Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads

Tobias Vinçon[*], Christian Knödler[*], Leonardo Solis-Vasquez[#], Arthur Bernhardt[*], Sajjad Tamimi[#], Lukas Weber[#], Florian Stock[#], Andreas Koch[#], Ilia Petrov[*]
[#]Embedded Systems and Applications Group, [*]Data Management Lab
[#]Technische Universität Darmstadt, [*]Reutlingen University

ABSTRACT

Today's Hybrid Transactional and Analytical Processing (HTAP) systems, tackle the ever-growing data in combination with a mixture of transactional and analytical workloads. While optimizing for aspects such as data freshness and performance isolation, they build on the traditional data-to-code principle and may trigger massive cold data transfers that impair the overall performance and scalability. Firstly, in this paper we show that Near-Data Processing (NDP) naturally fits in the HTAP design space. Secondly, we propose an NDP database architecture, allowing transactionally consistent in-situ executions of analytical operations in HTAP settings. We evaluate the proposed architecture in state-of-the-art key/value-stores and multi-versioned DBMS. In contrast to traditional setups, our approach yields robust, resource- and cost-efficient performance.

PVLDB Reference Format:
Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, Ilia Petrov. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. PVLDB, 15(10): 1991 - 2004, 2022.
doi:10.14778/3547305.3547307

1 INTRODUCTION

Modern data-intensive systems run Hybrid Transactional and Analytical Processing (HTAP) workloads combining long-running analytical queries (OLAP) as well as frequent and low-latency update transactions (OLTP) on the same dataset and even on the same system [55]. Such hybrid systems operate with continuous update rates on a hot portion of a large dataset, while performing complex analytical tasks on both the hot and the much larger cold part of the dataset. Consequently, large data transfers of cold data occur that are partly due to poor data locality, but also due to traditional (*data-to-code*) system architectures. Such transfers entail non-robust performance, scalability issues, and poor resource efficiency.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution, i.e. as close as possible to the physical data location. NDP leverages the trend towards *smart/computational storage* as hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device. Furthermore, with semiconductor storage (NVM/Flash), the *device-internal* bandwidth, parallelism, and latencies are much better than the external ones (device-to-host). Both trends lift major drawbacks of prior approaches like ActiveDisks [3, 35, 61] or Database Machines [16], such as bandwidth limitation and expensive proprietary hardware. Interestingly, even commodity devices nowadays come with compute hardware used for running backwards compatibility firmware not for data processing.

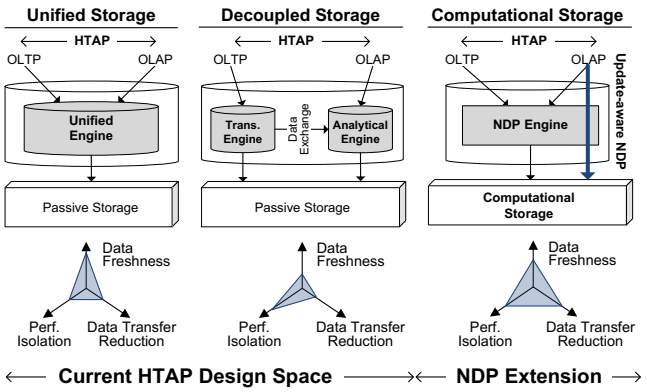

Figure 1: State-of-the-art HTAP architectures can be divided into unified and decoupled storage systems [60] and optimize either on data freshness or performance isolation. NDP and computational storage allow tackling both dimensions, while reducing cold data transfers for better performance.

Based on their storage design, two types of HTAP architectures can be distinguished [60]: unified and decoupled storage (Fig. 1). The former executes the OLTP and OLAP operations on the same dataset based on snapshotting and multi-versioning techniques, and is optimal for analytical processing on latest data. The latter separates the OLTP and the OLAP sub-system. It trades higher (but amortizable) OLAP response times, data freshness, selective access, workload adaptability, for higher OLTP throughput.

Problem 1: HTAP architectures cause transfer of cold data. In state-of-the-art large-memory settings, the *working dataset* fits in memory, yet the *complete dataset* is much larger (cold, historic data) and available on persistent storage. For instance, Umbra [52] is a novel system, representative of the new class of hybrid in-memory/SSD-based high-performance DBMS. HTAP workloads tend to process/analyze cold data, which is generally not in memory. Existing HTAP architectures (Fig. 1) assume passive storage, and thus OLAP processing entails large transfers of cold data. This impacts the system performance, and limits its scalability and resource efficiency (as shown in a motivating experiment – Fig. 2 –

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547307

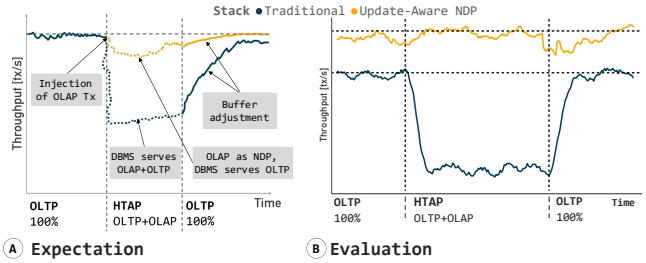

Figure 2: Expectation: Ⓐ traditional systems suffer the impact of data transfers after an OLAP query injection. Update-aware NDP executes OLAP operations in-situ, with robust performance. Our evaluation Ⓑ confirms this behaviour.

discussed later on). A key observation of this paper is that NDP naturally fits in the HTAP problem space, as NDP allows in-situ operations to process cold data without *moving* it to the host. NDP enables *intervention-free* execution, where the DBMS can continue processing, after asynchronously offloading NDP operations and delegating their execution to computational storage.

Problem 2: NDP necessitates transactional consistency. Despite all its advantages, NDP is currently utilized solely in read-only settings. Yet, in update-intensive HTAP settings, the most recent modifications of OLTP-style transactions are only available in the large DBMS memory [22], likely scattered across different data structures. However, analytical NDP operations from OLAP transactions, offloaded to computational storage, require that most recent data in-situ, alongside the cold persistent dataset, to achieve *consistency* and *freshness* guarantees. These properties are necessary since NDP operations execute in the context of the invoking transaction. The question of how the most recent data can be collected and propagated to smart storage, and how consistency and freshness can be ensured is still considered open [22].

Update-aware NDP. In this paper, we propose a *snapshot-based, update-aware NDP architecture* for computational storage and HTAP workloads. The *core idea* is to define a small shared state that accumulates modifications to main-memory data and DBMS state. Noticeably, the *shared state* is the only delta between the working set in the large DBMS memory, containing the most recent updates, and the much larger, but colder and complete dataset on computational storage. The shared state is regularly flushed to computational storage, whenever it reaches a pre-defined limit, but most importantly it is propagated as part of every NDP invocation. Thus, at the point of invocation the computational storage attains a complete and consistent *snapshot*, and the read-only NDP operation can execute with consistency guarantees. Moreover, the in-situ execution is asynchronous and free from DBMS/host intervention, as the computational storage now has a complete snapshot of the entire dataset. Although our architecture is aligned with disaggregated storage architectures [19, 46, 70], here we target in-situ processing.

We demonstrate the impact of NDP in HTAP settings with a motivating experiment (Fig. 2) in MySQL/MyRocks as traditional stack and MyRocks over nKV [69] as an NDP stack. In the first phase, both systems run LinkBench [7] as an OLTP workload. In the second HTAP phase, we inject an analytical OLAP operation (Betweenness Centrality, performing a graph analysis) in parallel to OLTP. Upon its completion, we switch back to pure OLTP in the last phase. The clear performance drop with the traditional stack during the HTAP phase is due to excessive cold data transfers. Noticeably, the OLTP throughput remains unchanged with the NDP stack, because of the intervention-free in-situ OLAP execution.

Our **contributions** are:

- We propose an *update-aware NDP architecture* that utilizes a small *shared state* to create a consistent snapshot on computational storage and execute the analytical NDP operation in-situ against it. Such NDP executions have snapshot transactional guarantees.
- We propose new NDP execution models. NDP storage can execute operations asynchronously in an *intervention-free* manner. Currently, we focus on read-only, analytical NDP operations.
- We describe how intermediary and final results can be handled in-situ to reduce data-transfers.
- A case study shows the implementation in two different systems: a key-value store (nKV), and a multi-version DBMS (neoDBMS). The evaluation is performed on real hardware - *COSMOS+* [54].

This paper is organized as follows. The next section provides necessary background and discusses related work. In Sect. 3 we go into the details of the proposed update-aware NDP architecture and describe the concepts behind its execution model. The experimental evaluation is discussed in Sect. 4 and we conclude with Sect. 5.

2 BACKGROUND AND RELATED WORK

We now discuss the current state-of-the-art HTAP systems from the industry [26, 27, 43, 44, 59] and academia [4, 6, 8, 9, 17, 29, 36, 40, 45, 49, 51, 60, 63] and classify their approaches into the HTAP design space. Our goal is to present: (a) the relevant background on the HTAP design space and the extension with NDP, (b) an overview of important aspects in regard to NDP and today's systems support [5, 10, 11, 21, 22, 30, 32–34, 37, 50, 64, 65, 73, 74, 76], as well as (c) a brief outline of the native storage concepts as they form the foundation of the present system architecture.

