nDT: The case for in-storage Data Transformations

Arthur Bernhardt
Johannes Kratz*
Ilia Petrov
arthur.bernhardt@reutlingen-university.de
johannes.kratz@student.reutlingen-university.de
ilia.petrov@reutlingen-university.de
Data Management Lab, Reutlingen-University
Reutlingen, Germany

David Volz
Sajjad Tamimi
Andreas Koch
volz@esa.tu-darmstadt.de
tamimi@esa.tu-darmstadt.de
koch@esa.tu-darmstadt.de
Embedded Systems and Applications Group,
TU-Darmstadt
Darmstadt, Germany

Abstract

In this paper we propose an approach for performing data transformations near- or in-storage. The currently prevailing approach of extracting the data and then transforming it to a target format suffers data movement and causes degraded system performance. To mitigate these challenges we propose an approach offloading data transformations as near-data processing operations. The results show robust performance of foreground workloads and lower resource contention. We present opportunities in multi-engine and multi-system settings, for ML pipelines and for reuse.

Keywords

Data Layout Transformation, Near-Data Processing

1 Introduction

Motivation. Data analytics pipelines are widely used to extract insights from OLTP or data ingestion systems or answer prediction queries. To this end, data is exported and transformed into open columnar data formats [41] such as Arrow [3], ORC [4], or Parquet [5] to be consumed by data analytics and machine learning frameworks. However, exporting and transforming OLTP data is slow and expensive [25, 30], as it is resource-intensive and incurs significant data movement. The latter is the root cause of many inefficiencies [8, 14] as fresh OLTP data must be scanned and extracted, incurring slow I/O, as well as memory and CPU contention with foreground workloads. Given the trend towards super-linear data growth rates, the impact of movement will continue to increase in the future.

What's more, data analytics and machine learning pipelines are deployed on popular Python frameworks such as Pandas, PyTorch, TensorFlow, or SciKit-Learn. Noticeably, such widespread analytics and machine learning tools do not integrate with DBMS beyond bulk dataset transfers [30]. To reduce the export and transformation cost, typically small samples are extracted, which inevitably reduces the accuracy of analytics algorithms [30]. As these are separate tools, the advantages of modern Hybrid Transactional Analytical Processing (HTAP) DBMS for analytics pipelines cannot be exploited. The pressure to export, transform and import large amounts of data across systems is likely to increase, since Python tool eco-systems and DBMS are separate systems.

*Work performed while at Reutlingen University.

EDBT '27, Lille (France)

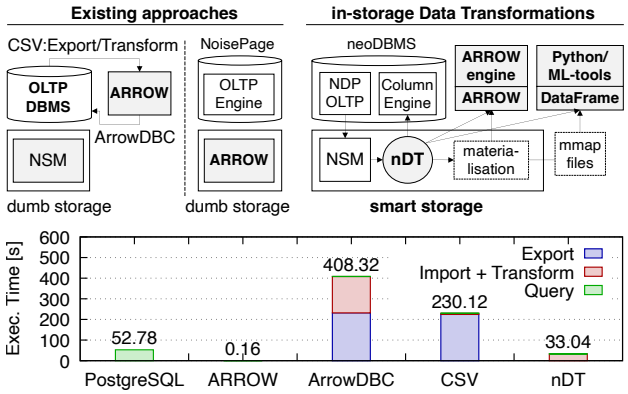

Figure 1: (Top) various alternatives for data transformations: by exporting DBMS data and importing it in the target system; directly accessing the source DBMS; and in-storage. (Bottom) comparison of these approaches under HTAP settings with simultaneous OLTP workload.

State-of-the-Art Overview. Given the strong focus on data analytics and the necessity to export data from various sources, there is a rapid spread of open formats [3–5]) and analytical systems that support them [20, 28]. Yet, prior work has shown that exporting and transforming data to the target format is a major source of inefficiency [25, 30] (Fig. 1, top left). To this end, new designs [25] (Fig. 1, top middle) investigate storing the main part of the dataset in Arrow, with a novel OLTP engine on top. DuckDB [20] or Velox [28] among other systems can natively operate on Arrow.

However, the majority of the proposed approaches, while efficient and universal are hardware-oblivious and do not primarily target data movement. Yet, there has been significant progress in the field of hardware, especially intelligent-storage or -memories. With the advance of semiconductor memories and fabrication processes, it has become economical to produce combinations of storage and compute elements packaged on the same device. Such intelligent storage devices allow offloading operations for on-device processing and offer better device-internal I/O characteristics (bandwidth, latencies) than device-to-host. Hence, they offer a possibility for data transformations and addressing data movement. [8] presents a vision for transparent data transformations that can be offloaded to smart memories. Along the same lines [32] demonstrates an approach for near main-memory row-column transformations.

