DANSEN: Database Acceleration on Native Computational Storage by Exploiting NDP

SAJJAD TAMIMI, Technical University of Darmstadt, Germany
ARTHUR BERNHARDT, Reutlingen University, Germany
FLORIAN STOCK, Technical University of Darmstadt, Germany
ILIA PETROV, Reutlingen University, Germany
ANDREAS KOCH, Technical University of Darmstadt, Germany

This paper introduces DANSEN, the hardware accelerator component for neoDBMS, a full-stack computational storage system designed to manage on-device execution of database queries/transactions as a Near-Data Processing (NDP)-operation. The proposed system enables Database Management Systems (DBMS) to offload NDP-operations to the storage while maintaining control over data through a *native storage interface*. DANSEN provides an NDP-engine that enables DBMS to perform both low-level database tasks, such as performing database administration, as well as high-level tasks like executing SQL, *on* the smart storage device while observing the DBMS concurrency control. Furthermore, DANSEN enables the incorporation of custom accelerators as an NDP-operation, e.g., to perform hardware-accelerated ML inference directly on the stored data. We built the DANSEN storage prototype and interface on an Ultrascale+HBM FPGA and fully integrated it with PostgreSQL 12. Experimental results demonstrate that the proposed NDP approach outperforms software-only PostgreSQL using a fast off-the-shelf NVMe drive, and significantly improves the end-to-end execution time of an aggregation operation (similar to Q6 from CH-benCHmark, 150 million records) by ≈10.6×. The versatility of the proposed approach is also validated by integrating a compute-intensive data analytics application with multi-row results, outperforming PostgreSQL by ≈1.5×.

CCS Concepts: • **Hardware → Reconfigurable logic and FPGAs**; **Hardware-software codesign**; **Hardware accelerators**; **External storage**; • **Information systems → Database query processing**.

Additional Key Words and Phrases: Near-Data Processing, Computational Storage, FPGA, Database Management Systems.

1 Introduction

Data movements have become a limiting factor in modern Database Management Systems (DBMS) as they use traditional (*data-to-code*) system architectures to process data by triggering massive data transfers. As an alternative, moving computation close to the storage devices (*code-to-data*) gains the potential of significant throughput improvement due to shorter / wide connections of high(er) performance busses. In addition, DBMS are working on *hybrid workloads* that combine long-running analytical (reading) transactions/queries and frequent and low-latency modifying (update) transactions [35]. However, analytical queries also operate on rarely accessed data that resides on slower "cold" persistent storage. Operating on cold storage causes massive data transfers from storage systems in the traditional stack [44].

Offloading computations from the host to the storage device, known as Near-Data Processing (NDP), is used to save bandwidth and increase performance. Active Disks [1] and Database Machines [12] as a pioneer in this era were not successful due to I/O boundedness and bandwidth limitations, and the high cost of the special hardware. However, with modern semiconductor technologies,

Authors' Contact Information: Sajjad Tamimi, tamimi@esa.tu-darmstadt.de, Technical University of Darmstadt, Darmstadt, Germany; Arthur Bernhardt, arthur.bernhardt@reutlingen-university.de, Reutlingen University, Reutlingen, Germany; Florian Stock, stock@esa.tu-darmstadt.de, Technical University of Darmstadt, Darmstadt, Germany; Ilia Petrov, ilia.petrov@reutlingen-university.de, Reutlingen University, Reutlingen, Germany; Andreas Koch, koch@esa.tu-darmstadt.de, Technical University of Darmstadt, Darmstadt, Germany.

it is now economical to fabricate combinations of powerful compute units and storage (e.g., non-volatile memories) close to or co-allocated in the same device (known as *computational storage*). Co-allocating computation and storage on the same device results in shorter access latency, higher bandwidth, and a higher level of parallelism for the internal compute unit (device-internal) compared to the external one (host-to-device). This is due to the independent channels between the on-device processing element and storage unit as well as the limited simultaneous access provided by the interconnect, i.e., PCI Express (short: PCIe). Leveraging the hardware properties and moving computation closer to the storage significantly enhances system performance, as device-internal bandwidth and latency are better than device-to-host. For example, the shorter access latency between the processing element and the physical storage unit facilitates leveraging the *byte-addressability* of novel non-volatile memory to improve further system performance, an approach that is less efficient when relying on the PCIe interface, which is optimized for bulk transfers. Offloading computation to storage is even more promising for DBMS systems operating under hybrid workloads, as it reduces resource contention and prevents performance interference between concurrently running host and device processes. By executing long-running analytical workloads on cold data, which exceeds the main memory capacity, on the device, and optimizing the execution of frequent and low-latency update workloads on hot data within main memory, system efficiency is significantly enhanced. This strategy improves performance, minimizes data transfers, and reduces memory pollution. These trends leverage the way for NDP-capable storage and overcome limitations of prior approaches like Active Disks or Database Machines. Please see Section 5 for an in-depth discussion of prior work.

This paper presents DANSEN, the hardware component of neoDBMS [8, 44], which together form an *end-to-end* system architecture for accelerating database applications on computational storage through the use of NDP techniques. The storage solution is PCIe-based and employs NVM, emulated on DRAM, for storage and an FPGA as the computing unit that handles queries and transactions from DBMS as NDP-operations. The DBMS is designed to perform NDP-operations on persistent storage to improve database performance while running hybrid workloads. To enable the DBMS to directly access and store database data on the device, as well as efficient push-down of NDP-operations to the device, we utilize *Native Storage* [18, 19, 37], which makes the DBMS aware of the specifics of the underlying storage medium. The concept of the Native Storage and DANSEN's implementation of an *Native Storage Interface* (NSI) is described Section 3.2.

The offloaded operations will be executed on an *NDP-engine* that is comprised of a pool of software-programmable soft-cores extended with custom hardware that acts as a *front-end* to the DBMS and interprets the data layout and concurrency mechanisms of the database. As the front-end required significant engineering effort and frequent changes due to the lack of existing best practices on how to directly integrate NDP into PostgreSQL, a more flexible software-based implementation was used here instead of hardwired acceleration. For performance, we leverage up to twenty 64-bit cores working in parallel to feed the NDP back-end with raw data, which then employs actual hardware-accelerated processing for aggregations or inference. The native storage approach leverages the byte-addressability of NVM to reduce read-amplification and improve overall system performance that would not be possible using page-addressed storage such as Flash. Read amplification is a phenomenon characterized by retrieving (much) more data from physical storage than is actually requested by the processing component, e.g., by the query executor. For instance, if the executor requests a single record of 100 B, the buffer manager will load the 8 KiB page containing that record, while the underlying file-system may load multiple blocks. Read-amplification results primarily from introducing levels of indirection, the data-organization or abstractions such as the block-device interface, file-systems and file-based storage.

To evaluate the proposed architecture, we build a prototype of the smart storage system on an AMD / Xilinx UltraScale+ FPGA, attached in the form of a Xilinx Alveo U280 PCIe board. We extended PostgreSQL 12 with the capability to perform NDP-operations using the FPGA on the persistent storage, enabling us to conduct a thorough evaluation of the system performance. The FPGA not only performs computational operations from database queries, but also directly processes the database internal formats while maintaining the transactional integrity of the database. This is achieved by creating a transactionally consistent snapshot of the database on the device and executing the NDP operation against it, thus enabling DANSEN to execute *intervention-free* (autonomously) from the host DBMS. Our evaluation shows that DANSEN improves system performance by decreasing costly data movement and enhancing the degree of computational parallelism. In addition, we use an ML-inference application from data analytics as an NDP operation to show how easy custom hardware can be integrated into the back-end pipeline of the proposed NDP-engine. As the NDP back-end is isolated from the DBMS specifics by the NDP front-end, it just has to deal with streams of raw data.

The novel contributions of this paper are as follows:

(i) We propose a PCIe-based computational storage that enables efficient "push-down" of the NDP-operation from the DBMS query engine to the NDP storage device. The interaction between the storage and DBMS is realized in the form of NSI and so eliminates costly intermediate abstraction layers (e.g., file systems, block devices) that are present in traditional DBMS stacks. In addition, we extended the DBMS to handle NDP on computational storage while running hybrid workloads.

(ii) We propose an NDP-engine architecture that is capable of performing in-situ data processing without any interaction with the host as an intervention-free NDP-operation. The proposed NDP-engine exploits the byte-addressable nature of NVM technologies to reduce read-amplification. The NDP-engine is not only designed and optimized for NDP-operations but also enables the user to easily integrate custom hardware and arbitrary software functions.

(iii) We investigate the generality and flexibility of the proposed architecture by the in-depth benchmarking of a compute-intensive application from the data analytics domain, in an end-to-end scenario (PostgreSQL DBMS down to FPGA-based hardware).

Note that since DANSEN was developed in an interdisciplinary research project between database researchers and computer architects, we sometimes have to use a mix of the two different communities' terminologies. For example, the term "push-down" from the DBMS research community could be seen to mean "hardware-acceleration" from the computer architects' perspective.

This text focuses on a detailed description of the *hardware* architecture and *hardware/software* interfaces, while our previous work in [8, 44] primarily focused on the *database architecture* and design principles/aspects. This work thus discusses three key contributions by the DANSEN system: (i) the architecture of an NDP-capable computational storage along with its implementation of NSI, (ii) the design of a novel hardware-accelerated NDP engine processing pipeline, and (iii) a detailed evaluation of a full-stack end-to-end usage scenario reaching from the full-scale host-side DBMS down to hardware-accelerated NDP for ML inference while maintaining transactional consistency in the presence of ongoing DBMS updates.

This discussion starts with a general introduction of some DBMS basics in Section 2. Afterward, the overall system architecture of DANSEN including the required modifications to PostgreSQL and the actual hardware architecture are explained in Section 3. Details of the experimental setup are given in Section 4.1, while Section 4.2 discusses the actual evaluation results. Prior related work is surveyed in Section 5, before we conclude in Section 6.

2 A Brief Introduction to Database Systems

As this is an interdisciplinary work, we give a necessarily brief introduction into DBMS for computer architects here down to the level we require to discuss the actual DANSEN system architecture in Section 3.

Modern DBMS handle thousands of transactions per second, while they are at the same time processing and analyzing large datasets. This advancement in technology has had several notable implications. First, especially in cloud settings, sudden workload changes from transactional (mostly writing) to analytical (mostly reading) and vice versa are inevitable. These changes cannot be anticipated as they occur due to user behavior and may happen frequently. Therefore, both types of processing typically co-exist on the same dataset and system. Second, modern DBMS do not limit their operations to processing hot data that can easily fit into large main memories. Concurrent analytical processing operates on both the hot, but also the much larger and cold, persistent portion of the dataset that typically resides in the persistent storage (much larger, but far slower than main memory). Especially in such cases, massive and slow cold data transfers severely impact system performance and scalability and hurt resource efficiency. These are inevitable because of the scarce I/O bandwidth in combination with poor data locality and traditional system architectures, mandating that data must be transferred to the host DBMS to be processed there (data-to-code). To tackle these challenges, NDP can be utilized to reduce data transfers by offloading parts of data-intensive processing to *computational* (sometimes also called *smart*) storage.

We now discuss these aspects in greater detail before considering how data is actually stored and accessed in a modern DBMS. The latter will have direct impact on the HW/SW architecture of DANSEN.

2.1 Workload

In database systems, sequentially handling individual operations is not the primary focus. Instead, these systems are designed to handle and process a *multitude* of *concurrent* transactions. A *transaction* refers to a sequence of database operations (i.e., SQL statements) that constitutes an *atomic* unit of work, demarcated by a beginning (i.e., begin work) and an end (i.e., commit/rollback work). Database *workloads* [16] encompass a predefined and parameterized set of different transaction types, involving modification operations and/or queries, that operate on a *dataset* with a *schema* describing one or more database objects (e.g., SQL tables).

Transactions are best described through their distinctive properties [17]. *Atomicity*, which refers to the all-or-nothing execution of a transaction, ensures that *all* operations are either successfully executed or are *all* automatically undone in case of failure. This guarantees that the dataset is in the same state as it was prior to the beginning of the transaction. *Isolation* mandates that concurrent transactions are executed by the DBMS as though they were processed *individually* and are *unaffected* by other modifications. This eliminates concurrency control anomalies [10] like the dirty read, which involves reading data modified by an active and concurrent transaction. *Transactional guarantees* are provided in accordance with the selected transaction isolation level (e.g., SERIALIZABLE). In addition, upon successful transaction completion (i.e., commit work), *all* modifications of that transaction become *durable*, enabling them to survive a range of failures (e.g., media crash), and the database is left in a *consistent* state, with all constraints satisfied.

In general, there exist two distinct workload types: Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP). OLAP involves long-running analytical transactions (consisting mostly of read queries), while OLTP typically involves frequent, low-latency modifying transactions (consisting mostly of modification write accesses). In modern database systems, both OLAP and OLTP workloads may be executed on the same dataset and system, known as Hybrid

Transactional and Analytical Processing (HTAP) [35]. However, this type of workload may result in costly transfers of cold data in the traditional DBMS stack [44], as analytics such as aggregations will need to read large amounts of data (e.g., "ISBN numbers of all books ordered in the last 10 years" for a histogram).