2.1 HTAP Workload and Systems

Today's database systems persist and operate on large and ever-increasing amounts of data. However, the processing no longer involves only OLTP-style workloads, operating on a small but hot portion of the entire data. Moreover, real-time analytical queries (OLAP), often with very complex algorithms, operate on the cold data from the storage tier as well as the freshest updates from the OLTP workload. The combined workload, termed Hybrid Transactional and Analytical Processing (HTAP), extends the problem space of database architectures with the following aspects [17, 49, 60]:

- *Data Freshness.* For the analytical portion of the HTAP workload, the given system architecture should aim for having the most recent version of data resulting from updates performed in the OLTP workload. Therefore, fast propagation of these updates to the analytical snapshot is required, and optimally avoids any performance drops for the transactional workload.
- *Data Consistency.* Regardless of the data freshness, the entire system must ensure transactional guarantees for its transactions so that transactional and analytical queries have a consistent view on the data. Various mechanisms have been proposed and

applied in databases to construct the so-called required snapshot. Two of the most prominent are the Copy-on-Write (CoW) approach and Multi-Version Concurrency Control (MVCC). The first ensures the visibility of older versions by creating a copy for modifications. The second creates a new version for every modified record and extends it with the current timestamp.

- *Data Transfers.* Independently of the separation of the analytical from the transactional engine, data transfers from the storage tier to the host processing units account for a large part of the overall performance. They result from: (a) HTAP processing of cold- and hot-data likewise, and (b) the cold data being much larger than the hot data and thus, causing buffer pollution and finally high eviction rates. Overall, the result is limited scalability, bandwidth boundness, and performance loss.
- *Performance Isolation.* DBMS are often used by business-critical applications, for which robust performance in terms of latency and throughput is essential. Interference between OLTP and OLAP workload must be prevented, in particular for hybrid scenarios that run both of them concurrently [49].
- *Memory Pollution.* While OLTP workloads operate on the hot data (working set) that fits in state-of-the-art large main memories, the major portion for OLAP processing is the cold data that exceeds the memory capacity. Hence, OLAP scans in hybrid scenarios inevitably entail buffer pollution in case the transactional and analytical engines share the same buffers, e.g. database buffers or OS page cache and device caches. Even though the buffer size is usually defined to be larger than the hot portion of the data set, the analytical queries have to fetch large parts of the cold data into the buffer, and thus cause evictions of the hot data.

Current state-of-the-art architectures proposed for HTAP scenarios can be classified into two major categories [60]. Firstly, Unified Storage Systems build snapshots for every occurrence of an analytical query. Consequently, this kind of system operates on the freshest data and is optimal for in-memory OLAP processing. Widely-known systems of this category are HyPer [36], Caldera [6], DB2 BLU [59] or SAP HANA [26]. Secondly, Decoupled Storage Systems continuously transfer modifications from the transactional engine to the separate analytical engine. Thus, workload optimizations and performance isolation can be introduced at the cost of data freshness. BatchDB [49], SQL Server [44] or Oracle's Dual Format [43] are representatives for this category.

Near-Data Processing emerges as another dimension in the HTAP architecture design space, which is not yet considered widely. Our *update-aware NDP architecture*, proposed here, can handle the freshest data and ensure transactional consistency without the drawback of buffer pollution or lack of workload optimizations from Unified Storage Systems. Noticeably, update-aware NDP allows to place data most efficiently on the computational storage device to leverage the hardware characteristics of the storage medium, but also introduces compute placement, as today's devices often come with multiple heterogeneous processing capabilities.

2.2 Near-Data Processing

Early approaches of Near-Data Processing date back to the 1980s-90s. *Database machines* [16] or *Active Disk/IDISK* [3, 35, 61] introduced proprietary magnetic/mechanical storage hardware. However, manufacturing costs combined with the low bandwidth and parallelism became limiting factors. With the advance in the semiconductor industry, Flash technologies and reconfigurable processing elements arose, and Smart SSDs [22, 64] were proposed. Since then, a variety of specific database and generic NDP frameworks were introduced such as IBEX [73, 74], Minerva [21], Willow [64], BlueDBM [50], JAFAR [10, 76], Kanzi [32], ISP [39], YourSQL [34], Biscuit [30], PapyrusKV [37], DoppioDB [5, 65], Caribou [33], Batched Writes [23], BlockNDP [11], Umbra [52], PolarDB [19] or nKV [68, 69]. Besides avoiding costly data transfers between host and device, each NDP approach optimizes for specific characteristics.

Storage Properties. By moving the execution closer to the storage, the opportunity to intensively leverage the hardware properties of storage technologies emerged. Flash, NVM, and HBM are widely utilized in NDP approaches due to their extremely parallel interfaces. Thus, significantly higher on-device bandwidths can be achieved in contrast to communications with the host. Indeed, [38] makes the case for 50 GB/s device-internal versus 6.4 GB/s device-to-host bandwidth. This is due to the physical organisation of semiconductor storage devices, which involves multiple chips, connected over independent channels to the on-device processing element. The chip-level bandwidth increases with chip density, which in turn increases due to modern 3D stacking technologies.

Similarly, latencies can be reduced as time-costly transfers through several OS layers are avoided and the arbitration over the parallel storage entities can be highly customized, reducing the load on waiting queues. Often, low-level interfaces that are usually not exposed to the host (e.g. multi-plane operations), allow for further optimizations.

Computation Models and Compute Placement. Apart from the storage technology, nowadays devices may comprise a variety of heterogeneous processing elements such as CPUs, GPUs, or FPGAs. Individual computations can be placed either traditionally on the host, or on-device. In case of the latter, it is further possible to split up and distribute processing across the various processing elements in heterogeneous hardware. For example, nKV [68, 69] demonstrated that moving computation to the device obtains significant performance benefits. Yet, offloading computation from ARM cores to parallel FPGA pipelines can improve throughput even further, especially for large scans. Depending on aspects such as the actual operation, workload, and underlying storage technology, the computation placement decision can vary and hardware can be configured individually for each NDP invocation.

Disaggregation and shared-storage architectures. The proposed NDP architecture is aligned to current disaggregated storage architectures [19, 46, 70]. These aim at elasticity and pushdown of database operations in the storage layer that decouples the CPU resources on the compute nodes from those of the storage nodes. The present work aims at NDP, which is a distinct subset of that problem space targeting in-situ processing.

2.3 Native Storage

Under native storage [56], the DBMS operates directly on the physical storage without intermediary layers of abstraction. Consequently, functionality like address mappings or garbage collection, that appears multiple times along the I/O path of traditional system stacks, can be combined and deeply integrated into the DBMS. This benefits not only the workload-aware scheduling, but also enables leveraging the hardware characteristics. Especially with the advent of Flash as general-purpose storage in today's data centres, the throughput is highly dependant on the utilization of parallel I/O units, e.g. channels and LUNs [68]. Thus, native storage also establishes novel storage abstractions like Regions and Groups [31] that can adapt to the workload at runtime per database object, and improve throughput, latency, and reduce write-amplification [67]. Other approaches, not yet as deeply integrated into the database as native storage, are pursued by [15, 57].

2.4 Update-aware NDP Systems

The components of the update-aware NDP architecture (Sect. 3) are generic, aligned with existing architectures of modern DBMS, and are easy to integrate. Throughout this paper, we focus on two systems: LSM-tree KV-Stores and multi-version DBMS.

Firstly, we employ nKV [68], a KV-store based on RocksDB, which can be exposed as a MySQL storage engine by means of MyRocks (*MyRocks over* nKV). It is a single-versioned, Copy-on-Write system, supporting `Repeatable Read` as the highest isolation level. The underlying data organisation is based on multi-level LSM-Trees [48], with C_0 being an in-memory skiplist-based MemTable, and $C_1 ... C_n$ organized as Sorted String Tables (SSTs) on persistent storage. The latter comprise data blocks with the actual KV-Pairs, and an additional index structure referencing these. As shown in Figure 3, each active transaction is assigned a separate *write batch* that contains transaction-local modifications before commit. Thus, transaction reads are first issued against their batch, before querying the MemTables or the persistent data. Modifications of a transaction are invisible to other transactions as they go to separate WriteBatches. Considering the example of Figure 3, a snapshot taken during TX_4 comprises only $key\_a = 11$ and $key\_b = 2$.

Secondly, we introduce neoDBMS [12], as a multi-version NDP-DBMS based on PostgreSQL. neoDBMS stores all updates as physically materialized version records (Fig. 3). As such they are identified by an implicit *RecordID* (<*PageNr,SlotNr*>). The version records

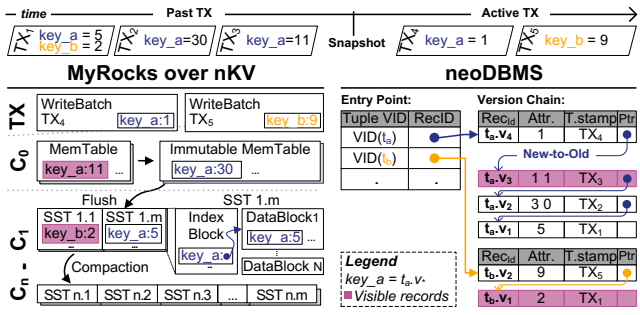

Figure 3: Storage organization under nKV and neoDBMS.

of each tuple form a version chain organized as a singly-linked list in a New-to-Old (N2O) manner, where every version has a forwards reference to its predecessor [28]. The invalidation of a version is handled implicitly by the presence of a successor version. All version records in a chain have the same *virtual id* (VID, e.g. t_a or t_b) as they belong to the same tuple. To mark the entry-point of a chain, neoDBMS introduces a VID_{Map} containing the RecordID of the latest version of each tuple. The N2O organisation yields fast visibility checks, especially for fresh data. In addition to the *VID* every physical record contains a transactional creation timestamp, unique for each version chain. 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 (Fig. 3) the version chain is traversed to determine the first visible version of each tuple to a transaction, e.g. $t_a.v_3$ and $t_b.v_1$ to a transaction starting at the time of the snapshot.

