TOBIAS VINÇON

DATA-INTENSIVE SYSTEMS ON MODERN HARDWARE

TECHNISCHE
UNIVERSITÄT
DARMSTADT

DATA-INTENSIVE SYSTEMS ON MODERN HARDWARE

LEVERAGING NEAR-DATA PROCESSING TO COUNTER THE GROWTH OF DATA

Doctoral thesis
by Tobias Vinçon, M.Sc.
from Leinfelden, Germany

submitted in fulfilment of the requirements for the
degree of Doctor of Engineering (Dr.-Ing.)

Computer Science Department
Technische Universität Darmstadt

Reviewers
Prof. Dr-Ing. Andreas Koch
Prof. Dr. Jens Teubner

Further Supervisors
Prof. Dr-Ing. Ilia Petrov

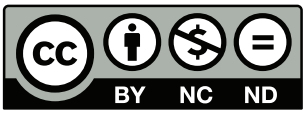
Date of the oral exam
December 8, 2022

Darmstadt, 2022

Tobias Vinçon: *Data-intensive Systems on Modern Hardware, Leveraging Near-Data Processing to Counter the Growth of Data*
Technische Universität Darmstadt
Data of the oral exam: December 8, 2022

Please cite this work as:
URN: urn:nbn:de:tuda-tuprints-230162
URL: https://tuprints.ulb.tu-darmstadt.de

This document is provided by TUprints,
The publication Service of the Technische Universität Darmstadt
https://tuprints.ulb.tu-darmstadt.de



ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, October 21, 2022

Tobias Vinçon

ABSTRACT

Over the last decades, a tremendous change toward using information technology in almost every daily routine of our lives can be perceived in our society, entailing an incredible growth of data collected day-by-day on Web, IoT, and AI applications.

At the same time, magneto-mechanical HDDs are being replaced by semiconductor storage such as SSDs, equipped with modern Non-Volatile Memories, like Flash, which yield significantly faster access latencies and higher levels of parallelism. Likewise, the execution speed of processing units increased considerably as nowadays server architectures comprise up to multiple hundreds of independently working CPU cores along with a variety of specialized computing co-processors such as GPUs or FPGAs.

However, the burden of moving the continuously growing data to the best fitting processing unit is inherently linked to today's computer architecture that is based on the *data-to-code* paradigm. In the light of Amdahl's Law, this leads to the conclusion that even with today's powerful processing units, the speedup of systems is limited since the fraction of parallel work is largely I/O-bound.

Therefore, throughout this cumulative dissertation, we investigate the paradigm shift toward *code-to-data*, formally known as Near-Data Processing (NDP), which relieves the contention on the I/O bus by offloading processing to intelligent computational storage devices, where the data is originally located.

Firstly, we identified *Native Storage Management* as the essential foundation for NDP due to its direct control of physical storage management within the database. Upon this, the interface is extended to propagate address mapping information and to invoke NDP functionality on the storage device. As the former can become very large, we introduce *Physical Page Pointers* as one novel NDP abstraction for self-contained immutable database objects.

Secondly, the on-device navigation and interpretation of data are elaborated. Therefore, we introduce cross-layer *Parsers and Accessors* as another NDP abstraction that can be executed on the heterogeneous processing capabilities of modern computational storage devices. Thereby, the compute placement and resource configuration per NDP request is identified as a major performance criteria. Our experimental evaluation shows an improvement in the execution durations of 1.4× to 2.7× compared to traditional systems. Moreover, we propose a framework for the automatic generation of *Parsers and Accessors* on FPGAs to ease their application in NDP.

Thirdly, we investigate the interplay of NDP and modern workload characteristics like HTAP. Therefore, we present different offloading models and focus on an *intervention-free execution*. By propagating the *Shared State* with the latest modifications of the database to the computational storage device, it is able to process data with transactional guarantees. Thus, we achieve to extend the design space of HTAP with NDP by providing a solution that optimizes for performance isolation, data freshness, and the reduction of data transfers. In contrast to traditional systems, we experience no significant drop in performance when an OLAP query is invoked but a steady and 30% faster throughput.

Lastly, *in-situ result-set management and consumption* as well as *NDP pipelines* are proposed to achieve flexibility in processing data on heterogeneous hardware. As those produce final and intermediary results, we continue investigating their management and identified that an on-device materialization comes at a low cost but enables novel consumption modes and reuse semantics. Thereby, we achieve significant performance improvements of up to 400$\times$ by reusing once materialized results multiple times.

ZUSAMMENFASSUNG

Über die letzten Jahrzehnte erkennt man in dem täglichen Leben unserer Gesellschaft eine gewaltige Veränderung hin zu der Nutzung von Informationstechnologie, welche einen enormen Anstieg an gesammelten Daten aus Web, IoT und KI Applikationen nach sich zieht.

Zeitgleich werden magnet-mechanische HDDs durch Halbleiterspeicher wie SSDs ersetzt, welche durch ihren modernen nicht-flüchtigen Speicher signifikant schnellere Zugriffszeiten als auch einen höheren Parallelismus aufweisen. Ebenfalls steigt die Ausführungsgeschwindigkeit der Recheneinheiten erheblich, da heutige Serverarchitekturen bis zu mehreren Hundert unabhängig arbeitenden CPU Kernen zusammen mit einer Vielzahl an spezialisierten Co-Prozessoren wie GPUs oder FPGAs aufweisen.

Allerdings ist die Belastung durch die Übertragung der stetig wachsenden Daten hin zu den bestmöglichen Verarbeitungseinheiten inhärent an die heutige Computerarchitektur gekoppelt, welche auf dem Prinzip *Daten-bei-Verarbeitungseinheit* basiert. Unter Berücksichtigung des Prinzips von Amdahl lässt sich schlussfolgern, dass selbst mit den heutigen leistungsstarken Recheneinheiten eine Beschleunigung der Systeme limitiert ist, da der Anteil an zu parallelisierender Arbeit stark an I/O gebunden ist.

Im Rahmen dieser kumulativen Dissertation untersuchen wir daher den Wechsel zum *Verarbeitungseinheit-zu-Daten* Prinzip, bekannt als Near-Data Processing (NDP), welches die Belastung vom I/O Bus nimmt, indem die Verarbeitung auf intelligente rechengestützte Speichergeräte ausgelagert wird, wo sich die Daten ursprünglich befinden.

Zunächst identifizieren wir *Native Storage Management* als essenzieller Grundbaustein für NDP, da es direkt den physikalischen Speicherbereich zentral innerhalb der Datenbank verwaltet. Darauf aufbauend wird die Schnittstelle zum Versenden von Adressverwaltungsinformationen und um Aufrufmöglichkeiten von NDP Funktionalitäten erweitert. Da das Erstere sehr groß werden kann, führen wir *Physical Page Pointers* als eine neuartige NDP Abstraktion für abgeschlossene, unveränderbare Datenbankobjekte ein.

Zum Zweiten wird die Navigation und Interpretation von Daten auf dem Gerät ausgearbeitet. Dabei führen wir schichtübergreifende *Parsers und Accessors* als eine weitere NDP Abstraktion ein, welche auf den heterogenen Verarbeitungsmöglichkeiten von modernen rechenfähigen Speichergeräten ausgeführt werden können. Dadurch ist die Platzierung der Verarbeitung und die Konfiguration der Ressourcen pro NDP Anfrage als ein wesentliches Leistungskriterium identifiziert worden. Unsere experimentelle Evaluierung ergibt eine Verbesserung der Ausführungszeit von $1,4\times$ bis zu $2,7\times$ verglichen mit traditio-

nellen Systemen. Darüber hinaus schlagen wir ein Framework zur automatischen Generierung von *Parsers und Accessors* für FPGAs vor, um deren Anwendung in NDP zu vereinfachen.

Zum Dritten untersuchen wir das Zusammenspiel von NDP und modernen Arbeitslastcharakteristiken wie HTAP. Dafür präsentieren wir verschiedene Auslagerungsmodelle und fokussieren auf eine *ununterbrochene Ausführung*. Durch das Verschicken des *Shared State* mit den jüngsten Änderungen der Datenbank an das rechengestützte Speichergerät ist dieses in der Lage, Daten mit transaktionalen Garantien zu verarbeiten. Somit erreichen wir auch eine Erweiterung des HTAP Gestaltungsraums durch NDP, indem wir eine Lösung aufzeigen, die sowohl auf Leistungsisolation, Neuheitswert der Daten als auch die Reduktion von Datenübertragungen optimiert. Im Gegensatz zu traditionellen Systemen erleben wir keinen signifikanten Leistungsrückgang, wenn eine OLAP Anfrage aufgerufen wird, sondern einen stabilen und 30% schnelleren Durchsatz.

Zuletzt schlagen wir *in-situ Ergebnisverwaltung und -Konsum* sowie *NDP Pipelines* zur flexiblen Datenverarbeitung auf heterogener Hardware vor. Da diese End- als auch Zwischenergebnisse produzieren, untersuchen wir deren Verwaltung und identifizieren, dass eine Materialisierung auf dem Gerät mit nur geringen Kosten verknüpft ist, aber neuartige Konsummodelle und Semantiken in der Wiederverwendung ermöglicht. Durch die mehrfache Wiederverwendung von einmal materialisierten Ergebnisse erreichen wir eine signifikante Leistungsverbesserung von bis zu $400\times$.

PUBLICATIONS

The following publications are a substantial part of this cumulative dissertation and are included in Chapter 8 to Chapter 16:

P1 [73] – Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vincon, and Christian Riegger. "Native Storage Techniques for Data Management." In: *Proc. ICDE* (2019) — Chapter 8

P2 [92] – 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 — Chapter 9

P3 [100] – Tobias Vinçon, Sergey Hardock, Christian Riegger, Andreas Koch, and Ilia Petrov. "nativeNDP: Processing Big Data Analytics on Native Storage Nodes." In: 2019 — Chapter 10

P4 [93] – Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *Proc. HardBD @ ICDE 2020*. 2020 — Chapter 11

P5 [98] – Tobias Vincon, Lukas Weber, Arthur Bernhardt, Andreas Koch, and Ilia Petrov. "nKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proc. DaMoN*. 2020 — Chapter 12

P6 [99] – Tobias Vincon, Lukas Weber, Arthur Bernhardt, Christian Riegger, Sergey Hardock, Christian Knoedler, Florian Stock, Leonardo Solis-Vasquez, Sajjad Tamimi, Andreas Koch, and Ilia Petrov. "nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing." In: *PVLDB* 12 (2020) — Chapter 13

P7 [101] – Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Tobias Vincon, Christian Knoedler, Arthur Bernhardt, Ilia Petrov, and Andreas Koch. "A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems." In: *Proc. RAW@IPDPS* (2021) — Chapter 14

P8 [95] – Tobias Vincon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads." In: *PVLDB* 15 (2022) — Chapter 15

P9 [94] – Tobias Vincon, Christian Knödler, Arthur Bernhardt, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch, and Ilia Petrov. "Result-Set Management for NDP Operations on Smart Storage." In: *Proc. DaMoN*. 2022 — Chapter 16

In addition, the author of this thesis has contributed to the following peer-reviewed publications:

P10 [17] – Justus Bogner, Carolin Dehner, Tobias Vincon, and Ilia Petrov. "Real time charging database benchmarking." In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2015, Brussels, Belgium, December 11-13, 2015*. Ed. by Gabriele Anderst-Kotsis and Maria Indrawan-Santiago. ACM, 2015, p. 78. ISBN: 978-1-4503-3491-4. DOI: 10.1145/2837185.2837258. URL: http://doi.acm.org/10.1145/2837185.2837258

P11 [96] – Tobias Vincon and Ilia Petrov. "Near Data Processing within Column-Oriented DBMSs for High Performance Analysis." In: June 2016. DOI: 10.13140/RG.2.1.1596.5687

P12 [97] – Tobias Vincon, Ilia Petrov, and Christian Thies. "cIPT: Shift of Image Processing Technologies to Column-Oriented Databases." In: *New Trends in Databases and Information Systems*. Springer International Publishing, 2016. ISBN: 978-3-319-44066-8

P13 [80] – Christian Riegger, Tobias Vincon, and Ilia Petrov. "Write-Optimized Indexing with Partitioned b-Trees." In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS '17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 296–300. ISBN: 9781450352994. DOI: 10.1145/3151759.3151814. URL: https://doi.org/10.1145/3151759.3151814

P14 [79] – Christian Riegger, Tobias Vincon, and Ilia Petrov. "Multi-Version Indexing and Modern Hardware Technologies: A Survey of Present Indexing Approaches." In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS '17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 266–275. ISBN: 9781450352994. DOI: 10.1145/3151759.3151779. URL: https://doi.org/10.1145/3151759.3151779

P15 [81] – Christian Riegger, Tobias Vincon, and Ilia Petrov. "Efficient Data and Indexing Structure for Blockchains in Enterprise Systems." In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS2018. Yogyakarta, Indonesia: Association for Computing Machinery, 2018, pp. 173–182. ISBN: 9781450364799. DOI: 10.1145/3282373.3282402. URL: https://doi.org/10.1145/3282373.3282402

P16 [43] – Sergey Hardock, Andreas Koch, Tobias Vincon, and Ilia Petrov. "IPA-IDX: In-Place Appends for B-Tree Indices." In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. DaMoN'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450368018. DOI: 10.1145/3329785.3329929. URL: https://doi.org/10.1145/3329785.3329929

P17 [82] – Christian Riegger, Tobias Vincon, and Ilia Petrov. "Indexing Large Updatable Datasets in Multi-Version Database Management Systems." In: *Proceedings of the 23rd International Database Applications and Engineering Symposium*. IDEAS '19. Athens, Greece: Association for Computing Machinery, 2019. ISBN: 9781450362498. DOI: 10.1145/3331076.3331118. URL: https://doi.org/10.1145/3331076.3331118

P18 [83] – Christian Riegger, Tobias Vinçon, Robert Gottstein, and Ilia Petrov. "MV-PBT: Multi-version indexing for large datasets and HTap workloads." In: *Adv. Database Technol. - EDBT*. Vol. 2020-March. 2020, pp. 217–228. ISBN: 9783893180837

P19 [58] – Christian Knoedler, Tobias Vincon, Arthur Bernhardt, Lukas Weber, Leonardo Solis-Vasquez, Ilia Petrov, and Andreas Koch. "A cost model for NDP-aware query optimization for KV-stores." In: *Proc. DAMON* (2021)

P20 [13] – Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Carsten Heinz, Christian Knoedler Tobias Vinçon, Andreas Koch, and Ilia Petrov. "neoDBMS: In-situ Snapshots for Multi-Version DBMS on Native Computational Storage." In: *Proc. ICDE* (2022)

P21 [14] – Arthur Bernhardt, Sajjad Tamimi, Florian Stock, Andreas Koch, Tobias Vincon, and Ilia Petrov. "Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage." In: *Proc. EDBT*. 2022

Knowledge is in the end based on acknowledgment.

— Ludwig Wittgenstein, 1969 in [104]

ACKNOWLEDGMENTS

As Ludwig Wittgenstein already stated in 1969, knowledge is in the end based on acknowledgment. Hence, I would like to thank all people and institutions that have contributed to the success of the publications, which are part of this present cumulative dissertation, and beyond. Furthermore, I would like to emphasize my gratitude toward a dedicated set of people who supported me extraordinarily along my journey.

First and foremost, I would like to mention my two PhD advisors Prof. Dr.-Ing. Ilia Petrov and Prof. Dr.-Ing. Andreas Koch, who gave me guidance in all regards throughout my entire PhD. Beginning with my Master's thesis, Prof. Dr.-Ing. Ilia Petrov assisted and advised me in an early stage in the area of database architectures and data-intensive systems. Thereby, he arose my interest in academic research and introduced me to all necessary methods and concepts to discover, elaborate, and publish scientific findings about Near-data processing. Prof. Dr.-Ing. Andreas Koch supported me in all kinds of questions regarding modern hardware aspects. His nearly unlimited knowledge about computer architectures and hardware-specific properties allowed me to investigate concepts properly.

Moreover, I would like to thank the supporting institutions: HAW Promotion, Herman Hollerith Zentrum, Hochschule Reutlingen, and Technical University of Darmstadt. In particular, my thanks goes to the other PhD students and Post-Docs for their extensive support in the publications, discussions, and unavoidable night shifts before some submissions.

Even though this thesis cannot be categorized as "industrial", I would like to speak out my gratitude to all colleagues and managers of Hewlett Packard Enterprise (HPE), DXC Technology, and Bosch for having confidence in my self-organization and the open discussions, while I worked part-time. Special thanks goes to Bernd Brennenstuhl, and his successor Daniel Kroell, from the HPE DualStudy program for initiating an industrial PhD program as well as providing financial support.

Lastly, my sincere gratitude goes to all my close friends and family that gave me moral support during the time of the PhD and motivated me in not getting discouraged by any backlash. Primarily, I'm incredibly grateful to my own little family and especially my wife, Denise, on whom I always could rely, who brought me joy along the entire journey, and always kept me motivated, even after rejections.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI Artificial Intelligence
ALU Arithmetic Logic Unit
ASIC Application-specific Integrated Circuit
BC Betweenness Centrality
BRAM Block Random Access Memory
CCIX Cache Coherent Interconnect for Accelerators
CPU Central Processing Unit
CXL Compute Express Link
DBMS Database Management System
DMA Direct Memory Access
DRAM Dynamic Random Access Memory
DSP Digital Signal Processor
FLOPS Floating Point Operations Per Second
FPGA Field Programmable Gate Array
FTL Flash Translation Layer
GC Garbage Collector

GPU Graphics Processing Unit
HBM High Bandwidth Memory
HDD Hard Disk Drive
HDL Hardware Description Language
HLS High-Level Synthesis
HTAP Hybrid Transactional/Analytical Processing
IDC Industrial Development Corporation
ISA Instruction Set Architecture
IoT Internet of Things
LBA Logical Block Address
LSM Log-Structured Merge
LUT LookUp Table
MIMD Multiple Instructions, Multiple Data
ML Machine Learning
NDP Near-Data Processing
NVMe NVM express
NVM Non-Volatile Memory
OLAP Online Analytical Processing
OLTP Online Transactional Processing
PBA Physical Block Address
PCB Printed Circuit Board
PCIe Peripheral Component Interconnect express
PCM Phase Change Memory
PPP Physical Page Pointer
RDMA Remote Direct Memory Access
RTL Register Transfer Level
SAS Serial Attached SCSI
SATA Serial Advanced Technology Attachment
SIMD Single Instruction, Multiple Data
SISD Single Instruction, Single Data
SSD Solid State Drive
SST Sorted String Table
STT-MRAM Spin-Transfer Torque Magnetoresistive RAM
SoC System on Chip

Part I

SYNOPSIS

1 INTRODUCTION

Over the last decades, a tremendous change toward using information technology in almost every daily routine of our lives can be perceived in our society. Together, the amount of data collected day-by-day on Web, Internet of Things (IoT), and Artificial Intelligence (AI) applications continuously grow with an ever-increasing speed, almost surpassing the exponential increase of Jim Gray's investigations [90] in 2006. At the same time, the need to derive insights from those data collections about customer behavior or analyze the performance of processes and products became indispensable to the classical transactional processing in such systems.

At the same time, the hardware of today's computer architectures is subject to rapid changes in technologies, properties, and performance characteristics. For instance, advances in the semiconductor industry have created storage technologies like Flash and Non-Volatile Memory (NVM) that are capable of: (1) persisting and accessing data at rates at least 100 to 10^5 times faster than classical mechanical storage media and (2) providing high levels of parallelism. With a high yield in manufacturing and low production costs, such storage technologies provide an economical way of storing Tera to Petabytes of data in contrast to in-memory solutions. Similarly, a variety of novel processing units besides classical Central Processing Units (CPUs) have become widespread in today's server architectures and cloud offerings. While the number of CPU cores in a single host system can easily reach multiple hundreds (4 to 8 sockets with 72 Cores for x86, or 128 ARM Cores), Field Programmable Gate Arrays (FPGAs) provide the ability to build custom hardware that is partially reconfigurable during runtime. With the latter, even Multiple Instructions, Multiple Data (MIMD) can be facilitated and flexible, scalable, and elastic operation pipelines enable new possibilities to process data.

However, the burden of moving data toward the best fitting processing unit is inherently linked to today's computer architecture that is based on the `data-to-code` paradigm. Thus, the continuously growing data has to be transferred from the storage to the compute to be processed there. Consequently, today's systems have several drawbacks. Firstly, the on-device bandwidth of the interconnected storage chips (e. g., Flash) cannot be fully-leveraged despite the fact that it is an order of magnitude higher than the bus between host and device. Secondly, the high clock frequency and parallel processing capabilities of modern CPUs cannot perform to their full potential as data movement entails I/O bus contention and stalling. Thirdly, with common hybrid

workloads, such data transfers cause also a high buffer pollution in the memory of the host system as processed datasets usually exceed their resource limits. Especially analytical queries process both, the latest modifications as hot data as well as the very large cold portion of the storage devices. Lastly, the overall power consumption of a system suffers from either the high amount of data movement or oversized in-memory configurations.

Nevertheless, by shifting the system paradigm toward `code-to-data`, i. e., pushing the processing as close as possible to the storage where the data is located, several benefits can be achieved. Firstly, the already available on-device hardware characteristics can be fully leveraged e. g., significantly higher on-device bandwidths and parallelism. Secondly, storage devices based on processing units of relatively low computing capabilities, such as embedded platforms equipped with simple ARM-Cores, can be utilized to perform simple filtering and calculations to reduce data movement, and thus, relieve the I/O bus contention. Thirdly, also fully-fledged processing pipelines can be offloaded to the storage device to gain significant performance benefits by utilizing the full set of available heterogeneous compute capabilities. Lastly, those aspects will not only improve the robustness of data-intensive systems but also provide an economically friendly way of processing large amounts of data, especially as storage like Flash is considerably cheaper than main memory.

While the fundamental concepts of Near-Data Processing (NDP) have already been studied in the early 80s, its revision is being mainly triggered by the advances in semiconductor technology. Contrary to early approaches, nowadays manufacturing processes allow to economically place processing units close to storage technologies, establishing modern intelligent computational storage devices. These reach from simple Solid State Drives (SSDs) with lightweight ARM processors up to enterprise-grade hardware equipped with large NVMs and several processing units such as CPUs, Graphics Processing Units (GPUs), or FPGAs.

Within this work, we discuss the beforementioned key aspects of NDP for data-intensive systems on modern computational storage devices. The following list presents the main contributions:

OVERVIEW We investigate and provide an in-depth overview of the relevant aspects of NDP in the context of data-intensive systems and modern hardware.
See Contributions: C1.1 C2.1 C2.3 C3.1 C4.1

CONCEPTS & ABSTRACTIONS We propose novel concepts, abstractions, and interfaces for shifting processing from the host to a computational storage device.
See Contributions: C1.2 C1.3 C2.2 C2.4 C3.3 C4.2

OPPORTUNITIES We propose new data processing opportunities like transactionally consistent NDP pipelines in Hybrid Transactional/Analytical Processing (HTAP) workloads or in-situ reuse of intermediary results on computational storage devices with heterogeneous processing units.
See Contributions: C2.5 C3.2 C4.3 C4.4

EFFECTS We evaluate and demonstrate the effect of NDP on the performance, robustness, and power consumption of a data-intensive system.
See Contributions: C2.2 C2.4 C3.4 C4.3 C4.5

As outlined in Figure 1.1, we discuss the challenge in-depth, give background information, and introduce the Central Research Question CRQ in the next Chapter 2, followed by elaborating why physical storage management is essential for efficient NDP in Chapter 3. Thereby, we introduce proper abstractions and interface definitions according to the Research Question RQ1. Likewise, Part ii provides insights on those aspects on basis of the Publications P1 P2 P3 of this cumulative dissertation.

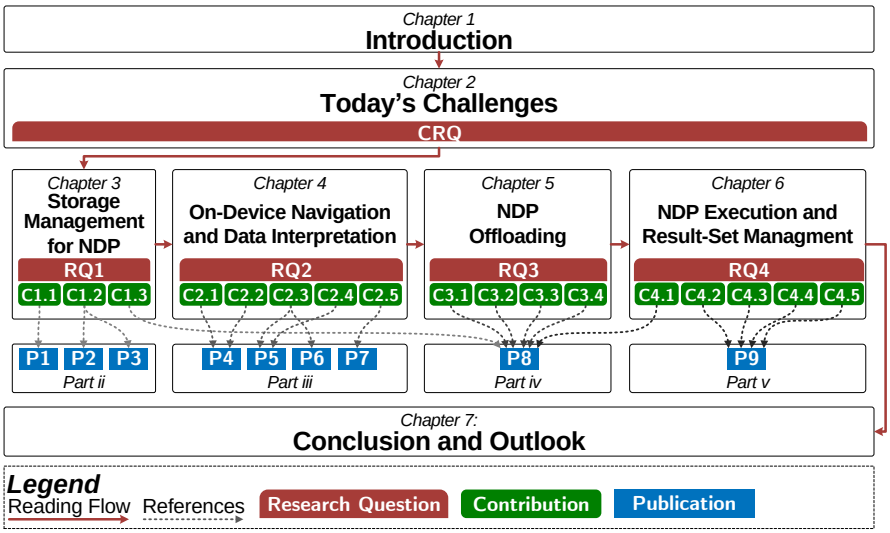


Figure 1.1: Structure and reading flow of this thesis.

