On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems

Tobias Vinçon [*1], Arthur Bernhardt [*2], Lukas Weber [#3], Andreas Koch [#4], Ilia Petrov [*5]

[*] *Data Management Lab, Reutlingen University, Germany*
[1] tobias.vincon@reutlingen-university.de,
[2] arthur.bernhardt@reutlingen-university.de,
[5] ilia.petrov@reutlingen-university.de

[#] *Embedded Systems and their Applications Group, TU Darmstadt, Germany*
[3] weber@esa.tu-darmstadt.de,
[4] koch@esa.tu-darmstadt.de

***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 viable.**

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 NoFTL-KV and the COSMOS hardware platform.

***Index Terms*—near-data processing, data format, data layout**

I. 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 traditional system 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

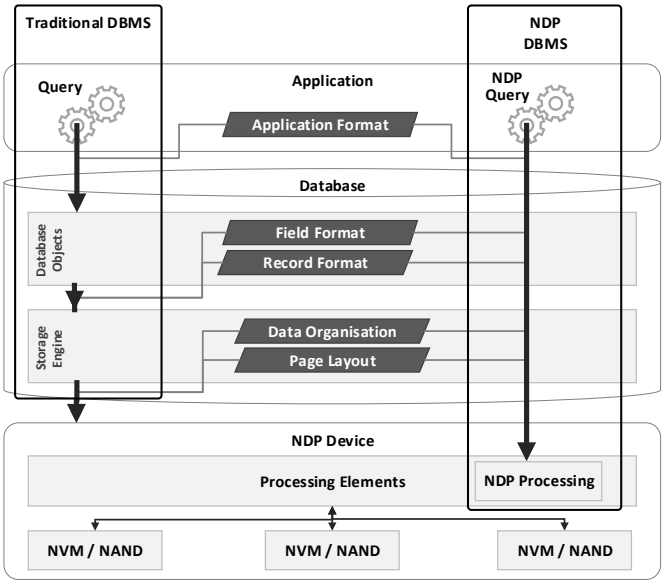

Fig. 1: In contrast to classical queries, NDP operations must have all necessary format and layout information to execute the respective operations in-situ without host interaction.

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 (Figure 1). Consequently, in SQL queries, format definitions of the upper layers are utilized to retrieve

and process data from the layer bellow. 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 K/V store-based NDP-ImageProcessor application. It naïvely 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:

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

The remainder of this paper is structured as follows: Section II reviews the basic concepts of NDP and NDP Operations, and provides detailed conceptional background information about formats and layouts in databases. An illustrative implementation of format pushdown is presented and evaluated via an ImageProcessor in Section III. We conclude with Section IV and discuss related work in Section V.

II. Conceptional Background

A. 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 operation in a QEP or also more general data-intensive operations like joins or UDFs.
2) Result set: In absence of proper result set management it is mandatory 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) Synchronization-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 with the host.

B. 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 (Figure 2).

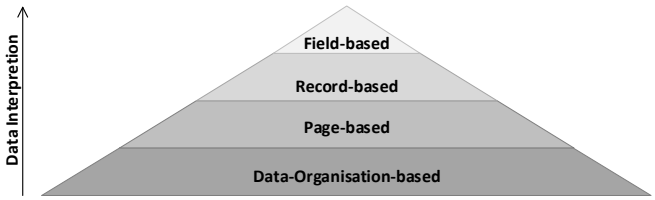

Fig. 2: Different NDP operation types build upon each other during the execution.

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 [1]. 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 Section II-C. 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 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 (e.g. Data Organisation) at runtime, as most interpretable DBMS kernels 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 (Section II-C).

C. Structural Elements: Formats and Layouts

The terms *format* and *layout* are often used interchangeably to describe the structure of the data in a specific area in memory or storage. However, in the context of this paper and NDP we distinguish between the two and provide their definitions below (Figure 3).

