Distributed and Parallel Databases (2022) 40:27–45
https://doi.org/10.1007/s10619-021-07328-z

On the necessity of explicit cross-layer data formats in near-data processing systems

Lukas Weber[1] · Tobias Vinçon[2] · Christian Knödler[2] · Leonardo Solis-Vasquez[1] · Arthur Bernhardt[2] · Ilia Petrov[2] · Andreas Koch[1]

Accepted: 25 February 2021 / Published online: 16 March 2021
© The Author(s) 2021

Abstract
Massive data transfers in modern data-intensive systems resulting from low data-locality and data-to-code system design hurt their performance and scalability. Near-Data processing (NDP) and a shift to *code-to-data* designs may represent a viable solution as packaging combinations of storage and compute elements on the same device has become feasible. The shift towards NDP system architectures calls for revision of established principles. Abstractions such as *data formats and layouts* typically spread multiple layers in traditional DBMS, the way they are processed is encapsulated within these layers of abstraction. The NDP-style processing requires an explicit definition of cross-layer data formats and accessors to ensure in-situ executions optimally utilizing the properties of the underlying NDP storage and compute elements. In this paper, we make the case for such data format definitions and investigate the performance benefits under RocksDB and the COSMOS hardware platform.

Keywords Near-data processing · Data format · Data layout · FPGA

1 Introduction

Besides substantial data ingestion, yielding an exponential increase in data volumes, modern data-intensive systems perform complex analytical tasks. To process them, systems trigger massive *data transfers* that impair performance and scalability, and hurt resource- and energy-efficiency. These are partly caused by the scarce system bandwidth in combination with poor data locality, as well as by traditional system

✉ Lukas Weber
weber@esa.tu-darmstadt.de

Extended author information available on the last page of the article

architectures and algorithms requiring data to be transferred from storage to computing elements for processing (*data-to-code*).

A shift towards *Near-Data Processing* (NDP) and *code-to-data* allows executing operations in-situ, i.e. as close as possible to the physical data location, leveraging the much better on-device I/O performance. This observation is supported by several trends. Firstly, hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device. Secondly, with semiconductor storage technologies (NVM/Flash) the *device-internal* bandwidth, parallelism, and access latencies are significantly better than the external ones (device-to-host). Combined, the two trends lift major limitations of prior approaches such as ActiveDisks or Database Machines.

Knowledge about the data organisation and the ability to interpret the data format in-situ are essential for performing NDP operations. Interestingly, NDP-able operations are defined on different levels of a DBMS or the I/O stack.

1. *DB-object-* or *Page-based* like fetch, update, scan or garbage collection;
2. *Field/Column-* and *Record-based* such as scan, record-materialization, selection or aggregation

Each operation type processes data according to the respective format or layout. Figure 1 shows common structures like the Field- and Record-Format, Data Organisation, and Page Layout, which are available in almost every classical database. In *classical (layered) DBMS architectures*, data formats and operations can be viewed as abstractions defined on the interface boundaries of the DBMS layers, which encapsulate their functionality (Fig. 1). Consequently, in SQL queries, format definitions of the upper layers are utilized to retrieve and process data from the layer below. Yet, in *NDP-system architectures* this is not possible anymore, as the query or operation is executed in-situ. Since data formats scattered across different layers of abstraction and encapsulated within them, and given the typical complexity of the I/O stack, NDP processing is not possible out of the box. As a result, every necessary format definition either needs to be available in advance on-device or it has to be enclosed to the NDP call.

To make the case for explicit cross-layer formats, this paper utilizes a simple KV store-based NDP-ImageProcessor application. It naively stores colours of images pixel-by-pixel, and defines a small set of operations, which can be executed as traditional queries or NDP calls.

The main contributions of this paper are: 1. We claim that explicit cross-layer data formats and transparent definitions of the data organisation are necessary in NDP scenarios. 2. We propose a definition for formats and layouts in the context of Near-Data Processing. 3. We present an approach to format pushdown in NDP-DBMS. 4. We prototyped the approach with a simple image processing application, on NoFTL-KV and the COSMOS OpenSSD as real hardware target, and gain performance improvements of up to 33%.

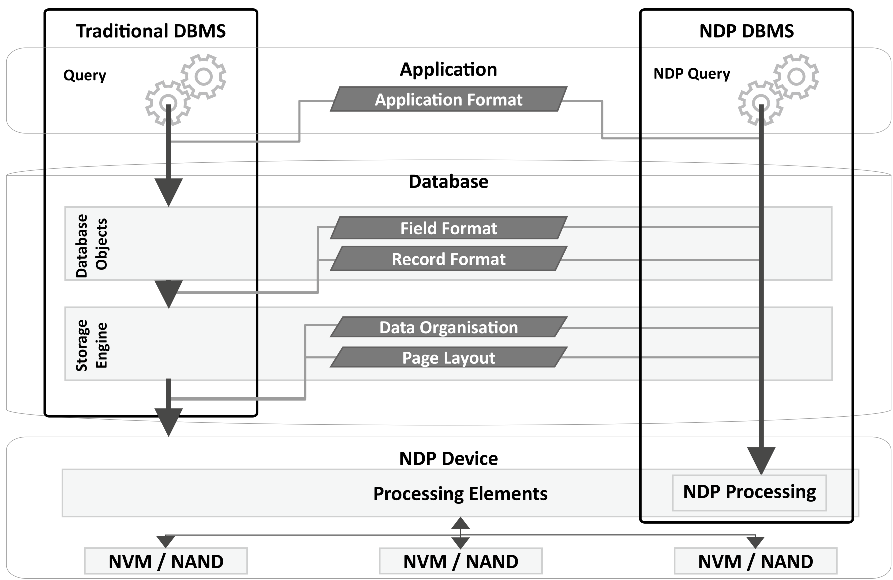

Fig. 1 While queries in traditional DBMS can directly apply format and layout information of the database within every layer of abstraction, NDP queries necessitate to pass them to the NDP device to execute the respective operations in-situ without host interaction

2 Conceptional background

2.1 Near-data processing

NDP targets executing data processing operations as close as possible to the actual physical storage location, instead of transferring the entire raw data to the host. Relevant NDP aspects are:

1. Which operations are NDP-able: only size-reducing or leaf operations in a query execution plan or also more general data-intensive operations like joins or UDFs.
2. Result set: In absence of proper result set management it is beneficial that the results of a NDP operations are significantly smaller than the actual raw dataset that they are operating on.
3. Faster processing: The NDP operations execute faster by leveraging hardware properties such as parallelism, which are not able to be utilized by the host.
4. Intervention-free NDP-executions: NDP may relieve the pressure on the system bus, reducing unnecessary stalls, and making room for further instructions by reducing the data operations given that in-situ executions can be performed without interaction or synchronization with the host.