3 UPDATE-AWARE NDP ARCHITECTURE

We begin with an overview of the components of the proposed architecture (Fig. 4) and elaborate on them in the sections to follow.

3.1 Shared State and NDP Execution Model

The core idea of the *update-aware NDP* architecture is to offload the processing of read-only HTAP operations (e.g. complex queries with massive scans) that require reading large parts of the cold data on device, while ensuring transactional guarantees in presence of frequent update transactions. To this end, we define a small *shared state* that accumulates modifications to main-memory data and DBMS state. The shared state is the delta between: the large *working set* in the main memory of the host DBMS with the most up-to-date data, and the much larger but colder and *complete* dataset on computational storage. The shared state is regularly flushed to computational storage whenever it reaches a pre-defined size, but most importantly, it is propagated as part of every NDP invocation. Thus, at the time of propagation, the computational storage attains a complete and transactionally consistent *in-situ snapshot*, and the read-only NDP operation can execute with consistency guarantees. The only caveat is that NDP processing must begin from the shared state and only then move onto the cold data, as data items in the latter may have been invalidated by their "newer versions" in the former. Interestingly, the NDP execution is *intervention-free*, as it is asynchronous and does not require any interaction with the host. Therefore, it can achieve better scalability and performance.

In the following sections, we consider details of snapshot creation in single-version and multi-version systems, concurrency control, the NDP interface, and the execution model.

Shared State. DBMS usually maintain the freshest data, mapping tables, status or system information, in the large and fast main memory of the host system. Modifications or newly inserted records are scattered across the database address space (Fig. 5.A) and remain there, until they get evicted. Beyond actual records, modifications spill across various auxiliary structures, such as status and mapping tables, e.g. logical-to-physical address mapping. However, NDP operations require all of the latest data and state, to ensure transactional guarantees. In the words of Do et al. [22]: *"If there is a copy*

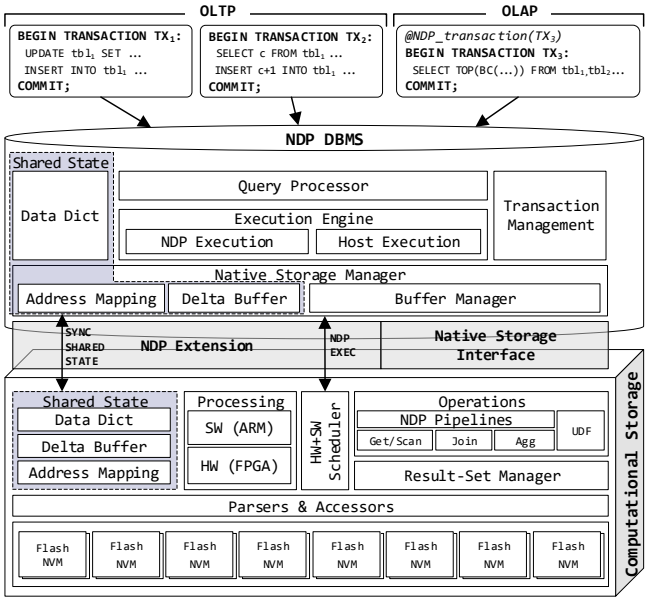

Figure 4: Update-aware NDP enables a transactionally consistent in-situ processing of OLAP operations.

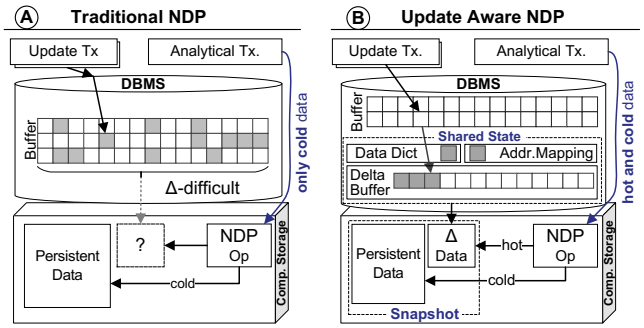

Figure 5: Modifications are gathered in a small in-memory shared state and propagated to computational storage, for consistent snapshot-based hot and cold data processing.

of the data in the buffer pool that is more current than the data in the SSD, pushing the query processing to the SSD may not be feasible."

In the update-aware NDP architecture (Fig. 5.B), we accumulate the modifications to all of those structures in an incremental way and place them together in shadow data structures that are collectively referred to as the *shared state*. As a result, the original data is left unmodified in the large memory of the DBMS. 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.

The *Delta-Buffer* is a key element of the shared state. It accumulates modifications as replacement records. Thus, records in the delta-buffer typically invalidate "older" versions present in memory or on the computational storage. Processing thus begins always with the delta-buffer and the shared state. In a multi-version DBMS, the delta-buffer accumulates the versions newly created by active transactions, while predecessor versions remain in memory and can be accessed by concurrent transactions. In a single-version DBMS (like RocksDB), the delta-buffer contains replacement records.

The delta-buffer is managed by an append-only double buffering strategy. Records are appended until the size reaches a certain threshold, upon which a new pre-allocated buffer is made available, while the old one is frozen. Committed records are prepared and compacted on fewer pages, while data from uncommitted transactions is pruned. Under specific systems like nKV, the process is straightforward as the delta-buffer (MemTables) is guaranteed to contain only committed versions due to the *WriteBatch* techniques. A possible low-space utilization is alleviated by lightweight compaction. Along these lines, the corresponding entries in the mapping tables are extracted and prepared. Both are then moved to the DBMS memory buffer, and are simultaneously flushed to the computational storage device. The traditional logging is orthogonal and remains unaffected. The shared state and the delta-buffer can be tailored to specific database objects. Thus, every DB-object can be assigned to a separate and individually-sized delta-buffer, to account for different levels of hotness and update patterns.

The concept of shared state is *aligned* with existing architectures of modern DBMS and is easy to integrate. For instance, it resembles the staging area in modern main-memory DBMS, such as SAP HANA [66], the delta storage in multi-version DBMS [75] or the batch-write transactional buffer in RocksDB.

Shared State Propagation Modes. The shared state is propagated to computational storage device in two distinct modes. First, *Flush & Append* is the regular mode, which is triggered by a flush of the shared state. The flushed state is prepared as described above, to contain data from a committed transaction, and the corresponding *log records* are written out in advance. Along the same lines, ahead of the flush, and based on the DBMS-controlled address mapping, the native storage manager has allocated clean nonadjacent physical pages. Noticeably, their physical addresses *need not* to be adjacent. Next, all shared state pages are flushed to storage and written to those allocated locations. Thus, the flush to persistent store is realized as a logical append that is placed on pre-assigned and nonadjacent locations. The flush is *atomic*, as only if all pages are successfully written, the storage manager *atomically* swaps the address-mapping entries. Otherwise, all pre-assigned address-mappings are dismissed, the corresponding locations are marked for later garbage collection, and the process repeats. The delta entries of the mapping table are merged atomically as well. However, the merge is performed only after the completion of active NDP-operations (which thus remain unaffected).

The second mode is *Pass Along & Cache*: At the time of an NDP invocation, the shared state is snapshotted and, together with the list of transactions currently in-flight, propagated to storage 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. This is possible, since the max. shared state size can be configured to be smaller than memory limits of the NDP device.

In this mode, applying the shared state to persistent storage is difficult in the general case since (a) it contains possible modifications of the invoking transaction, yet it is unknown whether it will commit; and (b) its space utilization may be low and incur overhead to successive space management operations (compaction, garbage

collection). With the shared state in place, the NDP operation executes in a shared-nothing manner without any intervention or synchronization with the host. Thus, the DBMS and device can operate independently and only synchronize at the end.

3.2 NDP Transaction Management

We now describe how transaction management must be adapted in the light of update-aware NDP, HTAP workloads, and existing concurrency control (CC) schemes. In that context, we face three issues: (a) transactional consistency, (b) intervention-free NDP executions, and (c) easy integration in various systems.

Any transaction containing NDP operations is called *NDP transaction* (annotated as `@NDP_Transaction`). Transactional consistency mandates that the NDP operations from an NDP transaction must only process modifications by transactions committed prior to its beginning, while ignoring modifications from concurrent transactions other than their own. The issue at hand is that, at the time of the NDP invocation, it is unknown whether concurrent transactions will commit or abort, and thus, which records should be processed. We tackle this by executing the NDP operation against a transactional snapshot created for it in-situ (described below). Noticeably, the snapshot construction and the execution are intervention-free, since inside the shared state, a list of the in-flight transactions is propagated alongside the NDP invocation, and is thus available on the device. This approach works well in the widespread *general MVCC case* [14], where the snapshot comprises only the latest committed version records prior to NDP transaction beginning. For example, NDP operations from transaction TX_3NDP (Fig. 6.A) can only operate on data from TX_1, ignoring modifications from TX_2.

Transaction Scheduling. Depending on the DBMS design and the CC flavor, modifications from concurrent transactions might be visible. For instance, MySQL/MyRocks [25] mandates that the visible record is the latest committed ahead of the NDP invocation (snapshot creation), rather than the NDP transaction start. Thus, modifications from TX_2 might be relevant to TX_3NDP (Fig. 6.B), but will not be present on device. Propagating them is difficult and unscalable. To this end, we propose a *transaction admission* mechanism for transactions with NDP operations (Fig. 6). Whenever such NDP-transaction arrives (e.g. TX_3NDP), it is assigned a transactional timestamp, as usual. However, its admission is delayed until after the completion of all transactions that were active when it arrived. The delay is typically very short (2ms in our setup), since OLTP transactions are fast relative to slow NDP/OLAP operations. Currently, we allow a single NDP invocation at a time. At the time the NDP-tx. is admitted for execution, no CC anomalies occur, since modifications from TX_2, but also from TX_4 are ignored (Fig. 6.C). Hence, TX_3NDP has at least snapshot-isolation guarantees.