2.2 Concurrency Control

Concurrency control ensures that no concurrency anomalies occur and that transactional guarantees (i.e., properties) can be achieved. Over the years, different types of concurrency control schemes have been designed, with *pessimistic* and *optimistic* being two general classes. The former seeks to prevent conflicts and avoid concurrency anomalies by blocking other transactions from accessing (writing or reading) a data item (e.g., record) through the use of locks. The latter seeks to increase transactional throughput by minimizing or avoiding locking/blocking at the cost of early conflict checking, multiple versions of a data item, and/or validation.

Pessimistic concurrency control schemes can significantly reduce overall throughput as either high-frequency update transactions will set locks, blocking the execution of long-running transactions or vice versa, since both types of transactions operate on some common data items. To this end, optimistic schemes come to the rescue. In this work, we used Multi-Version Concurrency Control (MVCC) as a common optimistic scheme to guarantee transactional properties and have higher throughput.

In terms of the DANSEN systems architecture, this implies that the NDP operations must be aware of the MVCC protocols, and be able to find the record versions current for the *specific* NDP transaction, which are not necessarily just the latest available versions! This is one of the main tasks of the NDP front-end described in Section 3.3.1.

2.2.1 Multi-Versioning and MVCC. Multi-version DBMS and MVCC schemes [7, 10] are well-suited for modern HTAP workloads. The fundamental concept of these systems is to maintain multiple versions of each record in a table. Whenever a transaction modifies a record (via updates, inserts, or deletes), a new version of that record is created. Thus for every transaction reading a record, the multi-version DBMS determines the latest committed version of the record prior to the start of that transaction (so-called *visible version*) and returns it. This process, known as *visibility checking*, effectively computes a virtual snapshot of the entire database. Importantly, this technique allows reading transactions, such as scan operations, to proceed with the visible version *without* requesting a lock and waiting. Consequently, in DBMS using MVCC schemes, read transactions are never blocked by write transactions, and vice versa, except in cases where multiple writing transactions modify the same data, resulting in *write/write conflict* [9, 42]. Nonetheless, this technique offers a good performance under HTAP workloads, where the long-running analytical (reading) transactions are not affected by the numerous short modifying OLTP transactions. Additional details on visibility checking can be found in Section 3.1.4.

2.2.2 Version Organizations. Each version of a record is physically represented as an independent and uniquely identifiable database version record, identified by its Record-ID (short: recID). Currently, there exist various version organizations with distinct properties [48], though they all share common characteristics. First, all versions of a record form a logical *version chain* with a predecessor and a successor version. Second, each version record stores the transaction ID of the transaction that created it (creation_timestamp), as well as the transaction ID of the transaction that invalidated it by creating a successor version (invalidation_timestamp), or a value of NULL if no successor version exists. Third, every version record includes a reference (pointer, recID) to the predecessor or successor version record.

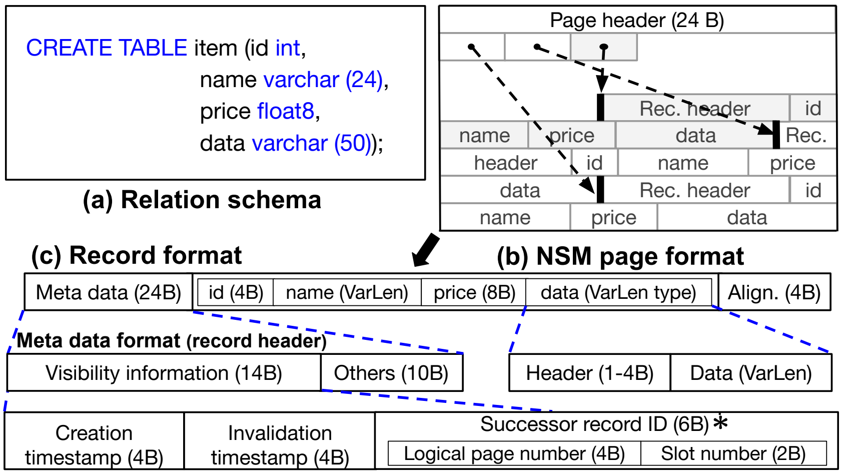


Fig. 1. N-ary Slotted Model (NSM) page format (b) and record format (c) for relation schema (a). *: In DANSEN, the successor record ID is replaced with the predecessor record ID.

2.3 Database Storage Model

2.3.1 Database Records and Pages. Data in a database is logically organized in tables, which are defined by a relational schema, as shown in Fig. 1-(a). This schema is the typed definition of a table and specifies the table's name, structure, and type of information stored in each column. Thus it defines the format of all records and how they are stored in memory or on disk. Each record has a recID, which remains *unchanged* throughout its lifetime and in its simplest form comprises <Page#, Slot#>. Records are stored on pages, which are contiguous blocks of memory with a specified format and configurable size ranging from 2KB to 32KB. Modern DBMSs offer a variety of page formats/layouts to host records, and the choice of page layout can have a significant impact on the overall performance of the DBMS. The N-ary Slotted Model (NSM) layout, as shown in Fig. 1-(b), is a widely used page layout in relational DBMS (so-called *row-stores*) that support frequent modifications and operate on whole records. As shown in this figure, the NSM format, contains a dictionary of slot pointers, located after the Page Header, which specify the offset of a record within the page and some status flags. NSM is particularly advantageous for variable-sized record modifications, where it can avoid costly record migration by hosting them on the same page as the original. However, it is sub-optimal for analytical workloads that require high cache efficiency, as it implies transferring all data.

In this paper, we use the NSM page format and assume a fixed-format/schema, variable-sized records, with a Record Header (i.e., metadata) consisting of version information followed by a body holding the field values as shown in Fig. 1-(c). Data types in the record format are either of fixed length, such as Integer or Float, or variable length, such as text or binary coded decimals. Variable length types require an additional header that encodes the length, and if the length exceeds one byte, a flag is used to indicate the header size, which may increase to 4 bytes. The metadata in the Record Header is used to store information about the transactions involved in creating or modifying records, including timestamps and predecessor information for record updates in MVCC. This metadata is crucial for ensuring transactional consistency and performing visibility checks. Records are always aligned on a 4-byte boundary, and if necessary, additional bytes append to the end of the record.

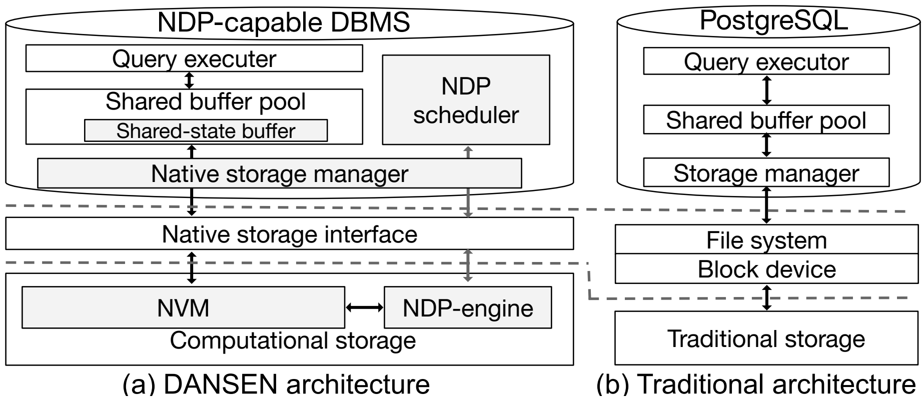


Fig. 2. A comparison of (a) DANSEN (with NDP-capable DBMS [8, 44]) and (b) traditional architecture.

2.3.2 Storage Management. When a new record is created in the DBMS, the storage manager requests from the Free-Space Manager a page with adequate space to accommodate the record. The page can either be an existing one or, if there are no suitable ones available, a new page is allocated. In a DBMS not using our NSI, blocks are usually stored in flat files in a file system. Interestingly, only the file system knows the logical block addressing for each Block#/Page# and only the DBMS knows the Page# for a table. This type of information hiding through all layers of abstraction along the I/O stack poses a significant challenge to NDP and makes identifying records and navigating across data on the device challenging. With our native storage approach, many of these abstractions are eliminated, giving an NSI DBMS detailed knowledge (and control) over actual storage.

3 DANSEN System Design

This section outlines a DANSEN system architecture that enables the offloading of computations from the query engine to the storage device. Fig. 2-(a) shows the overall layout of DANSEN including an NDP-capable DBMS (at the top) and computational storage (at the bottom) that interacts through a NSI (in the middle). In the following, we begin by introducing an NDP-capable DBMS. Next, we present a NSI that enables the DBMS to handle NDP-operations directly on the device. We then provide detailed information on the hardware architecture of the proposed storage solution and discuss the timing model of the proposed approach.

3.1 NDP-Capable DBMS

As shown in Fig. 2-(a), DANSEN is connected to an NDP-capable DBMS [8, 44], namely neoDBMS, that extends PostgreSQL by eliminating intermediary layers, such as the file system, along the critical I/O path and operates directly on NVM storage using physical pointers. DANSEN's NSI design enables the DBMS to control the physical data placement and address the properties of the underlying NVM. By leveraging the byte-addressability provided by the NVM storage, DANSEN significantly reduces read-amplification. As result, the computational storage can resolve physical records and page addresses in-situ, without frequent and expensive interaction with the host.

3.1.1 Version Organizations. The version organization in the NDP-capable DBMS differs from non-NDP-capable PostgreSQL, which employs the Old-to-New (O2N) organization in its linked list of record versions. In contrast, the used DBMS adopts a New-to-Old (N2O) organization for versions, which are stored as a singly-linked list, and introduces a novel version invalidation model. The O2N organization is well-suited for analytical (OLAP) operations, while the N2O facilitates fast updates and lookups (OLTP operations) under the HTAP workloads considered here. To this end, the NDP-capable DBMS uses a Virtual ID (VID) that is unique for all versions of the same record and replaces the successor recID with the predecessor recID in the record header (c.f. Section 1). Each version includes a timestamp of the creating transaction and is implicitly invalidated by the existence of a successor version. To manage the entry points of a version chain, a *VID-mapping* table is utilized. VIDs point to the logical page and slot number of the record (i.e., <Page#, Slot#>). Modifying transactions create new VID entries or update existing VIDs to point to the latest version, which enables the DBMS to function in an append-based manner.

3.1.2 Shared-State Buffer. When a transaction produces a record, a new record is placed in a *shared-state buffer* (i.e., of size 10 KB - 100 KB) in memory. The shared-state buffer contains one logically addressed NSM-format page per DB-Object, as shown in Fig. 3. This approach assumes that the working dataset can fit in memory, while the complete dataset, including cold historic data, is much larger and only available in its entirety on persistent storage. The shared-state buffer contiguously holds all differences between the in-memory data and the dataset on persistent storage. To propagate the shared-state buffer to the storage interface, the DBMS is required to operate in two modes:

Flush & append. Once the shared-state buffer is full, committed versions are *flushed and appended* to storage and persisted. A new logical page is then allocated in the buffer to hold further modifications, as shown in Fig. 3-(a). The NDP-capable DBMS employs the concept of NSI for NVM and NDP-operations (more information on this can be found in Section 3.2). NSI is based on physical addressing and native storage operations. For instance, when Logical Page Number (short: lpn) 10 gets full, the DBMS generates a physical page at the address 0x80000 and proceeds to evict the in-memory page to the persistent storage. The physical address is reserved when a new in-memory page is created. The mapping of logical pages to physical addresses is managed via a Logical to Physical (L2P-)mapping table, whose changes are also appended to storage during a regular flush operation. Additionally, VID-mapping changes due to version updates for the current shared-state buffer are also propagated to the device. Due to the contiguous nature of all shared-state within the buffer, the payload data is always easily accessible without having to "marshal" it from dispersed storage locations. In addition, it can efficiently be transferred to the device with a single DMA burst transaction.

Pass-along & cache. To ensure consistency between the in-memory data and the dataset on persistent storage before performing an NDP operation on the computational storage, DANSEN employs a strategy of just-in-time *passing-along and caching* before invoking an NDP operation. While the shared-state buffer pages containing committed versions are regularly flushed to the persistent storage once the buffer becomes full (i.e., Flush & append), even a partially filled buffer has to be flushed to storage as some in-memory pages may contain new versions that should be visible to the NDP transaction. To this end, DANSEN allocates a special caching region in the persistent storage exclusively used by the invoked NDP transaction, which is removed upon completion. This approach enables DANSEN to bridge the gap between the in-memory data and the dataset on persistent storage, allowing it to support large-memory scenarios and datasets that do not fit in memory. As a result, DANSEN provides a viable hybrid DBMS solution for a wide range of use cases.

