGA Smart SSDs [1], [8] well as IBEX [12] were proposed as intelligent storage for DBMS. JAFAR [14] is one of the first systems to target NDP for DBMS use. Much of the prior work on persistent KV-Stores and NDP focuses on *bandwidth* optimizations. Commercially, IBM Netezza or Swarm64 target NDP for RDBMS. But neoDBMS is the first system to describe snapshot computation with transactional guarantees.

**Acknowledgments.** The authors wish to thank the anonymous reviewers for the valuable comments. This work has been partially supported by *DFG neoDBMS – 419942270*.

**REFERENCES**