Contributions. This paper proposes the offloading of data transformations to smart storage. We argue that nDTs can be executed

10.48786/edbt.2027.07

as near-data processing (NDP) operations and are a natural fit for smart storage/memory to reduce data movement.

The main idea is to perform data layout transformations on-device on compute cores close to the physical data location, instead of effortfully moving all the OLTP data to the host (Fig. 1, top right). Thus, nDT benefit from the high on-device bandwidth. Furthermore, nDTs are executed fully autonomously on-device without host-DBMS intervention or resources contention.

A key aspect of nDT is that in HTAP settings, they execute against a transactionally consistent snapshot of the database computed on-device. They offer snapshot isolation transactional guarantees and high data freshness, thus enabling multi-engine settings. nDT also offer optional time-travel semantics.

In HTAP settings, nDT ensures robust co-execution by running autonomously and contention-free on-device, leaving foreground workloads to the host-DBMS.

Furthermore, nDT support streaming transformation results to the host for multi-engine settings, but also materializing them on device for later reuse, or for consumption by other systems. Notably, nDT provide for fast incremental updates of materialized and transformed data (so called delta-nDT), i.e., given a prior materialization, only the fresh delta can be computed, in a repeated transformation runs. While materialization and reuse are well-studied optimizations [18, 19, 27, 39] also leveraged in the ML-domain [16, 35, 40], our work demonstrates that those are also applicable in NDP-settings.

Lastly, nDT are economical. Transformations are faster in-memory, yet more expensive given the current cost of ~20 000$/TB server-DDR5 vs. ~500-1000$/TB of fast smart storage.

The potential of the proposed in-storage transformations is demonstrated in Fig. 1. We compare different approaches under HTAP settings and the CH-benchmark [12], while running the data intensive Q6 in parallel to the OLTP load.

We establish PostgreSQL and the ARROW engine [2] as our primary baselines to highlight the performance difference when each engine operates on its native format. In addition, we compare different import mechanisms (ArrowDBC, CSV) allowing the ARROW engine to obtain the most recent data from the DBMS in advance. The runtime under PostgreSQL indicates the overhead of data movement, while the other comparands indicate the additional overheads of data export and transformation. nDT is fast as it mitigates data movement by offloading the ARROW transformation to storage.

2 Related Work

Given the diverse modern workloads, HTAP systems manage row-organized layouts, transforming them to columnar layouts for OLAP processing [21, 24, 31, 33], or dynamically adapting layouts [1, 23] on the host. Prior work [9, 11, 22, 36] was able to accelerate analytical operations with NDP, reducing data transfers. RME [32] demonstrates on-the-fly layout transformations in main-memory, utilizing FGPA-based custom hardware, or DDR memory controller. To the best of our knowledge, nDT is the first to present transactionally consistent on-the-fly layout transformations running in-storage on simple PEs, supporting multi-engine/system settings.

3 Key Challenges

Transactional guarantees. Equipping nDT with strict transactional guarantees is essential for transactional processing within multi-engine DBMS and frameworks that lack transactional capabilities. A second key aspect is the ability to execute nDT asynchronously in an interruption-free manner. The main observation is that slow and blocking host-device roundtrips can be avoided, for instance for schema- or address-information. This way co-execution is achieved, i.e. while the host-DBMS continues processing the foreground workload, the device executes nDT autonomously without contention.

On-device materialization. A central challenge in nDT lies in enabling and orchestrating the materialization of results on-device, in contrast to streaming them up to the host. This way a richer set of execution models becomes possible, involving for instance vectorized or materialized execution, depending on the operations, the involved engines, or the on-device hardware, e.g. vector PEs. Materialization is economical on smart storage because of the lower (≥20×) price/TB relative to DRAM, or the re-computation savings.

Reuse. Another aspect that materialization enables is reuse. For instance, it may allow cooperative execution of analytics pipelines that share common input or intermediary data. Furthermore, its materialization fosters multi-engine settings. Lastly, if an older nDT result is already materialized, a fast incremental/Δ-nDT, should bring it up-to-date at low overhead.

4 DBMS design for in-storage transformations

A key idea of this paper is that nDT can be executed as a near-data processing (NDP) operation on smart storage, mitigating some of the well-known overheads that the layout dichotomy and transformation pose on DBMS designs [7]. To this end, nDT leverages snapshot-based NDP on multi-versioned storage. This strategy has been proven feasible in PostgreSQL, MyRocks, and MySQL, demonstrating broad system applicability [9, 11, 36–38]. Importantly, it fulfills the prerequisites for nDT, which include: mechanisms to synchronize the latest modifications with the device and multi-versioned storage. In the following, we overview the design of such NDP-DBMS for nDT on the basis of neoDBMS and it's PostgreSQL extensions [36] and describe the nDT execution model.