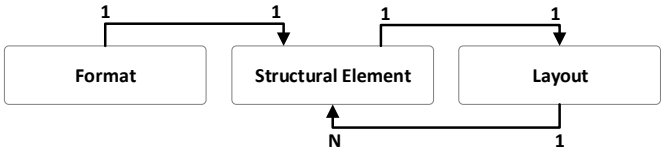

Fig. 3: Formats describe the properties of a single structural element, while layouts define the arrangement of multiple subordinate elements.

1) Data Formats: 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).

2) Data Layouts: In contrast to *Formats*, Layouts describe the spacial/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 Figure 4 – 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*.

D. 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 wide-spread *Physical Storage Organisations* and list examples for *Format Parsers* and *Layout Accessors*. As a running example Figure 4 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).

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 natural sort-order (see Figure 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 4 (and Figure 5) shows the Record Layout and Format and the necessary information for a Record-based NDP operation.

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 spread representatives are the N-ary Storage Model (NSM) [2], the Decomposition Storage Model (DSM) [3], and, the hybrid between those, the Partition Attributes Across (PAX) [4].

The difference is the arrangement of records within the space of a classical page as shown in Figure 5. However, there are various further layouts, such as Data Blocks [5] of HyPer, which optimize for different performance properties like scans and point queries on compressed data. IPA [6] and IPA-IDX [7] optimize for byte-level writes and write-amplification.

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.

```
CREATE TABLE `images` (
    `imageid` INT(10) UNSIGNED NOT NULL,
    `x` BIGINT(20) UNSIGNED NOT NULL,
    `y` BIGINT(20) UNSIGNED NOT NULL,
    `red` INT(10) UNSIGNED NOT NULL,
    `green` INT(10) UNSIGNED NOT NULL,
    `blue` INT(10) UNSIGNED NOT NULL,
    PRIMARY KEY(`imageid`,`x`,`y`)
);
```

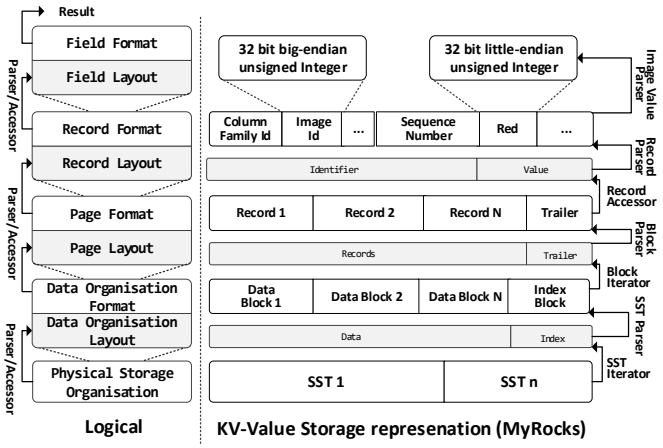

Fig. 4: (Left) logical data organisation and nested definitions with formats and layouts. (Right) record Format and Layout of the simple NDP-ImageProcessor divide fields in identifier and value for a simple table definition executed in MyRocks.

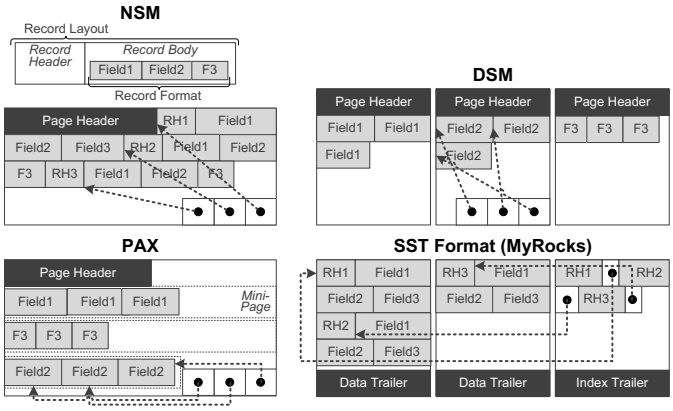

Fig. 5: Examples of different DBMS page layouts and the SST Layout, widely used in KV-Stores like RocksDB.