2.2 NDP operation types in databases

Operations that can been executed on the device are diverse. Interestingly, these frequently build on top of each other, forming a NDP-operation hierarchy (Fig. 2).

The lowest level constitutes the on-device *Data Organisation methods*. These process in-situ the physical storage segments allotted to a certain database object, performing full scans or on-device lookups. Such operations yield NDP accessors (hardware or software) that result from the way data is accessed in the respective data organization (i.e. heap) or the storage structure (i.e. LSM-tree).

Operations tied to the *Data Organisation-based* usually trigger physical on-device I/O operations, which are *Page/Block-based*. These perform physical I/O on-device without interpreting the contained data. Furthermore, they operate on units of physical granularity such as *Pages/Blocks* for Flash or *cachelines* for NVM. Depending on the storage stack, these can be triggered either by the Flash-Translation Layer on the device, any intermediate layers in the operating system, such as file systems or the kernel, or, in case of Native Storage Management, by the database itself [29]. In the context of databases, these *Page/Block-based* operations are usually connected to the Page Layout Accessors or Page Format Parsers to extract the physically embedded database records.

Record-based operations comprise among others full table scans, index lookups, or tree balancing. They make use of Page- and DB-Object-based operations and also interpret parts of the data according to *Structural Elements* as defined in Sect. 2.3. For instance, an index lookup might read several pages containing internal nodes to identify the correct leaf page. Depending on the database, this page is parsed likewise to retrieve either the position in the table or the actually requested data. All these operations process data according to the given page layouts and respective record formats. On top, higher-level database operators like selections, joins, or GROUP BYs can be implemented efficiently on the device.

If an operation needs to interpret individual fields within one or multiple records, another *Field-based* operation has to be executed. Closely linked to the DB-Object and Record Format, these kind of operations have to utilize the data definition (from the database catalogue) to extract the data types of necessary fields. While this is sufficient for a projection, other types of Field-based operations, such as aggregate-functions, must interpret these values to perform the NDP-operation.

In NDP scenarios it is unacceptable to have expensive round trips to the host to get any format or layout definitions at runtime (as most interpreting DBMS query

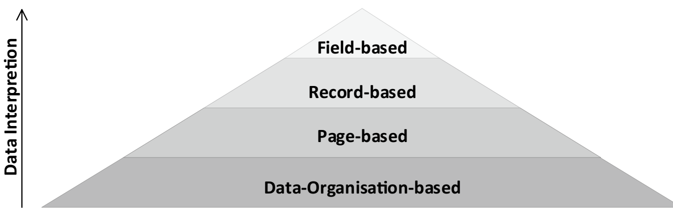

Fig. 2 Different NDP operation types build upon each other during the execution

execution models do), while executing queries or stored procedures. Rather such definitions need to be extracted and, together with page and record layouts, be passed to the NDP-device to ensure synchronization free NDP-execution. Hence, the need for explicit cross-layer format definitions arises (Sect. 2.3).

2.3 Structural elements: formats and layouts

The terms *format* and *layout* are often used interchangeably to describe the structure of the data held in a specific area of memory or storage. However, in the context of this paper and NDP, we distinguish between the two. First, *formats* describe the properties of a single *structural element*. Second, a *layout* defines the arrangement of multiple subordinate elements.

The *format* of an element defines the set of features (attributes, datatypes or sub-elements) as properties of that element. The *format* defines how an element is to be interpreted. Such properties can be user-, application-, or system-defined. Typical examples in databases are column/field-types, table definitions, and tuple formats. In NDP-environments dedicated software or hardware *format parsers* are required for such formats, as they need to be processed in-situ to execute NDP-operations (such as SUM or AVG, or to sort and compare, to name a few).

In contrast to Formats, *Layouts* describe the spatial/physical arrangement of elements within the memory space and the scope of a container element. Clearly, the contained elements can be of different formats. Typical examples are page- or record-layouts, or storage structures. The typical row-store record layout would comprise a record-header with a set of fields and flags, followed by a record-body, roughly containing the tuple-attributes as elements of the record format. Alternatively, the typical record layout in a KV-Store would comprise an *identifier/key* and a *value*.

In NDP settings various *layout accessors* (hardware or software) are needed on-device to retrieve the required elements efficiently from memory or storage. In contrast to *format parsers*, *layout accessors* have to be available entirely on the NDP device to retrieve the expected data storage locations. Depending on the NDP operation, a *format parser* might be applied on the result of a *layout accessor*.

Consider Fig. 3—a *Format Parser* will be required to process records or fields of an *image* table, while a *Layout Accessor* will be used to retrieve *Record2*.

2.4 Structural elements in databases

Formats and *Layouts* usually differ among DBMS types and are often optimized for their specific characteristics. In the following, we describe common concepts of widespread *Physical Storage Organisations* and list examples for *Format Parsers* and *Layout Accessors*. As a running example, Fig. 3 shows how these are mapped onto a Key/Value store (i.e. in MyRocks with RocksDB under the hood, which we use in the NDP-ImageProcessor scenario).