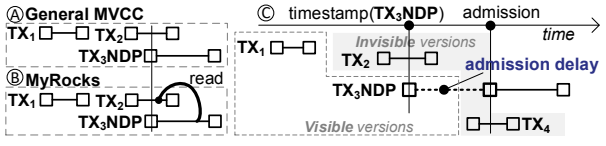

Figure 6: NDP transactions are delayed until concurrent OLTP tx. complete ensuring an intervention-free execution.

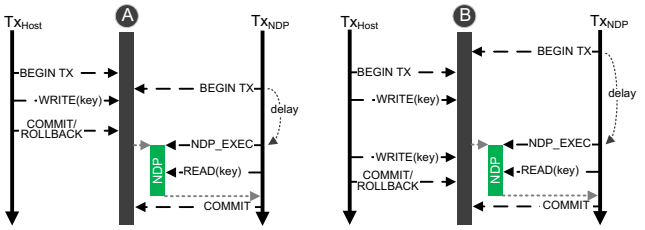

Figure 7: Update-aware NDP offers transactional guarantees depending on the integrated database.

Transactional Guarantees. We now analyze how update-aware NDP supports transactional guarantees for read-only operations in two transactional scenarios (Fig. 7). In particular, update-aware NDP within nKV is bound to MyRocks' highest isolation level `Repeatable Read` and its MVCC implementation, and can avoid Dirty Read and Non-Repeatable Read anomalies. Since NDP transactions wait for all other active OLTP transactions to complete (commit or rollback) before a snapshot of the current state is taken and the pushdown to the device is issued (see Fig. 7.A), it is ensured that the shared state and the delta-buffer only include transactionally consistent data for all previously started transactions. In case another transaction is started right after the NDP transaction, but before the NDP invocation (see Fig. 7.B), MyRocks stores all writes of this transaction in a separate *WriteBatch* (Sect. 2.4), ensuring that those updates will not be available to other transactions until its commit. Even if the transaction commits during the pushdown execution, the changes will not be present on device, as they have not been propagated with the NDP_EXEC call.

In-situ Snapshot Construction. In-situ snapshot creation is DBMS specific and can be realized with (a) a Copy-on-Write mechanism (MyRocks over nKV); as well as with (b) visibility-checking in multi-versioned DBMS. In Copy-on-Write based systems such as nKV, this snapshot is usually identified via a snapshot identifier, e.g. a sequence number (see Fig. 8). Records with a newer identifier are simply skipped during processing. This is possible because the write batching mechanism ensures that the delta-buffer only committed data. For instance, under nKV (Fig. 8), the NDP transaction

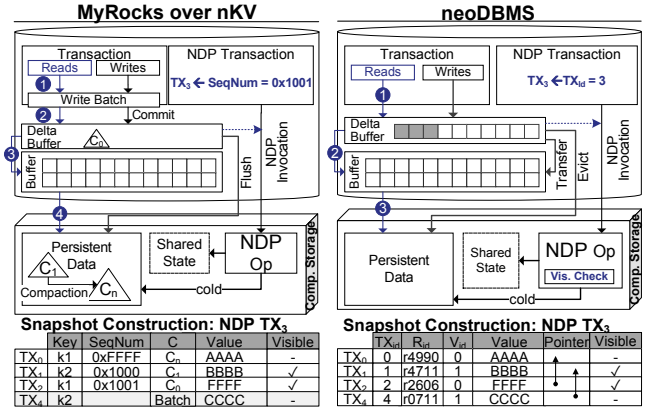

Figure 8: In-situ snapshot creation is DBMS specific.

and invocation get a sequence number of `0x1001`, and the in-situ snapshot comprises keys `k1` and `k2`, as `k1` with sequence number `0xFFFF` is skipped due to its lower component level.

In a multi-version system like neoDBMS, an in-situ visibility checking is performed. To this end, the shared state also comprises the version chain information and the list of in-flight transactions at the time of the NDP invocation. Given the invoking *transaction id*, the visibility check can now traverse the version chain backwards starting from the VID_{MAP} entry point (Fig. 4) to find the version, visible to the NDP transaction. We utilize newest-to-oldest order and thereby can ensure fast visibility checks, especially for fresh data [28]. For instance, neoDBMS (Fig. 8) will only construct an in-situ snapshot for $TX_{id}=3$ comprising version records `r4711` for VID/tuple `1` (as `r0711` as higher creation timestamp) and `r2606` for VID/tuple `0` (as its creation timestamp is the highest $\leq TX_3$).

3.3 NDP Interface

To enable an efficient pushdown of NDP commands, the lean interface definition of native storage [68] is extended. It builds upon NVMe, yet as a user-space module to avoid high user-/kernel-space switching overhead. Our native NVMe leverages SPDK [2].

Interface Design. Native storage [56] allows operating directly on physical memory, without any intermediary layers, by means of read/write/erase commands. This interface is extended by a command that transmits the current shared state via the NVMe payload to the device. Furthermore, an NDP_EXEC command extension sends parameter sets to device and can trigger a variety of executable functions (see Sect. 3.6). Its parameter set includes: (1) the shared state, and (2) the operation-specific parameters. Moreover, metadata and schema information are also included, i.e. column families and their respective data formats, number of LSM levels, assignment of SST per level, and many more.

Native integration. The NDP interface is deeply integrated into the DBMS. The entire stack is optimized to avoid copies of memory (zero-copy approach). Calls to computational storage are issued either synchronously through a central polling manager, or asynchronously through a callback function. The logical-to-physical address mapping is maintained within the storage manager, and updated on-the-fly with every I/O. Invalidated pages (e.g. after compaction) are marked for later garbage collection.

3.4 Parsers and Accessors

NDP operations must access and interpret persistent binary data in-situ without any interaction with the host. To this end, schema and data dictionary information must be present on device, and is propagated with the NDP call. It comprises information about DB-objects, their columns, types, sizes, or their physical representation. The on-device NDP infrastructure employs schema information to support data layout accessors for in-situ navigation, and format parsers for data interpretation [68, 72]. We also introduce physical page pointers to reduce the overhead of large address mappings.

Layout accessors exist for every element of the persistent data layout and help to navigate through the binary data organization and to access sub-elements. For instance, for a given key (Fig. 9), accessors allow navigating through the index block of an SST to the physical location of a record within a data block. Accessors are

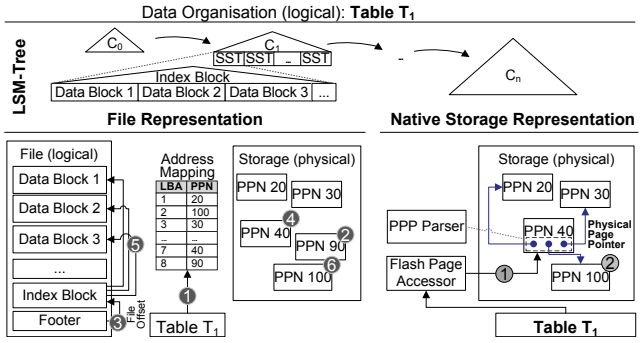

Figure 9: Physical Page Pointers eliminate the overhead of logical-to-physical address translation in file-based designs. The numbers indicate the necessary navigation steps.

simple to realize, with a microarchitecture resembling load units. They can be instantiated multiple times to increase the parallelism.

Format parsers. While accessors handle in-situ navigation, format parsers are required to *extract* persistent binary elements (records, values), interpret them semantically, and allow for further processing, mathematical operations, or comparisons. We actually distinguish field, record, and page formats and layouts for this purpose. For instance, in MyRocks, each element of the LSM-Tree based data organization (Fig. 9, right side) corresponds to a specific parser and accessor. The index block is interpreted according to its format, and the physical page pointers to the data blocks are extracted. Similarly, the data block is processed by the respective parsers and accessors to obtain the actual records, which themselves contain elements such as (a) an identifier, including a *column_family_id*, all *primary key* fields, the sequence number, and the key/value type; and (b) the actual value formatted according to the DDL definition.

Generation. Parsers and accessors are not necessarily static. As formats and layouts are declarative, parsers and accessors and can be *automatically* generated as software and/or hardware counterparts to support heterogeneous hardware, schema evolution [71].

Physical Page Pointers. Under native storage [56], the DBMS has direct control over the physical storage and manages the logical-to-physical address mapping. However, NDP-executions also necessitate address information in-situ, for on-device address resolution and intervention-free execution. The propagation of this information incurs high synchronization overhead. For instance, the size of the page-level address-mapping can be as large as 1 GB for 1 TB of storage. To this end, we introduce *Physical Page Pointers* (PPP, Fig. 9) that complement parsers and accessors, such that any reference within the persistent dataset is based on a PPP. They are designed for append-based storage (e.g. with LSM-Trees [48, 53], or Partitioned B-Trees [62]), since persistent data is immutable and is only modified by DBMS-controlled storage maintenance (i.e. garbage collection, compaction). PPPs eliminate the overhead of in-situ address resolution and address-mapping synchronization. The latter is still maintained, but only within the DBMS. For example, the index block of an SST utilizes PPP parsers and accessors to refer to the data blocks (Fig. 9). In contrast, traditional DBMS mostly use files and offsets within them, which require on-device address-mapping for in-situ navigation. To process an SST file (Fig. 9), the

DBMS extracts the address mappings for the file (1), loads (2) and processes (3) the index block (4). For each index block entry, the DBMS resolves (5) the address for the data block and (6) loads it.