Chapter 4 continuous to utilize these interfaces and abstractions and explores on-device navigation and data interpretation (see Research Question RQ2). Thereby, the concept of Parsers and Accessors as well as the usage of heterogeneous hardware for NDP are discussed, supported by Publications P4 P5 P6 P7 in Part iii.

Novel interaction and offloading models for NDP are evaluated in the Research Question RQ3 . Thus, Chapter 5 and Publication P8 of Part iv discuss different types of NDP offloading and propose the Shared State as a concept for supporting transactionally consistent NDP executions in Database Management Systems (DBMSs).

Lastly, we look into the NDP execution and result-set management in Chapter 6. Various ways of executing NDP are presented before we discuss NDP pipelines in-depth. Moreover, final and intermediary result-set handling is evaluated and an NDP finite state machine is proposed. Substantiated by the Publication P9 of Part v, the final Research Question RQ4 is examined. We conclude this thesis in Chapter 7 by summarizing the beforementioned contributions and exploring the problem space of NDP for data-intensive systems to give an outlook on future work.

2 TODAY'S CHALLENGES

In Chapter 1 we motivated the shift toward NDP and provided a first glimpse into the main drivers and expectable benefits. In this part, a detailed investigation of today's challenges is provided by first, discussing Amdahl's Law in light of the steady data growth in Section 2.1. Secondly, Section 2.2 describes how modern hardware can be employed to solve the challenge with NDP and gives background information about it. Moreover, governing trends in data-intensive systems that affect NDP are outlined in Section 2.3 which leads to the Central Research Question CRQ in Section 2.4.

2.1 THE NEED TO CHANGE SYSTEM PARADIGMS

Today's society creates impressive amounts of data. With the advent of the internet and especially the mobile phone, web, and social media platforms have become part of our daily lives. Nowadays, these collect data from their users in volumes of multiple thousand Giga- or even Petabytes per day. Widely-known examples are Facebook (500+ TB data created per day in 2012 [24]), Instagram (95 million shared photos and videos per day in 2021 [65]), Twitter (500 million tweets per day in 2013 [16]), and Youtube (500 hours of videos uploaded per minute in 2020 [89]). In addition, IoT applications became mainstream with connected smart devices such as building technology, e-Bikes, vacuum robots, mowers, or simple light blubs, to just name a few. Equipped with various sensors, they produce tons of data stored in cloud infrastructures, ready for being analyzed. Likewise, advances in camera and satellite technology improve the resolution and recording frequency of images e. g., in astronomy, and thus, increase the rate at which data is growing. According to the Industrial Development Corporation (IDC) and [49], the overall worldwide creation and processing of data comprise up to 97 Zettabytes in 2022.

> *The amount of scientific data is doubling every year.*
>
> — Alex Szalay and Jim Gray, 2006 in [90]

Already in 2006, Jim Gray and Alex Szalay perceived this rapidly growing amount of data and predicted an exponential growth for the future [90] as demonstrated in Figure 2.1. Today, data growth is subject to various research [23, 46, 63], and even though a large portion of the data volumes mentioned in those statistics may only be transient, it is reasonable to assume that just the persisted volume grows exponentially today.

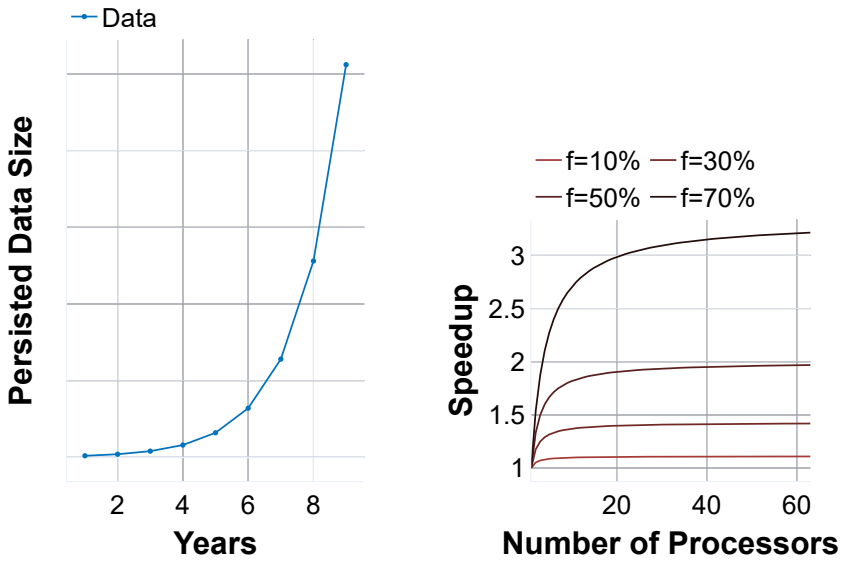


Figure 2.1: While the data size is continuously growing at an exponential rate (left), its processing is limited due to Amdahl's Law (right). The speed-up heavily depends on the fraction of time that the improved part is actually used but most of the time is spent on moving the data to any processing unit.

To cope with the increasing data volumes, the semiconductor industry firstly improved the performance of its processors by increasing the clock frequency [84]. For example, the clock frequency of CPUs improved from less than 0.5 GHz in the early 90s up to around 4 GHz in 2004, when they reached their peak. Unfortunately, with higher clock frequency, the heat development and power consumption increase similarly. Heat dissipation became one major issue and the relation between processing speed and power consumption was not economically beneficial anymore. Consequently, the semiconductor industry moved away from its designs of single-processor models with high cycle rates per second but rather focused on combining multiple lower-clocked processing units that work in parallel. By distributing the work across these cores, it is possible to significantly boost the number of performed instructions per second without jeopardizing the heat dissipation.

Today's manufacturing processes allow CPUs to comprise per socket up 72 cores in models of the x86 family [50] and even 128 cores in ARM-based models [25]. Multiplied by the number of sockets in todays server architectures, a single host instance can easily reach 512 cores. Moreover, GPUs have recently entered the data centers as highly parallel processing units. With their Single Instruction, Multiple Data (SIMD) instructions, they are specialized for algorithms that process large data blocks in parallel and achieve more than 10^{14} Floating Point Operations Per Second (FLOPS) [69].

Yet, such processing units imply that the data processing can be partitioned evenly across the cores to avoid any bottlenecks in a single core. This leads us to one of Amdahl's laws for balanced systems [4, 47] which is also seen as a basic rule of thumb in data engineering [39]. It states that by parallelizing a single part of the work, the overall performance improvement is limited by the fraction of time that the improved part is actually used (see Equation 2.1).

$$\begin{aligned} S &= \frac{1}{(1-f) + \frac{f}{k}} \\ \text{where } f &= \text{fraction of work in faster mode,} \\ k &= \text{speedup while in faster mode or number of cores} \end{aligned} \tag{2.1}$$

In Figure 2.1, we show sample speedups in relation to the number of cores for various factions f according to Amdahl's law. Depending on the parallel portion, the speedup increases significantly with 4 to 32 cores but stagnates beyond this number of parallelism approaching asymptotically a theoretical limit. However, perfectly parallelizing an algorithm is almost impossible in practice and can often only be applied in theoretical considerations. But even more importantly, with the exponential increase in data [90] described above, we cannot expect to have the entire data already in the memory or caches of the processing units anymore, but rather require to read it from persistent storage devices such as SSDs. Consequently, the time spent for I/O is also included in f.

Key Insight: The theoretical speedup of Amdahl's law is hurt by the I/O boundness of today's data-intensive systems.

However, most of the time, data-intensive systems become I/O-bound due to the scarce bandwidth of today's interconnects and transfer protocols. For instance, with Serial Advanced Technology Attachment (SATA) - 0.6 GB/s, Serial Attached SCSI (SAS) - 1.5 GB/s, or Peripheral Component Interconnect express (PCIe) - 32 GB/s, the data cannot be moved to the powerful processing units as fast as they can execute operations. As a consequence, the processing cores are blocked by the I/O and have to wait for new data to continue processing. In the end, such stalling has a negative impact on f, and thus, hurts the speedup of the entire system.

To sum up, even though modern processing units can process data with incredible levels of parallelism, the constant increase in the volume of data requires today's computer architectures to store it on storage devices and fetch it prior to processing. Thus, systems have become I/O-bound, harming the theoretical speedup for parallel processing units as described by Amdahl [4].

2.2 THE SHIFT TOWARD NEAR-DATA-PROCESSING

To date, the majority of computer architectures embrace a `data-to-code` paradigm that requires transferring all data from the different levels of

the storage hierarchy over bus systems to one of the processing units. Only after the data is available, instructions can be performed on it. As discussed in Section 2.1, such a paradigm can profit from the highly parallel processing units as long as most of the data is already close to them, e. g., using in-memory computing. And even in such cases, today's data-intensive systems struggle with the memory wall [107]. Additionally, with the continuously increasing data volumes, most of those systems cannot expect to have most of the dataset in memory without immense investments in expensive memory.

However, instead of moving all data from the storage device to the processing units, one can also shift the paradigm toward `code-to-data`, formally known as Near-Data Processing (NDP), or in-situ processing. Thus, the key idea is to offload processing to a compute unit that is as close as possible to the storage where the data is physically located and circumvent the burden of costly data transfers. While this paradigm is also applicable in main memory systems to circumvent the Von Neumann bottleneck [10], this thesis considers mainly the connection to and data transfers from mass storage.

Already in the early 80s and 90s, the concept of in-situ processing was investigated and can be traced back to David DeWitt who proposed database machines [18, 30]. Experimenting with magneto-mechanical storage devices, processors were equipped on either the tracks or the moving heads. Thus, for the first time, processing could be directly performed on mass storage and data movement be reduced. Unfortunately, several drawbacks were limiting factors: e. g., the low I/O bandwidth and parallelism at that time, and the high costs of manufacturing such proprietary hardware.

As this is not surprising, given the fact that those storage devices are limited by mechanical moving parts, the idea was adapted with advances in the semiconductor industry. First prominent approaches emerged in the late 1990s like ActiveDisc [1], IDISK [53], and Active storage/disk [78] followed by Smart SSDs [31, 86] in 2013.

Today, the semiconductor industry provides novel ways of combining powerful processing elements close to modern storage technologies with tremendously high bandwidths and parallelism. For instance, the on-device interconnect can nowadays achieve an aggregated data transfer rate of up to 2.4 GB/s for a single Flash chip [66]. Considering that multiple of those independent chips are used, we can expect a sustainable bandwidth of 19.2 GB/s and even 38.4 GB/s with 8 and 16 chips per module, respectively. But, as if that was not enough, modern computational storage devices can easily fit 2 [88] to 4 [75] of such modules on a single Printed Circuit Board (PCB), providing an overall bandwidth of 153.6 GB/s.

Likewise, FPGAs became popular besides the traditional CPUs and GPUs. Even though FPGAs support building elastic processing pipelines, their advantage is in providing low but also deterministic latencies

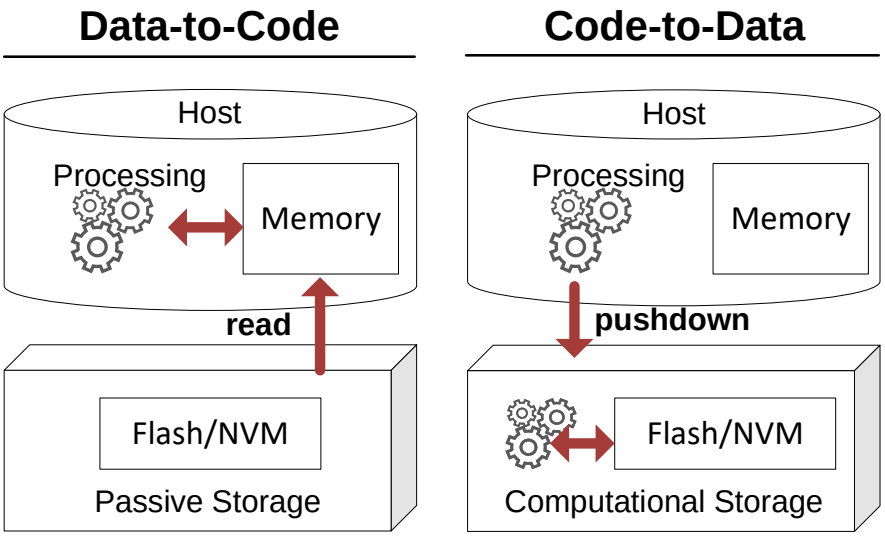


Figure 2.2: Near-data Processing entails a paradigm shift: From a data-to-code toward a code-to-data.

as well as low jitter. Once deployed, a hardware model continuously produces the same output in equal time, given that the input data and start conditions are similar. In contrast to fixed-function Application-specific Integrated Circuits (ASICs), FPGAs can be reprogrammed, sometimes even partially. Thus, deployed hardware models or portions of it can adapt to changing data formats or workloads. With almost a million LookUp Tables (LUTs), several Gb of Block Random Access Memory (BRAM) and multiple specialized logic elements like Digital Signal Processors (DSPs) or AI tiles, nowadays FPGAs are capable of performing highly complex algorithms.

Key Insight: Heterogeneous compute hardware, appropriate for different types of operations, is available on computational storage devices.

In addition, via modern 3D-Stacking, it became economically feasible to combine a variety of processing units on the same chip, namely System on Chip (SoC). These also include large caches or storage technologies such as High Bandwidth Memory (HBM). Additionally, such chips can be integrated on a PCB with mass storage technologies such as Flash, building the foundation for modern computational storage devices required for efficient NDP.

With those computational storage devices also a multitude of novel approaches, besides the one considered in this thesis, was proposed focussing on a variety of specific challenges, e. g., IBEX [105, 106], Minerva [29], Willow [86], BlueDBM [67], JAFAR [9, 108], Kanzi [45], ISP [56], YourSQL [52], Biscuit [41], PapyrusKV [55], DoppioDB [3, 87], Caribou [51], Batched Writes [32], BlockNDP [11], and PolarDB [22].

In general, nowadays computational storage devices enable a shift toward NDP. With various types of processing power close to storage, such devices are predestined to offload even complex processing pipelines (see Chapter 4). In this manner, they can leverage the on-device bandwidth which outperforms mass storage interconnect protocols like PCIe 3.0 x16 with 15.75 GB/s or PCIe 4.0 x16 with 31.5

Key Insight: Advances in the hardware enable sophisticated computational storage devices but their proper integration into data-intensive systems is subject to research.

GB/s. Consequently, NDP provides an efficient solution to either pre- or fully process the massive amounts of raw data collected nowadays, by limiting the data movement between the storage device and the host to only qualifying data, and thereby, relieving today's bus systems. Depending on the offloaded processing, this useful data equates to either: (a) the final results, (b) just a necessary subset of the data, or (c) the raw data transformed to improve later processing steps on the host. Moreover, NDP offers multiple new opportunities, as discussed in Chapter 5 and 6, in particular for data-intensive systems.

2.3 TRENDS AND FACTORS IN TODAY'S DATA-INTENSIVE SYSTEMS

In addition to the possibility offered by computational storage devices to shift processing closer to the storage where data is located, several trends and factors in database and system architectures can be perceived that affect the application of NDP in the context of this thesis.

2.3.1 *Workload*

Besides the rapid increase in data volume discussed before, the workload on data-intensive systems is also constantly evolving as applications change. Driven by trends from the web and mobile platforms, data comes in different shapes and sizes nowadays, reaching from simple key-value pairs, over graphs and relational models, to completely unstructured image and video recordings. Likewise, the update frequencies of data records increased significantly as more and more users operate on such platforms, putting pressure on today's concurrency controls.

Moreover, the urge to gain insights from operational data entails Hybrid Transactional/Analytical Processing (HTAP). Instead of having only highly frequent transactions that read, write, and modify the dataset as in classical Online Transactional Processing (OLTP) scenarios, HTAP combines them with long-running transactions from Online Analytical Processing (OLAP) workloads. In practice, such HTAP workloads can be found in various applications that require analyzing the data of a productive system in near real-time like social media networks or enterprise resource planning systems.

Key Insight: Modern workloads comprise OLTP and OLAP that compete for resources on the host.

Thereby, transactions are typically performed on a hot part of the data while OLAP queries usually process large parts of, or even the entire data, as shown in Figure 2.3. As a consequence, both workloads compete for resources such as buffers or processing units.

In Chapter 5 and Part iv we discuss how NDP improves the robustness in the execution of such HTAP workloads by offloading OLAP queries to the computational storage device.

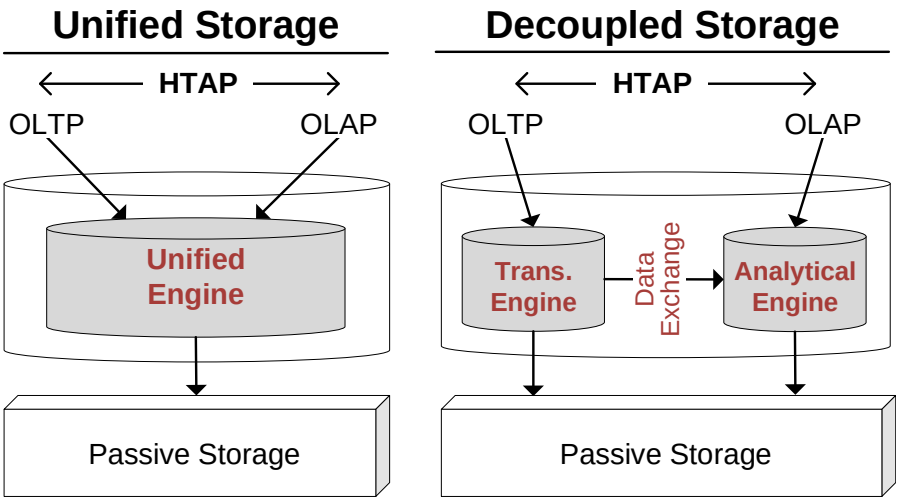


Figure 2.3: Today's architectures of data-intensive systems for HTAP can be divided into Unified Storage and Decoupled Storage Systems.

2.3.2 *Architecture*

In parallel to the evolving workloads, changes in database architectures are also investigated. For instance, in order to adequately perform HTAP workloads several systems were proposed in the industry [33, 36, 60, 61, 76] and academia [2, 5, 7, 8, 19, 40, 54, 57, 62, 64, 68, 77, 85].

These can be categorized into unified storage systems that introduce a single database engine to process both types of workloads (OLTP and OLAP), and decoupled storage systems that foresee one engine per workload operating on a single passive storage (see Figure 2.3). In the latter, recurring data exchange procedures transfer data from the OLTP to the OLAP engine to provide the freshest data in the analytical transactions. In this manner, the systems are optimized on different aspects such as data freshness, data consistency, data transfers, performance isolation, or memory pollution.

The extent to which NDP can be part of the solution space, and optimize in those aspects, is discussed in Chapter 5 as well as in the respective Publication P8.

2.3.3 *Abstractions and Interfaces*

Over the past years, several abstractions have been established to encapsulate a certain functionality or behavior within computer systems and especially in DBMS. Interfaces are responsible to define the way to interact with those abstractions. In particular, the access to storage devices is subject to multiple layers of abstractions and interfaces that have to be revised with NDP.

As presented in Figure 2.4, database architectures employ several general abstractions for their internal storage management. A database

could have multiple tablespaces for its segments like tables, indices, and others. Extents are used as entities of allocation whenever new space is required. In some cases, those extents are further subdivided into blocks.

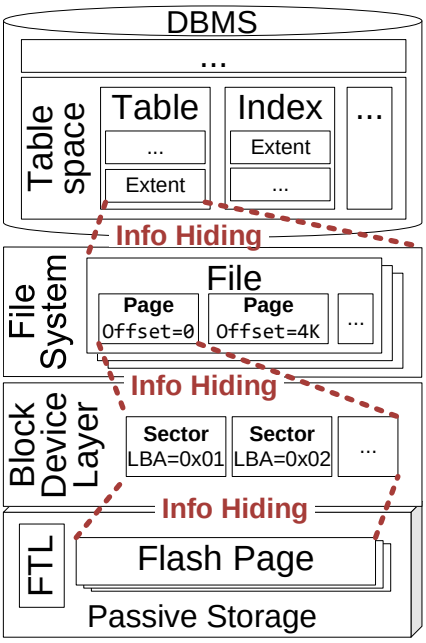


Figure 2.4: Databases organize their data in several abstractions like Tablespaces, Database Objects, and Extents. Those are further subdivided along the I/O Stack until they are physically stored on the storage device. Yet, every abstraction also introduces information hiding.

While those abstractions occur in slightly different variations for each DBMS, at some point they all rely on the commonly known files of operating systems. Files represent a virtually contiguous memory space that can be expanded almost without any limitations. Under the hood, file systems are responsible to manage those chunks of data, often referred to as pages. In most file systems, the size of such pages can be configured but usually comprise 4 KB. Within the block device layer of the operating system, these pages are subdivided into sectors of a size of 512 Bytes each and are assigned to Logical Block Addresses (LBAs). Depending on the interface of the storage device, these sectors are gathered and sent via a single request to the device. Once there, the Flash Translation Layer (FTL) takes over the responsibility to finally place the data in a physical location of the underlying storage technology, thereby mapping a Physical Block Address (PBA) to the LBA.

Similarly, the interface of a certain abstraction matches its given purpose, but hides the complexity of underlying structures. E.g., one can read, write, and modify files to any extent without worrying about the physical storage management of this data. While this opens the possibility to exchange certain layers of abstraction without considering the backward compatibility, this also masks essential properties of the underlying levels. For instance, the interface of files is consistent when changing from magneto-mechanical disks with cylinders, tracks, and sectors to Flash-based SSDs with dies, planes, and pages. Yet, as

already described in Section 2.2, NVMs like Flash have fundamentally different properties than magneto-mechanical disks.

Consequently, to fully leverage the properties of todays storage technologies, abstractions and interfaces have to adapt. In particular, when moving processing toward storage, such levels of abstraction become a serious issue as they introduce information hiding. In Chapter 3, we will focus on those aspects and provide solution proposals to enable NDP which can exploit the benefits of NVMs like high bandwidth and parallelism.

Key Insight: Abstractions and interfaces of today's I/O stack introduce information hiding that prevents direct processing on lower layers.

2.3.4 *Hardware*

In Section 2.2, we already described that FPGAs entered data centers recently. In contrast to CPUs and GPUs, they are not based on a classical Instruction Set Architecture (ISA) but on various building hardware blocks, such as LUTs, which are interconnected with reconfigurable wires to perform even complex operations. Consequently, the executable code is no longer any machine code with instructions that are processed by an Arithmetic Logic Unit (ALU) but rather a bitstream with a binary bit pattern to configure the FPGA architecture when being programmed.

While FPGAs provide a high potential to highly parallelize and accelerate processing logic, the development process for the required bitstreams is quite tedious. Most commonly, the first step is to define the Register Transfer Level (RTL) design using a Hardware Description Language (HDL) like VHDL or Verilog. Alternatively, modern design entry approaches such as High-Level Synthesis (HLS) can be used to model the behavior of an FPGA in a more abstract and productive manner from a developer's perspective. In both cases, the subsequent synthesis aims to optimize the logic according to several design constraints such as the FPGA area, clock frequency, power consumption, or reliability. In the final place and route step, the location and wiring of logic elements are created considering the layout limitation of the target FPGA.

Besides the complex design entry with HDLs, as well as the time-consuming synthesis, placing, and routing, the debugging and testing using simulations make developments even more time-consuming and error-prone. As a consequence, when utilizing FPGAs as accelerators in NDP, especially with changing workloads, novel ways of automation are required. In Chapter 4 we discuss first proposals of automation.

Key Insight: Even though FPGAs offer a promising potential to accelerate NDP, its development is cumbersome.

2.4 CENTRAL RESEARCH QUESTION

Considering all the background introduced here about the current system architecture along with its drawbacks regarding processing large ever-growing amounts of data with modern storage and process-

ing technologies, and the overview of today's trends and factors in data-intensive systems, the overarching Central Research Question of this work is:

CENTRAL RESEARCH QUESTION – CRQ

» *How can architectures of data-intensive systems leverage near-data processing to cope with today's hybrid workloads and increasing dataset size while addressing the challenges of today's computer architectures?* «

The following chapters of this thesis will investigate different aspects of an answer, and provide more profound details about the challenges, solution proposals, and contributions.

3 STORAGE MANAGEMENT FOR NDP

With modern computational storage devices, the technology for persisting data is also changing from magneto-mechanical moving disks to Non-Volatile Memorys (NVMs) like Flash. To reach the full potential of NDP, their physical characteristics, like access speed and patterns, parallelism, and interfaces must be leveraged.

RESEARCH QUESTION – RQ1

» *Which novel abstractions are necessary to extend the native storage interface for NDP?* «

Thus, we investigate the Research Question RQ1 by revisiting the physical storage management of modern storage technologies and setting it into the context of databases. Thereby, we elaborate interfaces toward and on the device itself. Moreover, necessary abstractions to issue and process NDP calls are established. Figure 3.1 shows the areas considered within a database as an architectural guidebook. The main contributions presented in this chapter are:

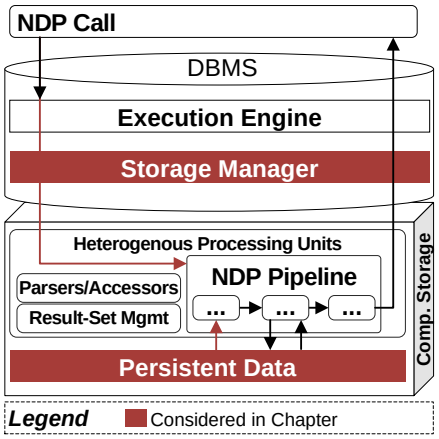


Figure 3.1: Architectural guidebook.