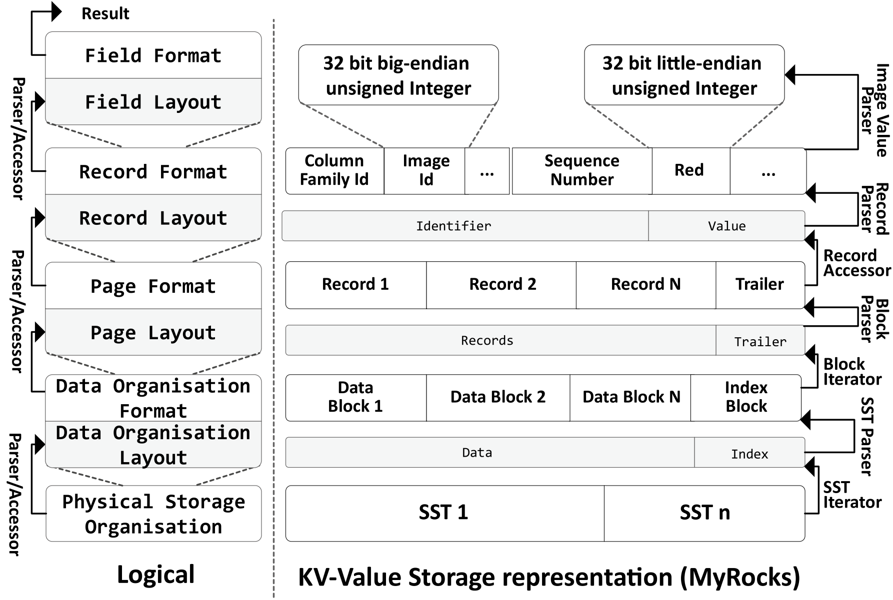

Fig. 3 (Left) Logical data organisation and nested definitions with formats and layouts. (Right) Record Format and Layout of the simple NDP-ImageProcessor divide fields in identifier and value for a simple table definition executed in MyRocks

2.4.1 Field format

Based on the DDL DB-object definitions in relational databases, the list of column data types, their Field Formats and their physical representations are known in advance or are engine-specific and therefore predefined. For instance, MyRocks defines an entire hierarchy of various number, decimal, string and date representations. Their Format describes the size in Bits or Bytes and a logic to translate the physical representation into an interpretable format for a given instruction set of the processing unit. For instance, the Format of the SQL clause INTEGER is trivially mapped to a 32-bit little-endian signed integer. Yet, if this field is part of the record identifier its physical representation is changed to big-endian to ensure a memory-comparable sort-order (see Fig. 3).

2.4.2 Record layout and format

In the typical DBMS, a physical record has a unique identifier. For instance, in the case of MyRocks, which utilizes RocksDB as a storage manager under the hood, this identifier includes a *column_family_id* and all *primary key* fields. In addition, RocksDB appends further information such as the sequence number and the key/value type. To reduce the physical space consumption, fields included in the identifier are not stored redundantly in the value. The following example depicts a simple table definition for the simple ImageProcessor, which stores every pixel of an image

as a single record. Figure 3 (and Fig. 4) shows the Record Layout and Format and the necessary information for a Record-based NDP operation.

2.4.3 Page layouts

Page layouts are a distinguishing characteristic of different DBMSs and have a major performance impact. They account for different access properties in terms of access and data locality, cache-awareness, prefetching as well as operation and maintenance costs. Three widely used examples are the N-ary Storage Model (NSM) [23], the Decomposition Storage Model (DSM) [5], and, the hybrid between those, the Partition Attributes Across (PAX) [2], while HPL [8] bridges the gap between the two. The difference is the arrangement of records within the space of a classical page. For the format we are working with, this is depicted in Fig. 4. However, there are various further layouts, such as Data Blocks [19] of HyPer, which optimize for different performance properties like scans and point queries on compressed data.

Another commonly used layout is the RCFile [11] which is typically applied in combination with MapReduce-Frameworks like Hadoop. While RCFile is the de-facto standard for such distributed storage systems, it is also specifically tailored for those use-cases. In contrast, SST-Files are more tailored towards general-purpose systems, which offers higher adoption potential for smaller systems. Independent of the used storage format, a major hurdle for implementing NDP-functionality is the implementation of record-accessors, which are able to retrieve a single record or key-value pair from the corresponding storage format. Since RCFiles do not rely on pointers within their data-structures, it would generally be easier to implement record-accessors for full scans. In turn though, the resulting system would be less applicable to general-purpose use-cases as lookups cannot be served efficiently. With the use of SST-Files, we aim to show that it is possible to implement NDP-functionality independent of the storage format, as long as the fields and records can be accessed.

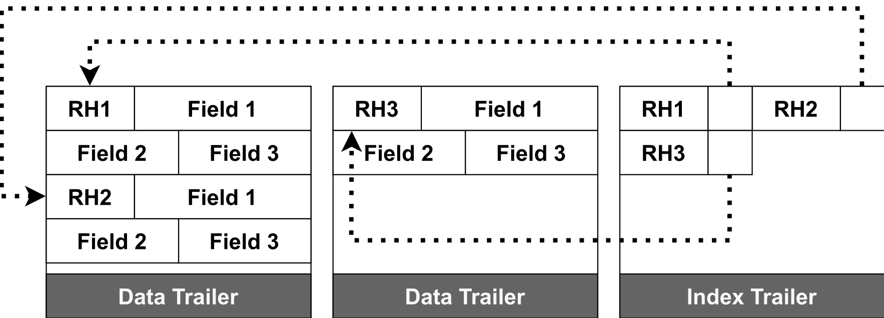

Fig. 4 Graphical representation of the SST file format used by RocksDB and MyRocks

2.4.4 Data storage organisation

Databases utilize various data structures to store records of different database objects. Hence, the most trivial storage organisation is a heap file – flat set of records placed on pages without any specific order. Alternatively, typical persistent Key/Value stores use multi-level LSM-trees.

For NDP calls, operating on the granularity of DB-Objects or even finer granularities (see Sect. 2.2), the organization of the underlying data structure is of importance, as the NDP-device needs to be able to: (a) navigate and iterate over the physical storage; and (b) should be able to perform address resolution in-situ. Consequently, depending on the operation, the *Layout Accessors* have to retrieve the requested data from storage.

3 Pushing down operations with format

After motivating the necessity of format pushdown from the conceptional perspective, we introduce a simple NDP-ImageProcessor. It uses a KV-Store, namely NoFTL-KV [29], to manage its images. It is based on the the widely-known RocksDB but integrates Native Storage management capabilities to leverage modern device characteristics. Moreover, it allows to call NDP functionality of the device. For the sake of simplicity, the ImageProcessor application disassembles each pixel of an image into its basic colours Red, Green and Blue, resulting in a record format similar to Fig. 3. The operations triggered by the application comprise a simple Get to retrieve colour information about a single pixel, and a histogram calculation, which counts the frequency of each colour within a certain area of an image.