3.1.3 Table Modifications. To execute transactions that modify data, such as inserts, updates, or deletes, the host query executor must first verify if there is enough free space in the current shared-state buffer to store the record. This verification is performed by looking at the page header that encodes the start and end of the free space and comparing it with the actual record size. After this check is performed, the modifying transactions will create a new slot pointer and store the record data along with the record header at the next available offset. This process has been shown

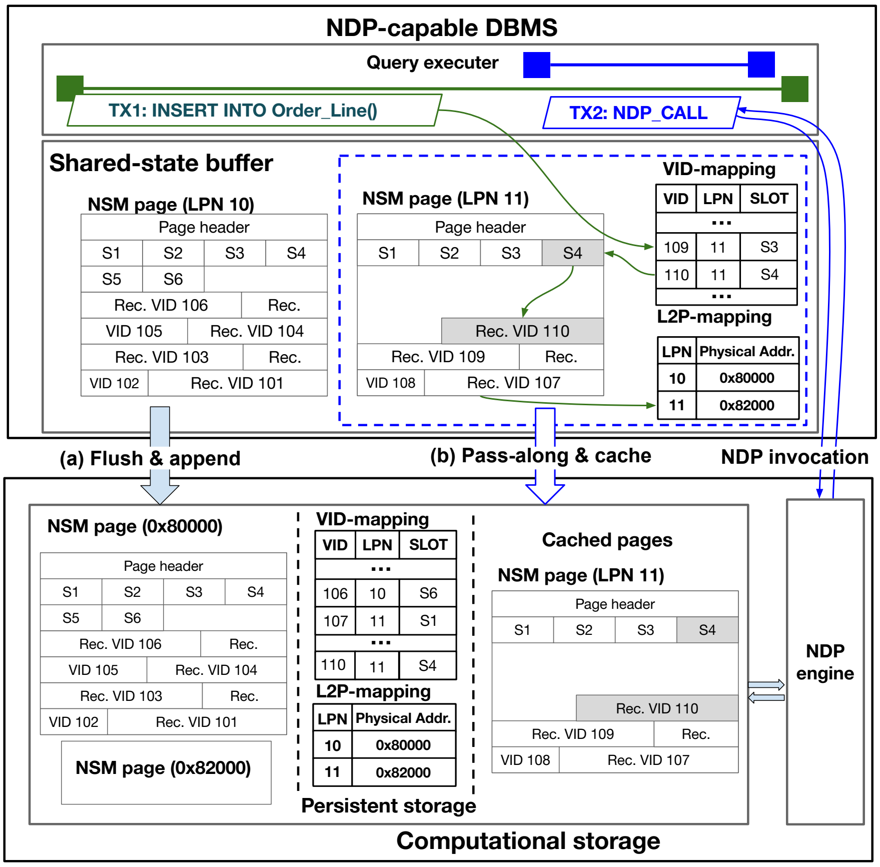


Fig. 3. Shared-state buffer propagation modes: (a) Flush & append and (b) Pass-along & cache (indicated by blue box and arrow). TX=Transaction, LPN=Logical Page Number, VID=Virtual ID, L2P=Logical-to-Physical

in Fig. 3 using a green arrow. The creation timestamps in the record header are set to the transaction ID that executed the modifying statement. If this is the first version of the record, no predecessor information needs to be stored, and therefore it is set to NULL. In case a previous version of the record exists, the predecessor information is set to the logical page and slot number of that version. The slot pointer within a page increases from top to bottom, while records append from bottom to top.

3.1.4 Version Visibility Checking. In DANSEN, each record version is considered a distinct physical entity with its own unique identity. These versions contain both a creation timestamp and a reference to their predecessor version, both of which are observable through the metadata format shown in Fig. 1. During each NDP invocation a transaction timestamp is provided which indicates the unique number (transaction IDs are monotonically increasing) of the calling transaction. Using this timestamp, the NDP-engine is able to calculate version-record timestamps for every record on the device, enabling it to determine whether a record is visible to the current transaction (belongs to the NDP transactional snapshot) or not. If a record is found to be invisible to the current transaction timestamp, the NDP-engine module must *traverse back* to the record's predecessor(s) to find the

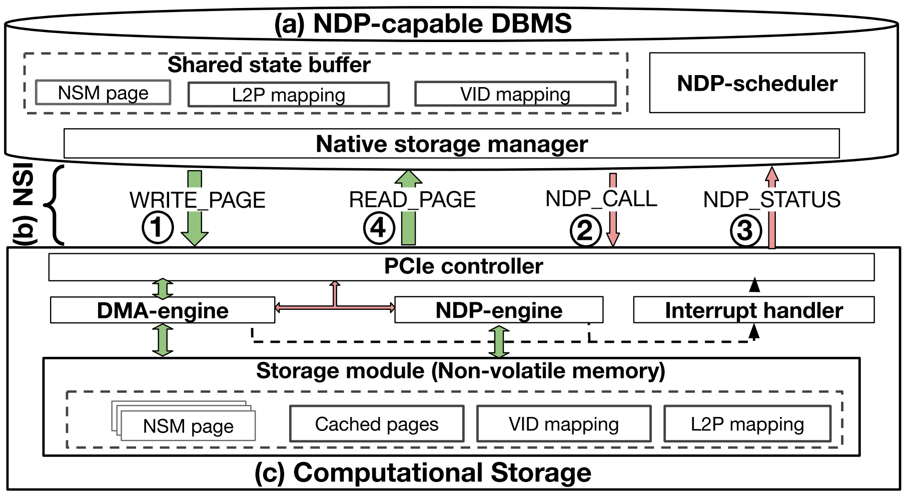


Fig. 4. Global layout of DANSEN including (a) NDP-capable DBMS, (b) native storage interface, and (c) computational storage.

version that is visible to the requested timestamp. This process is referred to as *version visibility checking*.

3.1.5 NDP-Operations. Deciding which operations to offload onto the NDP-engine is challenging considering factors such as data volume, computation complexity, and hardware capabilities. The NDP approach is particularly beneficial for size-reducing operations, such as filters (*NDP-filter*) [51] and aggregations (*NDP-SUM/MIN/MAX*). These operations involve processing significant data volumes to extract subsets or summaries. Operations that go beyond the built-in capabilities of the DBMS are implemented as UserDefined functions (*NDP-UserDefined*). This mechanism enables users to define and implement custom operations within the DBMS and incorporate them into the execution plan. The seamless integration of UserDefined functions and the principles of NDP enables the NDP-capable DBMS to make intelligent decisions about offloading certain computations to custom accelerators in the computational storage.

This study focuses on offloading *analytical* (read-only) queries to the computational storage device. As discussed later in Section 3.3, the NDP engine is currently capable of performing aggregation operations (e.g., SUM, MIN, MAX, etc.) having a scalar result, with optional record pre-filtering, using built-in functionality. Arbitrary operations, including multi-result ones, are realized as user-defined hardware functions that are easily integrated into the DANSEN NDP pipeline using standard AXI interfaces. For the experiments conducted here, we have manually selected which functions to execute NDP-style on the device. We perform fine-grained measurements, including different data transfer and execution phase times, in Section 4.2.2 **Experiment 3**, to justify our choice of NDP operations. These results could later be used in DBMS execution planning to *automatically* choose between traditional and NDP processing.

3.2 Native Storage and its Interface

Offloading an NDP operation onto the storage device requires processing capabilities on the storage device, as well as an active interface to the host system enabling their use. Relying on a traditional read/write storage interface (e.g., a block-device view) interface with many abstraction layers (e.g., file system) would complicate the NDP engine significantly, as it would need to be cognizant of all the abstraction layers as well. An alternative solution is to employ a *native storage interface*, which eliminates these layers and provides direct access to hardware resources [44]. Moreover, eliminating the intermediate abstraction layers between the storage and the host results in leaner I/O stacks [18, 19, 37] and spares the NDP engine from implementing the abstractions. To this end, we relied on an NSI that enables the DBMS to access/store data directly on the storage device to avoid information hiding, as shown in Fig. 4-(b). In the following sections, we explain what functionality is required in the DBMS and the NSI.

3.2.1 In-Situ Data Interpretation. To execute transactions in NDP mode, it is necessary for the NDP-engine to *interpret* the data layout, including table definitions, record- and page-layout and to navigate across persistent physical data. To this end, the *NDP scheduler* translates the table definition and page format into specific calling parameters, such as type and order of datatypes, for the NDP-engine. These administrative parameters which carried with each NDP invocation include NDP-operation (NDP-engine operation), TransactionID (NDP operation's transaction timestamp), ResultSetInfo (Result handling behavior), AddressInfo (physical address ranges for VID/L2P-mapping table and cached pages on device), SchedulerInfo (resource allocation and workload partitioning for the NDP-operation), Miscellaneous (in-situ data interpretation and optimization details). By utilizing this approach, the NDP-engine can transparently interpret database pages and record formats to process record versions and compute the correct snapshots for transactions.

3.2.2 NSI Commands. The NSI provides two categories of commands, namely *data-commands* and *NDP-commands* as shown in the middle part of Fig. 4. The data-commands support page-transfers (via READ_PAGE and WRITE_PAGE). These two commands completely replace the traditional storage interface that relies on the file and operating system. Page-transfer commands are best suited for regular flushes of database pages as well as NDP-related transfers like VID-mapping or L2P-mapping table.

The NDP-command enables the NDP scheduler to offload an NDP-operation to the NDP-engine via NDP_CALL. The NDP-engine uses the NDP_STATUS command to notify the NDP scheduler of the status of the NDP-operations as well as transfer control information. For example, consider a scenario where the NDP scheduler identifies an operation to execute on the NDP-engine. Prior to invoking the NDP-operation, the native storage manager propagates (pass-along & cache) the shared-state buffer to the device (in Fig. 4 ①). Next, the NDP scheduler initiates the invocation of the NDP-engine by specifying administrative parameters (②). Once the operation is completed on the NDP-engine, the status of the operation is forwarded to the NDP scheduler (③). Finally, the native storage manager configures the on-device DMA engine to read the *final* results (④) back to the DBMS for further processing. Note that *intermediate* results can be held on device between different NDP operations and do not need to be copied back to the host.

3.3 Computational Storage: Interface and Architecture

The underlying infrastructure of DANSEN, as shown in Fig. 4-(c), includes several key components such as the storage module, NDP-engine, PCIe-controller, DMA-engine, and interrupt handler. The storage module is where the data actually resides. It is partitioned into *persistent* storage for storing cold historical data, and a *shared-state buffer* that is used for administrative tasks. The interrupt handler is responsible for collecting interrupt signals from the internal modules and forwarding them to the host through the PCIe-controller. The DMA-engine is responsible for transferring data between the host and the device via the PCIe bus, while the PCIe-controller enables communication between the storage module and the host.

The NDP-engine is responsible for executing NDP operations, as shown in Fig. 5. The NDP-engine executes the NDP_CALL operation by involving three distinct pipeline stages, namely *NDP front-end*,

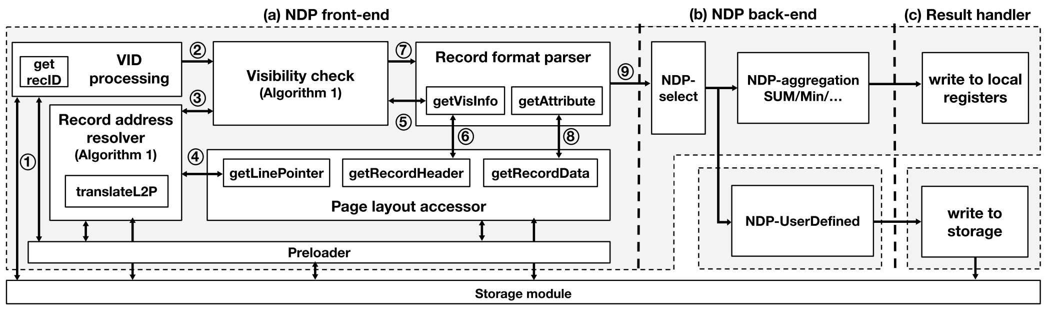


Fig. 5. The NDP-engine processing pipeline with three stages: (a) NDP front-end, (b) NDP back-end, and (c) result handler.

NDP back-end, and *Result Handler*. Only the NDP back-end needs to be customized for different operations. The front-end and the results handler are reusable parts of the infrastructure. In the following sections, we provide an in-depth overview of each stage's functionality, followed by a discussion of the microarchitecture required to implement this processing pipeline in hardware.

3.3.1 *NDP Front-End.*

After receiving an NDP_CALL, the engine proceeds to the first stage, which is processing the VID-table to identify the transactionally-consistent record versions that are visibile to the call, extracting the record data, and forwarding it to the NDP back-end stage. This operation necessitates the use of several modules, as shown in Fig. 5-(a):

VID processing. This module extracts recID entries from the VID-mapping table (①) and forwards them to the visibility checking module (②). Given that each recID can be handled independently, parallel execution of the first stage is possible.

Visibility checking. This module computes the transactionally consistent snapshot of the record version timestamp to determine whether the record belongs to the NDP_CALL transactional. Algorithm 1 shows the module functionality, which begins by resolving the logical recID (③). Following that, it requests the Visibility Information field (visInfo) from the record header using the *record format parser* module (⑤). Then, it extracts and compares the creation and invalidation times of each record against the transactional timestamp (txID). If the record belongs to the called transactional timestamp, it forwards the physical record address to the record format parser (⑦). If not, the module extracts *predecessor* information from the record's visInfo (i.e., new recID) to consider whether an *older* version of the record might be visible in the current NDP transaction, and repeats the process. In this manner, records are examined from newest-to-oldest to determine the most recent version current for this NDP_CALL.

Record address resolver. This module resolves recID and calculates the *physical* record address. As outlined in Algorithm 1, the module begins by extracting the lpn and slot number of the recID. Then, it translates the logical page number into a physical page number (ppn) by using the L2P-mapping table. Afterward, it extracts the line pointer (linePtr) of the corresponding recID by passing the ppn and slot number to the *page layout accessor* module (④). The module then calculates the physical record address based on the obtained linePtr.