- C1.1 Elaboration of drawbacks of today's storage management and the introduction of physical storage management for NDP.
- C1.2 Proposal of an interface to shift processing to computational storage devices.
- C1.3 Introduction of necessary abstractions for NDP like Physical Page Pointers.

The corresponding publications are:

P1	[73] – "Native Storage Techniques for Data Management"	Chapter 8
P2	[92] – "NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management"	Chapter 9
P3	[100] – "nativeNDP: Processing Big Data Analytics on Native Storage Nodes"	Chapter 10
P8	[95] – "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads"	Chapter 15

3.1 PHYSICAL STORAGE MANAGEMENT IN DATABASES

For decades, industry and academia have been constantly trying to optimize data-intensive systems to fully leverage the bandwidth and parallelism of the storage tier. While some systems started to provide proprietary filesystems [71] or even operate on raw storage [48, 72] to interact with devices, most of today's data-intensive systems utilize the widely known abstraction of files as a foundation, which is also very common in almost every operating system. The latter has a crucial advantage, as files resemble an almost ever-expandable contiguous logical memory space that can be modified in place without any considerations. Thereby, they provide a flexible abstraction for databases while encapsulating the complexity of the underlying physical storage management. Moreover, this allows to seamlessly swap lower levels of the I/O stack, such as necessary kernel modules of the operating system, the bus interconnection to the device, or even the storage technology itself, without adapting the DBMS.

However, by swapping some of those layers, the expected behavior and performance are changing tremendously as the internal properties of those layers likewise differ. In particular storage technologies have been subject to a fundamental change over the last decades with the advances in the semiconductor industry. In Figure 3.2, we sketch a broad overview and comparison of the major integral parts of traditional Hard Disk Drives (HDDs) and modern SSDs or computational storage devices that are equipped with Flash as NVM.

HDDs comprise heads that read and write data from and to a track on one of the multiple fastly-rotating cylinders. Therefore, to access a specific sector, the cylinders have to be rotated to that certain section and the head has to be positioned to the associated track. Not surprisingly, the corresponding seek time to access random sectors heavily depends on the current position of the mechanical parts of the HDD and consequently, is significantly higher than accessing sequential sectors. Moreover, the internal parallelism is limited to the number of heads.

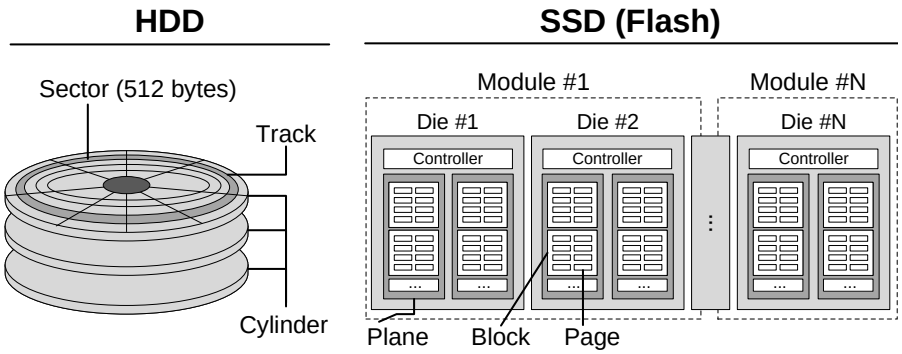


Figure 3.2: The composition of traditional HDDs differs significantly from those of SSDs. Similarly, access latencies and patterns, operation parallelism, and wear-out differ as well.

In contrast, NVMs of SSDs or computational storage devices, like Flash, consist of special semiconductors that can persistently store a specific state. Those semiconductors are packaged in dies that can be accessed independently of each other with their associated controller. Often, computational storage devices comprise even multiple modules (e. g., SO-DIMMs) that have 8 to 16 dies equipped [75, 88], resulting in a substantial on-device access parallelism. Internally, dies are further subdivided into planes that allow again some operations to be issued in parallel, also known as multi-plane operations. Lastly, within a plane, the pages as the smallest physically accessible unit are arranged in blocks. Accesses of the same operation take the same time, irrespectively whether the access pattern is random or sequential. However, NVMs differ from magneto-mechanical disks in having highly variable access latencies per operation. In particular, with SSDs, writing a page takes an order of magnitude longer than reading one but is still around 100× faster than HDDs. Another idiosyncrasy is that pages, once written, cannot be subject to another write operation until the block, they are belonging to, is erased. In consequence, all other valid pages of the same block are either erased or have to be copied to another block beforehand which entails a significant write amplification [42]. Unfortunately, the number of erase cycles per block is limited in such technologies as the semiconductors wear out. To provide block-device compatibility in that every page can be overwritten, but still expand the lifetime of modern SSDs, highly complex FTLs are responsible on the device for talking erase-before-overwrite, equalizing the block erase counts (wear-leveling) and managing the physical placement of the data and thereby, maintaining the logical-to-physical address mapping.

Key Insight: Modern storage technologies differ fundamentally from traditional HDDs and require a proper management.

In general, NVMs of SSDs provide significantly higher bandwidths and levels of parallelism in contrast to traditional HDDs. To take advantage of those benefits, their interface requires a careful data placement to issue operations on parallel operating units (e. g., dies and planes)

and to mitigate the wear-out by reducing block erases. Moreover, applications have to adapt to the operation-specific access latencies and storage granularities (e. g., blocks and pages) C1.1 .

Key Insight: Using a backward-compatible I/O stack on modern SSDs cannot leverage their actual performance.

But even with the latest FTLs, filesystems are not able to expose all of the beforementioned important properties of SSDs. Quite the contrary, precious on-device processing resources are employed to mimic backward-compatible interface abstractions like LBAs but could be utilized for reasonable NDP instead. Consequently, applications cannot fully leverage characteristics such as the bandwidth of the Flash chips. Even worse, with all these compatibility layers between the DBMS and the device, a significant write amplification of up to 19× can be observed [42, 92].

To counter this, modern approaches like Native Storage [42, 44, 73] provide the ability to directly issue storage commands like `PAGE_READ`, `PAGE_WRITE`, or `ERASE_BLOCK` against a physical entity (physical Page or Block) of the device. As a consequence, the database (1) has direct control over the storage without any intermediate layers of abstractions, (2) is able to integrate the properties of the underlying storage into the data placement, and (3) can adapt to workload characteristics e. g., for scheduling the Garbage Collector (GC). In Publication P1 we give an overview of the architectural approaches and techniques, describing how interfaces and abstractions have to be adapted according to the design of modern storage technologies, and how these can be integrated into database systems. In Publication P2 we present NoFTL-KV, an integration of Native Storage Management into the widely known RocksDB [74]. We investigated the concept of Regions [44] in the context of KV-Stores that are suitable for modern insert and update intensive workloads, represented by Linkbench [6] in the experimental evaluation. Overall, we improve the transactional throughput by 33% while reducing the response times of queries by up to 2.3× with NoFTL-KV. Moreover, with the sophisticated possibilities of Native Storage Management, to place data aligned with the internal database abstractions, the write-amplification is reduced to $\frac{1}{19}$th which, in fact, improves the endurance of SSDs C1.1 .

Key Insight: While several layers of the traditional I/O stack provide high flexibility, they also cause information hiding.

Besides these remarkable performance benefits for DBMSs, Native Storage Management provides another essential advantage over the traditional I/O stack, in particular for NDP. Having multiple layers of abstraction results in a number of logical-to-logical and logical-to-physical address mappings (see Figure 2.4). For instance, the offset in a file is mapped to a certain block of the filesystem which, in turn, is mapped to several LBAs. In the on-device FTL, these LBAs are assigned to specific PBAs as already discussed in Section 2.3.3. Gathering those mappings across the levels of the access stack during runtime is highly complex, or sometimes even impossible as interfaces are missing. Yet, this information is crucial for NDP to navigate through the physical location of database objects. Utilizing Native Storage

management moves the entire address mapping management into the storage manager of the database, and thus, builds a single point of truth for it. Consequently, NDP operations can access the most recent mapping information prior to execution and use it on-device. Thus, several ways for sharing are possible, which are discussed in the next section.

3.2 EXTENDING NATIVE STORAGE WITH NDP

Prior to an NDP execution, a variety of information must be present on the device. On the one hand, this includes execution-specific parameters like the type of operation and its arguments, and on the other hand, information about the data being processed like its logical-to-physical address mapping or format definitions, as shown in Figure 3.3. The necessity of the latter will be discussed in detail in Chapter 4. Likewise, this information is extended even further with the ability to perform even transactionally consistent NDP executions, as we elaborated in Chapter 5. While the operation-specific information is obviously part of the NDP call, the remainder can be shared in various ways.

Key Insight: Address mapping information of database objects must be present on-device prior to NDP executions.

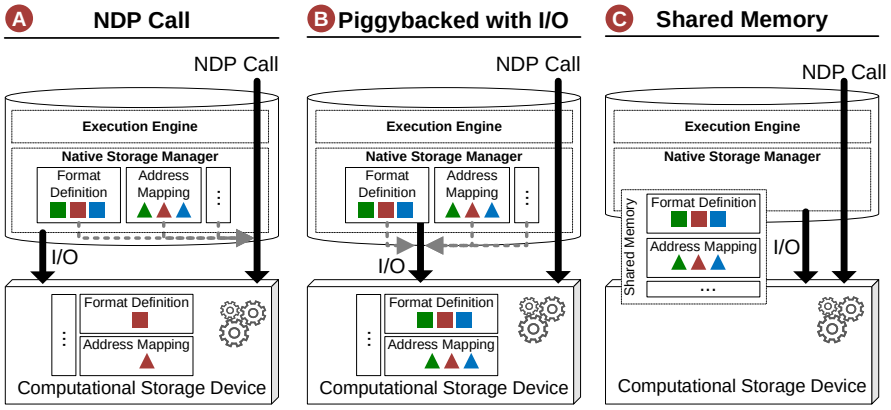


Figure 3.3: The logical-to-physical address mapping of native storage management is essential for NDP to resolve the location of database objects on-device. In addition to format definitions, it has to be present on the device prior to the NDP execution. Thereby, it can be propagated either (A) as part of the NDP call, (B) piggybacked with other I/O calls, or (C) exchanged via shared memory.

AS PART OF THE NDP CALL: Probably by far the simplest option is to extend the interface definition of an NDP command with all necessary information. On the one hand, this increases the size of the invocation significantly because address mappings can become very large with today's data volumes, as we discuss in Section 3.3. On the other hand, this does not require any additional on-device management functionality to keep the information in sync. In consequence, especially devices with constrained

resources benefit, as their memory capacity can be utilized for intermediary results as discussed in Chapter 6. Moreover, it is not necessary to push down the entire information, but it rather can be limited to the specific address mappings and format definitions required by the actual NDP command. Additionally, with novel abstractions like the Physical Page Pointer (PPP), discussed in Section 3.3, the size of the commands can be reduced even further. Another key benefit is that such propagation can be realized with existing interconnection protocols like NVM express (NVMe).

PIGGY-BACKING OTHER CALLS: Another way to continuously propagate the most recent version of address mappings and format definitions is to piggyback other calls to the device such as read or write operations. Since those calls usually also entail a change of the address table, they are suited to directly maintain a redundant version on-device. Likewise, existing standard protocols like NVMe can be exploited for this. As a consequence, the interface of the NDP command becomes lean, and the data being transferred prior to the execution is reduced. Low-latency commands from an OLTP workload will profit especially from this type of information sharing.

VIA SHARED MEMORY: Lastly, instead of having a redundant on-device version, modern cache-coherent protocols like Cache Coherent Interconnect for Accelerators (CCIX) [26] or Compute Express Link (CXL) [27] can be used to maintain the address mapping and format definitions in a shared memory. Changes to the shared memory from either the host or the device are synchronized in the background, allowing the remaining calls to focus on their purpose [91].

Within this thesis, we will focus on the first type of information sharing. To this end, we extended the concepts of Native Storage management with a well-defined interface for NDP commands. A first implementation is proposed in Publication P3 of this thesis, which is able to shift the processing of simple aggregations on a dataset from the distributed file system Ceph [103] to NDP-capable devices. Since then, the interface evolved along with our system nKV that includes further sophisticated features. In the next Chapters we will tackle for instance the filtering according to SearchKeyRanges on heterogeneous hardware P5 P6 (see Chapter 4), the propagation of shared data and state to fulfil transactional consistency P8 (see Chapter 5), and execution pipelines or result-set processing P9 (see Chapter 6) C1.2 .

3.3 REDUCING ADDRESS INFORMATION VOLUME

Key Insight: The size of address mappings for today's data volumes can easily grow up to multiple gigabytes.

As mentioned beforehand, address mapping information can become huge. For instance, database objects like tables can easily become 1 TB of data. Assuming that the physical page size is 16 KB, the mapping table for this single object will end up with 62.5 million entries. Using 64 bit to represent the LBA and PBA, the size of the mapping table will result in 1 GB. Transferring this with every NDP call will result in a significant overhead for the data movement and consequently impair the key idea of relieving the bus system. Moreover, low-latency NDP invocations will not be feasible.

However, many modern data organizations make use of the append-based behavior of logging, which in turn, can be utilized to significantly reduce the size of mapping tables of database objects. Prominent examples of append-based structures are Log-Structured Merge (LSM) Trees [70] or Partitioned B-Trees [38, 83]. The important aspect of those data organizations is that updates do not appear in place but rather are appended as a new version of the record. Hence, once written, data is not updated anymore until its deletion due to GC. While this does not only benefit the characteristics of wear-prone storage technologies like Flash, it also ends up in immutable parts of the data organization, which can have a stable physical location on the device.

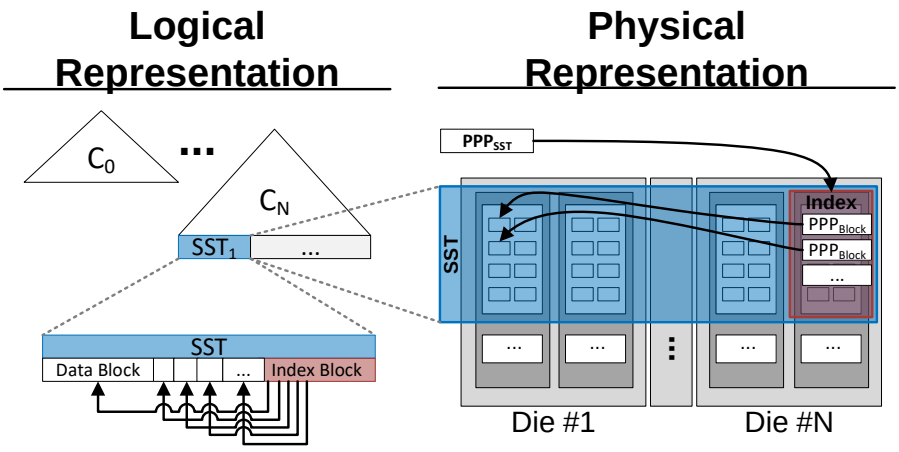


Figure 3.4: Many modern data organizations build on append-based logging and have immutable parts like data blocks of SSTs. To reduce the size of logical-to-physical address mappings for NDP calls, the entries for those immutable parts can be replaced by a Physical Page Pointer, e. g., references of the data blocks in the index block.

With Native Storage management (see Section 3.1), we are able to place those immutable parts in certain locations of the storage device. For instance, Sorted String Tables (SSTs) of LSM Trees have a configurable upper bound in their size. Aligning it with the characteristics of the underlying storage, e. g., multiple Flash blocks, allows mapping certain physical blocks to the logical object of an SST as depicted in Figure 3.4. As a consequence, NDP calls do not require a fine-granular

mapping table of pages anymore, but rather a pointer to the physical location, like PPP_{SST} in Figure 3.4. This concept of Physical Page Pointer (PPP) is proposed as one novel abstraction for NDP processing in Publication P8 C1.3 . Beyond the pointer itself, the respectively required format and layout for the given structure are discussed in Chapter 4. The application of PPPs is very versatile. On the one hand, immutable parts of data organizations can be directly addressed by NDP calls, avoiding the necessity to synchronize entire address mapping tables between the host and the computational storage device. On the other hand, virtual references, such as offsets in files, can be replaced with PPPs as shown as PPP_{Block} in Figure 3.4, which enables to access internal parts of an immutable object without having its address mapping. Prerequisites for PPPs are a Native Storage managed space and that the physical location of the data is guaranteed to be stable during the lifetime of the object. As a result, self-contained physical objects can be easily referenced with PPPs.

To quantify the benefit, we can contrast the PPPs approach with the 1 GB mapping table from the beginning of this section. Assume a single self-contained SST comprises 100 MB of data, which corresponds to 50 Flash blocks on the COSMOS+ [88], a wide-spread prototype board of a computational storage device. Thus, theoretically 10 000 SSTs would fit on a 1 TB storage space. As a Flash block can be referenced with a single 64 bit value, the total size of the PPPs is tiny 4 MB instead of a 1 GB large address mapping table in traditional systems.

» RECAP: INSIGHTS AND SOLUTIONS

This chapter elaborated on the drawbacks of today's widespread backward-compatible I/O stack, especially with the change from traditional HDDs to SSDs C1.1 . The key insight is that SSDs do not only fundamentally differ from HDDs, but also the backward-compatible I/O stack employed for SSDs hinders leveraging their significantly higher bandwidths and parallelism. Using Native Storage Management instead not only enables to exploit properties of the underlying NVM, it also allows managing the data placement according to database objects and workload characteristics, as well as introduces a single logical-to-physical address mapping as a single point of truth within the storage manager of the database. The deep integration into the database makes Native Storage Management superior to other state-of-the-art subsystems like LightNVM [15]. With Native Storage Management, we can resolve the information hiding due to backward-compatible I/O stacks. The address mapping table is also essential for NDP to resolve the location of database objects on-device. Hence, our investigations reveal that it has to be present on the device prior to the NDP execution. The proposed interface is defined accordingly C1.2 . Yet, we also discovered that such address mappings can become very

large with today's amount of data and require novel abstractions like the Physical Page Pointer (PPP) C1.3 .

In conclusion, by examining Research Question RQ1, our proposed general and lightweight interface for NDP is now aware of the on-device database objects, ensures to access them at full bandwidth and parallelism, has only a lean set of arguments, and can be easily mapped to low-level protocols like NVMe, Remote Direct Memory Access (RDMA), or development frameworks for the integration of heterogeneous computing platforms, e. g., TaPaSCo [59].

4

ON-DEVICE NAVIGATION AND DATA INTERPRETATION

In the previous Chapter 3 we introduced Native Storage management for NDP. However, to process data on a computational storage device, it is essential to find and extract it according to its given format.

RESEARCH QUESTION – RQ2

» Is in-situ data interpretation and navigation necessary for NDP and how is data navigated through and interpreted in-situ? «

By investigating RQ2 we firstly elaborate on the extent to which data interpretation and navigation are required for shifting a processing task to a storage device. Secondly, we will focus on an implementation that allows leveraging the heterogeneous processing capabilities of modern computational storage devices. Lastly, we discuss how the burden of hardware development, as outlined in Section 2.3.4, can be alleviated for NDP. Figure 4.1 shows the areas considered within a database as an architectural guidebook. The following main contributions are made in this chapter:

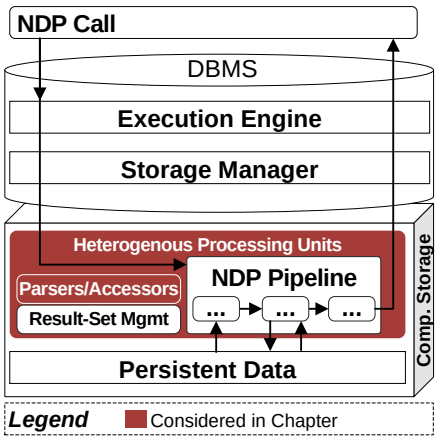


Figure 4.1: Architectural guidebook.

- C2.1 Elaboration on cross-layer data formats and layouts in databases.
- C2.2 Introduction of Parsers and Accessors for in-situ navigation and data-interpretation.
- C2.3 Investigation of heterogeneous processing hardware of computational storage devices.
- C2.4 Proposal for utilizing heterogeneous hardware with NDP and improving performance by 1.4× to 2.7×.
- C2.5 Proposal of a framework for automatic generation of Parsers and Accessors on FPGAs to lower the boundaries of hardware-accelerated NDP.

The corresponding publications are:

P4	[93] – "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems"	Chapter 11
P5	[98] – "nKV: Near-Data Processing with KV-Stores on Native Computational Storage"	Chapter 12
P6	[99] – "nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing"	Chapter 13
P7	[101] – "A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems"	Chapter 14

4.1 THE NECESSITY FOR CROSS-LAYER PARSERS AND ACCESSORS

In Chapter 3, we established that with Native Storage management, the database is responsible to place the data at specific physical locations of the storage device for performance and longevity. Information necessary for the address resolution is provided to the device (e. g., as part of the NDP call), to have it on hand when the processing of an NDP call starts.

Key Insight: Cross-layer layouts and formats of the database are required by the NDP call to navigate and interpret the data.

Yet, besides the logical-to-physical address mapping, further information about the structure of the data is necessary to extract specific parts from the physical storage representation of a database object, e. g., values of a record. In the context of databases, these definitions are often referred to as layouts and formats which are only present within the database itself. As a consequence, NDP calls may resolve the physical location of a database object on the device, as shown in Figure 3.4, but are not able to navigate or interpret the data without those definitions. Yet, by additionally providing this information to the computational storage device, NDP calls become not only aware of how to locate the physical location of the data but also of how to extract the values of a logical attribute by navigating through the various layers of the data organization and interpreting the different structures. In the end, on-device data-interpretation and navigation are necessary ingredients for reducing unnecessary host-to-device round-trips (e. g., to request a format or layout definition) and a prerequisite for an intervention-free offloading model, which will be the subject of Chapter 5.

Layouts and formats are related to each other as outlined in Publication P4. While formats define the contents of a structure, layouts describe their spatial arrangement within a certain scope e. g., a memory space. In database architectures, those layouts and formats are nested, starting with the physical storage organization of database objects, as depicted in Figure 4.2. For instance, the subdivision of the components of a LSM tree into SSTs can be seen as a first format

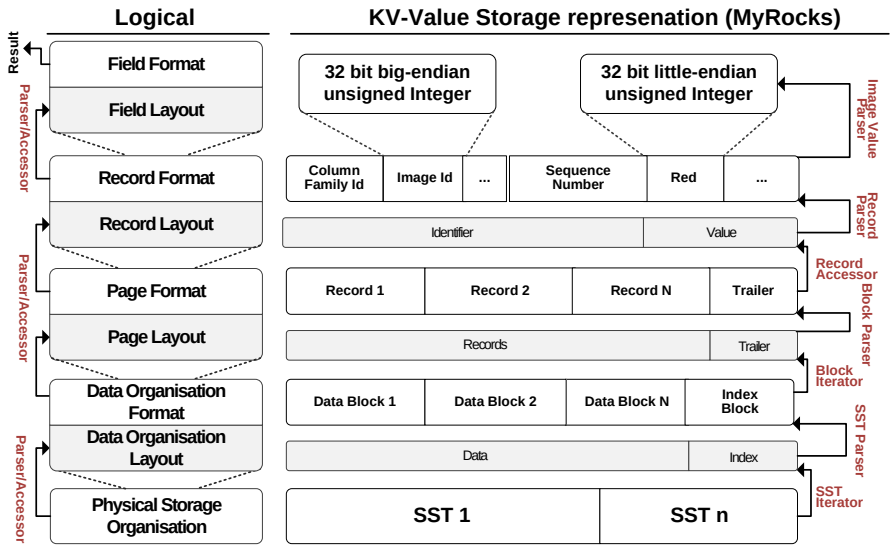


Figure 4.2: Along the hierarchy of physical storage representations there are multiple format and layout definitions. To navigate within and interpret the data of those, we propose cross-layer Parsers and Accessors in P4.

definition. Proceeding, the format of a SST usually comprises several data blocks and a single index block. Thereby, the layout of a SST defines the arrangement within the physical storage. As shown in Figure 3.4 of Section 3.3, Physical Page Pointers (PPPs) can be used to efficiently reference those elements. Similarly, further levels like blocks, pages, records, and fields are expressed by specific layout and format definitions C2.1 .

In Publication P4 and [102], Parsers and Accessors are proposed as the matching counterpart of formats and layouts C2.2 . With those, NDP calls are enabled to navigate and interpret the data of a given database object on-device. Thereby, the NDP call uses the Accessors to access the respective sub-structures until it reaches the necessary level of abstraction for its processing purpose. For instance, to process a simple `COUNT(*)` it is not required to access any specific field within a record but sufficient to count the records themselves. However, whenever a specific structure like `AVG(Img.Red)` of Figure 4.2 has to be interpreted, the NDP call uses its Parsers to extract the semantical meaning. For instance, to process a numerical field to calculate a `SUM`, an NDP call can use a Parser to extract the value that could be either big- or little-endian encoded.

As those Parsers and Accessors are also part of the different layers of a DBMS, we argue for having cross-layer Parsers and Accessors [93, 102]. Yet, even though the functional behavior might be identical, their implementations may vary since it has to be adapted to the characteristics of the given processing unit. Within the next section, we will discuss how the concept of Parsers and Accessors behaves

together with the heterogeneous processing capabilities of modern computational storage devices.