Design Overview. In nDT's execution model, the host-side OLTP engine safely manages concurrency and write-heavy workloads. As foreground transactions execute, neoDBMS incrementally accumulates modifications, and places them in a small shadow area called *shared-state* [36] (Fig. 2). The shared-state is regularly propagated at low overhead as it is kept small and

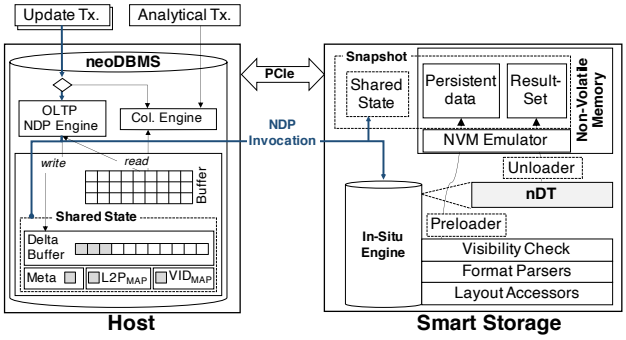

Figure 2: neoDBMS **NDP architecture and** nDT **integration.**

configurable in the range of a few hundred KB to a few MB. Notably, the shared-state is the only delta between the *working set* in the large host-DBMS main-memory with the most up-to-date data, and the much larger but colder and *complete* dataset on smart storage. Crucially, nDT performs no data transformations along the write path. All transformations occur on-the-fly along the read-path as part of nDT invocations, with the option for materialization and reuse. As a result, write-heavy workloads are not bounded by on-device compute resources.

A key point is that after a shared-state propagation, the smart storage attains all data necessary to construct a transactionally consistent DBMS snapshot in-situ. The on-device processing begins from the shared-state and only then moves to the cold data, as version records in the latter may have been invalidated by newer ones in the former.

The Delta-Buffer (Fig. 2) is a key element of the shared-state. It accumulates the versions newly created by active transactions, while predecessor versions remain in the main buffer. Furthermore, the shared-state accumulates incremental changes to auxiliary structures such as data dictionary definitions, the address-mapping table ($L2P_{map}$), or the VID table (VID_{map}), which we introduce below.

To support shared-state propagation and on-device snapshot construction, neoDBMS organizes version records as a singly backwards-linked *new-to-old (N2O)* list [9, 36]. To mark the entry-point of a chain, neoDBMS employs a VID_{map} containing the RecordID of the latest version record for each tuple. All version records in a chain have the same *Virtual ID* (VID) as they belong to the same tuple. Maintaining logical addresses is crucial as the host DBMS operates on virtual memory and file system abstractions, while the device operates on physical addresses. To resolve a RecordID in-situ, neoDBMS maintains a logical to physical address-mapping ($L2P_{map}$) that is incrementally propagated with the shared-state. This enables easy host-device coordination, as both can resolve VIDs independently, while ensuring snapshot isolation guarantees.

4.1 Transaction Consistency

All engines leveraging nDT for query execution operate on the latest committed snapshot/version of the database, attained through shared-state propagation ahead of any nDT invocation. In addition, these invocations run within a transaction context started by the DBMS transaction manager. As a result, nDT offers Snapshot Isolation at its minimum, but also `SERIALIZABLE` with transaction managers supporting SSI [10, 29]. nDT is executed in-situ on the on-device snapshot, while the columnar results are streamed up to the analytical engine.

4.1.1 NDP/nDT Invocation. Before offloading any NDP operation, e.g., nDT to smart storage, the NDP-DBMS prepares the execution. It retrieves the unique `TxID` of the calling transaction and creates a list of all transactions currently in-flight, both necessary for in-situ execution and in-situ snapshot creation. Next, an nDT invocation is constructed, which comprises the transaction context, the involved DB objects, and the operations to be executed. Along these lines, neoDBMS snapshots the current shared-state, to be propagated alongside the NDP call.

During the preparation, the NDP-DBMS estimates and allocates the storage and on-device resources necessary for the invocation. The estimation is performed based on the expected selectivity, the necessary compute resources or cache/FPGA-BRAM sizes, and the current device load. Storage space is provided as new DB pages, reserved and/or allocated for the pending nDT invocation. The NDP space management is performed solely by the DBMS. Therefore, if the nDT execution runs out of storage, it requests additional pages from the DBMS at the cost of an extra roundtrip. To this end, the current nDT execution is temporarily suspended and only resumes after the space allocation completes. Noticeably, neoDBMS configures and allocates on-device resources for each NDP call.

Upon nDT invocation, the worker threads of the NDP engine get suspended and sleep-wait for the response of the in-situ engine. The NDP engine may perform further invocations, which will execute against their own on-device snapshots.

4.1.2 On-device nDT Scheduling/Preparation. nDT processing on the individual smart storage processing elements (PEs) begins with the arrival of the nDT invocation command. The *in-situ engine* schedules the workload across the number of PEs specified in the invocation. This is done by partitioning the in-situ VID_{map} uniformly over all PEs in a round-robin manner and assigning each partition to a PE. The pre-allocated storage for the nDT invocation is uniformly distributed over and assigned to all PEs.