Record format parser. This module is responsible for in-situ record data extraction and interpretation. It achieves this by extracting the visInfo from the record header (⑥) using the getVisInfo() function. In addition, it extracts the attributes (⑧) from the record body and forwards them to the NDP back-end (⑨) using the getAttribute() function, based on the data format information. The

Algorithm 1: Pseudo-code for visibility checking and record address resolver.

```
Function visibilityCheck (recID, txID)
    recordAddress = getRecordAddr(recID);
    visInfo = getVisInfo(recordAddress);
    creation = visInfo → tx_create;
    invalidation = visInfo → tx_inval;
    if ( txID ≥ creation && txID ≤ invalidation) then
        return TRUE;
    else
        newRecID = visInfo → predecessor;
        if (newRecID == NULL) then
            return FALSE;
        else
            visibilityCheck(newRecID, txID);
        end
    end
end
Function getRecordAddr (recID)
    ppn = translate_L2P(recID → lpn);
    linePtr = getLinePointer(ppn, recID → slot);
    recordAddr = addressCalculator(ppn, linePtr);
    return recordAddr;
end
Function addressCalculator (ppn, linePtr)
    pageOffset = ppn × PAGE_SIZE;
    recordAddr = pageOffset + linePtr → recordOffset;
    return recordAddr;
end
```

getVisInfo() and getAttribute() functions enable NDP-engine to extract visInfo and attribute(s) from the record according to record format (c.f. Section 3.2.1).

Page layout accessor. This module enables the NDP-engine to interpret the database page by reading the record header (getRecordHeader) and data (getRecordData) by using the physical address of the record. In addition, it extracts line pointer (getLinePointer) from the page header by using ppn and slot number.

Preloader. This module is designed to improve the performance of the NDP-engine by preloading (prefetching) data from the storage module to local scratch-pad memories for each NDP PE, thereby mitigating NVM access latencies. Since the page layout and record format are known to the NDP engine, the preloader module can accurately locate the relevant information or record that needs to be accessed next. In addition, the module is capable of preloading database pages, as well as the L2P-mapping and VID-mapping tables. This allows the loading of *larger* data sets with a single request to the storage module as efficient long-burst data transfers (More details on exploiting the preloader is presented in Section 3.3.4).

3.3.2 NDP Back-End. This stage is responsible for actually executing the offloaded operations on the record data, after the prior stages ensured transactional consistency and interpreted the actual in-storage data layouts to extract the parsed attribute data. In DANSEN, the NDP operations can range from simple aggregation tasks to more complex data analytic tasks such as ML inference. As shown in Fig. 5-(b), the back-end stage is structured in two steps. In the first step, the NDP-engine applies *filter* criteria provided by the NDP_CALL. Filtering can be applied to single or multiple attributes and is implemented through conditional "if" statements in the software-programmable front-end. In the second step, the NDP-operation is either executed directly in the software-programmable front-end (e.g., simple aggregation operations as described in Section 3.1.5), or passed on to the NDP-*UserDefined* module for processing later in the pipeline.

3.3.3 Result Handler. The result handler stage is responsible for processing the output generated by the NDP back-end so that it can be returned to the software database. If only a single scalar value has to be returned (such as an aggregation result), a hardware register is used to pass it back along with the current NDP_STATUS value to the host. Composite results requiring more space are written to a pre-allocated region of on-device storage. The memory range to be used for this was set by software-side DBMS, and passed along with the NDP_CALL. The software-side DBMS efficiently allocates sufficient memory on the storage module to accommodate the maximum possible generated result. The generated results are stored as a vector, eliminating the necessity for a page format, and are transferred back to the host where they are interpreted and converted into the actual record format by the software-side DBMS.

3.3.4 NDP-Engine Microarchitecture. As shown in Fig. 6-(a), the heart of DANSEN is a *dynamic elastic* pipeline microarchitecture, which is beneficial both for performance, as well for extensibility and flexibility reasons. For the latter, the pipeline employs an array of easily reprogrammable soft-core CPUs in the front-end to keep up with advances on the DBMS software side, while the latter stages employ hardwired logic for high performance.

In the following section, we provide a detailed explanation of the microarchitecture of the NDP-engine internal modules, including the *soft-core* array and the result handler.

NDP-engine communication. The engine is equipped with two communication infrastructures, namely external and internal communication. External communication refers to the communication that occurs between the engine and the storage devices, and to the host-side DBMS, and are realized via memory-mapped I/O interfaces as shown in Fig 6-(a) by the data bus and control bus. The data bus enables the engine to access and store data on the storage module, while the control bus allows the DBMS to manage the soft-cores and result handlers using the PCIe interface. Internal module communication, on the other hand, refers to the communication between pipeline stages and is provided via streaming interfaces.

Soft-core integration. The first stage of the DANSEN NDP pipeline involves the processing of recIDs and the extraction of record data (attributes). Since the software-side of DANSEN, namely a modified PostgreSQL DBMS was also under heavy development simultaneously with the hardware side, the required hardware-software interface flexibility would have been difficult to achieve using hardwired functions on the FPGA. As a solution, software-programmable processing elements (PE) were used for the first stages of the hardware NDP pipeline in the form of customized soft processor cores.

The selection of a suitable soft-core faced the somewhat unusual challenge in that the main design goals were small size (as multiple cores are required), high performance on FPGAs (which ruled out many ASIC-optimized cores), and support for 64b addressing (to keep up with data working sets larger than 4 GiB). The design choices for addressing these requirements are discussed in Section 4.1.1.

For integration into the processing pipeline, we have wrapped the core with a control unit, local memories, control and data interfaces, and a preloader module, as shown in Fig. 6-(b). The *control unit* is responsible for managing the soft-core by initiating its activation upon receipt of a start

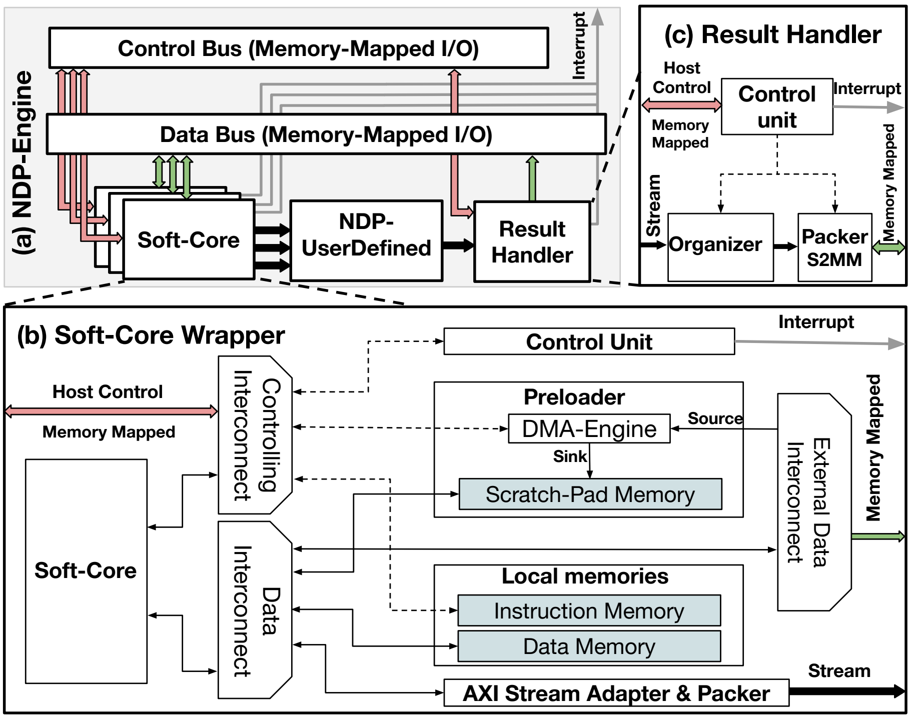


Fig. 6. System-on-Chip architecture of NDP-engine pipeline (a) including soft-core arrays (b), NDP-UserDefined, and result handler (c).

signal from the host. In addition, it provides memory-mapped registers that enable the core to indicate the completion of the program by writing a status value to them. When the operation executing on a soft-core PE is completed, the control unit generates an interrupt to signal the host. The *control interface* gives control over the core by allowing the software-side DBMS to set controlling registers and load new PE firmware through the PCIe bus. The *data interfaces* realize memory-mapped I/O and streaming accesses. The former enables the core to access data (such as database pages and records) from the entire device-side storage module.

The latter allows the core to forward data (such as extracted attributes from transaction-visible records) to the NDP-UserDefined module for hardware-accelerated processing. *Local memories* provide instructions and scratchpad data for the core and are mapped into the memory space of the soft-core. The instruction memory is connected to the control interface, which enables the loading of binary firmware into the PE under host control. The *preloader* is a custom DMA engine that moves data from the storage module to the scratchpad memory and back. It has a control interface, a source interface for reading from the storage module, and a sink interface for writing to the scratchpad memory, which is also mapped into the core address space. For each DMA transfer, the core is required to provide the source and sink addresses, as well as the data length. Using the hardwired preloader is considerably more efficient than using soft-core instructions for data transfers, as discussed in Section 4.2.2.

Result handler module. The module, as shown in Fig. 6-(c), consists of three sub-modules, namely organizer, packer, and control unit. The *organizer* re-organizes and changes the endianness of the generated results. The *packer* packs received data packets via the streaming interface (AXI-stream) into memory-mapped packets (AXI-MM) and stores them on the storage module in an address range pre-allocated by the DBMS. The *control unit* manages controlling registers, such as the address of the pre-allocated memory, and triggers an interrupt signal upon completion of the assigned task.

3.3.5 Execution Pipeline. Following the NDP invocation, the initial step involves loading binary firmware into the instruction memory of the soft-core and setting the controlling registers on the soft-core(s) and result handler module via the NDP_CALL. This step triggers the execution of the NDP-operation based on their designated tasks (i.e., NDP-operation as described in Section 3.2.1). DANSEN exploits not just pipeline parallelism in its NDP architecture. The front-end stage actually consists of an *array* of soft-core PEs that operate in parallel. To this end, the VID-mapping table is divided into equal-sized partitions and *distributed* among the soft-cores by the DBMS-side storage manager. This task distribution enables the DBMS to control the level of parallelization. Upon completely processing an assigned VID-mapping table partition, a soft-core PE raises interrupt signals to notify the host. Finally, the result handler generates an interrupt signal after storing all the generated results from the NDP back-end stage. Note that the resulting architecture truly is a *dynamic elastic* pipeline, as the front- and back-ends may progress at different throughput rates.

3.4 Timing Model

For the later discussion of the timing behavior of the DANSEN system, we now introduce a model that covers all of the relevant parameters. The execution time for an NDP_CALL is measured from the moment the DBMS invokes the call, to the point when it receives the "completed" response. This elapsed time, as shown in Eq. (1), includes the time required for DBMS to launch the call ($t_{\text{NDP-setup}}$), perform the NDP-operation on the NDP-engine ($t_{\text{NDP-engine}}$), and to actually transform the raw data result written by the result handler stage to the pre-allocated NVM storage memory range into actual database records ($t_{\text{NDP-teardown}}$). Note that the latter is a *software-side* operation, as the DANSEN NDP engine cannot alter the DBMS state directly.

Prior to invoking the NDP operation, as shown in Eq. (2), the software-side DBMS prepares the VID-mapping table (t_{VID}) and propagates the shared-state buffer onto the device ($t_{\text{propagation}}$) using data-commands (c.f. Section 3.2.2). After executing the operation, as shown in Eq. (3), the DBMS proceeds to retrieve the result from pre-allocated memory on the storage module (c.f. Section 3.3.3) and transfers it to a database buffer using data-commands (t_{transfer}). Subsequently, the software-side DBMS then interprets the raw result data blobs and creates the corresponding actual DBMS records (t_{format}). These different steps contribute to the following timing, which will be further broken down in the next paragraphs:

$$t_{\text{NDP}} = t_{\text{NDP-setup}} + t_{\text{NDP-engine}} + t_{\text{NDP-teardown}} \tag{1}$$

$$t_{\text{NDP-setup}} = t_{\text{VID}} + t_{\text{propagation}} \tag{2}$$

$$t_{\text{NDP-teardown}} = t_{\text{transfer}} + t_{\text{format}} \tag{3}$$

$t_{\text{NDP-engine}}$. The total execution time for processing an NDP_CALL, as shown in Eq. (4), includes the time required to start the NDP-engine (t_{invoc}), in addition to the *longest* processing time required by any front-end soft-core PE n for completing its assigned VID-mapping table partition VIDpart(n), as well as the latency of the second (S2) and third (S3) stages.

$$t_{\text{NDP-engine}} = t_{\text{invoc}} + \max_{n \in cores}\{t_{\text{core}}(n)\} + t_{S2} + t_{S3} \tag{4}$$

The execution time of each soft-core Eq. (5) is the sum of the latencies to process a recID (t_{proc}) and visibility checking (t_{vis}), and the latency to extract record data (t_{ext}) if the record belongs to the current NDP_CALL transaction. The probability that a record belongs to the current transaction

(P_{vis}) is determined by the likelihood of it being found in the current version (P_1) and its predecessor versions ($P_2,\ P_3, ..., P_n$).

$$t_{core}(n) = \sum_{r \in \mathrm{VIDpart}(n)} \left(t_{proc}(r) + t_{vis}(r) + P_{vis} \times t_{ext}(r)\right) \tag{5}$$

$$P_{vis} = P_1 + \sum_{m=2}^{M} \left(P_m \times \prod_{n=1}^{m\text{-}1} (1 - P_n)\right)$$

The average latency for the visibility checking of a record is shown in Eq. (6). The duration of each visibility checking operation is determined by the likelihood P_1 of finding a visible record in the *current* version vs. the probability $1 - P_1$ of having to recursively check its *predecessor* ($r \rightarrow$ pred) creation and invalidation times. Every visibility check requires the address resolution (t_{ar}) of a physical record address and processing of the actual visibility information ($t_{visInfo}$).

$$t_{vis}(r) = P_1 \times (t_{ar} + t_{visInfo}) + (1 - P_1) \times t_{vis}(r \rightarrow \mathrm{pred}) \tag{6}$$

The latency between the NDP-engine and storage module is part of the timing model (i.e., t_{proc}, t_{ar}, $t_{visInfo}$, and t_{ext}) as detailed above and in Section 3.3.1. An increase in this latency directly impacts the overall execution time, thereby reducing the utilized bandwidth. However, quantifying this impact precisely is challenging due to the differing interpretations of bandwidth from the database and hardware perspectives. This is especially true, considering the impact of read-amplification phenomena in multi-versioning systems, when the NDP engine must retrieve small, randomly distributed data segments *in addition* to the actual record data. For example, if a record version does not belong to the NDP transactional snapshot, the engine must check its predecessor records in storage to determine the correct version. This requires additional accesses to the storage module for address resolution (t_{ar}) and visibility information extraction of the predecessor record ($t_{visInfo}$). As discussed later in Section 4.2 **Experiment 4**, the preloader mitigates the latency between the NDP engine and storage, and makes NDP execution times robust against longer NVM latencies.