4.2 LEVERAGING HETEROGENEOUS PROCESSING CAPABILITIES

Section 2.3.4 already discussed that hardware is subject to rapid change. Besides storage technologies, today's processing units are available with diverse properties. While traditional scalar processing units optimize for executing Single Instruction, Single Data (SISD), vector processors are capable of executing Single Instruction, Multiple Data (SIMD). Often both types of Flynn's taxonomy [34, 35] are combined in modern CPUs. Additionally, there are GPUs as SIMD processors that have significantly higher data parallelism.

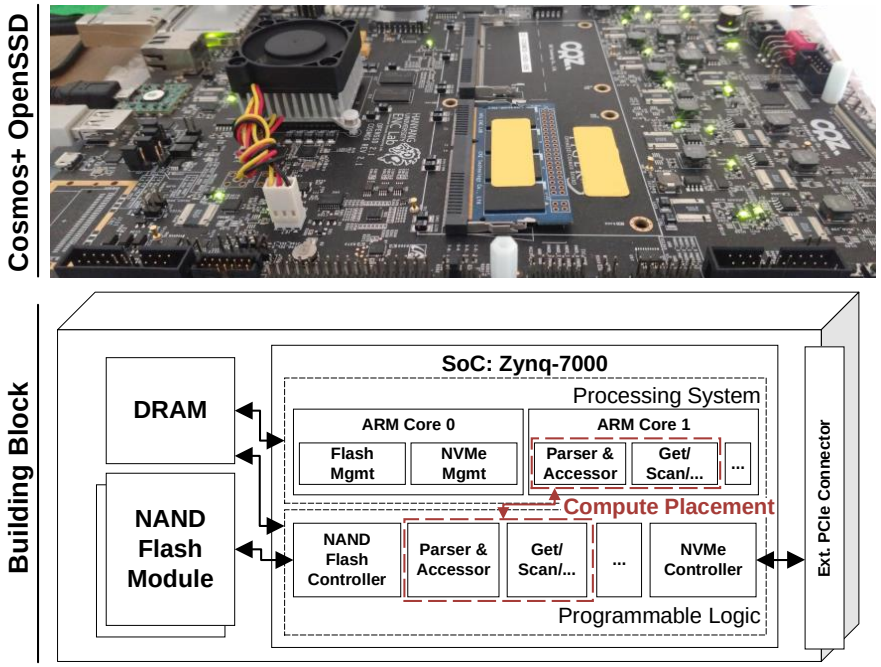


Figure 4.3: Computational storage devices comprise a variety of heterogeneous processing capabilities. For instance, the Cosmos+ OpenSSD, which is mainly used in the publications of this thesis, is equipped with an SoC Xilinx/AMD Zynq 7000 with two ARM cores as Processing System and an FPGA as the Programmable Logic.

Key Insight: Modern computational storage devices comprise a multitude of processing units with different characteristics.

Yet, all of those processing units can not be considered separately anymore as modern packaging techniques in the semiconductor industry are capable of combining those economically on a single chip. Usually referred to as a System on Chip (SoC), they are equipped with storage and memory on the PCB of a computational storage device. For instance, Figure 4.3 shows the Cosmos+ OpenSSD [28, 88] and its building block diagram as used in the experimental evaluations of this thesis. By executing operations on a SoC one can decide which pro-

cessing unit fits best for the given algorithm but also have to consider balancing the load equally.

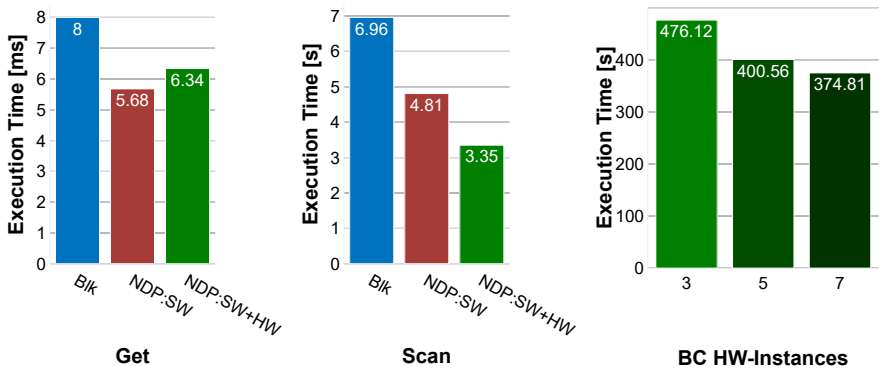


Figure 4.4: In Publication P5, Get, Scan, and BC requests are executed on a traditional block device stack (Blk), as purely software-based NDP (NDP:SW), and with FPGA support (NDP:SW+HW). While the execution time of some of these operations degrades with hardware support (Get), others profit significantly by leveraging multiple hardware instances running in parallel on the FPGA (Scan and BC).

Modern computational storage devices are equipped with SoCs, providing NDP the capability to dynamically place the computation on these heterogeneous SoC processing units C2.3 . In Publication P5 as well as in Publication P6, we investigate the benefits of offloading NDP executions to different kinds of processing units on real hardware. By this, we achieved improving the execution latencies of simple `GET` requests by 1.4× and `SCAN` requests on the values by 2×, when executing them as NDP in software in contrast to traditional systems employing a block device stack, as shown in Figure 4.4. More complex algorithms, like Betweenness Centrality (BC) [21], have improved even 2.7×, when executed in a hybrid hardware-software co-design C2.4. In our experimental evaluation, we identified that there is no one-size-fits-all, but rather different processing units perform best under given settings for NDP. For example, using FPGAs to execute a low-latency request like `GET` is not efficient since the high initialization costs cannot be amortized by the higher execution parallelism as data has to be read sequentially. However, performing data-intensive operations like BC on an FPGA outperforms pure CPU-based implementations, as it can exploit the parallelism by interleaving the execution and fetching of new data [98, 99]. Additionally, we can scale up the parallelism by employing multiple hardware instances, as shown in Figure 4.4, and thus, improve the execution time even further.

Key Insight: With the heterogeneous processing capabilities of SoCs NDP can be accelerated, but the performance highly depends on the utilized processing unit for a given setting.

However, executing NDP calls on heterogeneous processing units also requires that Parsers and Accessors, presented in Section 4.1, are present on the device for those (see Figure 4.3). Yet, in particular, for

FPGAs their development is extremely cumbersome as we will discuss in Section 4.3.

4.3 AUTOMATION OF NDP ACCELERATOR CREATION

Due to the fact that today's computational storage devices comprise heterogeneous hardware, NDP calls can be executed on various processing units. Especially FPGAs offer significant potential to accelerate data-intensive workloads as pointed out in Section 4.2.

Key Insight: FPGA-accelerated NDP is beneficial but cumbersome due to the tedious development process.

However, a major drawback is today's cumbersome development process of FPGA accelerators via HDL-based design entry, as outlined in Section 2.3.4. While the effort might be appropriate for a single static application, in the context of databases several Parsers and Accessors have to be developed for the different abstractions like Blocks, Pages, or Fields. Additionally, this is multiplied by the number of schemas, hence, the number of database objects with different record formats. To significantly reduce the effort spent in implementing NDP accelerators, and thus, ease their integration, their creation has to be automated.

In Publication P7, we propose a framework for the automatic generation of FPGA-based NDP accelerators, whose performance is similar to a manually optimized version C2.5. Thereby, we are able to lower the barrier in integrating hardware acceleration in NDP.

» RECAP: INSIGHTS AND SOLUTIONS

On top of the findings from the previous chapter, we detected and elaborated the cross-layer layouts and formats of databases as a key necessity for NDP to navigate and interpret the physical data C2.1. In contrast to past NDP research [9, 51, 105, 106, 108], which process data in-situ with simplistic or relatively static parsers, we argue for flexible Parsers and Accessors that are conveniently aligned to Formats and Layouts of traditional databases as novel NDP abstractions. They have to be present on-device prior to processing data but can adapt to schema evolution C2.2. Moreover, our research in the technology of modern computational storage devices reveals that those comprise a multitude of different processing capabilities that can be utilized to accelerate NDP C2.3. In contrast to related work, which focussed on a single processing unit [9, 41, 105, 106, 108], we discovered that a careful compute placement of a given logic on a specific processing unit is key for leveraging the full potential. By this, we achieve to accelerate simple queries by 1.4× to 2× and complex analytical algorithms, like BC, by even 2.7× in contrast to traditional systems relying on a block device stack C2.4. However, we also identified the tedious and cumbersome development process of FPGA-based accelerators as a barrier to the application in NDP. Thus, we propose a framework

for the automatic generation of Parsers and Accessors on FPGAs as a novelty C2.5 .

In summary, investigating Research Question RQ2 resulted in a clear approach to process data on-device while leveraging the benefits of the available heterogeneous processing capabilities of modern computational storage devices.

5 NDP OFFLOADING MODELS

In the previous Chapters 3 and 4 we outlined the prerequisites for NDP to access the data and process it on-device. Within this chapter we will set the focus on the following research question:

RESEARCH QUESTION – RQ3

» *How do novel interaction and offloading models for NDP-storage impact DBMS architectures?* «

With RQ3 we investigate the interplay of NDP and modern workload characteristics like HTAP, as presented in Section 2.3.1. We first give an overview of the different types of offloading models for NDP and discuss to what extent DBMS have to be adapted accordingly. Likewise, we discuss the extension of the solution space for today's DBMS architectures that are outlined in Section 2.3.2 to deal with HTAP workloads. As a foundation, we present the concept of transactional NDP and focus on a snapshot-based approach. Moreover, we provide more details on the necessary Shared State and its propagation toward the computational storage device. Lastly, we elaborate on the data freshness and transactional consistency aspects gained by an intervention-free execution. Figure 5.1 shows the areas considered within a database as an architectural guidebook. Thus, we make the following main contributions in this chapter:

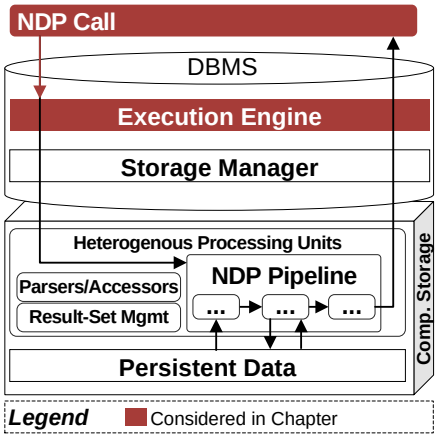


Figure 5.1: Architectural guidebook.

- C3.1 Definition of novel approaches to offload processing to computational storage devices.
- C3.2 Concepts for intervention-free NDP execution with transactional guarantees.
- C3.3 Extension of the solution space of HTAP by providing an architecture that is capable of NDP with the most recent data.
- C3.4 Evaluation of the benefits by shifting processing to a computational storage device, achieving 30% performance improvements while reducing the power consumption by 26%.

The corresponding publication is:

P8 [95] – "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads" — Chapter 15

5.1 TYPES OF NDP OFFLOADING MODELS

When issuing an NDP call to a computational storage device, its arguments comprise the logical-to-physical address mapping as well as information about formats and layouts to navigate and interpret the static data on the device, as we discussed in Section 4.1. However, this is only sufficient as long as we can guarantee that no concurrent transaction has modified the data at the same time. Past NDP approaches assumed a read-only static dataset [9, 67, 105, 106, 108]. Nevertheless, this assumption is not valid for all cases, especially in OLTP and HTAP setups. Consequently, the offloading model for NDP transaction has to ensure that the transactionally consistent result is computed on-device. To facilitate this, the host and the computational storage device have to exchange further information like concurrently running transactions or modified records located in the database buffer. In the following, we will list and elaborate on the different types of possible offload models that are depicted in Figure 5.2 C3.1 .

Key Insight: NDP is not limited to read-only setups but can also be applied in workloads with frequent updates.

FIRE AND BUSY WAIT: The first type of offloading model is to just issue the NDP transaction to the computational storage device and continuously poll for its return, as shown in Figure 5.2.A. This can occur either if the NDP transaction has been completed or if it requires more information from the host. For instance, after an NDP transaction has processed the persisted data on-device, it requests the latest committed updates from concurrent transactions of the host. While the implementation is very simplistic, it has the major drawback that the host requires one thread to actively wait for the return of the NDP transaction, and thus, cannot utilize it for parallel OLTP processing. In the worst case, this will end up in a high number of context switches that impair performance.

ASYNC. EXECUTION AND INTERRUPT: One way to improve this behavior is by switching to an asynchronous execution model. Thereby, the host issues the NDP transaction to the computational storage device in a similar manner but does not actively wait for any return as depicted in Figure 5.2.B. On the contrary, the host can utilize its waiting (i. e., idling) cycles for parallel OLTP processing. Whenever the device requires to communicate with the host, it sends an interrupt to break the current processing, which adds additional execution time.

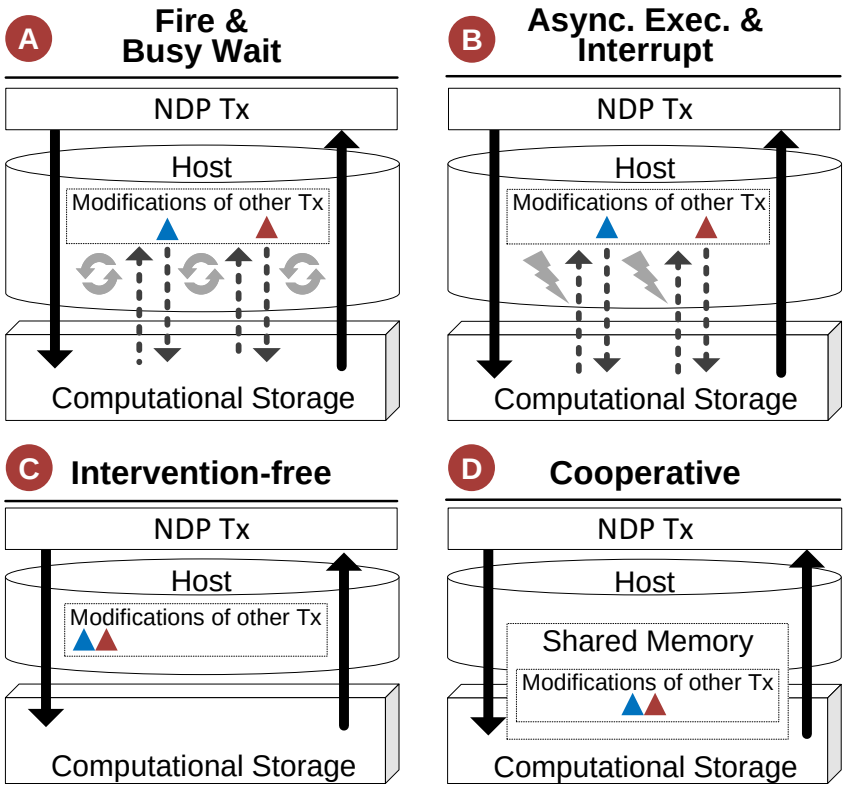


Figure 5.2: There exist several offloading models that can ensure transactional guarantees for NDP queries. (A) The host is actively waiting for the NDP to return, (B) the communication is based on interrupts, (C) all necessary data is gathered prior to the NDP execution, or (D) utilizing a shared memory to exchange data.

INTERVENTION-FREE: Since every interaction between the host and the computational storage device leads to an intervention in the processing on both sides, we argue to avoid any unnecessary round-trips. However, as we discuss in Chapter 6, there are cases where communication is required. For instance, for final result transferring or for overall device management. Consequently, the NDP transaction is issued in an intervention-free manner, shown in Figure 5.2.C. To apply such an offloading model, all relevant information about concurrent transactions has to be gathered and attached as additional attributes to the NDP transaction prior to its execution. This Shared State, as we propose it in Publication P8, builds the foundation and will be presented in detail in Chapter 5.2.

COOPERATIVE: Instead of sending the Shared State with the NDP transaction to the device, also modern protocols like CCIX [26] or CXL [27], enabling cache-coherence for shared virtual memories, can be used to exchange the information [91]. As shown in Figure 5.2.D, this provides the advantage that the NDP transac-

tion itself becomes lean while no interrupting handshakes are necessary between host and computational storage device.

In general, with all listed offloading models it is possible to process the most recent data from the host on the device, and thus, can ensure transactional consistency. In the remainder of this thesis, we will focus on the `INTERVENTION-FREE` offloading model.

5.2 PROPAGATING THE SHARED STATE

Key Insight: For a transactionally consistent execution, NDP requires the latest modifications from the host – the Shared State.

As explained in Section 5.1, the Shared State is the necessary foundation to ensure transactional consistent NDP execution. One part of this Shared State comprises the most recent data of transactions committed immediately before starting the NDP operation. Especially, with modern workload characteristics like HTAP (see Section 2.3.1), we can expect that the most recent modifications on the data, frequently triggered by the OLTP-part of the workload, still resides in the database buffer. Those modifications are not immediately flushed to the storage device, and thus, constitute the reminder to the much larger cold data on-device for a transactionally consistent snapshot. The other part of the Shared State comprises database-specific modifications on mapping tables, status, and system information as well as auxiliary data structures e. g., for version management.

Those modifications are spread across various data structures of today's database architectures. As a consequence, it becomes cumbersome or sometimes even impossible to gather them. Nevertheless, in Publication P8, we propose the Delta Buffer as a simple solution to collect all modifications. Thereby, these modifications are appended as replacements records in the Delta Buffer so that newer versions always overwrite older versions.

After the Shared State becomes available on the device, by using the previously presented `INTERVENTION-FREE` offloading model of Section 5.1, the NDP transaction is able to construct a transactionally consistent snapshot in-situ, and thus, can easily process the most recent data of other transactions by accessing the Delta Buffer C3.2 .

5.3 DATA FRESHNESS AND TRANSACTIONAL CONSISTENCY

Key Insight: Past approaches tackling HTAP suffer either in a steady performance or ensuring a high data freshness.

Data freshness became an important property in the context of HTAP workloads [20, 77]. As discussed in Section 2.3.1, modern applications require gaining analytical insights about the data that is rapidly modified in parallel by the foreground workload. Consequently, OLAP and OLTP queries either fight for the same resources on a single unified engine or are separated, with a continuously-running data exchange process in between, as depicted in Figure 5.3.A/B. While the first approach trades off a loss in performance against a higher data freshness

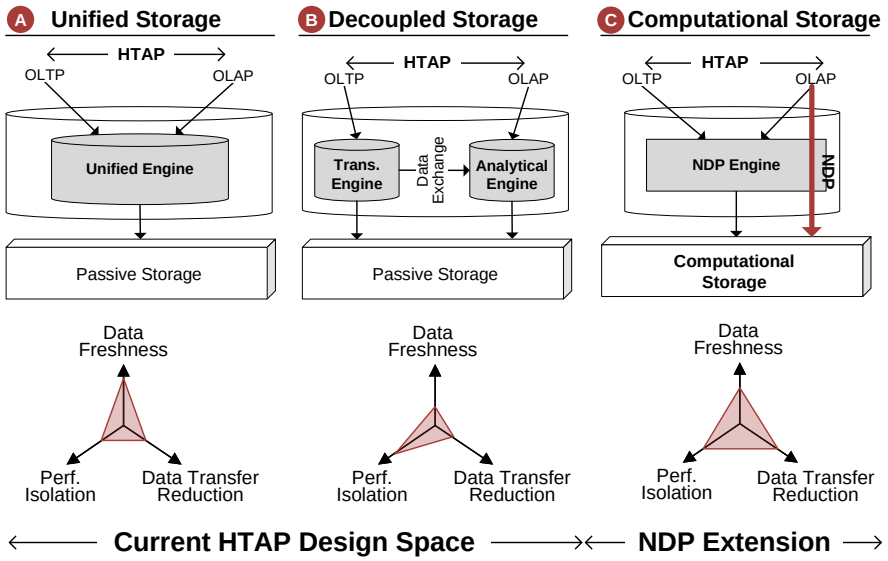


Figure 5.3: Besides (A) having a single unified engine to process OLTP and OLAP queries, and (B) having separate engines for both types of workload, we propose NDP as an extension of the solution space that improves the performance isolation while reducing data transfers and ensuring a high data freshness.

rate, the latter ensures a robust OLTP performance by including the most recent data in the analytical queries.

But with NDP as a novel extension for the HTAP problem space, we can optimize on both dimensions at the same time, as shown in Figure 5.3.C. In particular, with the previously mentioned offloading models of Section 5.1, and the Shared State of Section 5.2, as well as the contributions of Publication P8 a transactional consistent execution of NDP transactions can be guaranteed C3.3 .

In our experimental evaluation of Publication P8 , we execute a modified HTAP version of LinkBench [6] on our system. The system stack is set with two alternative configurations: the first one as traditional Block I/O stack, and the second one, with the previously described `INTERVENTION-FREE` offloading model, as Update-aware NDP stack. While the first performs OLTP and OLAP workload on the host, the latter shifts the processing of data-intensive long-running analytical queries to the computational storage device. As shown in Figure 5.4, we prove our expectation that, with the execution of an OLAP query a traditional Block I/O stack, the system throughput rapidly degrades and remains significantly lower during its execution. After its completion, it takes a considerable amount of time to remain the original performance, as the database buffer was highly polluted by the cold data of the analytical query. In contrast, with the Update-aware NDP stack, we improve the throughput in general by leveraging the previously discussed concepts of Native Storage Management of Chapter 3. But even more importantly, we ensure a steady and 30%

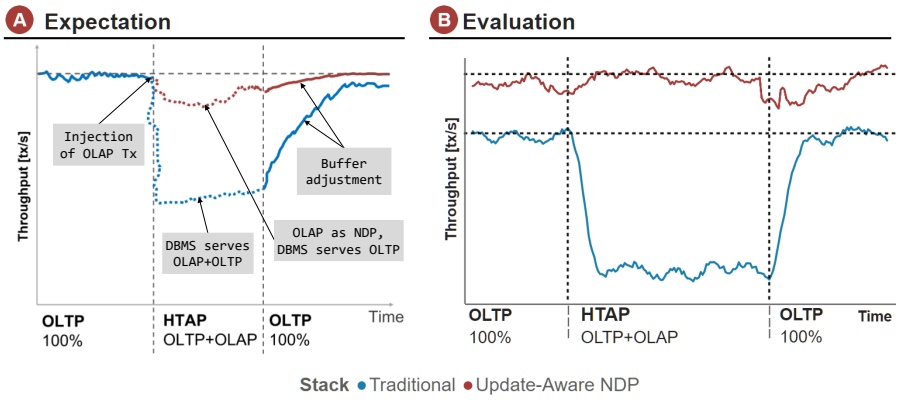


Figure 5.4: In Publication P8, we compare the execution of an HTAP workload on the traditional Block I/O stack with the Update-aware NDP utilizing an intervention-free offloading model. While the traditional stack suffers as OLTP and OLAP compete for resources and pollute their data locality in the buffer, the Update-aware NDP stack achieves steady and 30% faster performance.

faster performance by having an `INTERVENTION-FREE` NDP execution of the OLAP query with transactional guarantees. Notably, the CPU utilization on the host stays constant, as no waiting time is spent for reading the large cold datasets, and the database buffer on the host does not suffer from any pollution. Lastly, by significantly reducing the data movement between host and device, and by leveraging the on-device processing capabilities, the energy consumption can be reduced by 28% C3.4 .

» RECAP: INSIGHTS AND SOLUTIONS

In addition to the insights of the previous chapters, we discussed how novel interaction and offloading models for NDP impact DBMS architectures, and thus, investigated Research Question RQ3. Firstly, we identified that NDP is not limited to the read-only scenarios that have been presented by past approaches [9, 11, 41, 67, 105, 106, 108]. Novel approaches of offloading NDP to computational storage devices and facilitating the exchange of the most recent modifications and status information of concurrent transactions are required C3.1 . Focusing the research on concepts for attaining a transactionally consistent snapshot on-device to support transactional guarantees for in-situ processing, we identified the Shared State as a set of necessary information to be gathered prior to an NDP execution. Furthermore, we propose the Delta Buffer that gathers all those modifications in an append-based manner and can easily be shared with the computational storage device C3.2 . Novel offloading models like `INTERVENTION-FREE` and `COOPERATIVE` avoid costly device-to-host round-trips. Moreover, by elaborating the HTAP problem space, we detected that current approaches either optimize for performance isolation or data freshness.

However, with transactionally consistent NDP transactions, we not only optimize for both but also reduce the data transfers between host and storage device C3.3 . By this, we achieve an overall performance improvement of 30% while reducing the power consumption per transaction by 26% in contrast to a traditional Block I/O stack C3.4 .

So far, the NDP approach of this thesis is capable of resolving the physical data on-device (Chapter 3), navigating and interpreting it (Chapter 4), as well as ensuring transactional guarantees by using offloading models to exchange the Shared State between host and device.