Figure 5 gives a detailed view on the overall architecture. On the left-hand side, operations are executed over the conventional stack, while the left-hand side depicts the NDP execution model. To simplify the diagram, several layers, such as Kernel and FTL are omitted for the conventional stack. However, clearly visible is that, in case of NDP, format parsing and layout accessor executions happen on-device close to the physical storage instead of on the host, where NoFTL-KV is running. This requires both a modern Native Storage Manager as well as a pushdown mechanism ensuring that information required to configure and run the code for the respective Structural Elements is available on-device. For instance, current state information about the LSM-Tree (column families, levels, number and size of SSTs) and the record and field format as well as the SST layout must be provided to the NDP processing elements. In NoFTL-KV, such information is assembled along the way down to the device during run-time.

Since the entire processing flow is executed on the device, it can be optimized for the specific storage properties, e.g. number of concurrently addressable flash chips, or leveraging the pipelining effects of COSMOS's Flash Controller. The return path is lean since results are directly communicated to the ImageProcessor application without any intermediate layers.

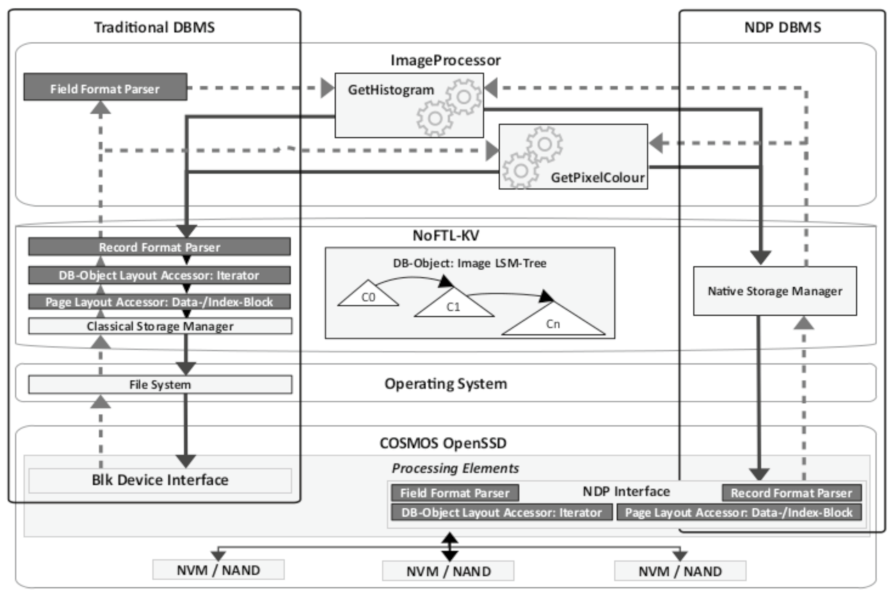

Fig. 5 The simple NDP-ImageProcessor application runs on top of NoFTL-KV, and operates either with conventional query requests via Block I/O to the COSMOS OpenSSD, or utilises the Native Storage Manager to issue NDP calls on the device. Depending on the stack, the Format Parser and Layout Accessors are executed within the KV-Store on the Host, or on processing elements of the NDP device

3.1 Testbed

For the evaluation, the system stack shown in Fig. 5 is set up on a host system, equipped with an Intel E6850 (3 GHz) CPU, 4 GB memory, and a 500GB SSD. The operating system is Debian 9.5 with kernel version 4.9.0. The host is connected to the *COSMOS OpenSSD* via an eight-lane PCIe 2.0 bus. The COSMOS platform [22] comprises a Zynq 7000 SoC, 1 GB RAM and a 512 GB NAND Flash module. The Flash and PCIe controllers are located on the FPGA part of the Zynq 7000 and are controlled by one of its ARM Cores (667 MHz). In case of the conventional stack, the COSMOS Flash storage is mounted as classical block device with an Ext4 file system. When running NDP experiments, the second ARM Core, which is running at a clock frequency that is more than four times slower than the host CPU, is responsible to run the NDP Format Parsers and Layout Accessors.

3.2 Evaluation

For the evaluation of both described operations, *GetPixelColour* and *GetHistogram*, are executed on the conventional stack as a baseline, and as NDP calls to compare the performance benefits. The pre-loaded dataset comprises 100,000,000 KV-Pairs

of pixels. Experiments are executed three times and the average result is reported. To ensure comparability, the page cache of the operating system is cleared every 2 s.

3.2.1 Record-based operation: GetPixelColour

Within the given architecture, this operation demonstrates a simple GET on the LSM-Tree. To look up the record, the LSM-Tree structure as data organisation of NoFTL-KV has to be processed. Therefore, firstly, the physical pages of the required Data Blocks have to be identified via a layout accessor. Secondly, requests for these pages can be issued to the on-device Flash Controller. With their successful completion, they can be parsed and records extracted according to the associated parses and accessors. The record itself is not interpreted, and no further calculations or extractions are necessary to retrieve the requested result. The NDP-operation, Layout Accessors and Format Parsers are executed on the slow ARM core without any FPGA support. Ahead of the experiment, 1000 pixel coordinates are pre-generated randomly to ensure an equal access pattern in all executions. We run the experiment once without any defined caches in NoFTL-KV or COSMOS (Fig. 6a), and once with caches for the index enabled (Fig. 6b).