Next, the in-situ engine launches the nDT (Sect. 4.2) as partitioned *NDP jobs* across the number of PEs. The jobs start by performing a visibility check (Sect. 4.1.4) for each tuple starting from the latest version record (referenced by VID_{map} entry) in their partition. To this end, nDT processing must be able to navigate on-device, which we briefly explain next.

4.1.3 In-situ Navigation and Data Interpretation. nDT must access, interpret and navigate the persistent binary data in-situ without costly host interaction and transfer-intensive roundtrips. To this end, schema information is propagated with the nDT invocation to the device. It comprises information about DB-objects, their schema (attributes, types, sizes), or physical representation. The on-device NDP infrastructure employs that information to support data layout accessors for in-situ navigation, and format parsers. These are pre-compiled MicroBlaze [26] binaries.

Layout accessors exist for components of the persistent data layout to navigate through the binary data organization in-situ. For example, for the NSM-layout, individual records or their headers.

Format parsers extract persistent binary elements (records, values), interpret them semantically, and allow for further processing, mathematical operations, or comparisons. We distinguish field, record, and page formats and layouts for this purpose.

In-situ address resolution. is an important part of in-situ navigation necessary to reduce data transfers. It occurs, for instance, during a version-chain traversal, when to retrieve a version-record, the in-situ engine resolves its *RecordID* to retrieve its physical location. To this end, the NDP-DBMS utilizes the logical-to-physical address mapping ($L2P_{map}$) that is incrementally propagated alongside the shared-state. It is then used to resolve logical addresses to physical locations without slow and blocking roundtrips to the host engine.

4.1.4 In-situ Snapshot Construction. To create a snapshot on-device, the visibility check must determine the latest version record of a version chain, committed prior to nDT invocation. To this end, the in-situ engine takes the snapshot information from the nDT invocation, i.e. the transaction ID (`TxID`) of the calling transaction, the shared-state (delta-buffer and the $L2P_{map}$ and the VID_{map}).

Given the *N2O* version organization, the visibility check is performed by traversing the records in a version-chain backwards, and comparing their creation timestamps against the `TxID`. The visibility task on each PE extracts the entry-point RecordID of each entry in the PE's VID_{map} partition 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* (Sect. 4.1.3) on the PE, which retrieves the corresponding NSM page slot. The *layout accessor* extracts the record header and passes it to the format parser to retrieve the transaction timestamp for comparison against `TxID`. The invalidation timestamp is available from the predecessor that has already been processed. The visibility decision is taken based on both timestamps. All visible versions are passed on to the nDT execution.

4.2 Near-storage Data Transformations

nDT is designed to efficiently handle generic data layout transformations in-storage. We illustrate nDT capabilities through a row-to-column conversion, transforming NSM-formatted (row-organized) OLTP-data to an Arrow compatible columnar storage format. Arrow serves as a cross-language, cross-platform data layer that enables seamless data interchange between various systems and programming languages. Arrow employs a memory layout that adheres to a zero-copy approach, optimizing data movement and minimizing overhead. Metadata (Arrow Schema) provide the blueprint for interpreting and processing the data without unnecessary conversions. Moreover, Arrow introduces record batches, a table-like data structure, which represents a logical grouping of data across columns.

To transform the NSM data to Arrow, the smart storage processing elements (PEs) employ a fast scratchpad memory (BRAM, 64 KiB) as shown in Fig. 3 and 6. The scratchpad provides a high-performance memory resource that facilitates quick data access and manipulation. To this end, a 8 KiB scratchpad partition is reserved and used for loading database records after snapshot computation and visibility checks (Fig. 3.A). The remaining free scratchpad space is partitioned equally for each attribute taking part in the transformation and requested by the projection. For each attribute that may contain `NULL` values, an additional validity bitmap partition is created. Moreover, variable-length attributes result in an extra scratchpad partition to store offset information. Our experiments indicate that the scratchpad can be sized much smaller without compromising performance, e.g. transformation performance of the CH/TPC-C `Orderline` is the same with 16KiB and 64KiB.

For efficient data rearrangement the row-to-columnar layout transformation relies on scratchpad-to-scratchpad copies (Fig. 3.B). Layout accessors would be sufficient for fixed-length

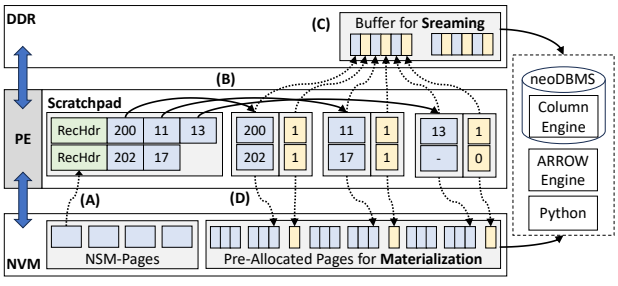

Figure 3: nDT **execution.**

data types during the transformation process, as they also account for NSM layout alignment rules in data types. However, to handle variable-length data types and type conversions, format parsers are employed to handle the specific transformation. The record header is processed to determine `NULL` values in any attribute and fill the validity bitmap (Fig. 6) partitions accordingly. When scratchpad partitions become full, they are flushed to pre-assigned pages (Fig. 3-C/D).