6 NDP EXECUTION AND RESULT-SET HANDLING

Lastly, we focus on the NDP execution and its final as well as intermediary result-set management. While the previously discussed abstractions for storage management of Chapter 3, Parsers and Accessors for navigation and interpretation of Chapter 4, as well as the offloading model of Chapter 5 build the prerequisites, we will have a deeper look into the following Research Question RQ4 in the following sections:

RESEARCH QUESTION – RQ4

» *Is in-situ result-set handling necessary for NDP?* «

We elaborate on a number of possibilities to execute NDP on a computational storage device. The main focus of this thesis is on NDP pipelines as one execution mode, which we discuss in more detail. Moreover, with NDP pipelines the need for managing intermediary as well as final result-sets arises, for which we introduce necessary concepts. Lastly, we present an NDP finite state machine as one approach to coordinate processing and result-set management on-device. Figure 6.1 shows the areas discussed within a database as an architectural guidebook. Thereby, the following main contributions are made in this Chapter:

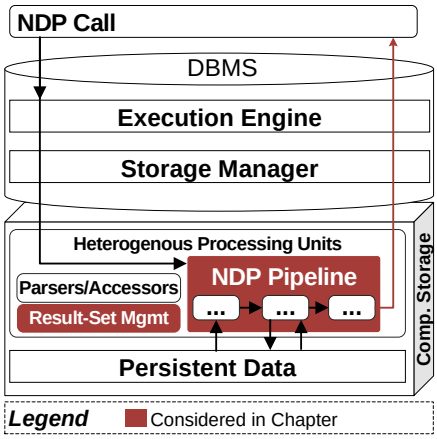


Figure 6.1: Architectural guidebook.

- C4.1 Overview of NDP execution modes with a focus on NDP pipelines.
- C4.2 Introduction of in-situ result-set management in the context of NDP pipelines.
- C4.3 Introduction of the reuse of in-situ materialized results without significant overhead, improving execution durations by up to 400×.
- C4.4 Enhancement of fault tolerance by reuse of materialized results e. g., in complex pipelines.
- C4.5 Introduction of the orchestration of interleaved NDP pipeline execution on computational storage devices.

The corresponding publications are:

- P8 [95] – "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads" — Chapter 15
- P9 [94] – "Result-Set Management for NDP Operations on Smart Storage" — Chapter 16

6.1 EXECUTION MODES

From the previous chapters, we know that NDP calls can be issued to the computational storage device in parallel to other workloads on the host. With the correct offloading model all information for an intervention-free processing is available on the device, including logical-to-physical address mappings, format and layout information as well as the most recent modifications from the host. Yet, there are multiple modes an NDP call can be executed. In the following list, we describe a number of them.

TASK-BASED: Firstly, an NDP call can reflect a single task that is executed at once. While its internal complexity can vary from simple tasks like filtering, to more sophisticated operations like sorting or even support matrix computation, the behavior seen by the caller is quite simple. The processing of an NDP task is started with all necessary parameters, and ends either successfully, or with some kind of error. For instance, [41] proposes a framework for NDP that defines SSDlets as simple programs that can be executed either on the host or on-device. Thereby, those SSDlets can be interconnected via ports to move data between them and allow implementations of more complex algorithms like MapReduce. In general, the orchestration of NDP tasks is simple. Yet, with their relatively static behavior, they cannot be used in every scenario, e. g., a stream of data.

STREAM-BASED: Another approach is inspired by network packet processing and attaches data processing to data movement. Thus, the key difference to the task-based execution mode is that the stream-based approach assumes a continuous stream of data. While this is very common for network environments and streaming-based applications like Kafka, it can be also applied to storage devices. For instance, [11] proposes a backward-compatible solution to introduce NDP along the traditional requests to a storage device. It is triggered by a read or write request to the device, which entails a data movement. While the data is transmitted, it is transformed on the way according to a given NDP operation e. g., filtering.

PIPELINE-BASED Lastly, we propose the pipeline-based NDP execution in Publication P8 of Chapter 15 that combines parts of the previously discussed execution modes. A pipeline consists of multiple interconnected operands, similar to the task of the task-based execution mode. However, data is streamed through those operands according to a given execution tree. Thereby, the pipeline-based execution mode partially resembles the classical volcano-style execution of many databases [37].

Depending on the surrounding conditions of the system architecture one of the previously presented execution modes fits best C4.1 . For the context of this thesis, we set the focus on NDP pipelines, which are described in detail in the next section.

6.2 NDP PIPELINES

NDP pipelines are proposed in Publication P8. They are defined as a demand-pull pipeline that is split into multiple operators. Each operator sequentially processes a block of data, and thus, the pipeline operates in a block-at-a-time manner. Whenever an operator finishes processing an input block, it demands from the preceding operator to provide the next block. Yet, the input of an operator can comprise outputs of multiple preceding operators. Thus, an NDP pipeline is not a single concatenation of operators but resembles an execution tree that may involve classical pipeline-breakers as well-known in DBMS and depicted in Figure 6.2 C4.1 . The goals of such an NDP pipeline are either to reduce the overall device-to-host data transfers, or to transform data into an adequate format for further processing, e. g., optimized for bandwidth utilization or Direct Memory Access (DMA) transfers.

An operator itself processes the data record by record whereas the applied transformation logic can vary. As in classical database systems, an NDP pipeline usually starts with a simple `Scan` operator followed by a `Selection` (σ) that filters the data according to some criteria. Subsequently, a `Join` ($\bowtie$), Group By, Sorting or any other user-defined function like Betweenness Centrality (BC) can be applied, as we evaluate in more detail in Publication P8 . [58] elaborated that in such NDP pipelines a `Projection` (π) should be executed as early as possible, also called early-projection, to reduce the data movement as soon as possible in the execution tree.

Key Insight: NDP pipelines provide a flexible way of processing data in-situ and leverage the heterogeneous processing capabilities.

Between two operators there is a buffer phase that handles intermediary results, which we discuss in Section 6.3. Each buffer phase uses a memory space, configurable in its size, to cache the outputs of the operators. This not only allows to process larger chunks of data in a row but also can be aligned to characteristics of underlying storage technologies, like Flash pages, which is of high importance as presented in Section 3.1. Moreover, it enables to execute operators on

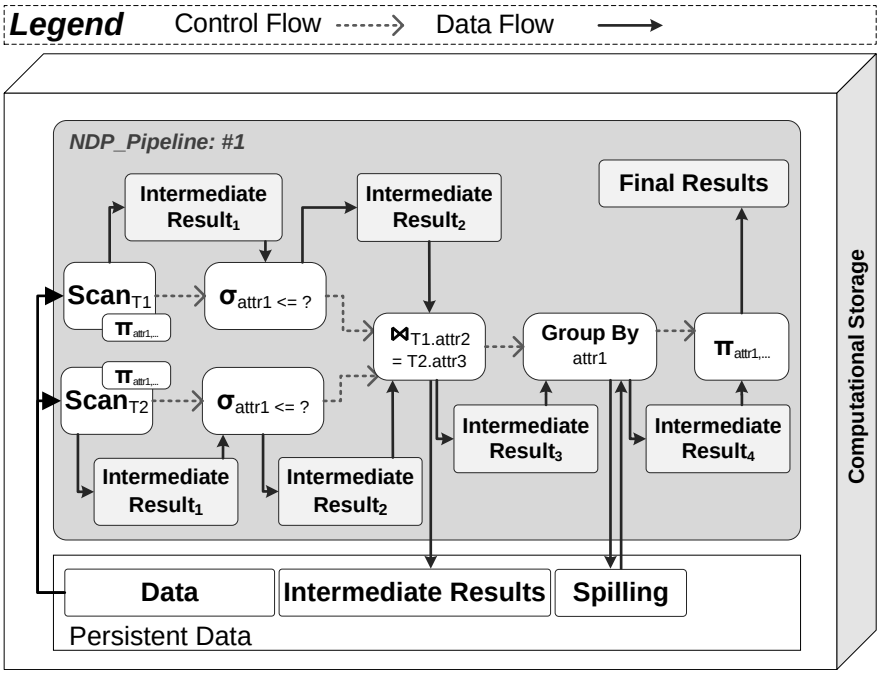


Figure 6.2: As part of this thesis, we propose NDP Pipelines as flexible demand-pull pipelines of operators that cache their intermediaries in buffer phases. This allows the execution of the operators on any of the heterogeneous processing capabilities. Moreover, operators can utilize persistent storage to materialize intermediary results and also spill data.

the best-fitting processing element, described in Section 4.2, without changing the NDP pipeline.

6.3 FINAL AND INTERMEDIARY RESULT-SET HANDLING

Key Insight: NDP pipelines produce final and intermediary results that need to be managed.

NDP pipelines, as presented in the previous Section 6.2, produce final but also intermediary results, which have to be managed. In particular, the latter have the characteristic that they have to be processed on-device. As a consequence, the computational storage device has to navigate within their layouts and interpret their formats similar to the original data of the database. Hence, the concept of Parsers and Accessors of Chapter 4 can be reused, but also have to be enhanced for writing out results in an expected format. In Publication P9, we revisit these concepts and extend them with result-set specific implementations.

Furthermore, we elaborate on the efficiency of transferring the final result-set to the host. Several approaches besides the well-known Block I/O are possible, as depicted in Figure 6.3.

BLOCK-LEVEL NDP: One approach to avoid reading pages that are discarded later by the execution engine, is to discard those pages already on-device. Thus, introducing NDP processing on block-

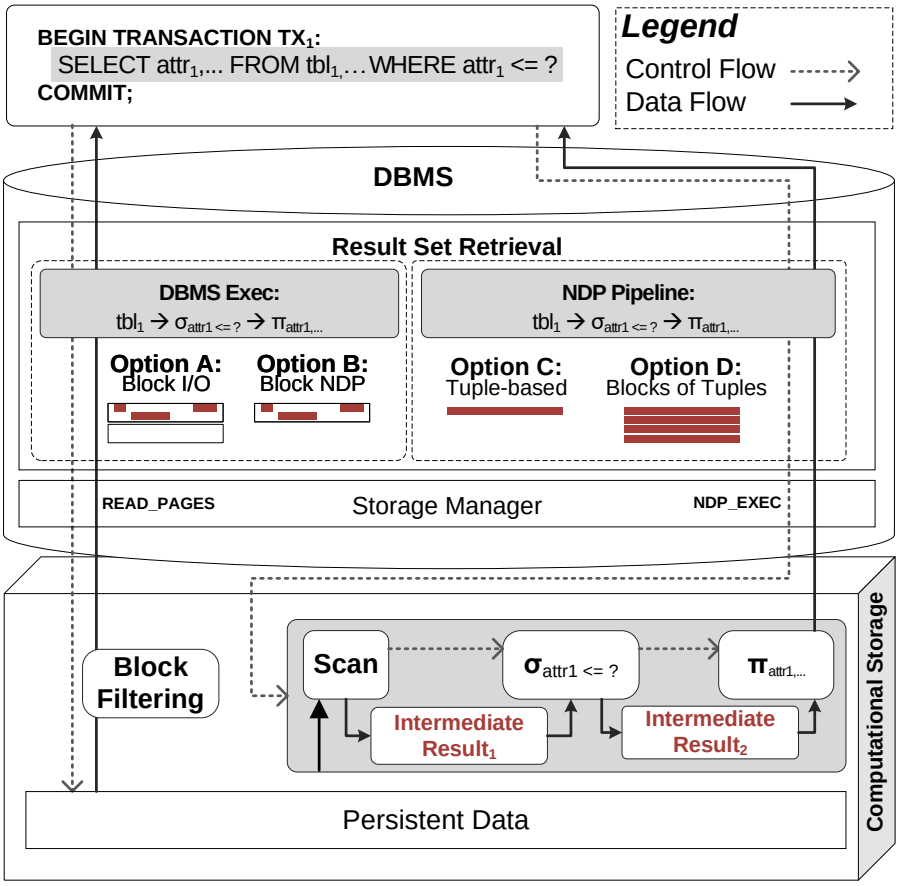


Figure 6.3: Results can be transferred from a storage device in several ways: (A) By using the traditional Block I/O, (B) by filtering some of the unnecessary pages on-device as Block NDP, (C) as an NDP pipeline tuple by tuple, or (D) even gathering multiple tuples in a block and transferring it as one.

level granularity (Figure 6.3, Option B), allows filtering the physical pages according to some criteria and transfer only the matching (e. g., BlockNDP [11, 12]). As a consequence, the bus system bandwidth is only spent for reading pages containing at least some relevant data. However, the physical pages also might comprise multiple records, sometimes even in different arrangements that constitute data unnecessary for the execution engine. Hence, the bus is still not fully leveraged efficiently.

TUPLE-BASED NDP: With NDP, it becomes viable to execute portions of a query execution plan, comprising multiple operators, on-device as discussed in the previous Section 6.2. Such NDP pipelines aim for reducing the overall data movement up to the host. One approach for transferring the data is to send them tuple by tuple (Figure 6.3, Option C), similar to a volcano-style execution engine [37]. This clearly eases the integration into volcano-style DBMS, and also allows for transferring only rele-

vant data. Unfortunately, this also entails a heavy communication overhead, especially with state-of-the-art bandwidth-optimized bus systems (PCIe) and protocols (NVMe), as shown in Publication P9.

BLOCKS OF TUPLES: To fully optimize the storage bus utilization by firstly transferring only relevant data, and secondly leveraging the bus and database engine properties, we propose an approach that can batch multiple result tuples into transfer units/blocks. In our approach, the size of these units can be configured from very small, to achieve a tuple-based behavior, up to very large, to align with present bus or system optimized settings, similar to the buffers between operands of NDP pipelines. Therefore, a pre-allocated set of on-device address locations is used and assigned to a certain format of tuples. Tuples are appended until the configured size is reached before the block is transferred.

Depending on the available I/O bus and system setup, one of the previously discussed final result-set handling techniques is beneficial. For instance, in the evaluation of Publication P9, we showed that transferring larger result-set blocks improves the performance by up to 27% for computational storage devices interconnected with a PCIe/NVMe communication C4.2.

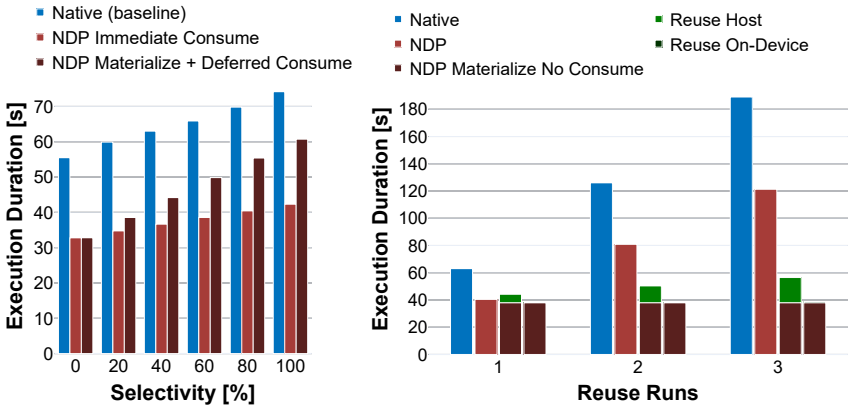


Figure 6.4: In Publication P9, the effects of consumption modes and reuse semantics on the execution performance of query are investigated. Clearly, on-device materialization comes at a low cost and can be reused to gain significant performance improvements.

Key Insight: On-device materialization of final or intermediary results introduces novel consumption modes and reuse semantics.

However, with NDP result-set management, it is also possible to materialize any kind of result-set (intermediary and final) on the storage device to reuse it in further recurring processings or to utilize large DMA transfers to improve data transfer rates. As a consequence, not only the processing of intermediary and final results is possible anymore but it also opens the way for novel concepts in terms of the `Consumption Mode` and `Reuse Semantics`. For instance, instead of

immediately consuming the results of an NDP pipeline (immediate consumption), they can be materialized and consumed to a later point in time (deferred consumption) and reused multiple times. C4.3 In the experimental evaluation of Publication P9, we elaborated the effects of those concepts on the performance of the query `SELECT id, type,...FROM nodetable WHERE type <= ?`. Figure 6.4 presents the numbers for the immediate and deferred consumption modes (left) and the multiple reuse of materialized data (right). The evaluation shows the marginal overhead of materializing results, and a significant performance improvement of 95% for reusing them on the host, and even $73 - 400\times$ for reusing them on-device. Especially the latter also benefits complex analytical calculations, which repeatedly require a pre-processed dataset and thus, increase the fault tolerance of such complex pipelines C4.4.

6.4 COMMUNICATION PROTOCOL AND STATE MACHINE

Besides the processing and result-set management, the orchestration of both on a computational storage device has not yet been deeply studied. In Publication P9, we propose a communication protocol that allows having necessary interactions with a host system, but avoids any unnecessary data transfers or costly roundtrips. Thereby, an `NDP Slot` can be reserved on-device as a single managing data structure for an entire NDP pipeline. In case a communication with the host is required, e. g., to transfer final results or to solve any resource constraint on the device, a `Host Interaction` is assigned to the `NDP Slot` that keeps all relevant information.

Key Insight: The on-device orchestration of NDP and result-set management is essential to transfer results while processing.

The `NDP Slots` and `Host Interactions` are orchestrated in an on-device finite state machine that can leverage today's widely-used multi-core processing units (see Section 2.3.4). Therefore, the finite state machine is split into a Managing Core that is responsible to coordinate the host-device communication and storage management, as well as multiple Processing Cores, which execute the NDP pipelines described in Section 6.2. A shared memory between Managing and Processing Cores allows for exchanging information. Combined with the `NDP Slots` and `Host Interactions`, this not only allows computational storage devices to process multiple NDP pipelines at once, but also paves the way for an interleaved pipeline execution where batches of final result-sets are transmitted to the host while new ones are continuously produced C4.5. This improves execution times by up to 30%, as shown in the experimental evaluation of Chapter 16.

» RECAP: INSIGHTS AND SOLUTIONS

Last but not least, we investigated in-situ result-set handling for NDP in Research Question RQ4. We firstly elaborated on different NDP

execution modes and identified NDP pipelines to be a flexible way of processing data in-situ, while leveraging the heterogeneous processing capabilities. In contrast, related work [11, 41] C4.1 either has a relatively static behavior or transfers data, not required in any subsequent processing. By introducing novel in-situ final and intermediary result set handling, we determined that the concrete implementation heavily depends on the I/O bus and system setup.

For NVMe communication over PCIe we achieved to improve the performance by up to 27% by transferring larger result-set blocks C4.2 . Additionally, our research has revealed that with on-device materialization, novel consumption modes and reuse semantics are feasible, and thereby, improved previous research on NDP query processing [105, 106]. In our experimental evaluation we speed-up execution durations by up to 400× C4.3 . Furthermore, reusing final or intermediary results also enhances the fault tolerance of complex pipelines, as they can be directly accessed, which is of high importance for analytical workloads with recurring calculations C4.4 . Investigations have shown that its orchestration in combination with the processing is essential to fully-leverage computational storage devices. Our proposed, interleaved NDP pipelines are managed by an on-device finite state machine and can process data while transferring the first results to the host, improving performance by up to 30% C4.4 .

Recapitulating, in the work of this thesis we investigated Near-Data Processing (NDP) for data-intensive systems by elaborating the physical storage management (Chapter 3) and on-device processing (Chapter 4), studying offloading models and transactional consistent NDP execution (Chapter 5) and finalized it with exploring the in-situ result-set management and its overall orchestration.

7 CONCLUSION AND OUTLOOK

7.1 CONCLUSION

Over the last decades, we observe an ever-increasing growth of data volume due to trends like Social Media, IoT, or scientific applications, which has to be managed by DBMSs. To keep the transactional throughput and execution times steady or even improve them, highly parallel processing units, like multi-core CPUs, and tremendously fast storage technologies, like NVM, are equipped in today's data-intensive systems. Yet, according to Amdahl's Law [4], the overall speedup of a system highly depends on the fraction of time the part, supported by that modern hardware, is actually used. Thus, we cannot fully leverage the increasingly fast processing units, as today's computer architecture easily become I/O bound.

Throughout the course of this thesis, we investigate the Central Research Question CRQ , elaborate novel concepts, and propose solutions for several aspects of an alternative computer architecture approach: With Near-Data Processing (NDP), we shift from a `data-to-code` toward a `code-to-data` paradigm.

First, we challenge today's storage management in Chapter 3 and conclude that due to its multi-layer approach, storage technologies might be easily exchangeable but they also introduce information hiding, which impairs processing on the device C1.1 . We believe to have found an essential prerequisite for NDP with Native Storage Management and investigate novel abstractions to extend the interface, according to Research Question RQ1. The Publications P1 P2 P3 of Part ii present initial approaches of the NDP interface extension, which is continuously extended with further features over the remainder of this thesis C1.2 . Moreover, we introduce Physical Page Pointers (PPPs) as one novel abstraction to tremendously reduce the exchange of logical-to-physical mapping information between host and device, and avoid expensive device-to-host round-trips for address resolution C1.3 . Overall, the foundation for on-device data processing is established.

Second, we address the problem of on-device navigation and interpretation as Research Question RQ2 in Chapter 4 and provide Publications P4 P5 P6 P7 in Part iii. Cross-layer data formats and layouts of databases are studied C2.1 and the concept of Parsers and Accessors as a solution is proposed C2.2 to avoid any costly host-device round-trips. Moreover, we investigate the utilization of heterogeneous processing hardware from modern computational storage devices C2.3 and achieve performance improvements of 1.4×

to 2.7× by executing portions of simple queries on an FPGA C2.4 . Thereby, the compute placement per NDP invocation and the hardware-software co-design play important roles. Yet, as the development of such accelerators is quite cumbersome, we also propose a framework to automatically generate those and lower the boundaries for hardware-accelerated NDP C2.5 . By this, we are now able to not only resolve data on-device but also to leverage the heterogeneous processing capabilities to navigate and interpret it.

Third, we focus on offloading models and their impact on the DBMS architectures in Research Question RQ3 . Hence, in Chapter 5, we present an overview on novel ways to offload processing to NDP-capable devices C3.1 . The concept of an intervention-free transaction execution of snapshot-based NDP is introduced in the Publication P8 as an indispensable prerequisite for processing frequently updating datasets on-device C3.2 . In particular, with respect to modern workloads like HTAP, this concept allows to process the most recent data on-device, and thus, provides even transactional guarantees C3.3. The experimental evaluation has shown that a performance improvement of up to 30% is possible while reducing the power consumption by 26% C3.4 . In general, our NDP approaches are now capable of processing the most recent data from the host as well as the large cold data persisted on the device with the available heterogeneous processing units of computational storage devices.

Lastly, we study possible NDP execution modes in Chapter 6 and propose NDP pipelines as flexible way of on-device data processing C4.1 . According to Research Question RQ4, we also have a look at their result-set management C4.2 . We introduce concepts for on-device materialization, which pave the way for novel concepts of consumption modes and reuse semantics. With those, we improve the execution times by up to 400× in the evaluation of the Publication P9 in Part v C4.3 . Likewise, we enhance the fault tolerance of complex queries as intermediate results can be easily accessed multiple times C4.4 , e. g., in analytics, Machine Learning (ML), or complex applications. Moreover, the orchestration of processing and result handling on computational storage devices is investigated and interleaved NDP pipeline execution is proposed to transfer final results in parallel to processing on-device C4.5 .

To recapitulate, we elaborate on Near-Data Processing (NDP) for data-intensive systems and propose valuable concepts and abstractions along all necessary phases of data processing in the context of this thesis – from reading the physical data, over leveraging heterogeneous processing capabilities, to navigate and interpret data. We propose novel offloading modes to ensure even transactional guarantees, and close by presenting the management of final and intermediate result-sets.

7.2 THE NDP PROBLEM SPACE AND FUTURE WORK

The major aspects of NDP for data-intensive systems are investigated and presented in this work. Yet, there is a lot more to explore in the broad range of NDP for future work. Summarized in Figure 7.1, there are interesting areas.

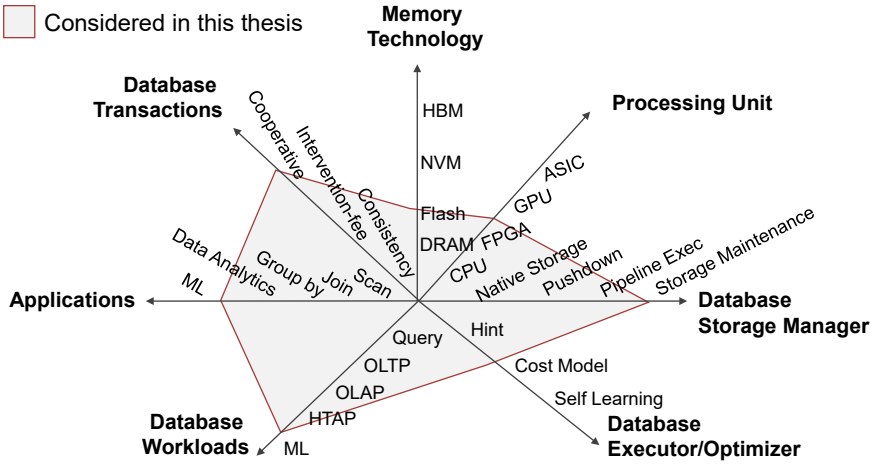


Figure 7.1: Throughout the course of this thesis, the fundamentals of the problem space for NDP in data-intensive systems are investigated. Yet, there is a lot more to explore.

For instance, new advances in the semiconductor industry broaden the spectrum of storage and memory technologies. While Dynamic Random Access Memory (DRAM) and Flash memories capture a charge for writing data and detect the voltage when reading, novel technologies like Phase Change Memory (PCM), Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM), or memristors are categorized as resistive memory. They need to have a pulsing current to write data but use the resistance for reading, which in turn, allows a higher density than DRAM, due to the large resistance ranges. In addition, they support byte-addressable accesses, which require new concepts for managing the large address space. With access latencies of up to 250 ns, they position themselves between DRAM and Flash in the memory hierarchy. Similarly, advances in 3D-Stacking of semiconductor memories yield extremely parallel technologies like HBM. As all of those memories can be closely combined with processing units, they are promising candidates to further improve NDP.