3.5 Software and Hardware-based NDP

Today's computational storage devices come with various heterogeneous processing elements, ranging from classical scalar ARM processors and SIMD units to highly flexible FPGAs. Different NDP processing tasks may profit from software- or hardware-based processing, or from a combination of both.

Software. Software-based NDP is especially viable for low-latency operations [68], such as point lookups, as these are less parallel, and benefit from the faster scalar units. The development of software-based NDP functionality is straightforward and their software compilation times are relatively short.

Hardware. In comparison, hardware design is relatively tedious, error-prone, and requires more extensive debugging and testing [71]. Moreover, hardware compilation (e.g. FPGA-bitstream generation) is time-consuming. However, hardware implementations can speed-up processing significantly. Particularly, large scans are good acceleration candidates, due to their intrinsically parallel execution [68]. Furthermore, hardware units typically have multiple instances. In the proposed architecture, we configure the number of instances individually for each NDP invocation.

Software-Hardware Co-Design. Often, software- and hardware-based processing can be *combined* to form a flexible execution model. In our full update-aware NDP architecture, we foresee a scheduling engine, running in software, that dynamically decides whether to schedule a processing task on a hardware processing element, or to use the software-based alternative [68, 69].

3.6 NDP Pipelines and Operations

Even though the actual operation execution is not the primary focus of this paper, we describe how the proposed architecture handles the execution of sequences of NDP operations. With computational storage, we propose a *hybrid execution model*, combining pipelined block-at-a-time [58] and materialized execution strategies.

Inspired by [77], the operations in a demand-pull pipeline are split into operation execution groups. *Block-at-a-time (BaT)* execution [58] (formerly termed *vectorization* [77] – not to be confused with SIMD-vectorization) is achieved by embedding a buffering phase between any two execution groups. All the operators in a pipeline are connected through a record-at-a-time interface. The output of an operator within a group is passed on-the-fly to the next one, while the buffering stage, caches records internally until a buffer budget is reached. Once full, the next execution group can pull the buffered records over the same record-at-a-time interface. This mode leverages the device-internal memory hierarchy and heterogeneous processing elements, as the buffer stage can be placed in the device DRAM cache. Nonetheless, the buffer/cache budget remains a key limiting factor.

To this end, an operator can *materialize* intermediary or final results on the device (Fig. 10), and the next operator can operate on the materialized data (more details in Sect. 3.7). Local materialization allows: (a) the creation of complex NDP-pipelines, possibly with size-reducing operators at the end; (b) in-situ handling of non-size-reducing operations like joins or grouping, (c) the reduction of data transfers to host and more efficient DMA handling.

Currently, nKV supports the following NDP operations. *Get* retrieves the *value* for a given *key* and benefits from in-situ execution with low-latency [68, 69]. *Scan.* Both key and value filter-scans can benefit from parallel in-situ executions [68, 69]. Furthermore, our format parsers realize *projection.* Depending on the query, the query planner embeds it as an *early projection* [41] in the initial pipeline stages to reduce the size of the result set. Furthermore, we support *Joins* such as Block Nested Loop Join and Grace Hash Join that spill intermediate partition results to the computational storage. This is especially advantageous as joins are non-size-reducing. Finally nKV also supports hash table-based *GROUP BY* and *aggregation.*

Betweenness Centrality (BC) is a UDF used as an analytical HTAP operation. It performs a classical analysis on social graph data, and measures the degree to which nodes stand among each other. Our BC implementation in nKV is inspired by [18], and utilises the Node and Link tables of LinkBench [7] as graph representation. The logic , outlined in Algorithm [18, 68], sequentially scans the nodes with the NodeTableParser. In case the type of the node complies to the given search criteria, its neighbours are looked up via the LinkTableParser and distances are calculated recursively. Finally, the BC results are calculated according to the original algorithm in [18]. Overall, BC yields a random and sequential I/O mix.

3.7 Result-Set Handling

A key goal of update-aware NDP is to leverage in-situ processing capabilities and reduce data transfers to host. This encompasses the intermediary or final results of NDP-operations. Clearly, a naïve block-at-a-time strategy would cause excessive transfers.

A key insight is that, computational storage offers fast local memory (BRAM, fast HBM or DRAM) as well as ample and cheap storage. Furthermore, a native storage DBMS can exclusively allocate and control on-device memory, allowing in-situ executions to materialize intermediary or final results there (see Sect. 3.7). The update-aware NDP architecture allows non-size-reducing operations, such as joins or grouping, to materialize their results in-situ to reduce data transfers (Fig. 10), while the next operator in a pipeline can operate on the materialized data.

Planning and execution. The planner estimates the upper bounds of the sizes of intermediary and final results along an NDP-pipeline.

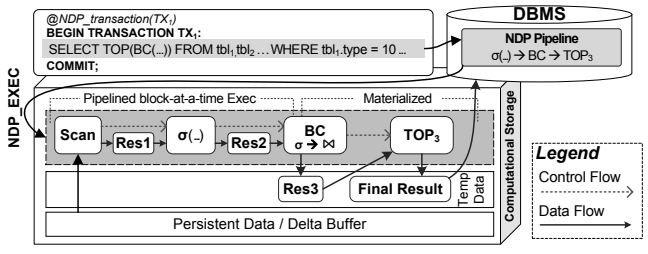

Figure 10: NDP-pipelines can be executed on the device, for faster processing or reduction of data transfers. Result set materialization is viable on computational storage.

If the estimate exceeds the buffer stage memory (BRAM, HBM), the *BaT* execution group ends, a *materialization stage* is injected in the NDP-pipeline, and another execution group begins.

Allocation. Depending on the size estimation, the planner and the storage manager employ an allocation strategy that targets fast levels of the on-device memory hierarchy first, i.e. static FPGA memories like BRAM or URAM, followed by fast on-chip HBM, and off-chip DRAM. If these resources are insufficient, a materialization and spilling strategy to *persistent* storage (e.g. NVM or Flash) is applied. To this end, every materialization stage is assigned an exclusive physical address range by the native storage manager. This may be the case for hash-join partitions, or aggregations with a high number of groups. If the space turns out to be insufficient during execution, the pipeline stalls and computational storage request more space from the DBMS in an extra roundtrip.

Space management and garbage collection. A native storage DBMS controls storage directly, manages logical-to-physical address mapping, and performs the garbage collection. It allocates and assigns exclusive physical address ranges to each materialization stage in a pipeline. Thus, the DBMS ensures that other transactions, pipelines or NDP operations do not overlap in the same storage space. Address ranges are preserved for the duration of the execution until the completion of the calling transaction. As part of commit/rollback processing upon its completion, the DBMS marks them for GC and schedules an asynchronous GC call.

4 EXPERIMENTAL EVALUATION

Experimental Setup. The experiments are conducted on two different system stacks (Fig. 11). The first, *MyRocks over* nKV, is based on MyRocks with nKV [68, 69] as storage manager and is used if not mentioned otherwise. The host is running Debian 4.9 OS and is equipped with a 3.4 GHz clocked Intel i5 CPU and 4 GB RAM. The *COSMOS+* board [54] is attached over PCIe Gen 2.0 ×8 and comprises a Zynq 7045 SoC with an FPGA, two 667 MHz ARM A9 Cores, and an MLC Flash module configured as SLC. *COSMOS+* is roughly equivalent to a consumer NVMe SSD or smart storage device (e.g. Samsung SmartSSD [22]) in terms of price and resources. The concrete configuration depends on the evaluation stack. MyRocks (MySQL 5.6) is configured with `Repeatable Read` as `Serializable` is not supported. Unless mentioned otherwise, the memory footprint is set to 7.5% of the dataset size (incl. 400 MB block buffer), and the mutable *memtables* are configured to 32 MB.

The second system stack, neoDBMS, is based on PostgreSQL12 and runs on an ARM Neoverse N1 System Development Platform (SDP) as host with 4 2.6 GHz ARM N1-CPUs and 3 GB RAM. A Xilinx Alveo U280 FPGA board with 2 GB DDR4 connected via PCIe Gen4 ×8 serves as enterprise-grade smart storage.

Baselines. We evaluate update-aware *NDP* against two baselines (Fig. 11): the *block* and the *native stacks* under nKV and neoDBMS.

Block/BLK (Baseline). The main baseline is the traditional, file-system stack with block-device storage. Out-of-the-box MySQL and PostgreSQL process OLTP and OLAP queries on the host, transferring all data from storage. We use *ext4* as file system and configure *Alveo U280* and *COSMOS+* as block devices. *COSMOS+* runs GreedyFTL with 1 MB DRAM cache for block device compatibility.

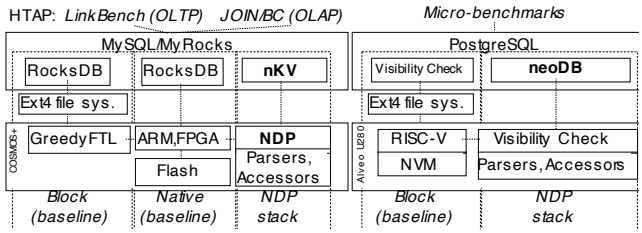

Figure 11: System Setup.

Native (Baseline) stack is lean and eliminates the file system and block-device layers, in contrast to *block*. Like *block*, *native* transports all necessary data from passive storage. It represents our second baseline as it builds the foundation of native NDP. *COSMOS+* is directly exposed to nKV userspace through the *native NVMe*.