4.3 Reuse and Materialization

nDT supports on-device result handling and different result consumption modes (Fig. 3), which address specific transformation requirements and help optimize the data processing pipeline.

Streaming. nDT allow the streaming of results up to different host systems as they are produced while transformation proceeds. This mode leverages the concept of Arrow streams, and the transient nature of Arrow. nDT ensures efficient interleaving of result production and consumption by the host engine. To this end, multiple small buffers in on-device DDR are reserved to host transformed results. Whenever they get full, a host notification is triggered and a new result batch can be loaded. The PEs continue to concurrently prepare the next batch of data, ensuring a continuous data flow.

On-device materialization. More importantly, the in-situ engine may also materialize the transformation results on device. In materialization mode, the PEs flush their scratchpad partitions into pre-allocated NVM pages that are uniformly distributed across all PEs, allowing for better parallelism. Upon nDT completion, PEs return the addresses and the sizes of every columnar structure and their validity bitmaps and offset vectors (Fig. 3), while unused space is marked free. In the rare case of insufficient result space, the in-situ engine temporarily suspends the nDT jobs and requests more free space from the NDP-DBMS.

4.4 Hardware Architecture

Our experiments are conducted on an ARM Neoverse N1 (N1-SDP [6]) as the host, equipped with 4× 2.6 GHz ARM N1-CPU cores and 16 GB RAM, connected to a computational storage device (CSD) via PCIe Gen3 x16. The CSD is built upon an AMD/Xilinx Alveo U280 (AU280) FPGA board, instantiated with up to 8 Processing Elements (PEs) executing NDP operations and nDT. These PEs are instantiated as MicroBlaze [26] scalar soft-cores running at 200 MHz with full access to the on-board memory. *A key intuition is that all on-device engines and NDP-operations (transformations, visibility-checker, accessors and parsers) are flexible software tasks written in C and therefore easily applicable to other architectures.* Compared to commodity system, the compute capabilities of the PE is low, reaching a Coremark [17] score of just 400 it/s. Because these wimpy PEs lack the robust memory architecture of modern embedded cores (e.g., out-of-order execution, HW prefetchers, deep caches), we extended them with scratchpad memories and software-managed data movement (preloader/unloader). To accurately emulate the latency characteristics of NVM storage, we use a *hardware* NVM emulator [34] The CSD is configured to match the performance of Intel Optane (3D XPoint) SSD and provides 32GB of NVM memory.

5 Multi-Engine System-Designs

nDT enables fully decoupled storage architectures and the use of multiple analytical engines to achieve both high data freshness

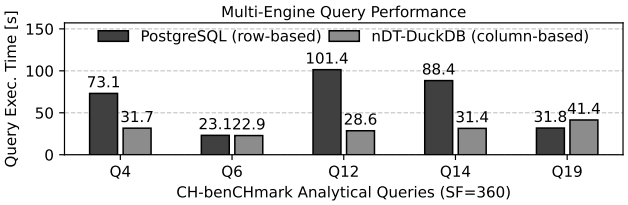

Figure 4: nDT enables the cooperative use of multiple engines, achieving both high data freshness and improved analytics.

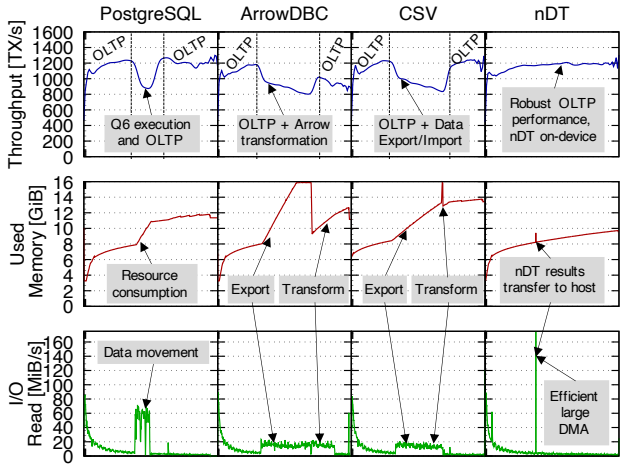

Figure 5: nDT lowers transformation impact on foreground workloads, maintaining low overhead analytics in HTAP.

and efficient analytics. While multiple engines do not share their formats, all operate on a tx. consistent snapshot of the database.