Moreover, this thesis has not deeply considered other processing units than CPUs and FPGAs. However, GPUs or ASICs have broad adoption in data centers and provide novel processing characteristics that have to be investigated in the context of NDP. Likewise, the utilization of FPGAs is addressed only on a superficial level in this thesis, and can be extended with other applications like ML.

In addition, it can be anticipated that computational storage devices will have a low-latency and cache-coherent interface, like CCIX [26] or CXL [27], besides the traditional high-bandwidth PCIe interconnect. In the context of offloading models and transactional consistency for NDP executions we already scratched the surface of cooperative transaction handling. Nevertheless, such modern shared memory architectures and protocols build the foundation for shared locking tables, and thus, extend the research area in terms of NDP that can also update the dataset on-device.

Furthermore, the storage manager of databases can be further improved by applying NDP. In particular storage maintenance operations are promising candidates as they constitute a significant amount in the data transfers between the storage device and host. For instance, compactions of append-based data structures, like LSM trees, could be executed completely asynchronously in-situ, relieving the processing units of the host as well as the bus system.

Lastly, the close linkage between NDP and the database executor or optimizer is not yet exhaustively investigated. First studies have shown the necessity for an adequate cost model that considers data movement between the device and host [58]. Another consideration could be to integrate NDP in the context of self-learning databases by utilizing ML to schedule processing in-situ.

BIBLIOGRAPHY

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. "H2O." In: *Proc. SIGMOD*. 2014, pp. 1103–1114. ISBN: 9781450323765. DOI: 10.1145/2588555.2610502. URL: http://15721.courses.cs.cmu.edu/spring2018/papers/10-storage/h2o.pdf%20http://dl.acm.org/citation.cfm?doid=2588555.2610502.

[3] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. "Tackling Hardware/Software co-design from a database perspective." In: *Proc. CIDR*. 2020.

[4] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

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

[6] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. "LinkBench: A Database Benchmark Based on the Facebook Social Graph." In: *Proc. SIGMOD*. 2013. ISBN: 978-1-4503-2037-5.

[7] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "Janus: A Hybrid Scalable Multi-Representation Cloud Datastore." In: *IEEE Trans. Knowl. Data Eng.* 30.4 (2018), pp. 689–702. ISSN: 10414347. DOI: 10.1109/TKDE.2017.2773607.

[8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. "Bridging the archipelago between row-stores and column-stores for hybrid workloads." In: *Proc. SIGMOD*. Vol. 26-June-20. 2016, pp. 583–598. ISBN: 9781450335317. DOI: 10.1145/2882903.2915231. URL: http://dx.doi.org/10.1145/2882903.2915231.

[9] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[10] John Backus. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579. URL: https://doi.org/10.1145/359576.359579.

[11] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. "BlockNDP: Block-storage near data processing." In: *Proc. Middlew.* 2020, pp. 8–15. ISBN: 9781450382014. DOI: 10.1145/3429357.3430519. URL: https://doi.org/10.1145/3429357.3430519.

[12] Antonio Barbalace and Jaeyoung Do. "Computational Storage: Where Are We Today?" In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

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

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

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

[16] Twitter Engineering Blog. *New Tweets per second record, and how!* URL: https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.

[17] Justus Bogner, Carolin Dehner, Tobias Vincon, and Ilia Petrov. "Real time charging database benchmarking." In: *Proceedings of the 17th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2015, Brussels, Belgium, December 11-13, 2015*. Ed. by Gabriele Anderst-Kotsis and Maria Indrawan-Santiago. ACM, 2015, p. 78. ISBN: 978-1-4503-3491-4. DOI: 10.1145/2837185.2837258. URL: http://doi.acm.org/10.1145/2837185.2837258.

[18] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: *Database Machines*. Ed. by A. R. Hurson, L. L. Miller, and S. H. Pakzad. Springer Berlin Heidelberg, 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

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

[20] Mokrane Bouzeghoub and Verónika Peralta. "A Framework for Analysis of Data Freshness." In: *In Proc. IQIS* (2004).

[21] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality." In: *Journal of Mathematical Sociology* (2001).

[22] Wei Cao et al. "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database." In: *Proc. FAST*. 2020, pp. 29–41.

[23] CL Philip Chen and Chun-Yang Zhang. "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data." In: *Information sciences* 275 (2014), pp. 314–347.

[24] CNET. *Facebook processes more than 500 TB of data daily*. URL: https://www.cnet.com/tech/services-and-software/facebook-processes-more-than-500-tb-of-data-daily/.

[25] Ampere Computing. *Ampere Altra 128 Core ARM*. URL: https://d1o0i0v5q5lp8h.cloudfront.net/ampere/live/assets/documents/Altra_Max_Rev_A1_DS_v1.05_20220728.pdf.

[26] CCIX Consortium. *CCIX*. URL: https://www.ccixconsortium.com/.

[27] CXL Consortium. *CXL*. URL: https://www.computeexpresslink.org/.

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

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

[30] David DeWitt and Jim Gray. "Parallel Database Systems: The Future of High Performance Database Systems." In: *Commun. ACM* 35.6 (June 1992), pp. 85–98. ISSN: 0001-0782.

[31] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. "Query processing on smart SSDs." In: *Proc. SIGMOD* (2013), p. 1221. ISSN: 07308078. DOI: 10.1145/2463676.2465295. URL: http://dl.acm.org/citation.cfm?doid=2463676.2465295.

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

[33] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview." In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33. URL: http://dblp.uni-trier.de/db/journals/debu/debu35.html%7B%5C#%7DFarberMLGMRD12%7B%5C%%7D5Cnhttp://sites.computer.org/debull/A12mar/issue1.htm.

[34] Michael J. Flynn. *Very High-Speed Computing Systems*. 1966. DOI: 10.1109/PROC.1966.5273.

[35] Michael J. Flynn. "Some computer organizations and their effectiveness." In: *IEEE Transactions on Computers* C-21 (9 1972), pp. 948–960. ISSN: 00189340. DOI: 10.1109/TC.1972.5009071.

[36] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. "Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads." In: *Proc. VLDB Endow.* Vol. 8. 12. 2015, pp. 1716–1727. DOI: 10.14778/2824032.2824069.

[37] Goetz Graefe. "Volcano-An Extensible and Parallel Query Evaluation System." In: *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING* 6 (1 1994).

[38] Goetz Graefe. "Partitioned B-trees-a user's guide." In: *In Proc. BTW* (2003).

[39] Jim Gray and Prashant J. Shenoy. "Rules of Thumb in Data Engineering." In: *Proc. ICDE*. 2000, p. 3.

[40] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. "HYRISE-A main memory hybrid storage engine." In: *Proc. VLDB Endow.* 4.2 (2010), pp. 105–116. ISSN: 21508097. DOI: 10.14778/1921071.1921077.

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

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

[43] Sergey Hardock, Andreas Koch, Tobias Vincon, and Ilia Petrov. "IPA-IDX: In-Place Appends for B-Tree Indices." In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. DaMoN'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450368018. DOI: 10.1145/3329785.3329929. URL: https://doi.org/10.1145/3329785.3329929.

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

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

[46] Martin Hilbert and Priscila López. "The world's technological capacity to store, communicate, and compute information." In: *science* 332.6025 (2011), pp. 60–65.

[47] Mark D Hill and Michael R Marty. "Amdahl's Law in the Multicore Era." In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 0018-9162. DOI: 10.1109/MC.2008.209.

[48] IBM. *DB2 Databases on RAW*. URL: https://www.ibm.com/docs/en/db2/9.7?topic=creation-attaching-dms-direct-disk-access-devices.

[49] IDC and Statista. *Worlwide Data Created*. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

[50] Intel. *Intel Xeon Phi 72 Core*. URL: https://ark.intel.com/content/www/de/de/ark/products/95830/intel-xeon-phi-processor-7290-16gb-1-50-ghz-72-core.html.

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

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

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

[54] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *Proc. ICDE*. 2011, pp. 195–206. ISBN: 9781424489589. DOI: 10.1109/ICDE.2011.5767867.

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

[56] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. "In-storage processing of database scans and joins." In: *Inf. Sci.* 327 (Jan. 2016), pp. 183–200. ISSN: 00200255. DOI: 10.1016/j.ins.2015.07.056. URL: http://dx.doi.org/10.1016/j.ins.2015.07.056%20https://linkinghub.elsevier.com/retrieve/pii/S0020025515006003.

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

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

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

[60] Tirthankar Lahiri et al. "Oracle Database In-Memory: A dual format in-memory database." In: *Proc. - Int. Conf. Data Eng.* 2015-May (2015), pp. 1253–1258. ISSN: 10844627. DOI: 10.1109/ICDE.2015.7113373.

[61] Per Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. "Real-time analytical processing with SQL server." In: *Proc. VLDB Endow.* Vol. 8. 12. 2015, pp. 1740–1751. DOI: 10.14778/2824032.2824071.

[62] Juchang Lee, Wook Shin Han, Hyoung Jun Na, Chang Gyoo Park, Kyu Hwan Kim, Deok Hoe Kim, Joo Yeon Lee, Sang Kyun Cha, and Seung Hyun Moon. "Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases." In: *VLDB J.* 27.3 (2018), pp. 421–444. ISSN: 0949877X. DOI: 10.1007/s00778-018-0503-z.

[63] Clifford Lynch. "Big Data: How do your data grow?" In: *Nature* 455 (Oct. 2008), pp. 28–9. DOI: 10.1038/455028a.

[64] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. "BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications." In: *Proc. SIGMOD*. Vol. Part F1277. 2017, pp. 37–50. ISBN: 9781450341974. DOI: 10.1145/3035918.3035959. URL: http://dx.doi.org/10.1145/3035918.3035959.

[65] Bernard Marr and Co. *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*. URL: https://bernardmarr.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/.

[66] Micron. *232-Layer NAND*. URL: https://investors.micron.com/news-releases/news-release-details/micron-ships-worlds-first-232-layer-nand-extends-technology.

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

[68] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "ScyPer: a hybrid OLTP&OLAP distributed main memory database system for scalable real-time analytics." In: *Datenbanksysteme für Business, Technologie und Web (BTW) 2044*. Ed. by Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 499–502.

[69] Nvidia. *Nvidia Titan V*. URL: https://www.nvidia.com/en-us/titan/titan-v/.

[70] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Inform.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903.

[71] Oralce. *Oracle Cluster File System*. URL: https://docs.oracle.com/cd/B19306_01/install.102/b14203/storage.htm#RILIN1024.

[72] Oralce. *Oracle Databases on RAW*. URL: https://docs.oracle.com/cd/B19306_01/install.102/b14203/storage.htm#RILIN1072.

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

[74] Meta Platforms. *RocksDB*. URL: http://rocksdb.org/.

[75] prodesign. *prodesign HAWK*. URL: https://www.prodesign-fpga-acceleration.com/products/prodesign-hawk-vc1902-acceleration-card.

[76] Vijayshankar Raman, Gopi Attaluri, and Ronald Barber. "DB2 with BLU Acceleration: So much more than just a column store." In: *Proc. VLDB* 6.11 (2013), pp. 1080–1091. ISSN: 2150-8097. DOI: 10.14778/2536222.2536233. URL: https://researcher.watson.ibm.com/researcher/files/us-ipandis/vldb13db2blu.pdf%20http://dl.acm.org/citation.cfm?id=2536233.

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

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

[79] Christian Riegger, Tobias Vincon, and Ilia Petrov. "Multi-Version Indexing and Modern Hardware Technologies: A Survey of Present Indexing Approaches." In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS '17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 266–275. ISBN: 9781450352994. DOI: 10.1145/3151759.3151779. URL: https://doi.org/10.1145/3151759.3151779.

[80] Christian Riegger, Tobias Vincon, and Ilia Petrov. "Write-Optimized Indexing with Partitioned b-Trees." In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS '17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 296–300. ISBN: 9781450352994. DOI: 10.1145/3151759.3151814. URL: https://doi.org/10.1145/3151759.3151814.

[81] Christian Riegger, Tobias Vincon, and Ilia Petrov. "Efficient Data and Indexing Structure for Blockchains in Enterprise Systems." In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS2018. Yogyakarta, Indonesia: Association for Computing Machinery, 2018, pp. 173–182. ISBN: 9781450364799. DOI: 10.1145/3282373.3282402. URL: https://doi.org/10.1145/3282373.3282402.

[82] Christian Riegger, Tobias Vincon, and Ilia Petrov. "Indexing Large Updatable Datasets in Multi-Version Database Management Systems." In: *Proceedings of the 23rd International Database Applications and Engineering Symposium*. IDEAS '19. Athens, Greece: Association for Computing Machinery, 2019. ISBN: 9781450362498. DOI: 10.1145/3331076.3331118. URL: https://doi.org/10.1145/3331076.3331118.

[83] Christian Riegger, Tobias Vinçon, Robert Gottstein, and Ilia Petrov. "MV-PBT: Multi-version indexing for large datasets and HTap workloads." In: *Adv. Database Technol. - EDBT*. Vol. 2020-March. 2020, pp. 217–228. ISBN: 9783893180837.

[84] Philip E. Ross. "Why CPU Frequency Stalled." In: *IEEE Spectrum* 45.4 (2008), pp. 72–72. DOI: 10.1109/MSPEC.2008.4476447.

[85] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. "L-Store: A real-time OLTP and OLAP system." In: *Proc. EDBT*. Vol. 2018-March. 2018, pp. 540–551. ISBN: 9783893180783. DOI: 10.5441/002/edbt.2018.65. arXiv: 1601.04084.

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

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

[88] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. "Cosmos+ OpenSSD: A NVMe-based Open Source SSD Platform." In: *Flash Memory Summit* (2016).

[89] Statista. *Hours of video uploaded to YouTube every minute as of February 2020*. URL: https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/.

[90] Alex Szalay and Jim Gray. "Science In An Exponential World." In: *Nature* 440.23 (Mar. 2006). URL: https://www.microsoft.com/en-us/research/publication/science-in-an-exponential-world/.

[91] Sajjad Tamimi, Florian Stock, Andreas Koch, Arthur Bernhardt, and Ilia Petrov. "An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing." In: *Proc. FCCM*. 2022.

[92] 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.

[93] Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *Proc. HardBD @ ICDE 2020*. 2020.

[94] Tobias Vincon, Christian Knödler, Arthur Bernhardt, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch, and Ilia Petrov. "Result-Set Management for NDP Operations on Smart Storage." In: *Proc. DaMoN*. 2022.

[95] Tobias Vincon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. "Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads." In: *PVLDB* 15 (2022).

[96] Tobias Vincon and Ilia Petrov. "Near Data Processing within Column-Oriented DBMSs for High Performance Analysis." In: June 2016. DOI: 10.13140/RG.2.1.1596.5687.

[97] Tobias Vincon, Ilia Petrov, and Christian Thies. "cIPT: Shift of Image Processing Technologies to Column-Oriented Databases." In: *New Trends in Databases and Information Systems*. Springer International Publishing, 2016. ISBN: 978-3-319-44066-8.

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

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

[100] Tobias Vinçon, Sergey Hardock, Christian Riegger, Andreas Koch, and Ilia Petrov. "nativeNDP: Processing Big Data Analytics on Native Storage Nodes." In: 2019.

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

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

[103] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System." In: *In Proc. OSDI* (2006).

[104] Ludwig Wittgenstein. *On Certainty*. New York and London: Harper Torchbooks, 1969.

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

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

[107] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious." In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588. URL: https://doi.org/10.1145/216585.216588.

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

Part II

NDP ABSTRACTIONS FOR PHYSICAL STORAGE MANAGEMENT

8

NATIVE STORAGE TECHNIQUES FOR DATA MANAGEMENT

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"Native Storage Techniques for Data Management"* by Ilia Petrov, Andreas Koch, Sergey Hardock, Tobias Vinçon and Christian Riegger in 2019 IEEE 35th International Conference on Data Engineering (ICDE). The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding author, Tobias Vinçon has contributed with the development of native storage abstractions and the implementation details of NoFTL in KV Stores. To the same extent, Sergey Hardock supported in the discussions on native storage techniques and provided implementations details of NoFTL in traditional relational databases. Ilia Petrov was responsible for the foundational concepts of native storage and the manuscript's text with feedback from all authors including Andreas Koch and Christian Riegger. «*

ABSTRACT

In the present tutorial we perform a cross-cut analysis of database storage management from the perspective of modern storage technologies. We argue that neither the design of modern DBMS, nor the architecture of modern storage technologies are aligned with each other. Moreover, the majority of the systems rely on a complex multi-layer and compatibility-oriented storage stack. The result is needlessly suboptimal DBMS performance, inefficient utilization, or significant write amplification due to outdated abstractions and interfaces. In the present tutorial we focus on the concept of *native storage*, which is storage operated without intermediate abstraction layers over an open native storage interface and is directly controlled by the DBMS. We cover the following aspects of native storage: (i) architectural approaches and techniques; (ii) interfaces; (iii) storage abstractions; (iv) DBMS/system integration; (v) in-storage processing.

8.1 OUTLINE

In the present tutorial we examine the influence of *native storage* on data-intensive systems and data management. We begin with a succinct description of the concept of native storage and a brief summary

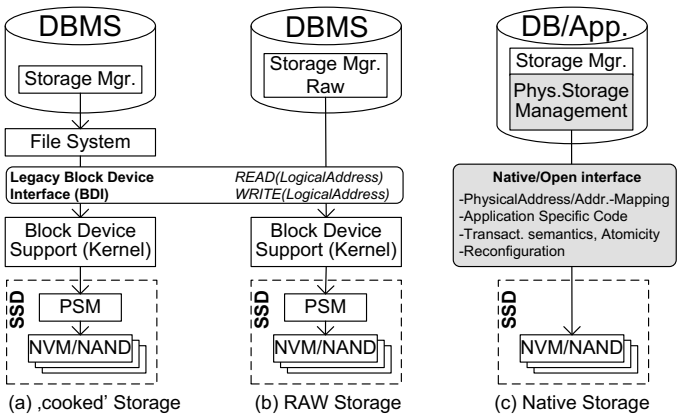


Figure 8.1: DBMS storage alternatives: (a) Traditional 'cooked' DBMS storage; (b) DBMS on RAW volumes/devices; (c) DBMS on Native Storage

of the characteristics of modern storage technologies. The main focus of this tutorial is on their influence on different aspects of data management. *Firstly*, we describe different architectural approaches and techniques for defining and using native storage. Existing native storage systems pursue different goals and result in various architectural blueprints. These range from small sensor data or graph processing to enterprise scenarios and in-storage processing. *Secondly*, we present a cross-cut analysis of different proposals for native storage *interfaces*. These are very different from the currently dominating compatibility block-device interface. *Thirdly*, novel storage abstractions are needed with open native storage interfaces in place. *Fourthly*, we analyze the different approaches for system integration of native storage and native storage interfaces. *Last but not least*, having considered all these aspects in isolation we provide a wrap-up in terms of data management techniques for native storage. The target audience is database researchers and practitioners with interests in storage management on modern storage hardware.

8.2 NATIVE STORAGE AND DATA MANAGEMENT

Over the last decade we witnessed several important breakthroughs in storage technologies: Flash and Flash SSDs have become omnipresent as database storage; Non-Volatile Memories (e.g., PCM or Intel/Micron's 3D XPoint) are gradually becoming real. Although these have very different characteristics from old-fashioned magnetic HDD storage, they are still treated as fast replacements and are embedded in a classical *"cooked"* storage stack. This typically comprises multiple layers: a low-level compatibility translation layer (typically running on-device), a block-device, OS kernel support, and a file system [10, 22]. In fact, file-systems are considered part of the NVM I/O stack even for non-volatile main-memory settings [1, 29]. Even though a compati-

bility stack fosters proliferation of modern storage technologies and simplifies systems development, it has a number of disadvantages [3, 5, 14, 23]: (i) performance lags and underutilization of SSD resources ([6, 10, 14, 30] report lags of several times); (ii) no cross-layer optimizations due to layered abstractions and information hiding as well as the rigidness of compatibility interfaces; (iii) I/O (write-/read-) amplification of several times in terms of size and count, leading to lower performance and faster wear [14, 21]; (iv) functional redundancy along the I/O stack [10, 21]; (v) inability to configure the semiconductor storage adaptively (depending on the application or the workload) [2, 22, 26]. In fact, with all of the above in place, [22] reports that only 40% of the raw Flash bandwidth is delivered to applications and that to achieve high performance only, 50%-70% of the raw capacity is effectively available to applications (the rest is reserved for write handling and error correction).

8.2.1 *Architectural Approaches and Techniques*

In pursuit of a better I/O stack design, the use of alternative architectural blueprints seem unavoidable [3, 5, 23]. Opening up storage confronts systems and applications with different physical/low-level abstractions, evolving interfaces, and the question of how to distribute physical storage management along the stack. *Physical storage management – PSM* (e.g. FTL for Flash SSDs) encompasses: (i) the logical-to-physical address mapping (L2PAM); (ii) wear-levelling; (iii) error correction (ECC) and bad-block management; (iv) physical metadata management, (v) physical garbage collection and storage optimizations; and (vi) write management. In a traditional I/O stack, *PSM* is typically performed on-device, hidden behind the compatibility block-device interface.

Various native storage architectures exist. LightNVM and open-channel SSDs envisage a host-based, shared L2PAM table and handle physical metadata management, GC and write optimizations, as well as wear-levelling, on the host as part of the LightNVM subsystem. BlueDBM [16] and NoFTL [10] likewise assume a native storage, which however is extensible. NoFTL [10] and NoFTL-KV [28] have proposed open native storage and follow the approach of deep database integration, which explores coherent integration of PSM in different modules of the DBMS and investigates algorithmic improvements and cross-layer optimizations. Architecturally, native storage can be realized as host-based storage [10, 28] or distributed storage [13, 16, 25].

Early proposals for an I/O stack redesign gravitate around the concept of *bimodal* [5] or *multi-modal storage*. Under *bimodal storage*[5] if the DBMS issues "constrained" I/O patterns, i.e. no in-place updates and no random writes, the SSD uses a minimal FTL, while for all "unconstrained" I/O patterns, the SSD switches to traditional full-scale FTL

[20]. Application Managed Flash (AMF) [19] explores append-based storage management over an extended block-device interface under a novel append-based file system. AMF explores the architectural coupling of free space management to physical GC, in the same time leaving ECC, bad-block management, and wear leveling on device. [17, 21, 30] assume partial exposure of the mapping (which still resides on device) to applications. Their design goals are to handle storage virtualization [30], storage management for append-mostly systems [21], or to explore transactional atomicity [17]. Some complementary aspects of native storage architectures are *reconfigurability* and *intelligent storage/In-Storage Processing (ISP)* [7, 8, 15, 18, 24, 27, 31, 32]. In-situ execution of data processing operations minimizes data movement, leverages internal storage characteristics (parallelism, bandwidth, on-device CPU/FPGA), to achieve performance improvement of several times as well as better resource and energy efficiency.

8.2.2 *Interfaces*

Current hardware and software interfaces to memory and storage are rudimentary. Such interfaces: (i) have limited support for parallelism and concurrency, hence they limit system bandwidth and throughput; (ii) are relatively rigid with limited extensibility mechanisms; (iii) have only a limited set of capabilities and expose outdated abstractions. Building on top of such outdated abstractions demands layers of backwards compatibility, which prevent algorithms and system architectures from efficiently using modern storage technologies as their true characteristics are masked. On the hardware level, for instance, the traditional RAS/CAS DRAM architecture offers limited parallelism, while increasing the number of DIMMs per channel typically decreases performance. Similar behavior is exhibited by SATA. Block Device Interfaces (BDI) and block I/O are ubiquitous, yet they are a major bottleneck [3, 23] as they do not match the properties of modern storage technologies and require: immutable logical addresses; fixed I/O granularity; a rigid set of operations, mainly read/write; symmetric and wear-proof storage.

The *basic native storage interface* typically comprises *READ_PAGE*, *WRITE_PAGE* and *ERASE_BLOCK* commands [4, 10, 16]. These are defined on *physical addresses*. Since overwriting is an issue on modern storage technologies (wear, erase-before-overwrite), physical addresses need to change. This is a stark contrast to the BDI commands that rely on immutable *logical addresses* (LBA), and raises the issue of logical-to-physical address mapping (L2PAM). Various extensions to the basic interface are proposed by different systems. LightNVM [4], for example, suggests that the above are vectored commands, i.e. they take sets or ranges of physical addresses as arguments, instead of a single address. NoFTL [10] suggests extensions such as *write_delta*

for writes of sub-page granularity, *copyback* to reduce data transfers incurred by the garbage collector, or *get_addr_table* to speed-up L2PAM table recovery.

I/O atomicity is an open issue in the traditional I/O stack, but it becomes viable with native storage and modern storage technologies. The key is the ability to control the L2PAM table and GC so that old physical pages and their address mapping entries are retained, while new contents are being written on a *different* physical location. Only if the write sequence succeeds as a whole the old address mappings are completely and atomically replaced with the new ones. I/O atomicity has inspired various architectural designs regarding *copy-on-write* (CoW) storage management and logging. [23] introduced a new I/O primitive "atomic-write", however without support for concurrency. [17] suggests transaction-awareness, allowing the DBMS to notify the SSD about the beginning and end of transactions. The *SHARE* [21] interface to Flash targets atomicity and CoW and proposes the *share(*$\texttt{LogicalAddr}_1$, $\texttt{LogicalAddr}_2$*)* command to allow two logical page numbers to be mapped on the same physical page. *SHARE* is defined to be a variant of *TRIM* that has also been explored in ANViL [30] predecessors called *ptrim()*.