4 Evaluation

This section evaluates the DANSEN system. In addition to benchmarking specific components of the system, we also perform end-to-end measurements for two NDP data analytics use-cases, namely scalar aggregation and multi-result ML inference.

4.1 Experimental Setup

Our experimental NDP-capable smart/computational storage system consists of two key components: The FPGA providing the compute capabilities, as well as the host and memory interfaces, and the actual storage. Our system is aiming at using modern byte-addressable storage-class non-volatile memory (NVM). With the demise of Intel Optane [24], such devices are no longer commercially available. We thus rely on *emulating* the characteristics of NVM that are relevant for DANSEN on the entire conventional DDR4-SDRAM, using NVMulator [40], on the FPGA. NVMulator is an open-source hardware module for FPGAs, which emulates NVM access characteristics by a controlled slow-down of the underlying DRAM used as actual storage.

We have realized this prototype storage on the AMD / Xilinx Alveo U280 FPGA (AU280) board, which features an UltraScale+-cass FPGA with in-package HBM that uses a 16-lane PCIe Gen3 host interface, along with two independently accessible banks of 16GiB DDR4-SDRAM each, realized as DIMMs. The system-on-chip on the FPGA is created and managed using the high-performance TaPaSCo [29] framework.

Table 1. Soft-core configurations examined for the NDP front-end PEs and Stage 1 execution times (single core and 16-core array)

Configuration (MicroBlaze)	**Pipeline structure**	**Cache specification**	**Functional units**	**F_{max} (MHz)**	**Exec. time (s) 1 core**	**Exec. time (s) 16 cores**
Microcontroller preset	3-stage, 64bit	D$: none I$: none	IntMult32	390	8.80	0.91
Minimum area	5-stage, 64bit	D$: none I$: none	none	380	14.67	1.02
Real-time preset	5-stage, 64bit	D$: 8KB, 32B I$: none	IntMult32, IntDiv	283.3	5.59	0.51
Application preset	5-stage, 64bit	D$: 8KB, 32B I$: none	IntMult64, IntDiv, FPU-basic	200	6.24	0.54
Maximum performance	5-stage, 64bit	D$: 16KB, 64B I$: none	IntMult64, IntDiv, FPU-extended	205	**4.90**	**0.42**
Orca (RISC-V)	5-stage, 32bit	D$: 32KB, 32B I$: 32KB, 32B	none	300	3.94	0.33

To reflect the trend to use ARM-based processors in modern data centers, e.g., Amazon's use of their own Graviton CPUs, or Ampere Altra CPUs used in Microsoft's Azure cloud, we use the same ARM Neoverse N1 cores that are also used in Google's TAU T2a cloud platforms for our host machine.

Our host consists of a four-core ARM Neoverse N1 System Development Platform (N1-SDP) [3] with 16 GiB of RAM that uses Linux to run the software-side DBMS, namely PostgreSQL 12 that has been modified for NDP operation as described above.

For our NVM emulation, we disregard low-level device characteristics such as reliability, error rates, and thermal buildup, and focus exclusively on timing behavior. To reflect a wider spectrum of future device timings and study their impact on the DANSEN architecture, we have configured NVMulator with three timing sets based on the published numbers in [15, 24, 36, 43]. From these prior studies, we selected fast-case (r/w: 305ns/100ns), medium-case (r/w: 350ns/170ns), and worst-case (r/w: 350ns/800ns) timings that have been observed in NVM systems such as Optane DC.

Unlike traditional smart SSDs that use page-addressed NAND Flash, the emulator mimics byte-addressable NVM types like Intel Optane DC. However, it only imitates the *timing* of NVM technologies and does not explore how NVM's basic device features, like error management or durability, affect overall system performance. The emulation allows for easily testing different NVM types to observe their effect on performance and leveraging FPGA as a robust computing element.

Before presenting the results, two key system components, namely the CPU cores for the software-programmable front-end PEs, as well as the microarchitectures of the hardwired PEs used in the NDP back-end for the two data analytics use-cases, will be discussed. Furthermore, we will also examine the workload and the system configuration in greater detail.

4.1.1 Soft-Core Selection and Configuration. As described above, our architecture relies on having flexible software-programmable processing in the DBMS-facing NDP front-end. Here, we discuss the various core options we evaluated. As already discussed in [41] and highlighted by our own experiments below, the choice of core and configuration parameters can have a major impact on performance.

Initially, we focused on cores using the open RISC-V architecture as our front-end PEs. However, it quickly turned out that the 64b cores available in open-source were all targeted at ASICs, and did not achieve high performance on FPGAs. Thus, we switched to the proprietary AMD / Xilinx

64b MicroBlaze core, which is optimized for FPGAs and has a plethora of parameters to tune it for specific applications. We still believe that there is much potential for FPGA-optimized RISC-V cores, though, and have evaluated the 32b version of such a processor, namely the Orca RISC-V core.

MicroBlaze. MicroBlaze is an IP core developed by AMD / Xilinx that offers a wide range of parameters for synthesis. To determine the optimal configurations for our specific application, we conducted an evaluation of various possibilities based on the Xilinx reference guide [23]. We carefully selected five different configurations for our application, as shown in Table 1, and integrated each of these cores into our proposed NDP-engine, allowing end-to-end benchmarking. For this broad experiment, we limited the front-end PE array to 16 cores. Note that it is possible using very careful manual tuning to exceed that number, we went to that effort for our final system configuration (Section 4.1.3).

To have a fair comparison between different MicroBlaze configurations, we execute a typical NDP use-case on each core, looking at the complete processing required for a single record, identified by its recID. The use-case consists of a query on a five-column table with small attribute sizes, four integers and a variable-length field per record, on a data set of three million records. The storage access pattern occurring during the visibility checking process and interpretation of the NSM page format containing this single record is mostly random (pointer-chasing), and of low processing intensity, as only a small volume of data is actually evaluated per record.

The query should return the sum of all entries in the first table column. The firmware for the MicroBlaze cores was compiled with the -O3 compiler optimization. The reported time in the last two columns of Table 1 corresponds to the $t_{\text{NDP-engine}}$ of Eq. 4, since the t_{invoc} was negligible (a few milliseconds) compared to the core execution time.

Our evaluation shows that configurations achieving a high clock frequency do not necessarily lead to a significant end-to-end speedups. For DANSEN, configuring the 64b MicroBlaze (MicroBlaze64) for *Maximum Performance* resulted in the fastest application-level execution time.

RISC-V. As discussed above, we did make an effort to include an open-source core that would enable easier experimentation with possible custom instructions for NDP acceleration, and thus considered a RISC-V-based processor. Among various RISC-V cores, the Orca RISC-V core stands out due to its high performance [20]. Orca is a five-stage 32-bit in-order RISC-V core designed to operate on FPGA [22]. In our evaluation, we replicated the same scenario as the one for MicroBlaze, and found that the Orca core actually achieved the fastest execution time. However, the main drawback of the Orca core is its limited address space of 32b, which makes it unsuitable for a potential “big data” DBMS workload.

4.1.2 *NDP Accelerators for Data Analytics Use-Cases.*

The DANSEN on-device architecture is highly flexible and allows the easy insertion of NDP accelerators for application-specific data analytics operations between the front-end and the result handler stages, which are always reusable.

An example of such an accelerator is the inference in Sum-Product Networks (SPNs), a probabilistic graphical ML model that can express uncertainty over its output. SPNs have been successfully applied in various applications, including robotics [50], medical imaging [38], as well as in databases, where an SPN is used for paper authorship prediction or social network link prediction [33]. SPNs can also be used to accelerate the DBMS itself, namely by performing fast and accurate cardinality estimation for query results [21].

SPNs are structured as a Directed Acyclic Graph (DAG) consisting of weighted sum-nodes, product nodes, and nodes representing univariate distributions. In this work, we focus on a specific flavor of SPNs so-called Mixed SPNs [31] where the SPN DAG evaluation begins with the leaf node distribution, followed by propagation of values upward through the graph, performing multiplication or weighted addition, until a final probability value is obtained at the root node.

SPN software implementation. For the software baseline, we implemented the trained SPN in C as a User Defined Function (UDF) in PostgreSQL, to be executed on the host. A UDF can be written in C or C++ and loaded at run-time via the CREATE_FUNCTION command, taking combinations of base (i.e., Integer and Float) or composite types as parameters, and returning individual result records or result-record sets when used with an SQL query (SELECT ...).
SPN hardware implementation. The hardware accelerator for SPN inference in NDP was created using the toolflow described in [39], which generates a fully pipelined accelerator from a description of the SPN DAG structure in a simple, text-based format [32]. The generated hardwired accelerator is connected to the rest of the NDP pipeline using AXI4 streams.

4.1.3 *Benchmarking Setup.*

All experiments described later employ the following configuration for both the workload and NDP smart storage.
Database workload. In this paper, we evaluated the efficiency of the DANSEN system by performing NDP-operations on the *Orderline* table from *CH-benCHmark* benchmark [13]. Our evaluation utilizes two types of queries, each with unique characteristics, to highlight different aspects of system efficiency. The first query, referred to as Query-SUM, aims at reducing the result set to a single scalar result using aggregation functions such as SUM, MIN, MAX, and COUNT, which are commonly found in analytical queries, especially in OLAP and HTAP benchmarks such as the CH-benCHmark and TPC-H. The second query, referred to as Query-SPN, shows multi-valued results having additionally computed attributes, and performs ML inference using a hardwired SPN accelerator integrated as user-defined function in our NDP pipeline. These two queries leverage FPGA capabilities and the system's ability to support a diverse range of workloads.

As *CH-benCHmark* is not directly suitable for this ML operation, we have to extend it appropriately. To this end, we repurpose and extend the variable length *dist_info* DBMS column to 80 bytes (plus one byte for length) to feed the SPN, which expect an input of 80 8-bit feature values. The modified column is randomly initialized by picking from a set of 10,000 precomputed input vectors, each consisting of 80 byte-sized features to be analyzed by the SPN. In this manner, the SPN UDF may be included in a SQL query (referred to later as Query-SPN) to retrieve inference results:

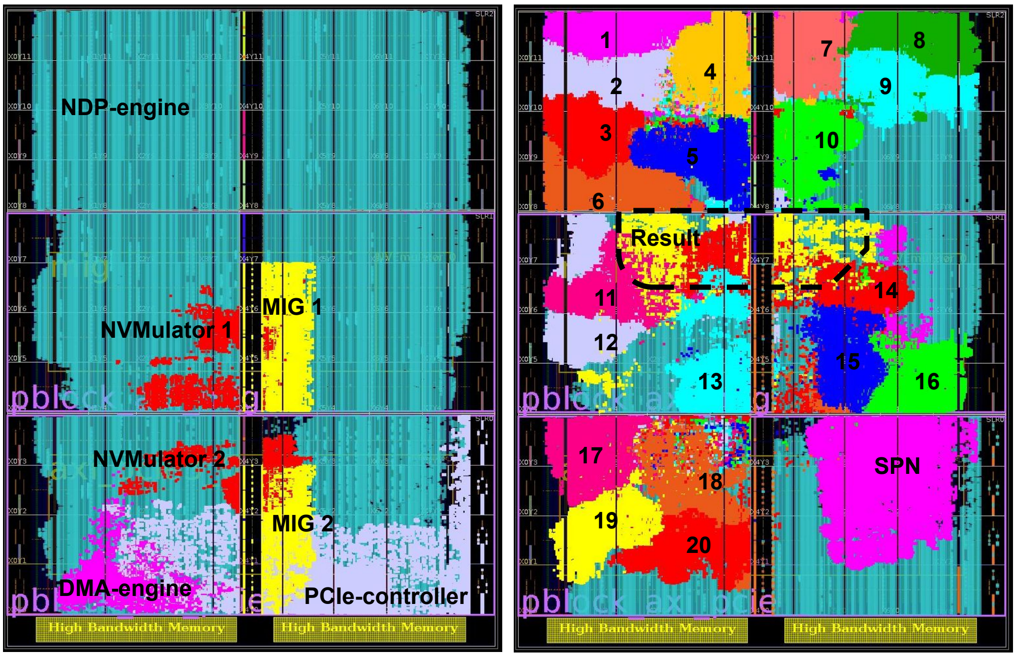
```
SELECT w_id, d_id, o_id, number, SPN(dist_info)
    FROM OrderLine WHERE d_id <= 2;
```

For this query, both DBMS on the host and NDP-engine on the device iterate over the entire table to determine visible records, and pass the byte array of each visible record containing the SPN-Sample having "an identifier less than or equal to 2" into the SPN hardware and software implementations, respectively. Then software implementation of the SPN computes the probability value based on the inputs and passes it into the regular query processing flow. The hardware implementation, after computing the probability value, passes the result to the result handler to be transferred back to the regular query processing flow on host. This process is repeated for all visible records. This is an example of an operation having a *multi-row* result.