While the conventional stack via the block device interface yields significant response time fluctuations, NDP executions exhibit robust performance and stable response times. In absence of any on-device caching, the NDP stack has a slightly inferior performance. Detailed analysis of I/O traces on COSMOS show that not every read request is served by the device due to non-deactivatable caches within

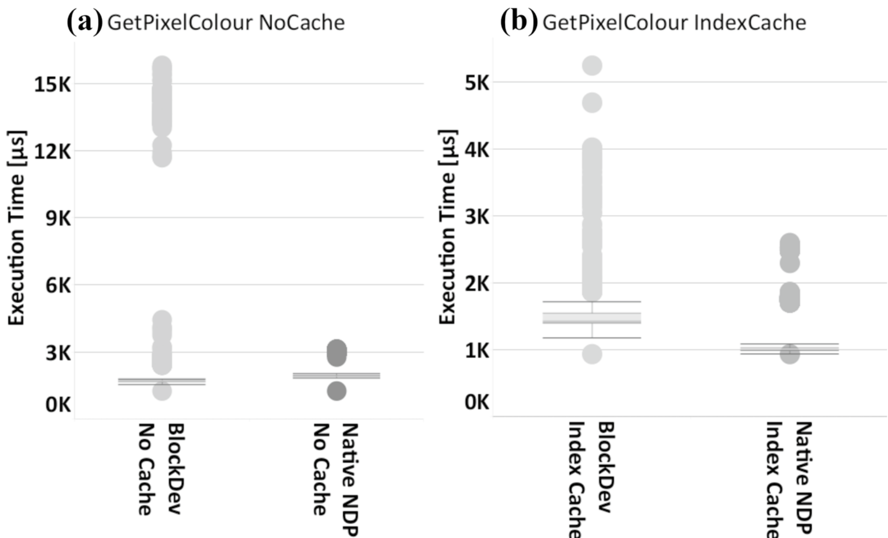

Fig. 6 NDP calls exhibit robust performance, in general. In absence of any on-device/NDP caches, the traditional stack executes slightly faster, due to non-deactivatable caches in the operating system. However, if small on-device index caches are enabled, NDP's performance improves around 33% against the baseline

the kernel or file system and distort the results somewhat. However, with a small on-device index cache, the NDP performance is around *33% better* than the conventional stack. Only when an index block has to be fetched, the performance drop behind the average execution time of the baseline.

3.2.2 Field-based operation: GetHistogram

This operation utilizes an Iterator Accessor to scan through pixels of a given area. Internally, this iterator leverages the format parsers and accessor similarly to the simple lookup of the previous experiment (Fig. 7). Yet, via a *Next* operation, Data Blocks are automatically fetched and parsed. By further applying a Field Format Parser the colours can be read and the respective bins of the histogram incremented. The NDP-operation, Layout accessors and Format parsers are executed on the slow ARM core without FPGA support. The experiment is run with different selectivities on the entire data set.

Both curves show a linear execution time, increasing with data size, due to same low-level NAND Flash I/O behaviour. However, by leveraging pipelining effects of the Flash Controller when fetching new pages, and exploiting the entire parallelism of the Flash chips, the performance can continuously be improved by approximately 27%.

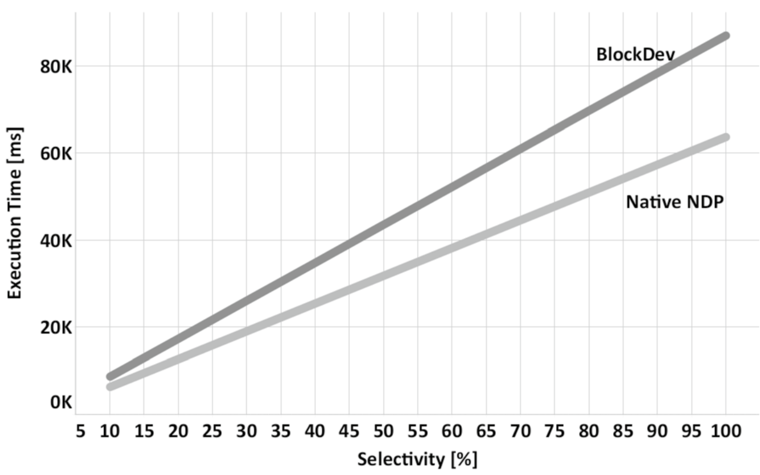

Fig. 7 The performance of both stacks increases linearly for the given data set size across a range of selectivities. However, due to optimizations exploiting COSMOS's hardware properties improve the performance of NDP by about 27%

4 Exploiting explicit data formats

The previous evaluation already provides motivation for the use of explicit cross-layer data formats. By using the explicit data formats, it becomes possible to exploit the on-device ARM-cores for computational loads. In systems without explicit data formats, all this computational load would have to be executed by the host CPU. An additional advantage that comes with the use of explicit data formats is the possibility of hardware acceleration. Specifically, the COSMOS hardware platform features a SoC comprised of an ARM-based processing System (PS) and a FPGA-based portion (PL). In the original design, the PL-part of the SoC is used to implement only the Flash controllers as well as the NVMe controller. Due to the under-utilization of the PL-portion, this opens up additional potential for hardware acceleration. Examples for this are given in [27, 28]. In both works, the COSMOS hardware design is extended using processing elements that carry out computational tasks. This section will discuss the performance of [27] as well as its applicability to typical use-cases.

4.1 Execution stack

First, we will give an overview over our system stack and discuss, how different parts of the stack interact with each other. Additionally, differences between this work and [27, 28] will be highlighted.

At the highest layer, a benchmark-application is run on the host. In this work, the application relies on a MyRocks-database to store the data processed by the Image-Processor. MyRocks in turn relies on an underlying RocksDB-KV store for actual persistant storage. Due to MyRocks being specifically designed to work with RocksDB, this integration is almost seamless. Instead of using a plain RocksDB, our stack relies on an enhanced RocksDB, called NoFTL-KV [29]. In difference to regular RocksDB, NoFTL-KV assumes a specific storage device underneath, which it can access natively from userspace, without any intermediary layers (i.e., OS-drivers or block device drivers). NoFTL-KV allows the DBMS to directly work on the actual flash chips instead of relying on intermediary compatability layers. Instead of using virtual addresses that have to be resolved, NoFTL-KV can directly access the corresponding physical flash pages on the COSMOS OpenSSD via the NVMe-protocol. To allow this direct access, it is also required that the COSMOS OpenSSD is running a specialized firmware that allows this kind of access to it. This firmware is responsible for actually performing incoming NVMe-requests. In our case, both NoFTL-KV and the COSMOS-firmware have been extended to also allow the execution of user-defined commands via NVMe.

While the setup of [27, 28] is very similar, there are two key-differences. In those works, the topmost layer is not MyRocks, but instead just NoFTL-KV. In both cases the stored data is structured in a specific manner, but in this work, the structure is defined by a MySQL-Schema, while [27] relies on the benchmark-application to define a clear data structure.