We extended PostgreSQL's OLTP-focused row-based engine with a column-based engine to demonstrate performance improvements for analytical queries in isolation and concurrent HTAP settings. A single thread is allocated per analytical query, prioritizing OLTP foreground workloads. Fig. 4 shows that employing multiple engines can be beneficial, especially for data intensive operations while nDT performs on-the-fly data transformation in-situ.

Furthermore, in HTAP configurations, performing analytical queries alongside exporting and transforming data is resource intensive and incurs significant data movement negatively impacting performance (visible as throughput drop in Fig. 5). nDT mitigates this by asynchronously and interruption-free executing the transformation in-situ and only returning the requested subset of data back to the engine, minimizing resource contention and resulting in better and robust performance, as nDT does not impact the foreground OLTP workload (Fig. 5). In addition, nDT greatly minimizes the memory footprint by eliminating intermediate in-memory data scans and buffering, mitigating the impact of data duplication during format transformation making it an economical solution.

Insight: nDT may lead to efficient multi-engine DBMS designs by reducing data movement and resource contention. We envision a new type of poly-model stores where different nDTs (e.g., row-column, or KV-to-column), are offloaded to smart storage, relayout persistent data in multiple data layouts on the way up to the host.

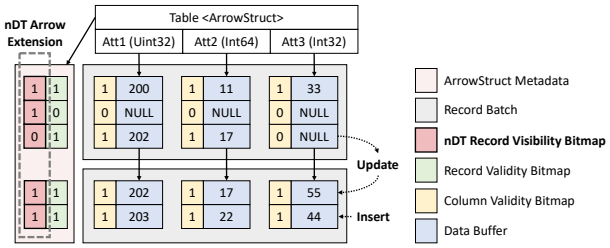

Figure 6: Integration of delta transformations in Arrow.

6 Opportunities for Reuse

A core challenge in supporting reuse is to avoid the full cost of re-materialization during transformations if the original OLTP data has changed. To this end, we propose delta-nDT and an extension to columnar-formats like Arrow. Δ-nDT efficiently determine the OLTP modifications to the OLTP database that occurred past the initial materialization on device. Comparing the calling `TxID` against the `TxID` of the latest materialization allows us to calculate the delta during in-situ snapshot construction (Sect. 4.1.4). Only the modifications are then transformed and appended to the existing materialization. To distinguish between outdated and new entries, nDT creates a light-weight visibility positional bitmap (Fig. 6) per delta creation, enabling snapshot-isolated access to transformed data.

To demonstrate the efficiency of Δ-nDT, we start with an initial materialization and increase the percentage of modifications, measuring the Δ-nDT creation time (Fig. 7-A). Noticeably, the creation time increases linearly with the number of modifications. Next we evaluate Δ-nDT in an end-to-end setting. With few modifications (e.g., 10%) Δ-nDT outperforms a native PostgreSQL execution by 1.95× - 3.12× (depending on how much data is buffered), indicating efficient delta construction and transformation, avoiding the full cost of re-materialization (Fig. 7-B).

Insight: Delta-nDT and reuse are equally applicable to both multi-engine and multi-systems settings. Reuse opens an economical tradeoff, as it reduces (re-)computation and increases performance at the cost of storage space, which is cheaper than DRAM.

7 Multi-System Architectures

The export and transformation overhead is especially prominent in multi-system settings, where OLTP data is exported and transformed to an open format, e.g., Arrow or Pandas DataFrame, to be imported into a different system, e.g. DuckDB, SciKit-Learn. We show the benefits of exposing OLTP data via nDT in Fig. 8, comparing different export/import mechanisms without restricting CPU resources, prioritizing transformation and analytical

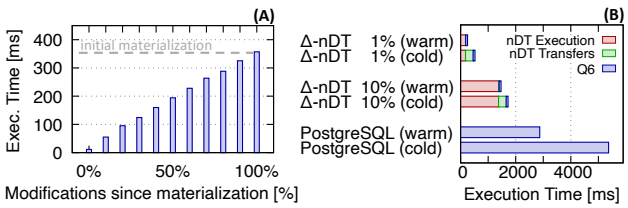

Figure 7: Efficient incremental transformations with delta-nDT offer opportunities for multi-system/engine designs.

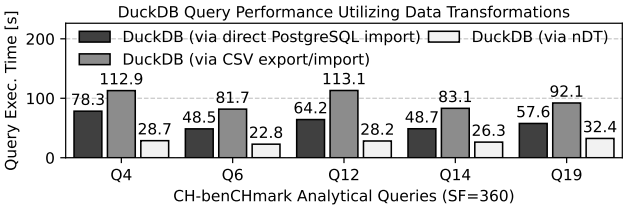

Figure 8: Impact of transformations on analytical workloads.