For NDP calls, operating on the granularity of DB-Objects or even finer granularities (see Section II-B), 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.

III. Pushing down Operations with Format

A. The ImageProcessor

After motivating the necessity of format pushdown from the conceptional perspective, we introduce a simple NDP-ImageProcessor. It uses NoFTL-KV [1], which is based on the pluggable storage engine MyRocks, to manage its images. For the sake of simplicity, each pixel of an image is disassembled into its basic colours Red, Green and Blue, resulting in a record format similar to Figure 4. 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.

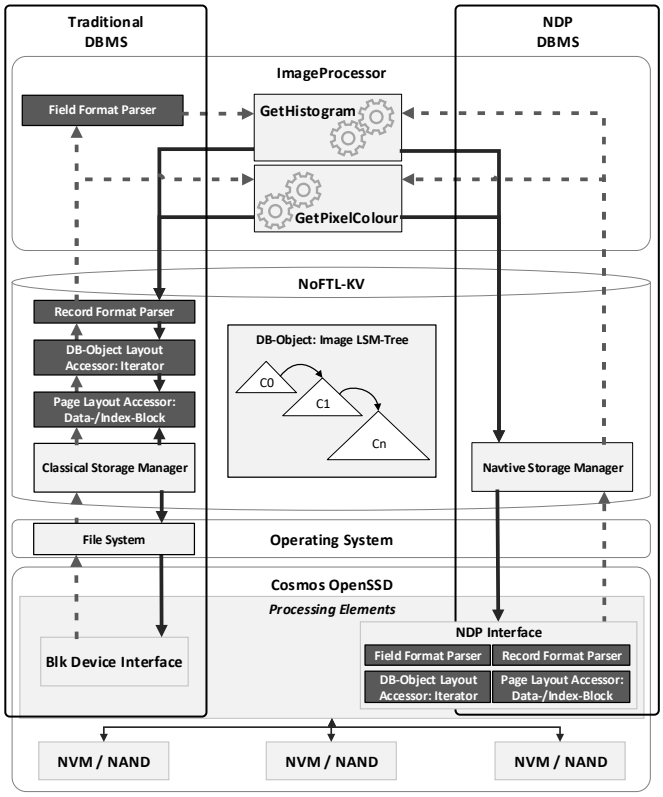

Fig. 6: The simple NDP-ImageProcessor application runs on top of NoFTL-KV, which is based on the pluggable storage engine of MyRocks, and operates either with conventional query requests via Block I/O to the Cosmos OpenSSD, or utilises the Native Storage Manager to issue NDP calls on the device. Depending 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.

Figure 6 gives a detailed view on the overall architecture. On the left-hand side, operations are executed over the conventional stack, while the right-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 executions for format parsing and layout accessors 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 and the record and field format as well as the SST layout must be provided to the NDP processing elements.

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 application without any intermediate layers.

B. Testbed

For the evaluation the system stack shown in Figure 6 is set up on a host system, equipped with an Intel E6850 (3GHz) CPU, 4GB 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* (see Figure 7) via a four lane PCIe 3.0 bus. The COSMOS platform [8] comprises a Zynq 7000 SoC, 1GB RAM and a 512GB 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.

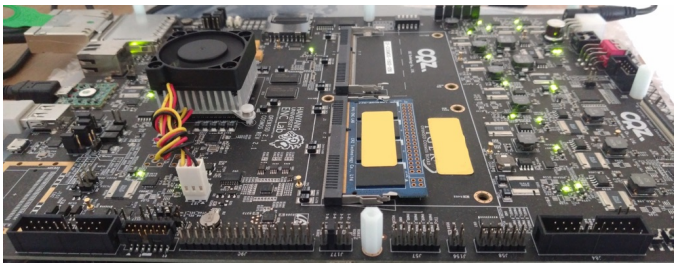

Fig. 7: The Cosmos OpenSSD board resembles a classical enterprise SSD connected via PCIe Gen3 x4 to the host. It comprises a Zynq 7000 SoC with an FPGA and a dual-core ARM, 1GB RAM and a 512GB NAND Flash module.