The second difference is the configuration of the COSMOS OpenSSD. The COSMOS device can be configured with different hardware-designs, enabling different hardware-functionality. In this work, the hardware-functionality is limited to running the interface via NVMe and the access to the on-device flash chips. In [27, 28], the functionality is extended by using hardware accelerators for certain database-related functions. Both hardware-configurations rely on the same baseline hardware-design and COSMOS-firmware. For actually using the hardware-accelerators, the firmware has to be extended to actually provide the functionality to higher layers. This is achieved, by implementing new user-defined NVMe-commands, which tell the firmware to call corresponding on-device functionality. This functionality can either be realized by using the on-device ARM-cores, or the accelerators implemented using the on-device FPGA-fabric. In this work, only the on-device ARM-cores are used, while [27, 28] also realize additional functionality via the FPGA-fabric.

In the case of FPGA-based functionality, it is then necessary to implement the corresponding control logic within the COSMOS-firmware. Depending on the complexity of the hardware accelerators, the control logic is typically performed by reading and writing a limited number of control registers, which are simply memory-mapped into address space of the ARM-cores. The resulting execution stacks are also shown in Fig. 8.

The following subsections will go into more detail on how the FPGA-fabric can be used to further increase the performance based on the findings of [27, 28]

4.2 Accelerator architecture

The architecture presented in [27] is relatively straight-forward. The processing element (PE) can access the on-device DRAM of the COSMOS. Additionally, the PE can be configured and controlled via control-registers that can be accessed by the PS. Within the PE, a pipeline is built, where entire data-blocks are first loaded from

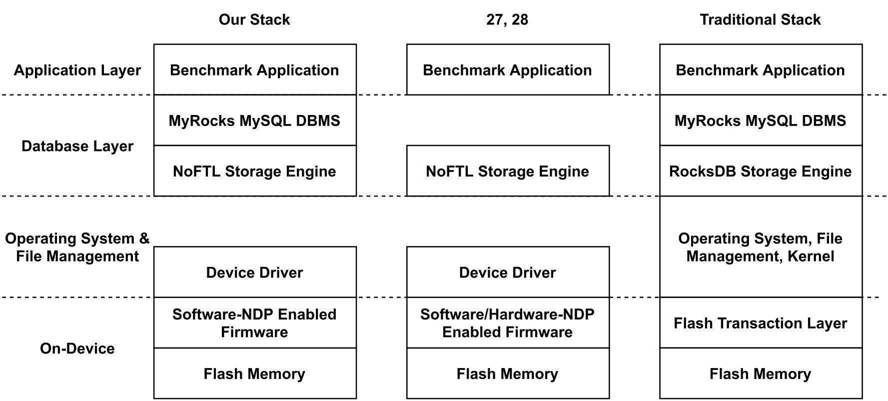

Fig. 8 Execution stack of our implementation in comparison to [27, 28] and a traditional execution stack, as described in Sect. 4.1

the DRAM and then grouped into a stream of tuples, each representing a single key-value pair. These key-value pairs are then passed into a filtering unit which can apply simple logical predicates to each key-value pair, and depending on the evaluation of the predicate, can discard some key-value pairs. Afterwards, a data transformation is performed, which allows the removal of unnecessary fields and the re-structuring of each individual key-value pair. The resulting key-value pairs are then automatically stored back to the on-device DRAM.

Depending on the underlying structure of the key-value pairs, the corresponding PEs may differ in the amount of FPGA resources they require. [27] report the hardware utilizations of two different kinds of PEs and the amount of required Slices ranges from 1.84 to 15.14% of the overall available Slices (54650). Especially for smaller key-value pairs, this enables a lot more parallelism. With using just the two available ARM-cores, the degree of parallelism is limited to two, while [27] uses up to *seven* concurrent PEs.

4.3 Accelerating database-operations

Exploiting the available compute-parallelism, as well as the heterogeneity of their system, [27] implements simple GET and SCAN operations. It evaluates the execution times of both using the traditional approach on the host-CPU (Blk), software-based NDP (SW-NDP) and hardware-accelerated NDP (HW-NDP). For their evaluation, a benchmark composed of key-value pairs built from publications, references, conference venues, and authors is employed. The overall number of key-value pairs in this dataset is roughly 48 million, with the total size being 2.4 GB.

Due to the use of the concepts presented in this paper, it is possible to reduce the GET execution time from 8ms (Blk) down to 5.68 ms (SW-NDP), and that of the SCAN operation from 6.96s (Blk) down to 4.81 s (SW-NDP). With the high potential for concurrency of the SCAN operation, it is possible to further reduce the SCAN execution time down to 3.35 s (HW-NDP) using hardware acceleration. Additionally, [27] implements the Betweenness Centrality (BC) algorithm and also measure the impact of software-based and hardware-accelerated NDP. In the case of BC, the original execution time of 1027.84 s (Blk) is reduced down to 426.62 s (SW-NDP) and 374.81s (HW-NDP) using software-based and hardware-accelerated NDP, respectively. The BC is implemented as software for the ARM core that can be configured to exploit the presented PEs (HW-NDP). Alternatively, the corresponding parts of computation are done using just the ARM core (SW-NDP).

4.4 Integration into existing systems

Independently of the potential for performance increases, another issue for new database paradigms and concepts is the complexity of adoption as well as usability. Since databases are very important in academic and industrial applications alike, they are already widely used. Many of these applications rely on a specific DBMS like MySQL or Postgre SQL and switching the DBMS is often not an easy task, due to applications running on top of the DBMS. Therefore, it is important to achieve a

degree of compatibility to these systems that enables a simple and easy switch from the existing stack to a hardware-accelerated stack.

To give a perspective on this, we want to take a closer look at MyRocks, which is a RocksDB-based storage engine that supports MySQL. Under the hood, a MyRocks database uses RocksDB for storing the data, while still allowing typical MySQL-Queries. Therefore, many of the performance increases achieved with a RocksDB database also apply to MyRocks. The two major restrictions to this statement are the following: 1. It might be necessary to adapt MyRocks to support command push-down to allow the full use of Native Storage and NDP. 2. Since our hardware-accelerated NDP is only able to process selected data-types, it would also be necessary to transform the MyRocks database in a way that removes variable-size data from as many relations as possible, since relations with variable-size data cannot be processed by our current NDP-PEs.