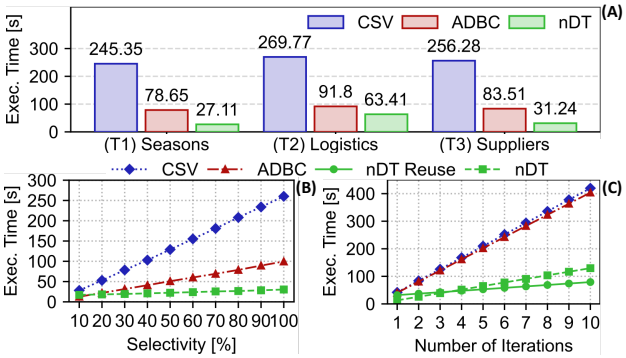

Figure 9: Impact of nDT on ML pipelines. On-device materialization facilitates data reuse, improving iterative accesses inherent to training algorithms.

performance. In this setting, data is requested, transformed, and consumed directly in a python environment, not as DBMS engine.

Interface. nDT introduces an interface called NDBC, resembles other python DBMS and SQL interfaces. In addition to specifying the SQL query, NDBC allows the configuration of the result-set format (currently Arrow or Pandas DataFrame). If requested, nDT reifies the result (transformed OLTP-data) as memory-mapped files [13] (Fig. 1, right). To this end, they can be readily consumed by various tools, e.g., DuckDB, SciKit-Learn. This mitigates wire-protocol overhead [25, 30] visible in Fig. 1-ADBC and enables zero-copy access to the same data across multiple engines/systems. Importantly, nDT requires no changes to the analytical engines as NDBC exposes the transformed data as a streaming data source.

Execution Models. Materialization and streaming enable different execution modes, e.g., materialized and vectorized. An important observation is that with such models, different system types can be served, e.g., OLTP, OLAP (vectorization), and machine learning (materialization). However, the workloads exhibit diverse consistency requirements. Operational analytics demands the freshest OLTP data, whereas ML pipelines require stable, consistent views over longer durations. nDT can leverage snapshot-based materialization and reuse to address both needs. Beyond avoiding re-materialization, concurrent systems can transactional consistent share transformed data even when operating on different snapshots.

8 Machine-Learning Scenarios

We evaluate nDT in an end-to-end machine learning pipeline using SciKit-Learn to train a k-Means clustering model. Our evaluation utilizes the TPC-H `LINEITEM` table (scale factor 10) across three distinct clustering tasks. (T1) Seasonal pattern analysis: Aggregates average daily return rates, grouped by month and day to detect fluctuations. (T2) Logistics optimization: Calculates average shipping duration per shipping mode to identify delays. (T3) Supplier segmentation: Clusters suppliers based on recency, frequency, and monetary metrics. For each task, `LINEITEM` data is extracted from the DBMS, transformed into Pandas *DataFrames*, and preprocessed prior to model training. The workload is read-only and therefore distinct from the HTAP scenario in Fig. 1.

Fig. 9-A compares the end-to-end training utilizing CSV export, ADBC, and nDT. While all approaches only request a subset of the data from the DBMS, only nDT forwards and executes transformations directly in-storage, applying selections in-situ and therefore reducing data movement to achieve an overall speedup of 1.45× - 9.05× across all tasks. In addition, the impact of selectivity on the first clustering task is shown in Fig. 9-B.

Much of nDT reuse potential (Sect. 6) lies in data analytics and ML pipelines. For instance, for cooperatively deployed data analytics and machine learning pipelines [15] on the same dataset. Reuse is also relevant for iterative analytics over the transformed nDT-results. In addition, reuse fosters materializing statistics computed in previous pipeline stages or different pipelines, thus trading computation, analytical performance, and resource consumption for space. Notably, on-device materialization for reuse is economical as smart storage space is much cheaper than DRAM and low-overhead, leveraging the high on-device bandwidth (~30 GiB/s) and the price/TB is more than 20× lower.

To demonstrate the benefits of nDT's reuse mechanism in ML pipelines, we iteratively execute the first clustering task described above and measure the training time across increasing iterations. Unlike the nDT baseline without materialization, the reuse strategy fully materializes all requested columns during the initial request, intentionally ignoring all filter criteria. The transformed data is solely persisted on-device. After materialization finishes, all subsequent requests are served directly from this pre-materialized snapshot, eliminating overhead from parsing, visibility checks, and repeated transformations on device. While the initial materialization incurs higher costs than on-the-fly transformation, this cost is quickly amortized after just three iterations (Fig. 9-C). Unlike CSV and ADBC, nDT achieves fast iterative accesses without requiring the DBMS to buffer the entire dataset in main-memory, potentially increasing resource contention as demonstrated in Fig. 5.

Insight: nDT allows efficient consumption of transformed OLTP data in multi-system settings. The efficient materialization of nDT-results on smart storage emerges as a key mechanism. Furthermore, low-overhead interfaces such as NDBC, `mmap` and DataFrames support efficient integration in different eco-systems.