Another aspect is the management of the *logical-to-physical address mapping*. With native storage it can be: (a) host- or device resident, depending on the available resources; (b) completely exposed (and managed by the application), or partially exposed (through special commands) but managed by PSM. [4, 10, 16] assume full exposure and host-based L2PAM table management. [10] investigates full DBMS integration. [17, 21, 30] assume partial exposure of the mapping (which still resides on device) to applications. This yields new commands and abstractions. ANViL [30] considers exposing the logical-to-physical address mapping table to applications and proposes commands such as *clone()*, *move()* or *delete()*.

In-Storage Processing (ISP), targets the execution of application/system specific functionality in-situ, and is an additional factor for interface extensions. [8] defines a new session-based DBMS-SmartSSD communication protocol, comprising operations like OPEN, CLOSE, GET, and a set of APIs for on-device functionality, such as Command API, Thread API, Data API and Memory API. Willow [25] proposes similar concepts for a user-programmable SSD. IBEX [31, 32] investigates a DB-record based interface.

8.2.3 *Abstractions*

As an open native storage interface replaces traditional BDI, the different physical organization of native storage is exposed to the DBMS. Typically, native storage comprises chips, channels, the on-device controller, and its resources. Adapting storage management and data

processing for this type of organization is non-trivial. Therefore there is a pressing need for new storage abstractions that ease the DBMS integration of native storage.

AMF [19] assumes flash *blocks* and contiguous *segments*. Segments are introduced as a unit of allocation and physical distribution to achieve better bandwidth. More importantly AMF segments need to be explicitly deallocated by *TRIM* to physically reclaim the occupied space. A segment is subdivided into *sectors* that are a unit of I/O. As AMF targets append-based storage, a sector can never be overwritten (unless the whole segment is deallocated).

NoFTL introduces the concepts of *regions* and *groups* to manage native storage [12]. A *NoFTL region* comprises a set of physical chips and data channels as well as physical storage management strategies, such as address management, garbage collection and data placement. Regions are coupled to standard DBMS logical storage structures such as segments or tablespaces. A storage device is thus viewed and maintained by the DBMS as a set of regions. Every database object is then assigned to a certain region based on its properties, while every region can hold multiple objects (i.e. one-to-many relationship). *Groups* [12] serve as means to improve hot/cold data separation and thus decrease unnecessary GC activity, reduce erases and improve performance and longevity.

ANViL [30] proposes the *snapshot* at the level of a volume or a file as a native storage abstraction. A snapshot allows to checkpoint the state of a file/volume with little space and performance overhead, as only mapping entries are cloned. ANViL [30] also introduces *deduplication* as an abstraction based on *range cloning* that identifies and collapses identical blocks.

8.2.4 *System Integration*

There are different approaches to integrate native storage and physical storage management (PSM) into the DBMS or other applications. We distinguish *partial integration* and *deep integration*. Many approaches offer non-intrusive *partial integration* of native storage and rely on multi-modal native storage interfaces. Such systems tend to preserve existing I/O interface, however they also incorporate new features as extensions. Systems such as AMF [19], SHARE [21], XFTL [17] represent partial integration. Some of the advantages are the high degree of reuse, proliferation, and low implementation footprint. Furthermore, some of the native storage systems such as BlueDBM [16] or ANViL [30] offer multi-modal interfaces, leaving it to application to decide on the use.

Systems such as [10, 33] support *deep integration*, i.e. the PSM is coherently integrated into different modules of the system. The key insight is that deep integration results in a surprisingly simple and

lightweight implementation. This is the case, since many DBMS subsystems *already* implement similar functionality, which only needs to be leveraged and extended for deep integration.

8.2.5 *Reconfigurability*

Storage built on top of semiconductor storage technologies can be dynamically reconfigured depending on the workload. This type of *reconfigurability* can be applied on various levels: physical storage; on-device controller/processor; native storage abstractions. [2, 26] explore reconfigurability in compatibility storage settings, whereas [22, 25] consider native storage. For *native storage abstractions*, NoFTL [9] offers an advisor to derive region properties from I/O access patterns to different DB-objects and an offline transformer. On the level of *physical semiconductor memory* it is possible to configure an MLC Flash chip in SLC mode, to achieve near-SLC performance and endurance properties on that chip. With the recently introduced QLC Flash transitions from QLC to TLC to MLC and SLC may be envisaged. NoFTL [9, 11] has the ability to perform this reconfiguration on the level of a *region* for a single DB-Object in *pseudo-SLC, MLC* or *TLC modes*.

8.2.6 *In-Storage Processing*

The ability to execute application/system specific functionality in-situ (*In-Storage Processing – ISP*) is a very relevant trend and a revival of past ActiveDisc/DatabaseMachines efforts. [8] is one of the first works to explore offloading parts of data processing on Smart SSDs, indicating the potential of significant performance improvements of up to 2.7x and energy savings of up to 3x. [8] defines a new session-based DBMS-SmartSSD communication protocol, comprising operations like OPEN, CLOSE, GET, and a set of APIs for on-device functionality, such as Command API, Thread API, Data API and Memory API. Willow [25] proposes similar concepts for a user-programmable SSD. [8] identifies two research questions: (i) how can ISP handle the problem of on-device processing in the presence of a more recent version of the data in the buffer; and (ii) what is the efficiency of operation pushdown in the presence of large main memories.

The initial ideas of [8] have recently been extended in a complementary approach called *In-Storage Processing/Computing* [15, 18, 24]. [18] demonstrates a performance improvement of 5x and 47x for scans and joins on embedded CPUs. Further approaches stress the importance of in-situ analytical processing on on-device stream processors or embedded CPUs [7, 27]. Still, all of the above target *read-only ISP*, assuming that the on-device data is immutable.

IBEX investigates how data processing on FPGAs can be used as explicit co-processors, or implicitly as part of an intelligent storage

system [31]. IBEX exploits reconfigurable computing and the capabilities of custom-hardware to accelerate certain database operations. Yet, operations are not performed *in-situ* as data and results need to be transferred form storage to the FPGA and vice versa.

8.2.7 *Data Management on Native Storage*

In this part of the tutorial we provide a brief overview on recent solutions in the industry and academia for architecting, organizing, and utilizing native storage. The two major directions here are (i) the utilization of storage-specific out-of-place update strategy, as well as (ii) usage of database semantics for optimizations in FTL (e.g., reconfigurable SSDs).

8.3 BIOGRAPHIES OF THE PRESENTERS

Ilia Petrov is a professor at Reutlingen University, Germany and a head of the Data Management Lab. His research focus is on data management on modern hardware. He holds a Ph.D. from the University of Erlangen-Nürnberg.

Andreas Koch is a professor at the Technische Universität Darmstadt, Germany and head of the Embedded Systems and Applications Group (ESA). His research interests are in the area of specialized computing systems ranging from low power embedded systems up to accelerators for data-center high-performance computers.

Sergej Hardock is a Ph.D. student at the Databases and Distributed Systems Group at the Technische Universität Darmstadt, Germany. His research interests are in database systems on modern hardware, native Flash database storage, lean Flash-aware database systems.

Tobias Vincon is a Ph.D. student at the Data Management Lab at Reutlingen University, Germany and at the Technische Universität Darmstadt. His research interests are in database systems on modern hardware and Near-Data Processing.

Christian Riegger is a Ph.D. student at the Data Management Lab at Reutlingen University, Germany. His research interests are on indexing large datasets with high-bandwidth continuous insertions.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems." In: *Proc. SIGMOD*. 2015.

[2] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. "A Design for High-performance Flash Disks." In: *SIGOPS Oper. Syst. Rev.* (2007).

[3] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. "The Necessary Death of the Block Device Interface." In: *Proc. CIDR*. 2013.

[4] Matias Bjørling, Javier González, and Philippe Bonnet. "LightNVM: The Linux Open-channel SSD Subsystem." In: *Proc. USENIX/FAST*. 2017.

[5] Philippe Bonnet and et al. "Flash Device Support for Database Management." In: *Proc. CIDR*. 2011.

[6] Feng Chen, Binbing Hou, and Rubao Lee. "Internal Parallelism of Flash Memory-Based Solid-State Drives." In: *ACM ToS* 12.3 (2016).

[7] Sangyeun Cho, Chanik Park, Hyunok Oh, and et al. "Active Disk Meets Flash: A Case for Intelligent SSDs." In: *Proc. ICS*. 2013.

[8] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, David J. DeWitt, and et. al. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD*. 2013.

[9] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. "Selective In-Place Appends for Real: Reducing Erases on Wear-prone DBMS Storage." In: *Proc. ICDE*. 2017.

[10] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. "NoFTL: Database Systems on FTL-less Flash Storage." In: *Proc. VLDB* 6.12 (2013).

[11] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. "From In-Place Updates to In-Place Appends: Revisiting Out-of-Place Updates on Flash." In: *Proc. SIGMOD*. 2017.

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

[13] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou: Intelligent Distributed Storage." In: *Proc. VLDB Endow.* (2017).

[14] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. "I/O Stack Optimization for Smartphones." In: *Proc. USENIX/ATC*. 2013.

[15] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, and et al. "YourSQL: A High-performance Database System Leveraging In-storage Computing." In: *Proc. VLDB* (2016).

[16] Sang-Woo Jun, Ming Liu, Sungjin Lee, Arvind, and et al. "BlueDBM: Distributed Flash Storage for Big Data Analytics." In: *ACM TOCS* 34.3 (2016).

[17] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. "X-FTL: Transactional FTL for SQLite Databases." In: *Proc. SIGMOD*. 2013.

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

[19] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. "Application-managed Flash." In: *Proc. USENIX/FAST*. 2016.

[20] Dongzhe Ma, Jianhua Feng, and Guoliang Li. "A Survey of Address Translation Technologies for Flash Memories." In: *ACM Comput. Surv.* 46.3 (2014).

[21] Gihwan Oh, Chiyoung Seo, Sang-Won Lee, and et al. "SHARE Interface in Flash Storage for Relational and NoSQL Databases." In: *Proc. SIGMOD*. 2016.

[22] Jian Ouyang, Shiding Lin, Song Jiang, and et al. "SDF: Software-defined Flash for Web-scale Internet Storage Systems." In: *Proc. ASPLOS*. 2014.

[23] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, and David Flynn. "Beyond block I/O: Rethinking traditional storage primitives." In: *Proc. HPCA*. 2011.

[24] Samsung. *In storage computing*. 2015. URL: http://www.flashmemorysummit.com/English/Collaterals/Proceedings/%5C%5C2015/20150813%5C_S301D%5C_Ki.pdf (visited on 10/08/2018).

[25] Sudharsan Seshadri, Mark Gahagan, and et al. "Willow: A User-programmable SSD." In: *Proc. OSDI*. 2014.

[26] Ji-Yong Shin, Zeng-Lin Xia, and et al. "FTL Design Exploration in Reconfigurable High-performance SSD for Server Applications." In: *Proc. ICS*. 2009.

[27] Devesh Tiwari, Simona Boboila, and et al. "Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines." In: *Proc. FAST*. 2013.

[28] Tobias Vincon, Sergey Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. "NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management." In: *Proc. EDBT*. 2018.

[29] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, and et al. "Aerie: Flexible File-system Interfaces to Storage-class Memory." In: *Proc. EuroSys*. 2014.

[30] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "ANViL: Advanced Virtualization for Modern Non-volatile Memory Devices." In: *Proc. USENIX/FAST*. 2015.

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

[32] Louis Woods, Jens Teubner, and Gustavo Alonso. "Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances." In: *Proc. SIGMOD*. 2013.

[33] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. "Bluecache: A Scalable Distributed Flash-based Key-value Store." In: *Proc. VLDB*. 2016.

9

NOFTL-KV: TACKLING WRITE-AMPLIFICATION ON KV-STORES WITH NATIVE STORAGE MANAGEMENT

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management"* by Tobias Vinçon, Sergey Hardock, Christian Riegger, Julian Oppermann, Andreas Koch and Ilia Petrov in 2018 21st International Conference on Extending Database Technology (EDBT). The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Tobias Vinçon was responsible for the design and implementation of NoFTL-KV with essential support from Sergey Hardock. Moreover, the experimental evaluation was executed by him and supervised by Ilia Petrov. Likewise, he was in charge of the the text with intensive feedback from all authors including Christian Riegger, Julian Oppermann and Andreas Koch. «*

ABSTRACT

Modern persistent Key/Value stores are designed to meet the demand for high transactional throughput and high data-ingestion rates. Still, they rely on backwards-compatible storage stack and abstractions to ease space management, foster seamless proliferation and system integration. Their dependence on the traditional I/O stack has negative impact on performance, causes unacceptably high write-amplification, and limits the storage longevity.

In the present paper we present NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly consume the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB. The performance evaluation under LinkBench shows that NoFTL-KV improves transactional throughput by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves storage longevity by imately the same factor.

9.1 INTRODUCTION

Over the last decade, various specialized DBMSs have been intensively investigated to meet the demand of new workloads, applications or data models. Persistent Key/Value stores (KV-stores) are specialized for high-throughput and predominantly update-intensive, OLTP-style workloads.

KV-stores exhibit a characteristic *lightweight architecture*, simplifying the deployment and integration process for large infrastructures and lowering maintenance demand in production. *Scalability* is intrinsically supported in terms of partitioning and distribution schemes, making KV-stores an excellent choice for current data-center architectures. The *simplicity* of their interface (with *put* and *get*) as well as data model matches wide range of modern insert and update intensive applications running high-throughput OLTP-style workloads. Last but not least, the ability to *serve as DB-Engines* in traditional and modern NoSQL databases (e.g. MyRocks[13] or MongoRocks), allows for the *integration* as meta stores into applications and distributed file systems (e.g. Ceph[10]), or serve as a backend for OLTP services.

Persistent KV-stores leverage the properties of modern hardware due to the lean architecture, interface and flexibility, yet native hardware support is rare. The majority of such KV-stores rely on backwards-compatible storage, to ease administration and foster proliferation. Furthermore, the use of file systems simplifies space management, support for various storage architectures and the embedding in existing data center environments. *The underlying assumptions are that: (1) files and file-based I/O are the appropriate storage abstractions, and (2) use of standard/compatibility interfaces (and abstractions) on each individual layer of the I/O stack does not harm performance.*

The traditional I/O stack was developed with the characteristics of HDDs in mind, with the block-device interface, block I/O operations and files as abstractions. New storage technologies such as Non-Volatile Memories or Flash exhibit very different characteristics. However, to utilize them, persistent KV-stores require multiple layers of backwards compatibility, having a negative impact on performance and longevity. (1) Hardware resources are not fully exploited because of the hardware-oblivious abstractions. (2) DBMS access patterns result in suboptimal physical I/O patterns due to the presence of multiple abstraction layers along the critical I/O path. (3) KV-store information about the current workload cannot be used for better physical data placement. (4) Functionality along this critical I/O path is redundant. Significant write-amplification and suboptimal performance are the inevitable consequences.

To verify the above claims we perform an experiment under RocksDB running LinkBench and measure the end-to-end write-amplification along a backwards-compatible, file-system based stack. The results

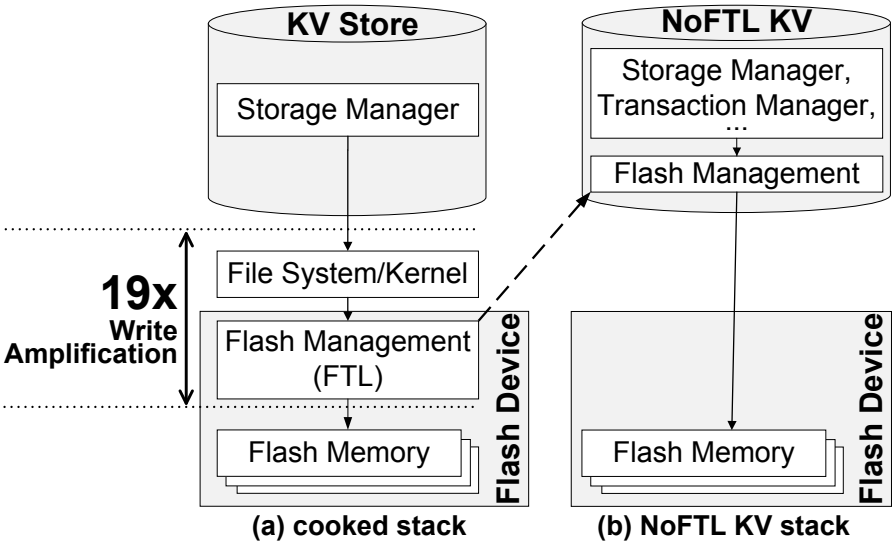


Figure 9.1: Write-Amplification along a traditional I/O stack in contrast to NoFTL-KV.

(Fig. 9.3) indicate a 19x physical write volume increase, lower performance and longevity.

In this paper we present NoFTL-KV (Fig. 16.1), an approach that avoids backwards compatibility and targets the above disadvantages by controlling the underlying physical storage directly. NoFTL-KV integrates physical storage (Flash) management natively in the KV-store. Subsequently, it opens up ways for workload adaptability within the storage layer and new abstractions for native storage.
The main contributions of this paper are:

(1) The extension of the concept of native storage management (NoFTL) to persistent KV-stores. We show that by coherently integrating address mapping, data placement, GC and free space management into the KV-store, storage characteristics, on-device parallelism and wear-leveling are addressed.

(2) NoFTL-KV is implemented under RocksDB.

(3) The performance evaluation under LinkBench[1] shows that NoFTL-KV improves the transactional throughput by 33%, while the response times improve up to 2.3x. Furthermore, NoFTL-KV improves physical storage management. In terms of *write efficiency*, NoFTL-KV performs 87% less physical page writes (including maintenance I/O and GC). Moreover, NoFTL-KV performs 19x less erases, improving the endurance by approximately the same factor.

The rest of the paper is structured as follows. NoFTL-KV and the integration into RocksDB are described in Section 9.3. Experimental results are discussed in Section 15.4. We conclude in Section 9.5.

9.2 Related Work

Modern workloads (Social Media, Big Data or IoT) not only have become write-intensive and require high sequential throughput, but also demand low latencies [16]. *Read-* and *Write-Amplification* are major performance factors [16].

These can either be approached by utilising compression to decrease I/O in general [5] or by aligning better with the characteristics of modern storage devices. The latter is addressed in terms of either new data structures [3, 4, 19], or new software interfaces [2] as well as Flash interface extensions [9, 12, 14]. However, neither of those takes the issues with the *cooked stack* into account. [6] and [7] present a full integration of native storage support within traditional DBMS. A few lightweight KV-stores address the concept of direct native storage integration [8, 15, 17, 18] by moving the entire KV-store onto the device. Yet, physical storage management is only partially addressed.

With NoFTL-KV we address the deep integration of native storage management to tackle all issues regarding the traditional cooked stack while avoiding to overload the device controller with database functionality and maintaining a mature KV-store.

9.3 NoFTL-KV: Native Storage KV-Store

We investigate the concept of native storage management and NoFTL under persistent KV-stores to address and evaluate the above mentioned claims. RocksDB exhibits an append-only I/O pattern for various write-intensive workloads, because of its LSM-Tree-based persistent storage. LSM-trees perform regular compactions to remove old records, to ensure optimal tree structure and to perform hot-cold-separation. Compactions reorganize levels of the LSM-Tree, removing updated or deleted KV-Pairs, at regular intervals or given a certain threshold. As a consequence, frequently changing data is placed in the upper levels of the LSM-Tree, while the lower levels contain the cold data.

Under NoFTL-KV we pursue coherent integration of Flash management into existing modules of the KV-store as shown in Fig. 9.2. Firstly, NoFTL-KV has direct control over hardware resources through a native storage interface (NSI). NSI allows the DBMS to operate with I/O operations, in granularity and with addressing schemes supported by the the underlying storage technology. Furthermore, NSI eliminates the need to support backwards compatibility. Secondly, we revisit hardware-oblivious abstractions and propose using physical storage abstractions such as *Regions* to: (a) reduce read/write amplification along the I/O path, (b) utilize available I/O parallelism more efficiently, (c) provide better hot-cold data separation to (d) improve space management and (e) increase longevity.

Moreover, unnecessary DBMS data transfers can be reduced by pushing tasks down to the storage device. For instance, parts of garbage collection and compaction can be planned by the NoFTL-KV storage manager for certain Regions, but are executed onto the device to reduce I/O contention and data transfers. Likewise, queries, i.e scans, can be pushed down and executed on the storage device. Especially in combination with Regions, such queries can profit by the involved address mapping and level of on-device-parallelism. Also worth to mention is that processors on such storage devices usually exhibits the characteristics of common co-processor (ASIC or FPGA). These are perfectly aligned to the characteristics of modern storage technology (Flash, NVM) e.g. in respect to parallelism.

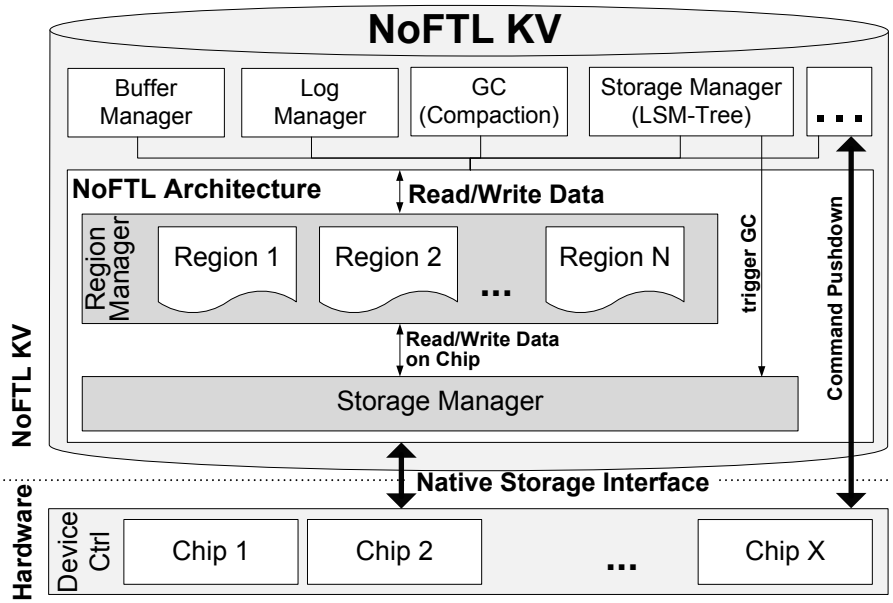


Figure 9.2: NoFTL-KV: Design of a deep integration of the NoFTL concept within an entire KV-store for native storage management

By integrating address mapping into the storage manager of the KV-store, the latter gets control over the physical data placement on Flash. Hence, the KV-store can utilize available information about data semantics, statistics and the access pattern (e.g., desired level of I/O parallelism) to perform efficient placement. Individual levels of the LSM-Tree can be physically separated on different chips to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device. Consequently, new storage abstractions can be defined besides files.

We introduce *Regions* as physical storage abstractions spanning multiple chips/dies (i.e. parallel unit of the storage device). They can be effectively optimized for different access patterns (sequential, random, append) of various KV-store components (e.g. Levels of LSM-Tree and Log Manager). Regions allow for flexible physical storage

management as the parameters of hot-cold data separation, garbage collection etc. are part of the definition. The level of supported I/O parallelism per Region can be defined in terms of the number of chips/dies it spans or whether these run in pseudo SLC, MLC or TLC Flash mode. Region definitions are not static, but can evolve over time to reflect properties of the workload.

```
CREATE REGION rgBlockMapping (
     MAX_CHIPS=4, MAX_CHANNELS=4,..., ADDR_MAPPING=BLOCK,
     NAND_MODE=MLC, ...);
CREATE TABLESPACE MyRocks.tblBlock (
     REGION=rgBlockMapping, UNIFORM EXTENT SIZE 128K );
CREATE TABLE MyRocks.nodetable(...)
     TABLESPACE MyRocks.tblBlock;
```

Furthermore, the functional redundancy along the *cooked I/O stack* is reduced. While, the file-system and the FTL distort the append-based access pattern and amplify the read/write data volume, NoFTL-KV simplifies the critical I/O path, exhibits a physically sequential I/O pattern, and offers better physical storage management. Consequently, write-amplification is significantly reduced. Similarly, the integration of the garbage collection within the compaction process of the LSM-Tree, allows for elimination of the time and resource-expensive merges common for traditional FTL-based SSDs. As a result, the KV-store is able to trigger the GC only when necessary and under the current workload. Higher longevity through less block erases and better throughput are the consequence.

9.4 EXPERIMENTAL EVALUATION

Testbed. Our testbed comprises a server equipped with an Intel Xeon E5-1620 v3 3.50 GHz CPU-core, 32GB RAM and an Intel DC 3600 SSD under Ubuntu 12.04 LTS, kernel 3.13.0.