Assuming that the MyRocks-relations meet these restrictions, the accelerated NDP could be integrated at different points in the stack. In our use-cases, it was integrated at the level of RocksDB, which provides the most basic operations such as SCAN and GET. Since MySQL-Queries performed on MyRocks will invoke the corresponding functionality in the underlying RocksDB, no further implementation changes would be required. Other functions of RocksDB could also be adapted to exploit the performance increases of NDP. Obvious candidates would be RANGE_SCANs and MULTI_GETs. To adapt these, their filtering functionality has to be replaced by calls to the PEs. These operations can be wrapped into function-calls without significant overhead, making this overall approach very simple. A major downside of this approach is the introduction of additional layers in the overall stack. Since a significant portion of our performance increases are achieved by *removing* unnecessary compatibility layers, it would potentially make more sense to instead allow MyRocks to directly access the on-device functionality. While this would incur much higher implementation effort, it might also yield even greater performance increases.

4.5 Discussion

From the results presented in [27, 28], it is clear that the potential of hardware acceleration is great. To unveil this potential, it is absolutely necessary to use explicit cross-layer data formats. Especially for the operations allowing more concurrency, such as SCAN and BC, the potential of hardware-acceleration is large and may provide great performance improvements. In addition, computational load is shifted from the host-CPU to the storage-device, which in theory allows the CPU to carry out more complex operations, while simple operations are handled using on-device NDP across *multiple* smart storage devices.

Even though these initial results are promising, the use of hardware-accelerated NDP introduces a new problem: To exploit hardware-acceleration, [27, 28] rely on *manually-designed* PEs and corresponding control logic. To address this problem, several approaches could be used. The most obvious is the use of High-Level Synthesis (HLS), which allows the generation of hardware accelerators from high-level code (e.g., C/C++). While HLS is generally easier to use for software- or

database-engineers, it generally produces sub-optimal results in comparison to manually-written or generated Hardware Description Language (HDL) based accelerators. Considering the accelerators used in [27], it is clear that many of the sub-modules depend on the underlying structure of the processed key-value pairs.

Exploiting this structure, it could be possible to implement the *automatic* generation of HDL-based accelerators. This would also enable the automatic generation of the software interface for the control of the accelerators. Such a specialized NDP-PE generator could potentially create better hardware accelerators than general-purpose HLS, while also providing all of the control software, without requiring any additional knowledge about hardware design.

Apart from this caveat, the work presented in [27, 28] is a very good example for a field-based NDP-operation, where processing and computation can be performed close to storage, and the processed data can be handled at the finest degree of granularity. According to the model depicted in Fig. 2, the relevant cross-layer data formats are implicitly encoded into the accelerator to allow it the interpretation of the data down to single fields.

5 Conclusion

We motivate the necessity for data format pushdown in NDP scenarios. To this end, we put the terms format and layout in an NDP context and discuss a type hierarchy for NDP operations. Processing data format and layout definitions on device and creating/generating dedicated parsers and accessors allows optimizing for the given hardware properties and improving the execution time. The evaluation demonstrates the impact of NDP by improving a Record-based operation by around 33% and a Field-based operation by approximately 27%. Additionally, it is worth mentioning that the NDP operations are executed on an ARM Core, which is clocked at one-quarter of the host-CPUs frequency. Additionally, we discuss the works by Vinçon et al. [27, 28] on exploiting explicit data formats using hardware-acceleration using the FPGA-portion of the COSMOS OpenSSD platform. Their results further motivate and explain the necessity of explicit cross-layer data formats regarding a more hardware-focused context.

6 Future work

From these findings and the prior works [27, 28], it becomes increasingly clear that explicit data formats may offer great advantages for Near-Data Processing. While these works already achieve significant improvements, they are also reliant on detailed knowledge about the hardware architecture of the COSMOS platform. An interesting avenue of research could be the automation of the hardware-development process. This would also increase the accessibility of the presented methods to a wider audience. Exploiting the regularity of the two accelerators presented in [27] could yield similar performance improvements while streamlining the development process.

Another interesting path of research could lie in the use of a different hardware platform. Despite its out-of-the-box functionality and the fact that it offers all of the required controllers and firmware, the COSMOS OpenSSD used in this work is becoming an outdated platform. Especially considering the sheer age of the Xilinx Zynq-7000 chip (released 2011), it could be interesting to test the limits of the presented concepts on a newer platform like the Samsung SmartSSD used in [20], which features a more recent FPGA-SoC. The increased number of hardware-resources could be used to implement more advanced hardware-acceleration, which in turn could allow higher performance-increases or the execution of more complex NDP-operations.

7 Related work

Using Formats and Layouts to describe storage elements are concepts from the early beginning in the research and development of databases. Page layouts such as NSM [23] and DSM [5] date back to at least the 80s. Yet, also recently, new variations were proposed like PAX [2], BLU [24] of DB2, or DataBlocks of HyPer [19]. Some layouts even make use of the hardware properties of Flash-like Delta Records in [10].

Likewise, the concept of Near-Data Processing is deeply rooted in *database machines* [4] developed in the 1970s-80s or Active Disk/IDISK [1, 16, 25] from the late 1990s.

With the advent of Flash technologies and reconfigurable processing elements Smart SSDs [7, 15, 26] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [30]. Biscuit [9] is a proposal for a general NDP framework. JAFAR [3, 31] is one of the first systems to target NDP for DBMS (column-store) use, whereas [14, 18] target joins besides scans. The use of NDP in the realm of KV-Stores has been investigated in [6, 17]. Kanzi [12], Caribou [13] and BlueDBM [21] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on NDP focus mainly on either bandwidth optimizations or on the execution of specific algorithms. Yet, this paper gives a broad overview of necessary formats and layouts, in particular for databases to issue several types of operations as NDP calls.

Funding Open Access funding enabled and organized by Projekt DEAL. This work has been supported by BMBF PANDAS - 01IS18081C/D, DFG Grant neoDBMS - 419942270 and HAW Promotion, MWK, Baden-Würrtemberg, Germany.