NDP. Both nKV and neoDBMS introduce the concept of *native NDP* and build on top of *native*. This allows offloading the OLAP processing to the device where most of the data is already located, while the OLTP workload fetches the required data to the host on-demand. In MyRocks over nKV, one ARM core of the *COSMOS+* exclusively handles foreground I/O, while the other one performs the NDP/OLAP execution. Thus, NDP execution on the *COSMOS+* is limited to a single execution at a time, while the host benefits from its 4 core CPU. On-device, 200 MB DRAM are reserved as a hashtable-based block buffer for reading pages that can be used by the OLAP operations. neoDBMS relies on 16 RISC-V [1] processors on the FPGA that are operated via the TaPaSCo framework [42].

Workload. The workload is based on an HTAP-extended version of LinkBench [7] (if not specified otherwise). LinkBench [7] represents a social graph that is larger than the database memory. The graph is frequently updated by the OLTP-style transactional workload of LinkBench [7]. In addition, we introduce new analytical workload portions, performing graph analysis with either BC or `JOIN/GROUP BY` queries (see Sect. 3.6). The initial dataset comprises a graph with 10M nodes and 20 GB of data. The workload is controlled by several parameters described below.

- $OLTP_{SKEW}$: The OLTP workload operates on the hot portion of the dataset. The workload parameter $OLTP_{SKEW}$ sets the ratio of hot to cold data accesses.
- $OLAP_{SEL}$: To vary the complexity and runtime, the number of input nodes to BC is limited to a certain threshold - $OLAP_{SEL}$ - by filtering on the type of the `NODE` table (normal distribution).
- $OLAP_{PAUSE}$: It controls the time between two OLAP query injections. Due to the limited number of ARM cores on *COSMOS+* (one used for I/O, the other for NDP), the OLAP workload is currently restricted to only sequential executions.

Experiment 1: Update-aware NDP enables transactionally consistent NDP executions of OLAP operations in presence of OLTP updates in HTAP systems, without performance drops. We open with a general experiment, demonstrating that with NDP as part of the HTAP design space, analytical queries are executed without degrading the performance of the concurrent transactional workload, while analytical queries operate on the freshest data. To conduct this experiment, the HTAP workload is configured with $OLAP_{PAUSE} = 100s$ and $OLTP_{SKEW} = 40\%$.

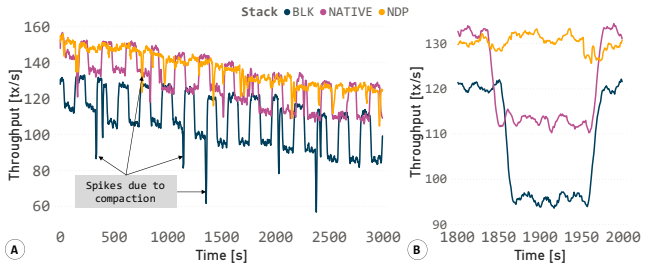

Figure 12: (A) LinkBench with HTAP extension is executed on the Block, Native, and NDP Stack. The throughput drops during OLAP queries due to increased I/O and the related buffer pollution. (B) Enlarged detail of one drop.

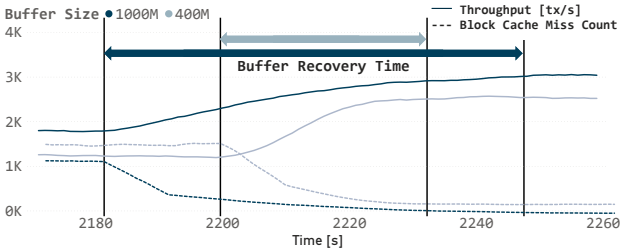

Figure 13: After an OLAP query, the cache misses (dashed) decrease as the buffer retains the hot dataset. The buffer recovery time (solid arrows) for the OLTP throughput (solid) to reach the original level depends on the buffer size.

Figure 12.A shows the OLTP throughput over time for all stacks. The *native* and *block* baselines exhibit significant performance drops whenever an OLAP query is injected. These are due to the increased number of read I/Os, as the cold data for the OLAP execution must be fetched from storage. In contrast, no buffer misses occur in the NDP stack due to the in-situ OLAP execution (Fig. 12.B).

Several aspects need to be considered. Firstly, OLAP processing incurs significant buffer pollution, as hot OLTP working set pages are evicted to make room for cold data. Even after the completion of OLAP processing and a workload switch back to OLTP, it takes time for the buffer to recover and retain the hot OLTP working dataset in memory (Fig. 12.B). We investigate this effect in a further experiment by varying the buffer size (Fig. 13). Clearly, the larger the DBMS buffer, the longer the adjustment time upon a workload change. Secondly, *NDP* and *native* have higher throughput (tx/s) compared to the *block* stack baseline, due to the leaner I/O stack. Lastly, each stack exhibits regular and sharp performance drops. These relate to compactions and flushes of the LSM-tree, and explain the gradual performance degradation over time (Fig. 12.A). Overall, the OLTP throughput of *NDP* is 30% better than *block* in the HTAP phase, and 12% better than *block* during the OLTP phase.

Insight. Extending the HTAP design space with update-aware NDP improves the overall performance. Offloading OLAP operations to computational storage preserves transactional consistency, reduces data transfers, and minimizes DBMS buffer pollution.

Experiment 2: Update-aware NDP is intervention-free, yielding robust and resource-efficient performance. Now we investigate the hypothesis that with *intervention-free* NDP in HTAP settings, in-situ OLAP processing does not impact host-side OLTP processing, yielding better CPU utilization and robust performance. The experiment (Fig. 14) sets the HTAP phase so that the time between two successive OLAP requests is $OLAP_{PAUSE} = 1000s$. We report the host CPU utilization, for host-only HTAP (*native* baseline), and for *NDP* OLAP-execution with concurrent host OLTP.

We observe significant drops in CPU utilization (Fig. 14), during the OLAP phase under the *native* stack. These are due to CPU stalls, while waiting for I/O to fetch cold data from storage for host-only HTAP processing. With *NDP*, these drops are minimized, as OLAP processing is offloaded to computational storage, and the in-situ execution is asynchronous and intervention-free. Therefore, the free host CPU resources are utilized for concurrent OLTP processing, as the working OLTP dataset typically fits in memory. Moreover, *NDP* leverages storage device resources that would otherwise remain idle. In particular, we utilize both *COSMOS+* ARM cores, the FPGA, and exploit the full Flash parallelism. In addition, *intervention-free* NDP translates into robust transactional throughput, as shown in the previous experiment (Fig. 12). Insight. Intervention-free NDP frees up host resources, making them available to other tasks.

Experiment 3: NDP can handle different types of OLAP operations. Our architecture handles different types of OLAP operations with good overall HTAP performance, utilizing on-device I/O properties and due to intervention-free NDP. To this end, we investigate `BC/TOP` and `JOIN/GROUP BY` as NDP-pipelines.

First, we consider `Selection/BC/TOP` to show how NDP dampens the effect of varying selectivity on OLAP executions. Notably, these are *size-reducing* operations. To this end, we vary $OLAP_{SEL}$, which determines the number of `NODE` table records that BC is processing. Thus, higher $OLAP_{SEL}$ yields higher OLAP read-intensity and more data transfers, as well as more nodes to be processed and longer OLAP runtimes (Fig. 15.B). Given the HTAP workload, Figure 15.A shows the throughput of a frequent concurrent OLTP transaction `GetLinkList`, with varying $OLAP_{SEL}$. With increasing $OLAP_{SEL}$ (Fig. 15.A), the average OLTP throughput decreases and its variance expands under the *block* and *native* stacks. This is due to the increasing OLAP duration, which causes more data transfers and a larger performance drop as observed in Experiment 1. Insight. With *NDP*, the throughput remains stable with varying $OLAP_{SEL}$.

Second, we consider `JOIN/GROUP BY/AGGREGATION` pipeline to show that NDP can handle *non size-reducing* operations, because of the hybrid execution model. We use the query: `SELECT n.type, SUM(c.count) FROM node n JOIN count c ON n.id = c.id`

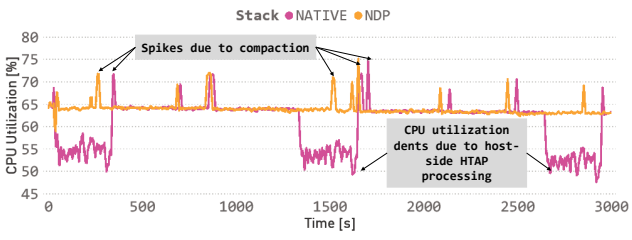

Figure 14: OLAP processing on the host, degrades CPU performance due to I/O wait time. NDP yields robust utilization of host resources, by leveraging on-device capabilities.

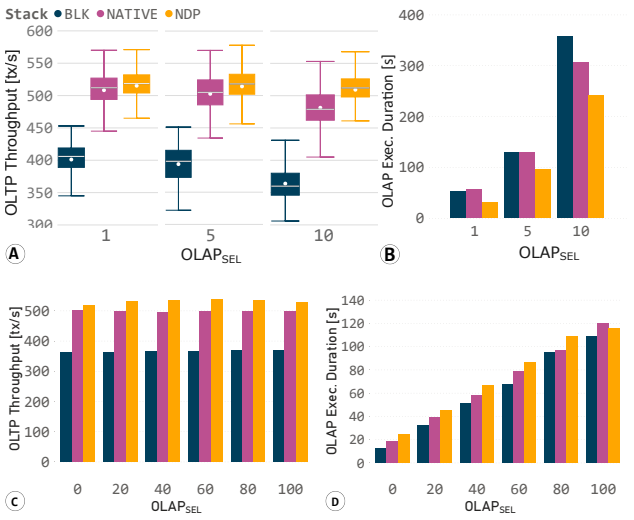

Figure 15: Executing BC as OLAP workload avoids dropping OLTP throughput Ⓐ, with increasing selectivities and OLAP runtimes Ⓑ. NDP outperforms native/block under OLTP, Ⓒ although JOIN/GROUP BY queries are slower on-device Ⓓ.