C. 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. These are application-specific versions of typical database operations like lookup and scan. 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 seconds.

1) Record-based Operation – GetPixelColour: Within the given architecture, this operation demonstrates a simple GET on the LSM-Tree. The record is not interpreted, and no further calculations or extractions are necessary to retrieve the required result. The NDP-operation, Layout Accessors and Format Parsers are executed on the slow ARM core without 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 (Figure 8.a), and once with caches for the index enabled (Figure 8.b).

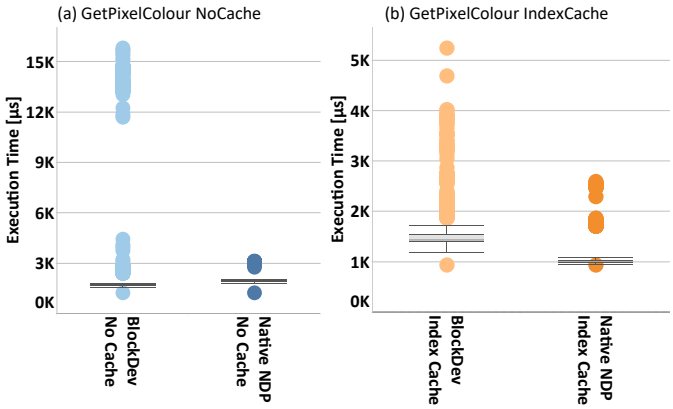

Fig. 8: NDP calls exhibit robust performance, in general. In absence of any on-device/NDP caches, the traditional stack executes slightly faster, due to non-deactivatable caches in the operating system. However, if small on-device index caches are enabled, NDP's performance improves around 33% against the baseline.

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 the kernel or file system 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.

2) Field-based Operation – GetHistogram: This operation utilizes an Iterator Accessor to scan through pixels of a given area. By 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, and exploiting the entire parallelism of the Flash chips,

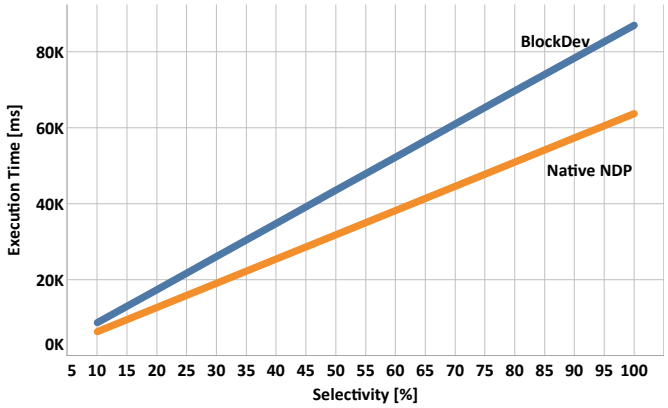

Fig. 9: The performance of both stacks increases linearly for the given data set size across a range of selectivities. However, due to optimizations exploiting Cosmos's hardware properties improve the performance of NDP by about 27%.

the performance can continuously be improved by approximately 27%.

IV. Conclusion

In the present paper, the necessity for format pushdown in NDP scenarios is clearly motivated. 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 only ¼ of the host CPU.

V. 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 [2] and DSM [3] date back to at least the 80s. Yet, also recently, new variations were proposed like PAX [4], BLU [9] of DB2, or DataBlocks of HyPer [5]. Some layouts even make use of the hardware properties of Flash like Delta Records in [6].

Likewise, the concept of Near-Data Processing is deeply rooted in *database machines* [10] developed in the 1970s-80s or Active Disk/IDISK [11]–[13] from the late 1990s.