For the second use-case, namely a query that returns just a single scalar result, we use the *Q6* aggregation operation:

```
SELECT SUM(amount) AS revenue FROM OrderLine
    WHERE '2000-01-01' <= delivery_d AND delivery_d < '2022-01-01'
    AND 1 < quantity AND quantity <= 100000;
```

In this query, which we will refer to as Query-SUM, we aggregate the *amount* column of the *OrderLine* table and compute the revenue generated from orders with delivery dates between '2000-01-01' and '2022-01-01', and quantities between 1 and 100,000.



(a) System level view (b) NDP-engine level view

Fig. 7. Layout of the DANSEN accelerator on the AU280 FPGA, showing the infrastructure components in (a) and the individual PEs in (b). Note that the numbers indicate the individual soft-core CPUs.

Table 2. FPGA-resource available and utilization.

Module name	LUTs	Registers	BRAM	DSPs	F_{max} (MHz)
Available resources	1303680	2607360	2016	9024	
DANSEN SoC	66.18%	32.39%	53.27%	8.82%	
PCIe-controller	4.62%	2.94%	3.13%	-	250
DMA-engine	0.88%	0.69%	0.74%	-	250 300
Storage module	7.09%	4.59%	3.38%	0.07%	300
└ **NVMulator**	0.55%	0.19%	0.42%	-	
NDP-engine	53.09%	23.85%	45.78%	8.75%	180
├ **Single MicroBlaze64**	2.09%	0.69%	2.18%	0.14%	
├ **SPN**	4.18%	2.52%	-	5.87%	
└ **Result-handler**	2.07%	0.22%	-	-	

NDP hardware architecture configuration. The DANSEN hardware architecture can be flexibly configured, with the number of soft-core PEs in the NDP front-end being a key parameter. We conducted a thorough design space exploration, trading-off FPGA area (number of PE cores) with maximal clock frequency to achieve the best *wall-clock* end-to-end performance.

With careful optimization, we were able to realize 20 MicroBlaze cores in the front-end array, and fully use both of the two separate DRAM memory banks of the AU280 board, each with its own NVM emulation logic. As can be seen in Fig. 7, this led to a very complex FPGA design that

Table 3. A bandwidth comparison between NVMe SSD and NVMulator as pure storage using the FIO tool.

Storage Device	Read (random)	Write (random)	Read (sequential)	Write (sequential)
NVMe SSD	498 MiB/s	299 MiB/s	3181 MiB/s	1748 MiB/s
NVMulator (fast-case)	429 MiB/s	343 MiB/s	1162 MiB/s	1130 MiB/s
NVMulator (medium-case)	423 MiB/s	340 MiB/s	1112 MiB/s	1132 MiB/s
NVMulator (worst-case)	425 MiB/s	298 MiB/s	1103 MiB/s	1097 MiB/s

timing closure was difficult to achieve for. Subfigure (a) highlights the system-level interfaces (MIG memory controller, NVM emulators, PCIe, DMA), subfigure (b) the NDP pipeline with its 20 front-end PEs, the SPN, and the result handler stages.

Detailed resource utilization data for each module in the design, which was created with AMD / Xilinx Vitis 2022.2, is reported in Table 2.

Non-NDP Software Baseline. To evaluate the effectiveness of the DANSEN system, we compared it against a traditional stack running PostgreSQL 12 (short: pgSQL) on a traditional storage device, namely a 500 GiB Samsung 970 Pro SSD, which is a PCIe 3.0 NVMe SSD and commonly used as the storage system in a commodity DBMS.

For comparability, we wanted to make sure that using our emulated NVM storage in "dumb" (non-computational) mode has a similar performance to this traditional baseline, so that we only show speedups achieved due to our NDP architecture, and *not* due to the use of potentially faster storage memories (NVM vs. Flash), which would not be the point of this work.

To this end, we developed a simple Linux block device driver to access the emulated NVM on the FPGA board from the host system as "dumb" storage, without any NDP capabilities. Then, we used the Flexible I/O (FIO) [5] benchmarking tool to generate I/O traffic and measure the bandwidth of our emulated storage and the commercial NVMe SSD. The FIO tool was configured to operate with 8 KiB pages, matching the NSM page size, and to generate a workload of 32 GiB, the maximum capacity of the emulated hardware, across four threads, as the host has four CPU cores. For the underlying filesystem, we used ext4.

As shown in Table 3 for different emulated NVM latencies, the performance of the DANSEN storage when used in "dumb" mode closely tracks that of the commercial SSD for random accesses, with the SSD being *faster* for sequential accesses. This is most likely due to it using internal full-speed DRAM read-ahead/buffering for these cases, which is faster than our always slowed-down DRAM mimicking NVM. Given these numbers, all of the performance benefits we show will be due to our NDP, and not artificially fast storage.

To ensure realism, we limited the memory available to the host-side database to 4 GiB. Without this limitation, the host-side DBMS would see an unrealistically large memory compared to the storage capacity, which in our setup is limited by the 32 GiB DRAM-emulated NVM on the AU280 card. A production smart storage system using actual NVM would provide much larger storage capacity, and thus would not need this workaround.

4.2 Results

In this section, we present a series of experiments conducted to evaluate the system performance in a real-world application. The experiments are categorized into two kinds: system-level and device-level experiments.

4.2.1 System-Level Experiments. We begin with system-level experiments that measure the *I/O throughput* of the DBMS. Next, we proceed to analyze the impact of *selectivity* on the overall system performance.

Experiment 1: DBMS I/O throughput. The objective of this experiment is to examine the impact of DANSEN on the I/O throughput of the DBMS. For this purpose, we compared the I/O throughput of the DBMS in normal and NDP modes while executing Query-SUM and Query-SPN of our benchmark workload with a dataset of 150 million records. In this experiment, the NDP-engine is configured to use 20 front-end PEs, with the preloader enabled. More detailed experiments regarding the NDP-engine performance are presented in Section 4.2.2, including on-device storage throughput in **Experiment 5**.

We begin by looking at the data transfer throughputs occurring between the PostgreSQL DBMS on the host and the traditional ("dumb") storage, and the DANSEN NDP-capable smart storage. Fig. 8-(a) shows this comparison for the DANSEN and baseline systems while executing Query-SUM over time. The execution of Query-SUM on DANSEN takes 7.6 seconds. However, the execution of Query-SUM on the host takes 80.2 seconds as it involves loading database pages from the storage device and processing them. During the execution of this query, the host CPU is under-utilized, as it has to wait for I/O operations to the storage device. As shown in this figure, after ≈6 seconds, the pgSQL begins to flush old pages to the storage, resulting in cache page contentions. In contrast, the proposed approach processes data in close proximity to the storage, and does not require waiting for I/O operations. Fig. 8-(c) shows the I/O throughput of the DANSEN system in detail for four different NV-emulator access latencies. As shown in this figure, there is only a single write request to pass-along & cache (at second 0 as explained in Section 3.1.2) modified data from the DBMS to the NDP device before invoking the NDP operation, and one read request that carries the final result back to the DBMS. Note that, due to preloading, our NDP processing performance is insensitive to the selected NVM latencies.

Fig. 8-(b) shows a similar experiment for Query-SPN. As shown in this figure, executing Query-SPN on PostgreSQL takes a longer time, as it requires loading more database pages from the storage device (at second 16). In contrast, DANSEN again requires just the single write request to pass-along & cache pages (at second 0) and one read request (at second ≈50) for reading the raw data result, which now encompasses the separate ML inference results for all records processed, taking a total of 3.2 GiB. This is transferred back to the host at a rate of 3.2 GiB/s. The I/O throughput of the DANSEN system is shown in detail in Fig. 8-(d).

This experiment demonstrates that moving computation closer to the storage device significantly reduces the volume of data that has to be transferred within the system, and that the transfers that still need to occur can be performed at greater efficiency (higher throughputs). This behavior is one of the key benefits of DANSEN, achieved by reducing the number of intermediate (abstraction) layers between processing and storage.

Experiment 2: Impact of selectivity. In DBMS terminology, *selectivity* refers to the percentage of records passing through filtering (e.g., the condition in a SELECT ... WHERE *condition*) forward to further processing. In this experiment we consider the impact of different selectivities on traditional DBMS and DANSEN. To this end, we vary the selectivity over the `d_id` attribute of the dataset. Fig. 9 shows end-to-end execution times for Query-SPN on a dataset of 150 million records, with different selectivities ranging from 0% to 100%. We also show the impact of page buffer sizes by increasing the host memory from 4 GiB to 16 GiB.

As expected, the dataset generally exceeds the capacity of the DBMS memory, which results in longer execution times due to page accesses from slow storage in the baseline. The 0% selectivity represents the raw scan and filter performance of both pgSQL and DANSEN. Thus, when increasing the selectivity rate, more records meet the filter condition and undergo SPN processing, leading to longer execution times due to the increased processing demands and the generation of more result records, which affect both DANSEN and pgSQL. Host-only processing under pgSQL mandates the transfer of all cold database pages from storage to main memory to determine which records

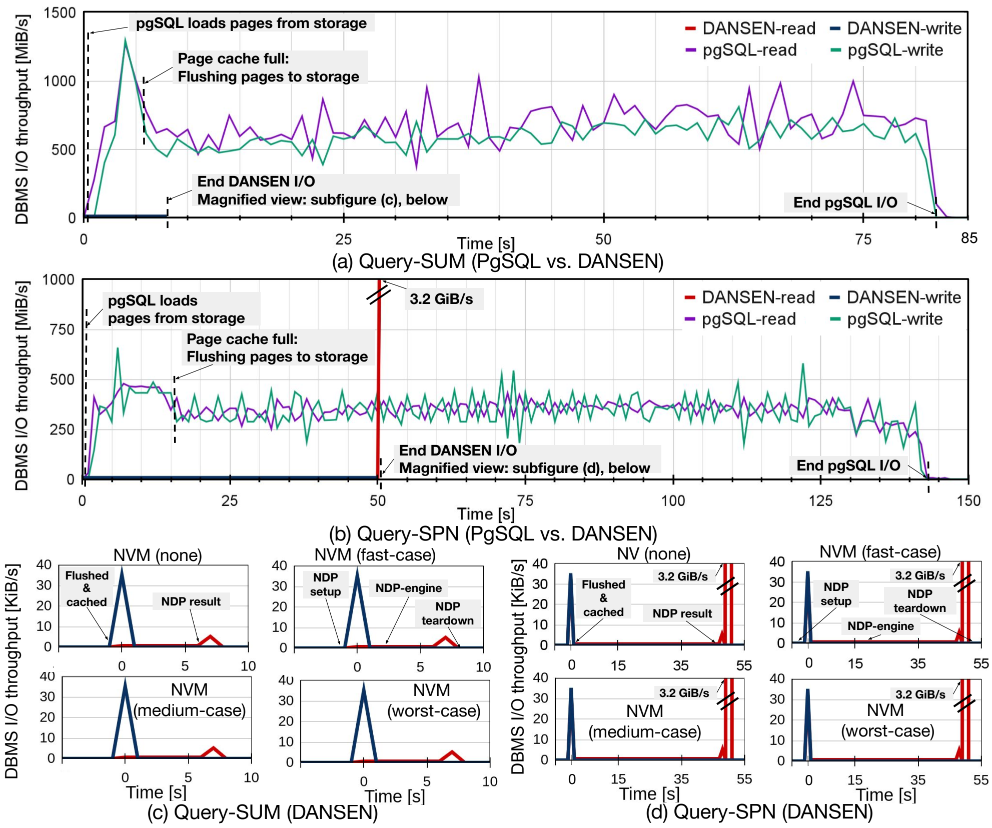


Fig. 8. A comparison of DBMS I/O throughput for (a, c) Query-SUM and (b, d) Query-SPN between DANSEN and PostgreSQL. Note that the smaller figures (c),(d) are zoomed-in views of the overall graphs (a),(b), showing the negligible impact of increasing NVM delay on DANSEN performance.

to filter. This leads to increasing execution times of 92s in a 4 GiB scenario and 82s in a 16 GiB scenario solely for the scan, filtering, and visibility checking.