`WHERE n.type <= ? GROUP BY n.type;`. NDP and host plans resort to a BNLJ, while the on-device we resort to hash-based grouping, which does not spill to flash in this query. Again we vary the selectivity $OLAP_{SEL}$. Fig. 15.D shows increasing OLAP execution times with higher selectivities. Noticeably, NDP OLAP becomes compute-bound due to the slow on-device ARMs. Nonetheless, we achieve better overall HTAP performance due to intervention-free NDP.

Third, we vary the host resources for `Selection/BC/TOP`. In the first step, we increase the block buffer (Fig. 16.A/B), varying memory footprint from 8% (1.6 GB) to 13% (2.6 GB) of the dataset size. State-of-the-art approaches [20, 24, 47] aim at 10%. More host memory, yields better OLTP throughput (Fig. 16.A) and OLAP times (Fig. 16.B) under all stacks. With larger $OLAP_{SEL}$ and more memory, the OLAP gap between *BLK* and *NDP* shrinks, as larger memories shorten the OLAP performance drop length (Fig. 12) by reducing the buffer pollution. Next, we attach COSMOS+ to another host with a more powerful CPU. More logical cores improve OLTP performance (Fig. 16.C), but entail higher buffer pollution that slows down OLAP queries on *native* (Fig. 16.D) due to increased buffer contention, while NDP OLAP remains unaffected due to intervention-free NDP. In fact, the higher the host parallelism, the higher the potential improvement through update-aware NDP, due to better relative OLAP execution times. NDP preserves its advantage, whenever both cores and memory are increased in a lockstep, since the buffer pollution caused by better OLTP (more cores) counters the positive OLAP impact (more memory).

Experiment 4: Update-aware NDP reduces data transfers. One major benefit of NDP is that data is processed close to its physical storage location, and thus, reduces costly data transfers. To quantify this effect, we execute the read-only OLAP operation in isolation on each stack. Again, we vary the selectivity $OLAP_{SEL}$ to to increase the number of neighbours processed by *BC*, and the number of *Join/Grouping* nodes for OLAP query from Experiment 3.

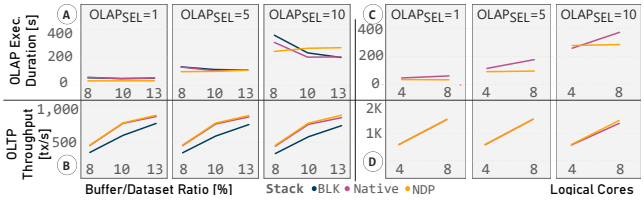

Figure 16: System performance behaviour with larger host memory footprints Ⓐ Ⓑ and more logical cores Ⓒ Ⓓ.

Figure 17.A clearly shows that the *native* stack baseline outperforms the *block* under all settings. This is due to the leaner I/O stack, reducing the amount of data to be read and transferred to host, and due to the advanced native NVMe storage manager that reduces I/O latencies (Fig. 17.B/C). Yet, *NDP* improves OLAP runtime by 52% over *native* and 48% over *block* for a lower number of neighbours (5K, 10K). With more neighbours, the number of nearest neighbour searches within the analytical operation rises, as does the number of nodes to be revisited by the algorithm as well. This behaviour benefits vastly from large buffers, which is a major constraint on commodity computational storage devices, given the limited *COSMOS+* DRAM capacity. Thus, buffer misses on the device entail more Flash reads under *NDP* relative to *native* and *block* (see Fig. 17.B). Regardless of these limitations, Figure 17.C clearly indicates that the device-to-host data transfers can be reduced significantly.

Figure 17.D shows the execution time of the OLAP query from Exp. 3 under the same conditions. The low NDP performance is due to the NDP BNL-JOIN compute-boundness, but also due to its I/O intensity (Fig. 17.E), which grows as expected, due to the small on-device join buffer. Nonetheless, the whole NDP-pipeline is size-reducing, keeping the number host-transfers *low* (Fig. 17.F).

Insight. NDP reduces reduces data transfers to host even further, but is constrained by the on-device processing capabilities (ARM) are significantly weaker than host CPUs.

Experiment 5: Update-aware NDP can operate on fresh data with low overhead. Operating on fresh data and supporting transactional guarantees is achieved at the expense of transferring the

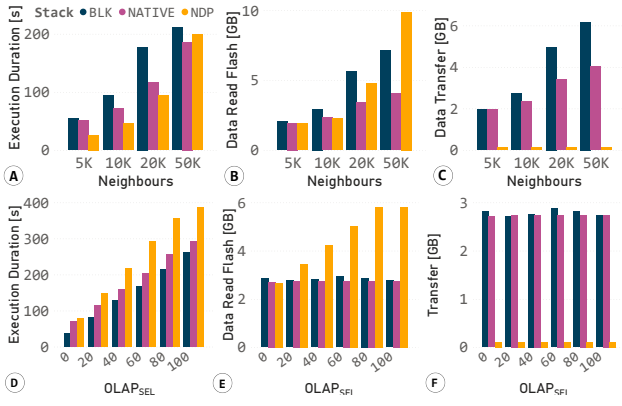

Figure 17: Ⓐ Processing BC, Native and NDP outperform Block. Ⓑ More neighbours yield more NDP I/O. Ⓒ Yet, host-device data transfers are reduced significantly. Similar effects are visible with JOIN/GROUP BY/AGGR query Ⓓ,Ⓔ,Ⓕ.

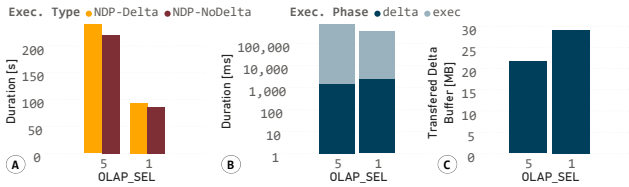

Figure 18: Ⓐ NDP operates on fresh data with low overheads as Ⓑ transfer times are low due to Ⓒ small delta-buffer sizes.

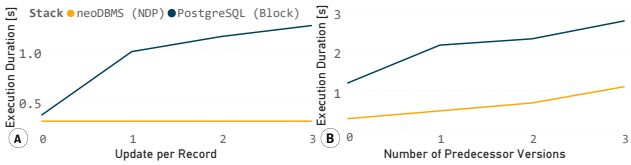

Figure 19: Ⓐ The most recent tuple-version is retrieved with overhead due to N2O in neoDBMS. Ⓑ Accessing predecessor versions is sped up by leveraging the on-device parallelism.

shared state to computational storage. We now quantify this overhead by executing the HTAP experiments with and without shared state transfers. We vary $OLAP_{SEL}$ to achieve different shared state sizes. Figure 18.A shows the average OLAP execution duration in both settings. This execution time is broken down (Fig. 18.B) into shared state transfer time to device, and the subsequent processing time on a logarithmic scale. Figure 18.C shows the average size of the shared state for those breakdowns.

The shared state transfers amount to just 1 second, irrespectively of $OLAP_{SEL}$, and represent a negligible portion (0.7%) of the overall execution time. The different shared state sizes are due to the parallel OLTP update activity: lower $OLAP_{SEL}$ entail lower OLAP runtimes and more OLTP transactions that yield more updates and larger shared states. The shared state size and thus the transfer overhead can be controlled through configuration parameters.

The remaining portion is due to operation dependent fresh data processing. As BC's execution time depends on the number of input nodes and $OLAP_{SEL}$, the gap of both scenarios (with and without shared state propagation) increases with higher selectivities.

Insight. Even though the shared state increases the data transferred from host to device, the time overhead is negligible: 0.7% of total NDP execution time of analytical queries.

Experiment 6: Computational storage can efficiently return the visible version or the transactionally consistent snapshot by means of NDP. So far we have investigated how CoW shared state and in-situ snapshot creation facilitates NDP processing. In this experiment, we investigate the impact of in-situ version visibility checking for multi-version DBMS.

To this end, we execute a micro-benchmark on top of TPC-C `OrderLine` table in the DB stack. It is subdivided into four phases. In each phase, an update (T_U) and a read (T_R) transaction are executed after each other. The update transactions update all tuples of the `OrderLine` table (the `ol_amount` column) and commit, thus producing a new version of each tuple and increasing the dataset size. The read transaction computes `SUM(ol_amount)`.

In a follow-up micro-experiment, we start each T_R, but leave it open, while all T_U commit, thus increasing the number of versions to four. Now compute `SUM(ol_amount)` for each reading transaction T_R (Fig. 19.B) in-situ and on the host. Figure 19.B shows the overhead of creating a snapshot at different points in time, and traversing the version chain to different predecessors. The in-situ snapshot creation time increases with the number of versions per tuple to be skipped. Nonetheless, the in-situ creation is 2× faster.

Insight. With update-aware NDP the storage device can provide the visible version and construct snapshots on the device. The runtime improves up to 4× by leveraging hardware parallelism.

Experiment 7: Update-aware NDP reduces the power consumption per transaction. Besides throughput, power consumption plays an important role for the spread of NDP. We now present the *end-to-end* power consumption of both the host and *COSMOS+* during the executions of Experiment 1. Noticeably, the host is not optimized for power measurements and has idling energy consumers, e.g. a graphics card. Overall, *block* demands the most power with 0.16 Watt/tx; *Native* is in the midfield with 0.14 Watt/tx; and *NDP* improves the power consumption to 0.12 Watt/tx and thus, by up to 26.1%. Thereby, the power draw of the storage device increases from 13.8 Watt (*block*) to 14.7 Watt (*NDP*). This is expected, as NDP offloads processing to device. Host power consumption increases as well from 44 Watt (*block*) to 50.5 Watt (*native*) due to the current native storage manager implementation. Its power footprint can be lowered with a better thread-management in future work. Nevertheless, update-aware *NDP* relieves the host and decreases the consumption to 47 Watt. Changing to a synchronous interface lowers the host-side power consumption of *native* and *NDP* by approx. 5% below that of *block* at the cost of some throughput.