With the advent of Flash technologies and reconfigurable processing elements Smart SSDs [14]–[16] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [17]. Biscuit [18] is a proposal for a general NDP framework. JAFAR [19], [20] is one of the first systems to target NDP for DBMS (column-store) use, whereas [21], [22] target joins besides scans. The use of NDP in the realm of KV-Stores has been investigated in [23], [24]. Kanzi [25], Caribou [26] and BlueDBM [27] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on NDP focusses 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.

Acknowledgment

This work has been partially supported by *HAW Promotion MWK, Baden-Württemberg, Germany*; *BMBF PANDAS – 01IS18081C/D*, *DFG Grant neoDBMS – 419942270*

References

[1] T. Vincon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, and I. Petrov, "Noftl-kv: Tackling write-amplification on kv-stores with native storage management," in *Proc. EDBT*, 2018.
[2] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 2003.
[3] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," *In Proc. SIGMOD 1985*, 1985.
[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving Relations for Cache Performance," *Proc. VLDB 01*, 2001.
[5] H. Lang, T. Mühlbauer, F. Funke, and et al., "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation," in *Proc. SIGMOD 16*, 2016.
[6] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "From In-Place Updates to In-Place Appends," in *Proc. SIGMOD '17*, 2017.
[7] S. Hardock, A. Koch, T. Vinçon, and I. Petrov, "Ipa-idx: In-place appends for b-tree indices," in *Proc. DaMoN*, 2019.
[8] *COSMOS Project Documentation*, OpenSSD Project, January 2019, http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Technical_Resources.
[9] V. Raman, G. Attaluri, and R. Barber, "DB2 with BLU Acceleration: So much more than just a column store," *Proc. VLDB 13*, 2013.
[10] H. Boral and D. J. DeWitt, "Parallel architectures for database systems," A. R. Hurson, L. L. Miller, and S. H. Pakzad, Eds., 1989, ch. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28.
[11] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proc. ASPLOS*, 1998.
[12] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, 1998.
[13] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proc. VLDB*, 1998.
[14] J. Do, J. Patel, D. DeWitt, and e. al, "Query processing on smart ssds: Opportunities and challenges," in *Proc. SIGMOD*, 2013.
[15] S. Seshadri, S. Swanson, and et al., "Willow: A User-Programmable SSD," *USENIX, OSDI*, pp. 67–80, 2014.
[16] Y. Kang, Y.-s. Kee, and et al., "Enabling cost-effective data processing with smart SSD," in *Proc MSST*, may 2013.
[17] L. Woods, J. Teubner, and G. Alonso, "Less watts, more performance: An intelligent storage engine for data appliances," in *Proc. SIGMOD*, 2013.
[18] B. Gu, A. S. Yoon, and e. al., "Biscuit: A Framework for Near-Data Processing of Big Data Workloads," in *Proc. ISCA*, jun 2016.
[19] S. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," *Proc. DAMON*, 2015.
[20] O. O. Babarinsa and S. Idreos, "JAFAR : Near-Data Processing for Databases," 2015.
[21] I. Jo, D.-h. Bae, and e. al., "YourSQL : A High-Performance Database System Leveraging In-Storage Computing," in *Proc. VLDB*, 2016.
[22] S. Kim, S.-W. Lee, B. Moon, and et al., "In-storage processing of database scans and joins," *Inf. Sci.*, 2016.
[23] J. Kim and et al., "Papyruskv: A high-performance parallel key-value store for distributed nvm architectures," in *Proc. SC*, 2017.
[24] A. De, M. Gokhale, S. Swanson, and e. al, "Minerva: Accelerating data analysis in next-generation ssds," in *Proc. FCCM*, 2013.
[25] M. Hemmatpour, M. Sadoghi, and et al., "Kanzi: A distributed, in-memory key-value store," in *Proc. Middleware*, 2016.
[26] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent Distributed Storage," in *Proc. VLDB*, 2017.
[27] S.-w. J. Ming, Arvind, and et al., "BlueDBM: An Appliance for Big Data Analytics," *Proc. ISCA*, 2015.