At 100% selectivity (=all records actually processed), DANSEN still provides a noticeable reduction in data transfers via the slow I/O bus, as it only returns the result-set of the SPN query, which is smaller compared to transferring entire DB pages, including all attributes of the records, back to the DBMS. In addition, the DANSEN system leverages its SPN hardware accelerator and the interleaved transfers/processing enabled by the preloader module, both of which further improves the overall execution times.

Note that when increasing the host memory from 4 GiB to 16 GiB, more pages can be buffered in host memory, resulting in better execution times for pgSQL. However, as we target "Big Data" settings, provisioning main memory to completely hold huge datasets including cold data is generally uneconomical. Even when the main memory capacity is assumed to be at an unrealistically high 1:2 ratio compared to the storage capacity, which is usually in the range of *many* terabytes for commercial storage systems, DANSEN consistently outperforms pgSQL. This performance improvement ranges from 12× in the 4 GiB setting and 10.7× in the 16 GiB setting at 0% selectivity to 1.5× at 100% selectivity in both settings.

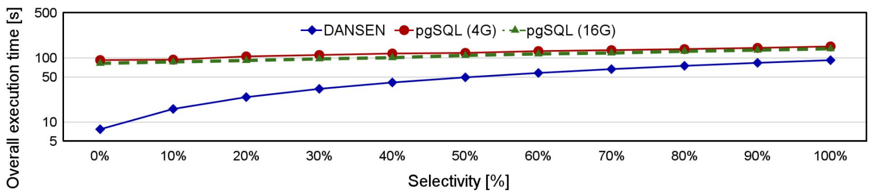


Fig. 9. Impact of selectivity on the overall execution time.

4.2.2 NDP-device Experiments. The prior experiments focused on end-to-end performance and host-device (storage or NDP) interactions. In the next paragraphs, we shift the discussion to a more detailed view examining the impact of *microarchitectural* choices in the DANSEN NDP hardware, such as the number of cores and our preloading scheme for data.

Experiment 3: Overall execution time. In this experiment, we compare the overall execution time of Query-SUM and Query-SPN with the baseline. Note that this baseline execution time for processing these queries *includes* the time to fetch the record data from the NVMe device and process them in pgSQL. The execution time for DANSEN (more details in Section 3.4) consists of $t_{\text{NDP-setup}}$, $t_{\text{NDP-engine}}$, and $t_{\text{NDP-teardown}}$. Our evaluation shows that $t_{\text{NDP-setup}}$ ($\approx 200\,\mu s$) and t_{invoc} ($\approx 30\,ms$ for NDP engine using 20 cores) are negligible relative to the overall execution time.

DANSEN uses the term *preloader* to collectively refer to explicitly managed scratchpad memories and the data transfer strategies for their use.

We examined the execution time of NDP-operations in two modes: with a preloader (With-preloader) and without preloader (No-preloader). Fig. 10 shows the overall execution time for Query-SUM (Fig. 10-a) and Query-SPN (Fig. 10-b) on the dataset sizes ranging from 30 million records (≈5GiB) to 150 million records (≈25GiB), executing on 20 MicroBlaze64 cores. The greenish color bars show the execution time without the preloader, while the pinkish ones show that with the preloader. The diagonally hatched section of each bar on this figure indicates the database overhead ($\text{Time}_{\text{NDP-teardown}}$). As shown in this figure, using the preloader in the proposed approach improves execution time compared to the conventional NVMe-drive by 1.5× for Query-SPN benchmark and 10.6× for Query-SUM with preloader enabled. As expected, the preloader improves system performance by accessing NVM storage in longer, more efficient bursts, and enables the NDP-engine to hide the NVM access latencies by moving data from/to storage to/from scratch pad memories in a double-buffered manner, in parallel to processing.

Note that Fig. 10 shows each of the execution times for different emulated *NVM latencies* (off/best/middle/worst), as explained in Section 4.1. Without the preloader, using slower NVM carries a significant execution time penalty. With the preloader, execution times stay mostly independent of increasing NVM latencies.

Experiment 4: Scalability. We build on the previous experiment to evaluate the NDP-engine performance, now varying the number of active MicroBlaze64 cores in the first stage of the NDP pipeline. Fig. 11 shows the NDP-engine execution time for different numbers of active cores, ranging from 1 to 20, while processing the largest dataset (≈25 GiB). As expected in No-preloader mode, using slower emulated NVM causes a longer execution time. However, enabling the preloader improves NDP-engine execution time and mostly hides NVM access latency, resulting in a roughly similar execution time for the four NVM latencies examined. Our evaluation indicates that other dataset sizes have a similar behavior (omitted in figures for clarity). As shown in Fig. 11-(c), the speedup of both queries is linear. This proves that increasing number of cores in the the NDP front-end results in even higher execution time.

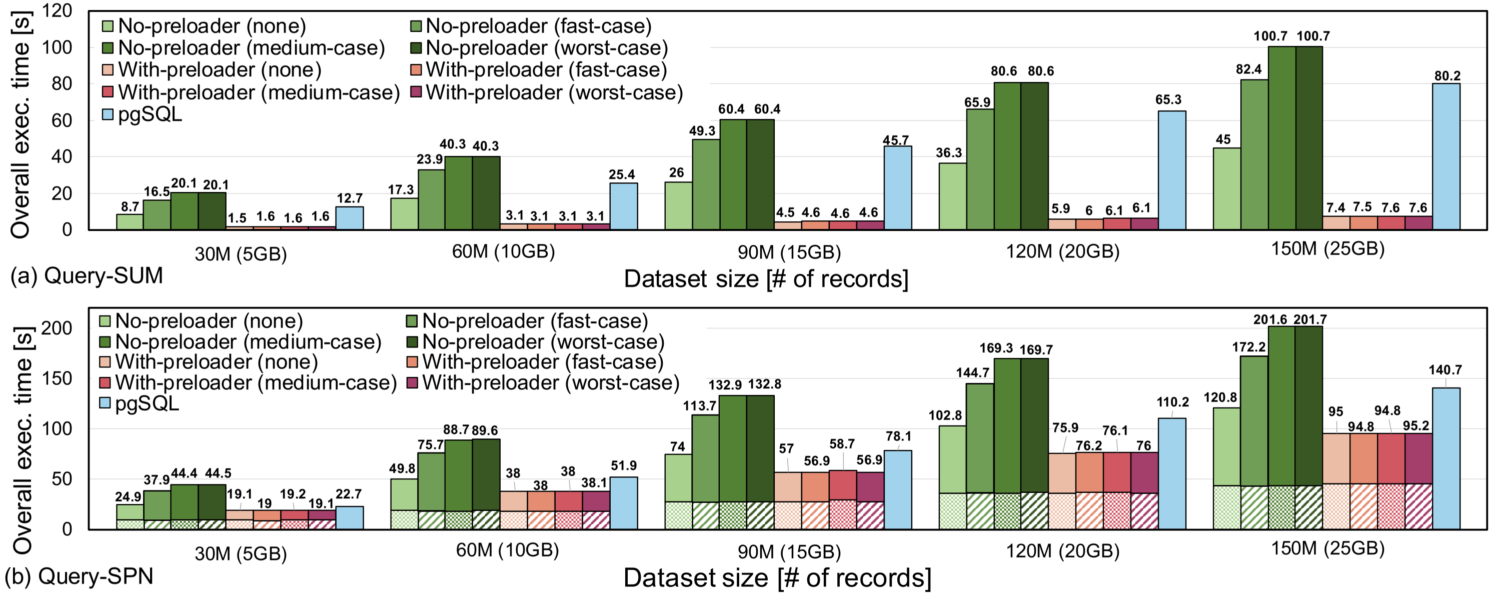


Fig. 10. Overall execution time for (a) Query-SUM and (b) Query-SPN with/without preloader for different dataset sizes and emulated NVM latencies (none/fast/medium/worst). Green and pink bars indicate No-Preloader and With-Preloader execution times, respectively, and hatched sections represent database overhead.

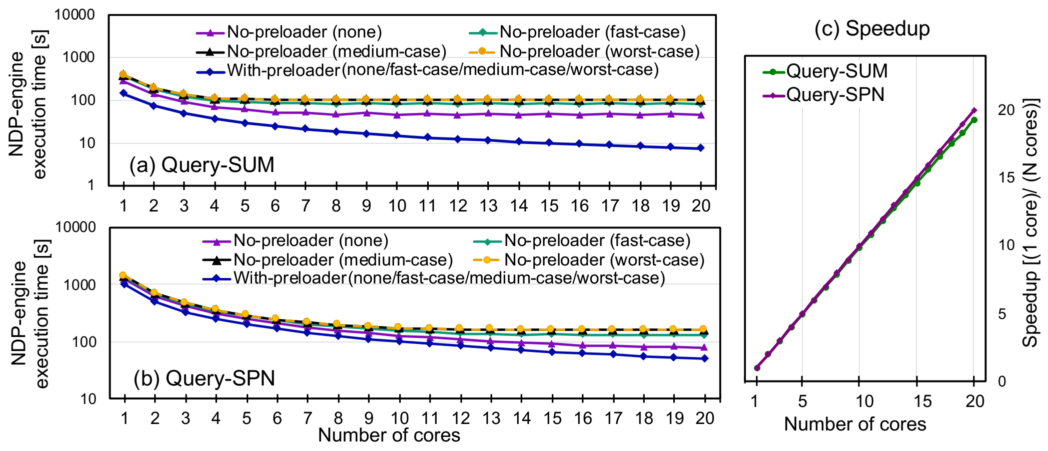


Fig. 11. Interaction between preloader use and increasing core counts on NDP-engine execution time for (a) Query-SUM and (b) Query-SPN, for a dataset size of 150M records and different emulated NVM latencies. (c) A speedup of Query-SUM and Query-SPN with increasing number of soft-cores in with-preloader mode.

Experiment 5: On-device throughput. To further investigate the NDP-engine performance, we measure the *on-device* throughput of the NDP-engine while running Query-SUM and Query-SPN on 20 MicroBlaze64 cores. Fig. 12 shows on-device read/write throughputs for the NDP-engine executing these queries. We found that the on-device read throughput for Query-SUM and Query-SPN are 4 GiB/s and 0.6 GiB/s, respectively. In addition, the on-device write throughput for Query-SPN is ≈0.07 GiB/s for storing the per-row SPN inference result vector, and for the Query-SUM is zero as it does not write to the storage module at all and just computes a single scalar result.

An interesting question arises as to why the read throughput for Query-SPN is lower than that of Query-SUM, given that they both process the same dataset. The reason for this difference lies in the fact that the Query-SPN requires the extraction of *more data* from the record body, including the primary key and SPN fields (16B keys, 80B SPN data). In contrast, the Query-SUM operation only requires the filtering of 16B and forwarding of a single 8B value. In addition, during the execution of the Query-SPN operation, the result for each row is continuously written to the storage module. The Query-SPN example is thus representative of operations which are more computation intensive, while Query-SUM with its very simple aggregation function is more I/O intensive.

The on-device throughput could be further increased by enhancing the level of parallelism in the NDP front-end (c.f. **Experiment 4**). However, the current NDP engine configuration does not fully utilize the internal bandwidth for two main reasons. First, the nature of the application, especially the visibility-checking process, performs random memory accesses in a pointer-chasing-based manner. This process, explained in Section 3.3.1 and considered in the timing model in Section 3.4, involves multiple steps to find the visible record version, requiring at least four cache-line size (64B) accesses to the emulated NVM per recID. These stages include retrieving recID, resolving the address, extracting the linePtr, and reading visInfo from the record header. In addition, attribute extraction for Query-SUM and Query-SPN from the record body requires an additional 1 and 1-3 cache-line accesses, respectively. Second, the irregular data access pattern to the emulated NVM lowers utilization, as addresses are dynamically calculated at each stage during run time and cannot be well predicted. In addition, each access currently passes through the Microblaze64's local L1 system cache (c.f. Section 4.1.1), followed by either direct access or via the preloader and scratchpad memory to the emulated NVM. The preloader optimizes the efficiency of data handling by interleaving transfers

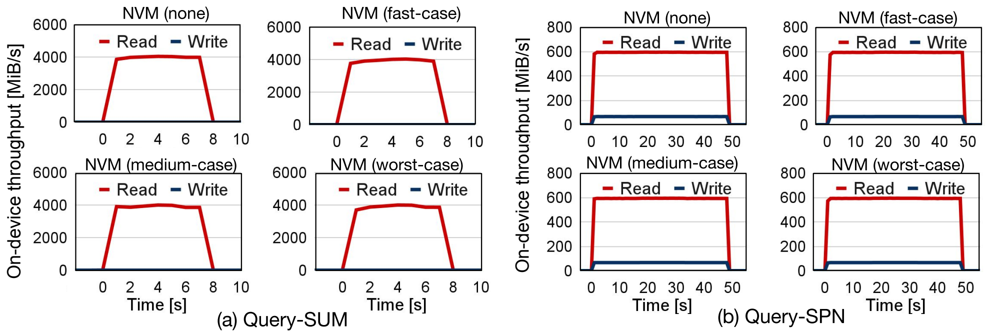


Fig. 12. The on-device throughput of the NDP-engine for (a) Query-SUM and (b) Query-SPN, while processing 150 million records on 20 MicroBlaze64 cores in the NDP front-end across four emulated NVM latencies.

with computation and requesting larger data sizes (64 B - 4 KiB) for more efficient long burst accesses.