Compliance with ethical standards

Conflict of interest Ilia Petrov was part of the program committee of Joint International Workshop on Big Data Management on Emerging Hardware and Data Management on Virtualized Active Systems (HardBD & Active 2020), but has not reviewed the original paper.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Acharya, A., Uysal, M., Saltz, J.: Active disks: programming model, algorithms and evaluation. In: Proceedings of ASPLOS (1998)
2. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: Proceedings of VLDB, 01 (2001)
3. Babarinsa, O.O., Idreos, S.: JAFAR: near-data processing for databases (2015)
4. Boral, H., DeWitt, D.J.: Parallel architectures for database systems. In: Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28 (1989)
5. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. In: Proceedings of SIGMOD (1985)
6. De, A., Gokhale, M., Swanson, S., Al, E.: Minerva: Accelerating data analysis in next-generation SSDS. In: Proceedings of FCCM (2013)
7. Do, J., Patel, J., DeWitt, D., Al, E.: Query processing on smart SSDs: opportunities and challenges. In: Proceedinds of SIGMOD (2013)
8. Graefe, G., Petrov, I., Ivanov, T., Marinov, V.: A hybrid page layout integrating PAX and NSM. In: Proceedings of the 17th International Database Engineering & Applications Symposium, IDEAS '13, pp. 86–95 (2013)
9. Gu, B., Yoon, A.S., Al., E.: Biscuit: a framework for near-data processing of big data workloads. In: Proceedings of ISCA (2016)
10. Hardock, S., Petrov, I., Gottstein, R., Buchmann, A.: From in-place updates to in-place appends. In: Proceedings of SIGMOD '17 (2017)
11. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., Xu, Z.: Rcfile: a fast and space-efficient data placement structure in mapreduce-based warehouse systems. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 1199–1208 (2011). https://doi.org/10.1109/ICDE.2011.5767933
12. Hemmatpour, M., Sadoghi, M., et al.: Kanzi: a distributed, in-memory key-value store. In: Proceedings of Middleware (2016)
13. István, Z., Sidler, D., Alonso, G.: Caribou: intelligent distributed storage. In: Proceedings of VLDB (2017)
14. Jo, I., Bae, D.h., Al., E.: YourSQL: a high-performance database system leveraging in-storage computing. In: Proceedings of VLDB (2016)
15. Kang, Y., Kee, Y.S., et al.: Enabling cost-effective data processing with smart SSD. In: Proceedings of MSST (2013)
16. Keeton, K., Patterson, D.A., Hellerstein, J.M.: A case for intelligent disks (IDISKs). In: SIGMOD Rec (1998)
17. Kim, J., et al.: PapyrusKV: A high-performance parallel key-value store for distributed NVM architectures. In: Proceedings of SC (2017)
18. Kim, S., Lee, S.W., Moon, B., et al.: In-storage processing of database scans and joins. Inf. Sci. **327**, 183 (2016)
19. Lang, H., Mühlbauer, T., Funke, F., et al.: Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: Proceedings of SIGMOD 16 (2016)
20. Lee, J.H., Zhang, H., Lagrange, V., Krishnamoorthy, P., Zhao, X., Ki, Y.S.: SmartSSD: FPGA accelerated near-storage data analytics on SSD. IEEE Comput. Architect. Lett. **19**(2), 110–113 (2020)
21. Ming, S.W.J., Arvind, et al.: BlueDBM: an appliance for big data analytics. In: Proceedings of ISCA (2015)

22. OpenSSD Project: COSMOS Project Documentation (2019). http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources
23. Ramakrishnan, R., Gehrke, J.: Database Management Systems (2003)
24. Raman, V., Attaluri, G., Barber, R.: DB2 with BLU acceleration: so much more than just a column store. In: Proceedings of VLDB, 13 (2013)
25. Riedel, E., Gibson, G.A., Faloutsos, C.: Active storage for large-scale data mining and multimedia. In: Proceedings of VLDB (1998)
26. Seshadri, S., Swanson, S., et al.: Willow: a user-programmable SSD, pp. 67–80. USENIX, OSDI (2014)
27. Vinçon, T., Bernhardt, A., Petrov, I., Weber, L., Koch, A.: nKV: near-data processing with KV-stores on native computational storage. In: Proceedings of the 16th International Workshop on Data Management on New Hardware, DaMoN 20. Association for Computing Machinery, New York, USA (2020). https://doi.org/10.1145/3399666.3399934
28. Vinçon, T., Weber, L., Bernhardt, A., Riegger, C., Hardock, S., Knoedler, C., Stock, F., Solis-Vasquez, L., Tamimi, S., Koch, A.: nKV in action: accelerating KV-stores on native computation storage with near-data processing. In: Proceedings of the VLDB Endowment, vol. 13 (2020)
29. Vincon, T., Hardock, S., Riegger, C., Oppermann, J., Koch, A., Petrov, I.: NoFTL-KV: tackling write-amplification on KV-stores with native storage management. In: Proceedings of EDBT (2018)
30. Woods, L., Teubner, J., Alonso, G.: Less Watts, More performance: an intelligent storage engine for data appliances. In: Proceedings of SIGMOD (2013)
31. Xi, S., Babarinsa, O., Athanassoulis, M., Idreos, S.: Beyond the wall: near-data processing for databases. In: Proceedings of DAMON (2015)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Lukas Weber[1] · **Tobias Vinçon**[2] · **Christian Knödler**[2] · **Leonardo Solis-Vasquez**[1] · **Arthur Bernhardt**[2] · **Ilia Petrov**[2] · **Andreas Koch**[1]

Tobias Vinçon
tobias.vincon@reutlingen-university.de

Christian Knödler
christian.knoedler@reutlingen-university.de

Leonardo Solis-Vasquez
solis@esa.tu-darmstadt.de

Arthur Bernhardt
arthur.bernhardt@reutlingen-university.de

Ilia Petrov
ilia.petrov@reutlingen-university.de

Andreas Koch
koch@esa.tu-darmstadt.de

[1] Embedded Systems and Applications Group, TU Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Germany

[2] DBLab, Reutlingen University, Alteburgerstrasse 150, 72762 Reutlingen, Germany