9 Conclusions

We propose near-data layout transformations that reduces host resource contention while leveraging high internal smart storage bandwidth. Within a reasonable cost/performance ratio, nDT accelerates HTAP and ML analytics via cheaper storage offloading.

Acknowledgments

The authors wish to thank the anonymous reviewers for the insightful comments. This work has been funded by the DFG (Deutsche Forschungsgemeinschaft) grant *neoDBMS – 419942270.*

References

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

[2] Apache Foundation. 2023. ACERO: Arrow execution engine. https://arrow.apache.org/docs/cpp/streaming_execution.html.

[3] Apache Foundation. 2023. Apache Arrow. https://arrow.apache.org/.

[4] Apache Foundation. 2023. Apache ORC. https://orc.apache.org/.

[5] Apache Foundation. 2023. Apache Parquet. https://parquet.apache.org/.

[6] ARM Corp. 2022. ARM Neoverse N1 Software Development Platform. Online. https://www.arm.com/-/media/global/products/processors/N1%20Solution%20Overview.pdf

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proc. SIGMOD*. 16 pages. doi:10.1145/2882903.2915231

[8] Manos Athanassoulis. 2021. Transparent Data Transformation: Can I Have My Rows and Eat My Columns Too? *SIGMOD Rec.* 50, 2 (aug 2021), 33–34. doi:10.1145/3484622.3484630

[9] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Carsten Heinz, Christian Knoedler Tobias Vinçon, Andreas Koch, and Ilia Petrov. 2022. neoDB: In-situ Snapshots for Multi-Version DBMS on Native Computational Storage. *Proc. ICDE* (2022).

[10] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *Proc. SIGMOD (SIGMOD '08)*. 729–738. https://doi.org/10.1145/1376616.1376690

[11] Wei Cao et al. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *Proc. FAST*. 14 pages.

[12] Richard Cole et al. 2011. The Mixed Workload CH-benCHmark. In *Proc. DBTest*.

[13] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*.

[14] Bill Dally. 2011. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *Proc. IPDPS*. 878. doi:10.1109/IPDPS.2011.420

[15] Behrouz Derakhshan, Zoi Kaoudi, Tilmann Rabl, Volker Markl, et al. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *Proc. SIGMOD*. 15 pages. doi:10.1145/3514221.3526186

[16] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1701–1716. doi:10.1145/3318464.3389715

[17] EEMBC. 2023. CoreMark - EEMBC Embedded Microprocessor Benchmark Consortium. https://www.eembc.org/coremark/.

[18] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: reusing results of MapReduce jobs. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 586–597. doi:10.14778/2168651.2168659

[19] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 75–88.

[20] Pedro Holanda and Jonathan Keane. 2021. DuckDB quacks Arrow: A zero-copy data integration between Apache Arrow and DuckDB. https://arrow.apache.org/blog/2021/12/03/arrow-duckdb/.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. doi:10.14778/3415478.3415535

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

[23] 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. doi:10.1109/ICDE.2011.5767867

[24] 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. doi:10.1109/ICDE.2015.7113373

[25] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (dec 2020), 534–546. doi:10.14778/3436905.3436913

[26] MicroBlaze, Xilinx Inc. 2022. MicroBlaze Soft Processor Core. https://www.xilinx.com/products/design-tools/microblaze.html.

[27] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. 3, 1–2 (Sept. 2010), 494–505. doi:10.14778/1920841.1920906

[28] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384. doi:10.14778/3554821.3554829

[29] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[30] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (jun 2017), 1022–1033. doi:10.14778/3115404.3115408

[31] 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. doi:10.14778/2536222.2536233

[32] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2023. Relational Memory: Native In-Memory Accesses on Rows and Columns. In *Proc. EDBT*.

[33] 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 pages.

[34] Sajjad Tamimi, Arthur Bernhardt, Florian Stock, Ilia Petrov, and Andreas Koch. [n. d.]. NVMulator: A Configurable Open-Source Non-Volatile Memory Emulator for FPGAs. In *Proc. ARC'23*.

[35] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1285–1300. doi:10.1145/3183713.3196934

[36] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proc. VLDB Endow.* 15, 10 (jun 2022), 1991–2004. doi:10.14778/3547305.3547307

[37] Tobias Vincon et al. 2020. nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing. *PVLDB* 12 (2020).

[38] Tobias Vincon, Lukas Weber, Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2020. nKV: Near-Data Processing with KV-Stores on Native Comp. Storage. In *Proc. DaMoN*.

[39] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in MapReduce framework. *Proc. VLDB Endow.* 7, 3 (Nov. 2013), 145–156. doi:10.14778/2732232.2732234

[40] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: holistic optimization for accelerating iterative machine learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. doi:10.14778/3297753.3297763

[41] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *CoRR* abs/2304.05028 (2023).