4.3 Discussion

Based on the end-to-end and low-level evaluations presented in the prior sections, we can now enter into some overarching discussion.

Potential for write transactions as NDP operations. DANSEN currently supports performing read (analytical) operations as NDP in a transactionally consistent manner, namely by using the latest visible record version on the device, in the presence of concurrent writing (update) transactions on the host, without having to wait for a lock request to succeed. Extending DANSEN to also allow NDP write (update) operations is possible, but becomes considerably more challenging, as multiple writes to the same data may now occur concurrently on the host and NDP device, leading to *write/write conflicts*. As a solution, handling write operations could be addressed by a fast synchronization mechanism between the host and NDP device [9, 42]. We are looking forward to the availability of CXL-capable hosts and FPGA boards to investigate the implementation of these mechanisms.

Soft-core limitation. In **Experiment 4**, we observed that an increase in the number of parallel active soft-cores in the NDP front-end results in a shorter execution time for the first stage. However, **Experiment 5** showed that the proposed NDP-engine only utilizes ≈4 GiB/s on-device throughput in Query-SUM, which is somewhat disappointing considering that the current design employs two memory controllers with a theoretical bandwidth of ≈38 GiB/s to the emulated NVM. This inefficiency is most likely due to the random memory accesses and on-the-fly address calculation performed in the pointer-chasing-based visibility checking algorithm (c.f. Section 4.2.2 last paragraph). While more cores could be used in the front-end to better hide the memory latencies, the DANSEN design is already quite large (see Table 2) and barely meets timing even with three clock domains. Thus, instantiating more MicroBlaze64 cores is infeasible. We are considering two options to address this: First, replacing the MicroBlaze64 core with a more modern FPGA-optimized 64-bit core based on Taiga/CVA5 [30], which can be configured to be smaller/faster than the MicroBlaze64. Second, now that the DBMS-facing data structures and algorithms in the front-end have stabilized, employing an optimized multi-threaded hardwired accelerator for the front-end functionality.

Applicability to different host DBMS / storage engines. The NSM layout we use for storage pages in DANSEN is widely used in row-oriented relational DBMS, e.g., from Oracle and Microsoft. Thus, our fundamental NDP architecture will be portable from the PostgreSQL we employed to these DBMS as well. However, NSM has a relatively poor cache locality and is thus suboptimal high-performance analytical workloads. These are better handled by column-oriented storage engines, using formats such as PAX [2]. To enable the use of DANSEN for DBMS using this approach, changes in the DANSEN NDP front-end would be required, as it is the only part of the hardware that directly interacts with database data structures and algorithms. However, the rest of the DANSEN NDP pipeline could be used as-is.

In addition, existing database data layouts are designed to maximize cache efficiency for conventional x86 and ARM CPUs and storage mediums such as Flash. Emerging computational storage devices that utilize NVM raise a very relevant research topic regarding the development of specialized data layouts tailored for computational storage and NDP. Furthermore, in-storage data transformations have the potential to decouple the data layouts employed in computational storage from the applications running above, paving the way for multi-engine operations.

Future works. Our evaluations show that the DANSEN system is a promising computational storage solution for database applications, with several research questions already planned for future investigation:

- To what extent can computational storage support the processing of database updates autonomously in update environments and hybrid workloads?
- What degree of parallelism, using either hardwired modules or smaller soft cores, in the NDP-engine's front-end would allow full internal bandwidth utilization in DANSEN?
- How can specialized data layouts for NVM-based computational storage improve NDP and enable decoupled, multi-engine operations? And what level of hardware support is required?

5 Related Work

In this section, we will start by examining the concept of native storage. Then, we will focus on prior work proposing the use of NDP to improve system performance, specifically within the domain of database systems.

5.1 Native Storage

The traditional interfaces used for storage devices present a significant obstacle to the effective implementation of Non-Volatile Memory (NVM) based Data Processing (NDP) approaches. This is primarily because these interfaces lack transparency in exposing the underlying storage properties. Both block device interfaces and file systems are designed with a focus on data transfers, specifically block-based access, which hampers the seamless utilization of NVM features like byte-addressability [11, 34]. The lightweight file systems proposed by [4, 28] offer improved storage management but provide little advantage for NDP since they mimic a contiguous address space and depend on *logical addresses*, which do not align well with the core principles of NDP.

On the other hand, the native storage solution allows the host application, such as database systems, to directly interact with the *physical storage*, eliminating the complexity of intermediate abstraction layers. This enables a more efficient exploitation of NVM features and enhances the overall performance of NDP.

Previous studies on persistent storage have proposed various approaches to data processing on raw Flash, including BlueDBM [25], NoFTL [19, 45], and CORFU [6]. Among these studies, both BlueDBM and NoFTL have presented end-to-end systems.

BlueDBM adopts a cluster-based architecture within a distributed system, which facilitates storage management and hardware acceleration for data processing on raw Flash, making it suitable for a wide range of applications. An example of such an application is BlueCache [49], which builds on BlueDBM and serves as a key-value storage engine.

In contrast, NoFTL introduced a general native storage interface for host-based storage configurations. This integration embeds deep physical storage management into the DBMS architecture. The utilization of a native storage interface has shown performance improvements not only in relational database systems [19] but also in key/value database systems [25]. Moreover, such an approach is crucial for efficient data processing on storage and enables the exploitation of the byte-addressability characteristic of NVM.

5.2 Near-Data Processing

Previous studies have proposed using NDP techniques to improve system performance. For instance, [14, 26, 27] suggested offloading certain data processing tasks, such as scans and joins, to embedded processors on a SmartSSD to enhance system performance. However, their methodology's applicability is constrained by its dependence on the current ARM processor architecture.

Another approach, called IBEX [47], explored the utilization of FPGAs as co-processors for executing specific database operations within the storage system. Nevertheless, this approach faced limitations due to the challenge of efficiently moving data and results to and from the FPGA.

Furthermore, these works have not fully capitalized on the unique features of NVM, such as device-internal bandwidth, parallelism, and access latencies. To more fully exploit these aspects, [45, 46] evaluated different NDP result set handling strategies, including result materialization and re-use. In their study, the task distribution was partitioned between an ARM processor and an FPGA operating in parallel. However, that work is closely tied to page-granularity NAND flash and not fully applicable to the byte-addressable NVMs we examine here.

Lastly, it is essential to highlight that these studies primarily concentrated on read-only scenarios with static datasets. However, modern DBMS are dealing with hybrid workloads that also include data updates and writes. In this regard, FOEDUS [28] made an attempt to leverage NVM/DRAM for managing cold/hot data and addressing modifying transactions on new hardware, resulting in excellent scalability. Nevertheless, FOEDUS fell short of fully exploiting NVM characteristics, such as byte-addressability, as it only utilized NVM technologies for page layout purposes.

In this work, we have addressed these challenges and combine the exploitation of NVM with heterogeneous NDP, using both flexible software running on a multi-core array with high-performance fixed-function accelerators.

6 Conclusion

We introduce DANSEN, a comprehensive end-to-end system architecture that enables query processing near storage. Within this system, we have developed a prototype storage, NDP-engine, storage interface, and NDP-capable DBMS. Our experimental results demonstrate the significant advantages of DANSEN over non-NDP PostgreSQL 12. With reduced data movement costs and improved computational parallelism, DANSEN achieves up to an impressive 10.6× improvement in end-to-end execution time while maintaining the transactional integrity of the database, even when processing 150 million records that traditionally would carry a high data transfer effort.

Furthermore, our results highlight the effectiveness of the NDP-engine, which efficiently utilizes the preloader to process queries near storage without the need for intervention, effectively hiding the NVM access latency.

Beyond the immediate performance gains, our evaluation also underscores the potential of DANSEN as a new computational storage solution for enhancing the performance of database management systems across various applications. This work opens up new possibilities for designing computational storage, capable not just of the analytical operations described here, but also able to autonomously perform database updates in NDP.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments and AMD (Xilinx) Inc. for their hardware donation. This research was funded by DFG project #419942270 *neoDBMS*.

References

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

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving relations for cache performance. *VLDB* (2001), 169–180.

[3] ARM. [n. d.]. Neoverse N1 System Development Platform (SDP) - Documentation and Support. https://developer.arm.com/Tools%20and%20Software/Neoverse%20N1%20SDP. Accessed October s, 2023.

[4] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. *Proc. SIGMOD* (2015), 707–722.

[5] Jens Axboe. n.d.. Fio tool source code. https://github.com/axboe/fio. Last accessed: 2021-12-16.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. USENIX Association, 1–14.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD* 24, 2 (1995), 1–10.

[8] 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. *ICDE* (2022).

[9] Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Tobias Vinçon, Andreas Koch, and Ilia Petrov. 2022. Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage. In *EDBT*.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[11] Matias Bjorling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. 2013. The necessary death of the block device interface. In *Proc. CIDR*. 1–4.

[12] Haran Boral and David J. DeWitt. 1989. Parallel Architectures for Database Systems. In *Advances in computers*. Vol. 28. 107–151.

[13] Richard Cole, Florian Funke, Leo Giakoumakis, et al. 2011. The Mixed Workload CH-BenCHmark. In *DBTest*.

[14] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. *Proc. SIGMOD* (2013), 1221.

[15] Sean Eilert, Mark Leinwander, and Giuseppe Crisenza. 2009. Phase Change Memory: A New Memory Enables New Memory Usage Models. In *IMW*. 1–2.

[16] Jim Gray. 1992. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[17] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[18] S. Hardock et al. 2016. Revisiting DBMS Space Management for Native Flash. In *EDBT'16*.

[19] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. 2013. NoFTL: Database Systems on FTL-less Flash Storage. *VLDB* (2013).

[20] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 2019. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In *ReConFig*.

[21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (mar 2020), 992–1005.

[22] VectorBlox Computing Inc. [n. d.]. ORCA. https://github.com/VectorBlox/orca Accessed August 13, 2019.

[23] Xilinx Inc. 2022. Vivado MicroBlaze Processor Reference Guide.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module.

[25] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. *Proc. ISCA* (2015).

[26] Yang Seok Ki et al. 2015. In-storage compute: an ultimate solution for accelerating i/o-intensive applications. *Flash Memory Summit* (2015), 1–30.

[27] 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.

[28] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. *Proc. SIGMOD* (2015), 691–706.

[29] 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 *ARC*.
[30] Eric Matthews and Lesley Shannon. 2017. TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*.
[31] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. 2018. Mixed sum-product networks: A deep architecture for hybrid domains. In *AAAI*.
[32] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. 2019. SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks. *arXiv* (2019). arXiv:1901.03704 [cs.LG]
[33] Aniruddh Nath and Pedro Domingos. 2015. Learning relational sum-product networks. In *AAAI*, Vol. 29.
[34] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and D. Panda. 2011. Beyond block I/O: Rethinking traditional storage primitives. *HPCA* (2011), 301–311.
[35] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*. 1771–1775.
[36] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *MEMSYS*. 304–315.
[37] Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. 2019. Native Storage Techniques for Data Management. In *ICDE*.
[38] Fabian Rathke, Mattia Desana, and Christoph Schnörr. 2017. Locally adaptive probabilistic models for global segmentation of pathological OCT scans. In *MICCAI*.
[39] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. 2020. Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs. In *FCCM*.
[40] Sajjad Tamimi, Arthur Bernhardt, Florian Stock, Ilia Petrov, and Andreas Koch. 2023. NVMulator: A Configurable Open-Source Non-Volatile Memory Emulator for FPGAs. In *ARC*. 35–50.
[41] Sajjad Tamimi, Zahra Ebrahimi, Behnam Khaleghi, and Hossein Asadi. 2019. An Efficient SRAM-Based Reconfigurable Architecture for Embedded Processors. *TCAD* 38, 3 (2019), 466–479.
[42] Sajjad Tamimi, Florian Stock, Andreas Koch, Arthur Bernhardt, and Ilia Petrov. 2022. An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing. In *FCCM*. 1–9.
[43] Alexander van Renen, Lukas Vogel, Viktor Leis, et al. 2019. Persistent Memory I/O Primitives. In *DaMoN*.
[44] 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. *VLDB* (2022), 1991–2004.
[45] 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*.
[46] 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).
[47] Louis Woods, J. Teubner, and G. Alonso. 2013. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proc. SIGMOD*.
[48] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *VLDB* (2017).
[49] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-Based Key-Value Store. *VLDB* (2016), 301–312.
[50] Kaiyu Zheng, Andrzej Pronobis, and Rajesh P. N. Rao. 2017. Learning Semantic Maps with Topological Spatial Relations Using Graph-Structured Sum-Product Networks. *CoRR* abs/1709.08274 (2017). arXiv:1709.08274
[51] Chen Zou and Andrew A. Chien. 2022. ASSASIN: Architecture Support for Stream Computing to Accelerate Computational Storage. In *MICRO*. 354–368.