Insight. Even though the total power draw is slightly higher on the device and the host, *native* and *NDP* execute more work, yielding 26.1% lower Watts/transaction compared to *block*.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we introduce update-aware NDP as a generic architecture for transactionally consistent in-situ processing in HTAP environments. The key idea is to propagate the most recent data, status, and system information to smart storage. As a result, a transactionally consistent snapshot can be constructed in-situ, on top of which read-only analytical NDP operations are executed. The evaluation indicates a 30% higher OLTP throughput in HTAP settings and update-aware NDP with 26% less Watts/transaction. We observe that shared state propagation overhead is marginal (≤0.7%) and that in-situ snapshot computation is 2×/4× faster.

Future Work. Offloading modifying NDP operations is an important challenge for reducing data transfers. They require synchronisation and invalidation mechanisms for disaggregated memory environments for transactional consistency [13]. Furthermore, efficient NDP logging space management techniques are necessary.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for the valuable comments. This work has been partially supported by *BMBF PANDAS – 01IS18081C/D*; *DFG neoDBMS – 419942270*; *HAW Prom, MWK, Baden-Württemberg, Germany.*

REFERENCES

[1] [n.d.]. RISC-V. https://riscv.org/.

[2] [n.d.]. SPDK. https://spdk.io.

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. ASPLOS* (San Jose, California, USA). 11.

[4] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O. In *Proc. SIGMOD*. 1103–1114. https://doi.org/10.1145/2588555.2610502

[5] 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*.

[6] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *Proc. CIDR*.

[7] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proc. SIGMOD*. 12.

[8] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Janus: A Hybrid Scalable Multi-Representation Cloud Datastore. *IEEE Trans. Knowl. Data Eng.* 30, 4 (2018), 689–702. https://doi.org/10.1109/TKDE.2017.2773607

[9] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proc. SIGMOD*, Vol. 26-June-20. 583–598. https://doi.org/10.1145/2882903.2915231

[10] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. 2015. JAFAR : Near-Data Processing for Databases. *In Proc. SIGMOD*.

[11] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. 2020. BlockNDP: Block-storage near data processing. In *Proc. Middlew.* 8–15. https://doi.org/10.1145/3429357.3430519

[12] 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. *Proc. ICDE* (2022).

[13] A. Bernhardt, S. Tamimi, F. Stock, A. Koch, T. Vincon, and I. Petrov. 2022. Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage. In *Proc. EDBT*.

[14] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221.

[15] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. *FAST*.

[16] 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.

[17] Amirali Boroumand, Saugata Ghose, Geraldo F. Oliveira, and Onur Mutlu. 2021. Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design. *CoRR* abs/2103.00798 (2021). arXiv:2103.00798 https://arxiv.org/abs/2103.00798

[18] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* (2001).

[19] 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*. 29–41.

[20] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proc. SIGMOD*. 505–520.

[21] Arup De, Maya Gokhale, Steven Swanson, and et. al. 2013. Minerva: Accelerating Data Analysis in Next-Generation SSDs. In *Proc. FCCM*.

[22] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs. *Proc. SIGMOD* (2013), 1221. https://doi.org/10.1145/2463676.2465295

[23] Jaeyoung Do, David Lomet, and Ivan Luiz Picoli. 2019. Improving CPU I/O performance via SSD controller FTL support for batched writes. In *Proc. SIGMOD*. https://doi.org/10.1145/3329785.3329925

[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *FAST)*.

[25] Facebook Inc., MyRocks. 2021. Transaction Isolation in MyRocks. https://github.com/facebook/mysql-5.6/wiki/Transaction-Isolation.

[26] 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, 1 (2012), 28–33. http://dblp.uni-trier.de/db/journals/debu/debu35.html{#}FarberMLGMRD12{%}5Cnhttp://sites.computer.org/debull/A12mar/issue1.htm

[27] 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.* Vol. 8. 1716–1727. https://doi.org/10.14778/2824032.2824069

[28] Robert Gottstein, Ilia Petrov, and et al. 2017. SIAS-Chains: Snapshot Isolation Append Storage Chains. In *ADMS@VLDB*.

[29] 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 (2010), 105–116. https://doi.org/10.14778/1921071.1921077

[30] Boncheol Gu, Andre S. Yoon, and et al. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proc. ISCA*.

[31] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro P. Buchmann. 2016. Revisiting DBMS Space Management for Native Flash. In *Proc. EDBT*.

[32] Masoud Hemmatpour, Mohammad Sadoghi, and et al. 2016. Kanzi: A Distributed, In-memory Key-Value Store. In *Proc. Middlew.*

[33] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. In *Proc. VLDB*.

[34] Insoon Jo, Duck-ho Bae, and et al. 2016. YourSQL : A High-Performance Database System Leveraging In-Storage Computing. In *Proc. VLDB*.

[35] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* (1998).

[36] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[37] Jungwon Kim and et al. 2017. PapyrusKV: A High-performance Parallel Key-value Store for Distributed NVM Architectures. In *Proc. SC*.

[38] Sungchan Kim, Hyunok Oh, and et al. [n.d.]. In-storage Processing of Database Scans and Joins. *Inf. Sci. 2016* ([n. d.]).

[39] 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 (jan 2016), 183–200. https://doi.org/10.1016/j.ins.2015.07.056

[40] Hideaki Kimura, Alkis Simitsis, and Kevin Wilkinson. 2017. Janus: Transactional processing of navigational and analytical graph queries on many-core servers. In *Proc. CIDR*.

[41] Christian Knoedler, Tobias Vincon, Arthur Bernhardt, Lukas Weber, Leonardo Solis-Vasquez, Ilia Petrov, and Andreas Koch. 2021. A cost model for NDP-aware query optimization for KV-stores. *Proc. DAMON* (2021).

[42] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. 2019. The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems. In *Applied Reconfigurable Computing*.

[43] 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. - Int. Conf. Data Eng.* 2015-May (2015), 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[44] 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.* Vol. 8. 1740–1751. https://doi.org/10.14778/2824032.2824071

[45] 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 (2018), 421–444. https://doi.org/10.1007/s00778-018-0503-z

[46] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated Data Centers: Challenges and Trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26. https://doi.org/10.1109/MCOM.001.1900612

[47] Chen Luo. 2020. Breaking Down Memory Walls in LSM-Based Storage Systems. In *SIGMOD*.

[48] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[49] 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*, Vol. Part F1277. 37–50. https://doi.org/10.1145/3035918.3035959

[50] Sang-woo Jun Ming, Arvind, and et al. 2015. BlueDBM: An Appliance for Big Data Analytics. *Proc. ISCA* (2015).

[51] 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) 2044*, Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen (Eds.). Gesellschaft für Informatik e.V., Bonn, 499–502.

[52] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[53] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (jun 1996), 351–385.

[54] OpenSSD Project 2019. *COSMOS Project Documentation.* OpenSSD Project. http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources.

[55] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proc. SIGMOD 2017*. 1771–1775.

[56] Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. 2019. Native Storage Techniques for Data Management. *Proc. ICDE* (2019).

[57] Ivan Luiz Picoli and Philippe Bonnet. 2020. Open-Channel SSD (What is it Good For). *Cidr* (2020).

[58] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards Practical Vectorized Analytical Query Engines *(DaMoN'19)*. Article 10, 7 pages.

[59] Vijayshankar Raman, Gopi Attaluri, and Ronald Barber. 2013. DB2 with BLU Acceleration: So much more than just a column store. *Proc. VLDB* 6, 11 (2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[60] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *Proc. SIGMOD* (Portland, OR, USA) *(SIGMOD '20)*. 2043–2054.

[61] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In *Proc. VLDB*.

[62] Christian Riegger, Tobias Vinçon, Robert Gottstein, and Ilia Petrov. 2020. MV-PBT: Multi-version indexing for large datasets and HTap workloads. In *Adv. Database Technol. - EDBT*, Vol. 2020-March. 217–228.

[63] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A real-time OLTP and OLAP system. In *Proc. EDBT*, Vol. 2018-March. 540–551. https://doi.org/10.5441/002/edbt.2018.65 arXiv:1601.04084

[64] Sudharsan Seshadri, Steven Swanson, and et al. 2014. Willow: A User-Programmable SSD. *USENIX, OSDI* (2014).

[65] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. DoppioDB: A Hardware Accelerated Database. In *Proc. SIGMOD*.

[66] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proc. SIGMOD* (Scottsdale, Arizona, USA). 12.

[67] T. Vincon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, and I. Petrov. 2018. NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management. In *Proc. EDBT*.

[68] 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*.

[69] 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. *PVLDB* 12 (2020).

[70] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[71] Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vincon, Christian Knoedler, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. 2021. A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems. *Proc. RAW@IPDPS* (2021).

[72] Lukas Weber, Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. 2021. On the necessity of explicit cross-layer data formats in near-data processing systems. *Distributed and Parallel Databases* (2021).

[73] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB* (2014).

[74] Louis Woods, J. Teubner, and G. Alonso. 2013. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proc. SIGMOD*.

[75] 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.

[76] Sam Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. 2015. Beyond the Wall: Near-Data Processing for Databases. *Proc. DAMON* (2015).

[77] Jingren Zhou and Kenneth A. Ross. 2004. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proc. SIGMOD 2004* (Paris, France) *(SIGMOD '04)*. 191–202.