The Flash storage device for the NoFTL-KV data is emulated by our real-time Flash Simulator[6], which is running as a kernel module. Configured with common latencies for reads, writes, and erases of current SLC NAND Flash it is able to simulate a modern enterprise SSD with either block- or char-device interfaces. For the block device, FASTer[11] is utilised as FTL with an over-provisioning area of 14%. In our setup, the simulator consumes 24 GB of memory to emulate an SSD of the same capacity with 256 pages (4KB) on 24576 blocks. The level of parallelism (emulated NAND chips/dies) is limited by the number of hardware threads. For our experiments we configured NoFTL-KV to only store RocksDB LSM-Tree files on the emulated device and the remaining files on the Intel DC 3600 formatted with an ext4 file system.

LinkBench. The experimental evaluation is performed using LinkBench[1], which is an OLTP-style workload on large updatable graphs. Under

LinkBench the working data set size is an order of magnitude larger than the database buffer. The request phase of LinkBench comprises common graph queries like adding, getting, counting, deleting, updating nodes or edges on the graph. The experimental dataset is a graph of 15M nodes (initial), amounting to 15GB raw data. The number of requests and duration vary depending on the experiment. The baseline utilises the same configuration (ext3, 4KB blocksize, active journal) for MyRocks with RocksDB, hereinafter referred to as RocksDB.

Figure 9.3: Amount of data written by the DB, FS and FTL during the load and request phases of LinkBench shows the write-amplification of RocksDB in contrast to NoFTL-KV

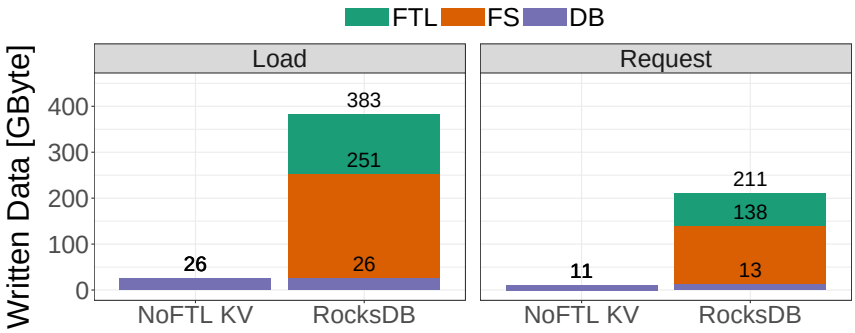
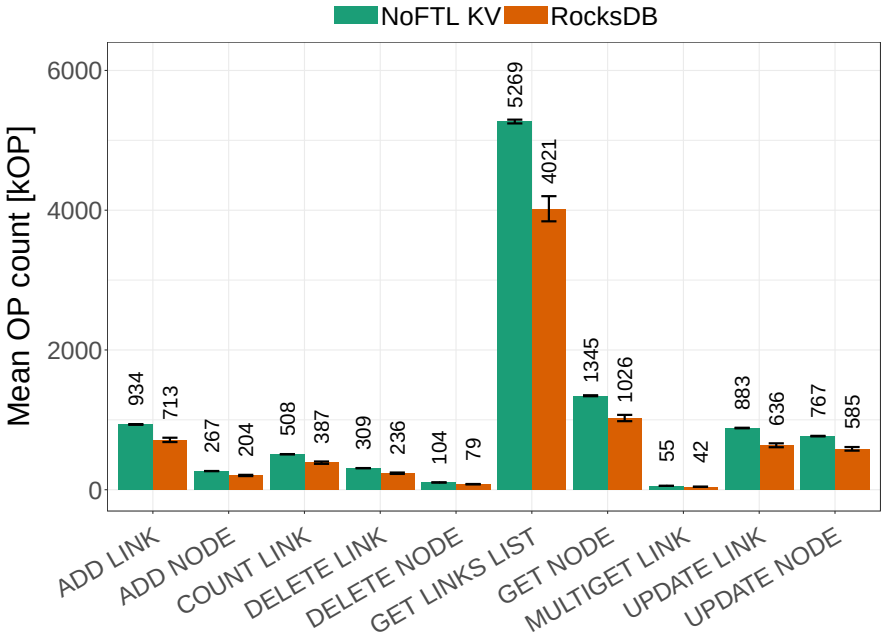


Table 9.1: Number of logical page writes of the DB's compaction and physical page writes of the device over 10 request phases demonstrates write-amplification and inconsistent logical to physical page write ratio.

	NoFTL-KV		RocksDB		Speedup
	Mean	StDev	Mean	StDev	
GC Calls	1	3	4769	2434	
GC Page Write	0	0	1932263	737453	
Block Erase	1127	3564	21389	7792	18.98x

Write-Amplification. To measure write amplification, hooks are placed in the storage engine of RocksDB (DB), the file system (FS), and the Flash emulator (FTL), i.e., in all layers along the I/O path. The number of requests is limited to 1M per thread with sufficient time (10h) to be executed completely. This ensures that, at the end, both variants have executed the same number of operations. The results

Figure 9.4: Throughput: The average number of executed operations over the last 7 request phases demonstrates that NoFTL-KV outperforms RocksDB



(Fig. 9.3) for NoFTL-KV and the baseline RocksDB represent average values of multiple runs.

Not surprisingly, the significant write amplification of the cooked stack becomes evident. The 26 GB of raw data, bulk-loaded during the *load phase*, swells up by more than 14 times to 383 GB. On top of that, the file system adds about 225 GB and the FTL increases this again by 132 GB. During the *request phase*, the disadvantages of the cooked I/O stack become even more visible. The average write-amplification here is more than 19x. This creates enormous I/O overhead, which is clearly reflected by the metrics to follow.

Throughput. The mean number of executed operations and their errors for every operation type is shown in Fig. 9.4. NoFTL-KV outperforms RocksDB in every type of query. This is because of the smaller data volume to be written, and the better utilisation of available Flash parallelism. The workload of LinkBench has a high write-intensity over the complete duration. Consequently, the throughput increases about 31% accross all operation types. The performance stability across different runs, indicated by the error bars increases by an order of magnitude.

Response Time. To investigate the impact on response time for common operations, we perform further experiments with 1M requests per thread. Fig. 9.5 shows the average duration, while the error bars indicate the standard deviation.

Figure 9.5: Response Time (the lower the better): Average operation latencies and std. dev. are better and more stable under NoFTL-KV vs. RocksDB

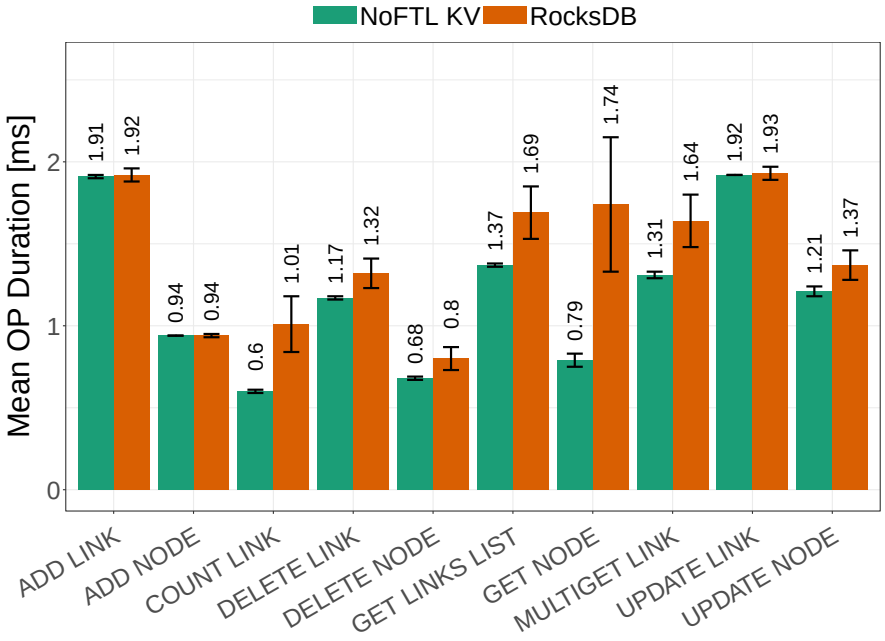


Figure 9.6: Results of the experimental evaluation of NoFTL-KV using LinkBench

One can clearly see that the latency is lower under NoFTL-KV. Especially reading operations like *GetNode()*, *GetLinksList()*, and *MultigetLink()* perform significantly better. This is even more relevant, since about 22% of these could not be served by database buffer (cache miss rate) and are read from the persistent device. On the other hand, inserting and updating operations like *AddNode()*, *AddLink()*, and *UpdateLink()* complete directly after pushing the data into an in-memory buffer and the WAL. This buffer is persistet only after a compaction, which is not taken into account for the operation latency. This explains the similar performance for both RocksDB and NoFTL-KV. The only exception is *UpdateNode()*, which might result in multiple gets that are slower with the traditional I/O stack.

Erases and Longevity. Write-amplification on Flash devices inevitably leads to more physical Flash erases, which has negative impact on device longevity. Table 9.1 captures the GC activity in terms of physical page writes and block erases during the benchmark runs of the previous experiment.

RocksDB performs almost 19 times the physical block erases than NoFTL-KV, which is primarily due to (i) the journal of the file system, which doubles Flash page writes, and (ii) FASTer's hybrid address

mapping scheme. It is worth noting that the erase overhead of FASTer would also be present in other FTLs, which utilize hybrid address translation (common for current SSDs). As soon as the so-called log block area of the device runs out of space, the GC kicks in and merges the updated data with the corresponding Flash blocks in the data block area. Each of those merges requires multiple page migrations (on-device write amplification), and one or two erase operations (partial or full merges). NoFTL-KV is configured to use BLM, which matches the append-only LSM-Tree based storage management of RocksDB.

9.5 CONCLUSION

In the present paper we propose NoFTL-KV, an approach that results in a lean I/O stack, integrating physical storage management natively in the Key/Value store. NoFTL-KV eliminates backwards compatibility, allowing the Key/Value store to directly exploit the characteristics of modern storage technologies. NoFTL-KV is implemented under RocksDB and evaluated using LinkBench. The transactional throughput improves by 33%, while response times improve up to 2.3x. Furthermore, NoFTL-KV reduces write-amplification 19x and improves endurance. In addition, our current integration on the file-based LSM-Tree can be further improved by a deeper integration into the KV-store's data structure in future work to gain additional performance improvements.

REFERENCES

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

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

[3] Niv Dayan, Philippe Bonnet, and Stratos Idreos. "GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices." In: *Proc. SIGMOD 2016*.

[4] Biplob Debnath, Sudipta Sengupta, and Jin Li. "SkimpyStash: RAM Space Skimpy Key-value Store on Flash-based Storage." In: *Proc. SIGMOD 2011*.

[5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. "Optimizing Space Amplification in RocksDB." In: *Proc. CIDR 2017*.

[6] Sergej Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. "NoFTL: Database Systems on FTL-less Flash Storage." In: *Proc. VLDB 2013*.

[7] Sergey Hardock and Ilia Petrov and Robert Gottstein and Alejandro P. Buchmann. "NoFTL for Real: Databases on Real Native Flash Storage." In: *Proc EDBT 2015*.

[8] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. "KAML: A Flexible, High-Performance Key-Value SSD." In: *In Proc. HPCA 2017*.

[9] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. "X-FTL: Transactional FTL for SQLite Databases." In: *Proc. SIGMOD 2013*.

[10] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. "Understanding Write Behaviors of Storage Backends in Ceph Object Store." In: *Proc. MSST 2017*.

[11] S. P. Lim, S. W. Lee, and B. Moon. "FASTer FTL for Enterprise-Class Flash Memory SSDs." In: *Proc. SNAPI 2010*.

[12] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. "NVMKV: A Scalable, Lightweight, FTL-aware Key-value Store." In: *Proc. ATC 2015*.

[13] Yoshinori Matsunobu. "InnoDB to MyRocks Migration in Main MySQL Database at Facebook." In: *Proc. SREcon 2017*.

[14] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. "SHARE Interface in Flash Storage for Relational and NoSQL Databases." In: *Proc. SIGMOD 2016*.

[15] Samsung. In: *Flash Memory Summit, 2017*. URL: http://www.samsung.com/semiconductor/global/file/insight/2017/08/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[16] Russell Sears and Raghu Ramakrishnan. "bLSM: A General Purpose Log Structured Merge Tree." In: *Proc. SIGMOD 2012*.

[17] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. "Willow: A User-Programmable SSD." In: Proc. OSDI 2014.

[18] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. "Bluecache: A Scalable Distributed Flash-based Key-value Store." In: *Proc. VLDB 2016*.

[19] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. "ParaFS: A Log-structured File System to Exploit the Internal Parallelism of Flash Devices." In: *Proc. ATC 2016*.

10

NATIVENDP: PROCESSING BIG DATA ANALYTICS ON NATIVE STORAGE NODES

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"nativeNDP: Processing Big Data Analytics on Native Storage Nodes"* by Tobias Vinçon, Sergey Hardock, Christian Riegger, Andreas Koch and Ilia Petrov in 2019 Advances in Databases and Information Systems (ADBIS). The contribution of the author of this thesis is summarized as follows.

» As the corresponding and leading author, Tobias Vinçon was responsible for the integration of NoFTL-KV into the widespread cluster file system Ceph. Moreover, he designed the client application and was in charge of configuring the entire system stack as well as executing the experimental evaluation. With constructive feedback from Sergey Hardock, Christian Riegger, Andreas Koch and Ilia Petrov the manuscript was written by him. «

ABSTRACT

Data analytics tasks on large datasets are computationally-intensive and often demand the compute power of cluster environments. Yet, data cleansing, preparation, dataset characterization and statistics or metrics computation steps are frequent. These are mostly performed ad hoc, in an explorative manner and mandate low response times. But, such steps are I/O intensive and typically very slow due to low data locality, inadequate interfaces and abstractions along the stack. These typically result in prohibitively expensive scans of the full dataset and transformations on interface boundaries.

In this paper, we examine *R* as analytical tool, managing large persistent datasets in *Ceph*, a wide-spread cluster file-system. We propose *nativeNDP* – a framework for *Near-Data Processing* that pushes down primitive *R* tasks and executes them in-situ, directly within the storage device of a cluster-node. Across a range of data sizes, we show that *nativeNDP* is more than an order of magnitude faster than other pushdown alternatives.

10.1 INTRODUCTION

Modern datasets are large, with near-linear growth, driven by developments in IoT, social media, cloud or mobile platforms. Analytical operations and ML workloads result therefore in massive and sometimes repetitive scans of the entire dataset. Furthermore, data preparation and cleansing cause expensive transformations, due to varying abstractions along the analytical stack. For example, our experiments show that computing a simple *sum* on a scientific dataset in *R* takes 1% of the total time, while the remaining 99% are spent for I/O and *CSV* format conversion.

Such data transfers, shuffling data across the memory hierarchy, have a negative impact on performance and scalability, and incur low resource efficiency and high energy consumption. The root cause for this phenomenon lies in the typically low data locality as well as in traditional system architectures and algorithms, designed according to the *data-to-code* principle. It requires data to be transferred to the computing units to be processed, which is inherently bounded by the *von Neumann bottleneck*. The negative impact is amplified by the slowdown of *Moore's Law* and the end of *Dennard Scaling*. The limited performance and scalability is especially painful for nodes of high-performance cluster environments with sufficient processing power to support computationally-intensive analytics.

Luckily, recent technological developments help to counter these drawbacks. Firstly, hardware vendors can *fabricate combinations of storage and compute elements at reasonable costs*. Secondly, this trend covers virtually all levels of the memory hierarchy (e.g. IBM's AMC for Processing-in-Memory, or Micron's HMC). Thirdly, the device-internal bandwidth and parallelism significantly exceed the external ones (Device-To-Host), for non-volatile semiconductor (NVM, Flash) storage devices.

Such *intelligent storage* allows for *Near-Data Processing (NDP)* of analytics operations, i.e. such operations are executed in-situ, close to where data is physically stored and transfer just the result sets, without moving the raw data. This results in a *code-to-data* architecture.

Analytical operations are diverse and range from complex algorithms to basic mathematical, statistical or algebraic operations. In this paper, we present execution options for basic operations in nodes of clustered environments as shown in Figure 16.1: (1) The computation is within the client and the cluster node is used as part of a traditional distributed file system; (2) The operation is transmitted to the cluster and processed within the cluster node itself; (3) The operation is executed in-situ, within the NDP devices of the cluster's node. The investigated operations are simple, yet they clearly give evidence for the NDP effects on internal bandwidth and the ease of system and network buses. The execution of more extensive operations

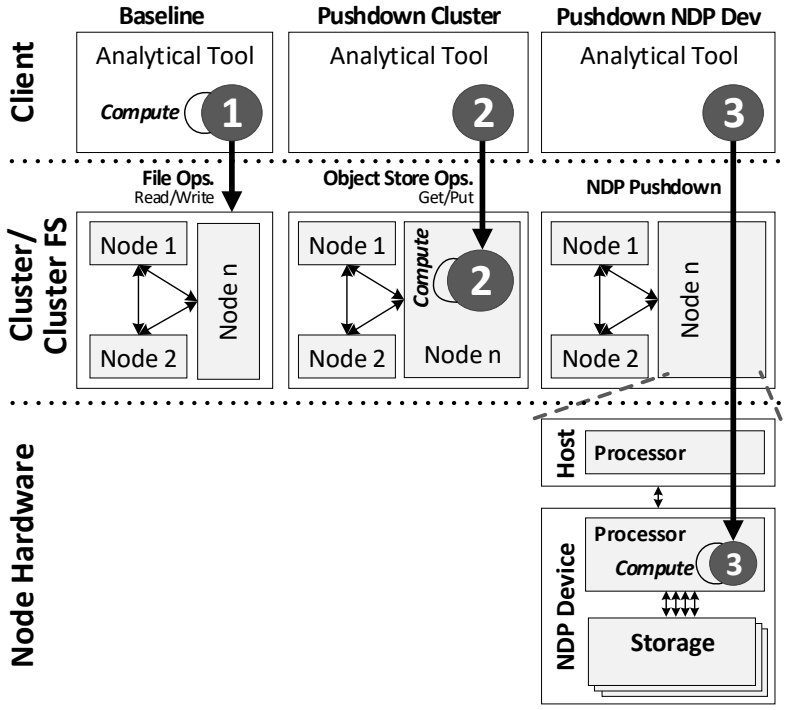


Figure 10.1: Three different options to execute analytical operations on a cluster environment. (1) Baseline: Execute on the client; (2) Pushdown Cluster: Execute on a cluster's node; (3) Pushdown NDP Device: Execute on the NDP Device of a cluster's node

like betweenness centrality within graphs or clustering and k-nearest neighbor searches are planed for future work.

The main contributions of this paper are:

- End-to-end integration of NDP interfaces throughout the entire system stack
- The performance evaluation shows improvements of NDP operation pushdown of at least 10x
- Analysis of the impact of and necessity for NDP-based abstractions and interfaces.
- We identify the following aspects as the main drawbacks to implementing NDP: Interfaces; Abstractions; Result-Set consumption semantics; Data Layout and NDP Toolchain

The rest of the paper is structured as follows. Section 13.2 presents the architecture of *nativeNDP*. In Section 10.4 we discuss the experimental design and performance evaluation. We conclude in Section 13.5.

10.2 RELATED WORK

The concept of *Near-Data Processing* is not new. Historically it is deeply rooted in *database machines* [2, 5], developed in the 1970 and 1980s. [2]

discuss approaches such as processor-per-track or processor-per-head as an early attempt to combine magneto-mechanical storage and simple computing elements to process data directly on mass storage and to reduce data transfers. Besides reliance on proprietary and costly hardware, the I/O bandwidth and parallelism are claimed to be the limiting factor to justify parallel DBMS [2]. While this conclusion is not surprising, given the characteristics of magnetic/mechanical storage combined with Amdahl's balanced systems law [7], it is revised with modern technologies. Modern semi-conductor storage technologies (NVM, Flash) are offering high raw bandwidth and high levels of parallelism. [2] also raises the issue of temporal locality in database applications, which has already been questioned earlier and is considered to be low in modern workloads, causing unnecessary data transfers. Near-Data Processing presents an opportunity to address it.

The concept of *Active Disk* emerged toward the end of the 1990s. It is most prominently represented by systems such as: Active Disk [1], IDISK [12], and Active storage/disk [15]. While database machines attempted to execute fixed primitive access operations, *Active Disk* targets executing application-specific code on the drive. Active storage [15] relies on processor-per-disk architecture. It yields significant performance benefits for I/O bound scans in terms of bandwidth, parallelism and reduction of data transfers. IDISK [12], assume a higher complexity of data processing operations compared to [15] and targets mainly analytical workloads and business intelligence and DSS systems. Active Disc [1] targets an architecture based on on-device processors and pushdown of custom data-processing operations. [1] focuses on programming models and explores a streaming-based programming model, expressing data intensive operations, as so called *disklets*, which are pushed down and executed on the disk processor.

With the latest trend of applying different compute units, besides CPUs, to accelerate database workloads, a more intelligent FPGA-based storage engine for databases has been demonstrated with Ibex [19]. It focuses mainly on the implementation of classical database operations on reprogrammable compute units to satisfy their characteristics, such as parallelism and bandwidth. A completely distributed storage layer, targeting NDP on DRAM over the network, is presented by Caribou [10]. Its shared-data model is replicated from the master to the respective replica nodes using Zookeeper's atomic broadcast. Utilizing bitmaps, Caribou is able to scan datasets with FPGAs only by the limiting factor of the selection itself (low selectivity) or the network (high selectivity). Moreover, [3, 4, 8, 14] investigate further host-to-device interfaces for general-purpose applications or specific workloads.

However, previous research focused mainly either on the concrete implementation of the reconfigurable hardware, or on single device instances. In this paper, we attempt to combine both topics and focus

on the abstraction and interfaces necessary to complete an efficient NDP pushdown.

10.3 NATIVENDP FRAMEWORK

The architecture shown in Figure 10.2 presents a bird's eye view of the essential components, interfaces, and abstractions of the nativeNDP framework. An analytical client executes an R script, triggering an analytical operation (filtering, simple computation - SUM, AVG, STD-DEV, or a clustering algorithm). It can be processed on different levels of the system stack:

- directly in R (Figure 16.1–*baseline*). This is a classical approach, which can be done with out of the box software, requiring little overhead. The downside is that the complete dataset needs to be transferred through the stack causing excessive data transfers and posing significant memory pressure on the client.
- within a *Cluster node* (Figure 16.1–*pushdown cluster*). The same function can be offloaded to the HPC cluster system and distributed across nodes. Hence the compute and data transfer load can be reduced, but not eliminated as such data transfers are performed locally on a node.
- on the *Storage Device* (Figure 16.1–*pushdown NDP dev*). With NDP, the operations are offloaded directly on the device, utilizing the internal bandwidth, parallelism and compute resources to reduce data transfers and improve latency.

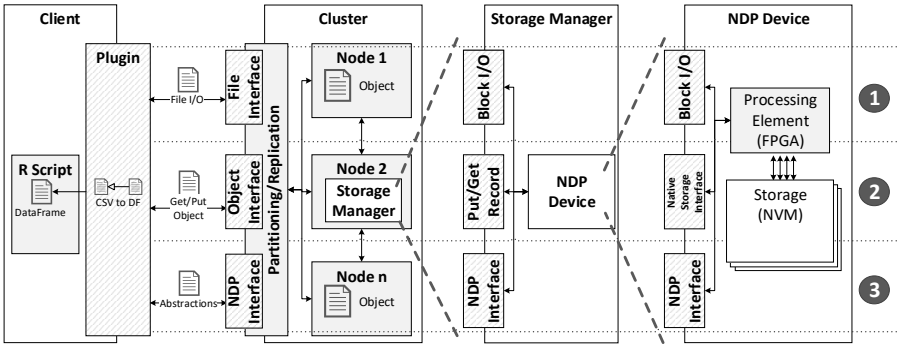


Figure 10.2: The high-level architecture showing the applied interfaces and data abstractions along the access path for the three compared experiments: baseline, pushdown cluster node, and pushdown device

10.3.1 *System Stack*

In the following we describe the layers of the analytical stack in more detail.

Client: We utilize R as one of the most popular client software for analytical and statistical computation. To interact with the Ceph cluster and the underlaying layers, we designed a custom R plugin, RCeph. It uses the RADOS API [18] to connect to the cluster and is able to issue specific commands with following features:

PUT/GET OF FILES/OBJECTS: To facilitate the first scenario, presented in Figure 16.1, the dataset file has to be retrievable from the cluster. Therefore, the standard file I/O API is reused. However, the transfer of results from the second and third scenario necessitates further interfaces such as RADOS's provided Object API as explained in Section 10.3.2.

PUSHDOWN OF DOMAIN-SPECIFIC OPERATIONS: This feature is mainly addressed with the second and third scenario, where domain-specific operations, usually executed within the client, are pushed down to either a cluster's node or even throughout the node's storage engine to the NDP Device. I.e. such domain-specific operations comprise R-native operations on their storage abstraction *DataFrame* or could even be extended to small algorithmic expressions.

FORMAT CONVERSION: As interfaces and abstractions of lower levels often rely on backwards compatibility in nowadays complex systems, format conversions of the results or CSV-encoded files and objects into the R-specific abstraction *DataFrame* are necessary.

The RCeph is complied using Rcpp [6] to a plugin package and can be installed, loaded, and applied within the R runtime environment of the client.

Cluster: To process todays datasets with analytical or statistical workloads in an acceptable time, both data and calculation are distributed over a cluster environment. This becomes even more crucial with focus on high performance in particular. To simplify low latency data accesses, distributed file systems are applied in such environments nowadays. Therefore, Ceph [17], which is a wide-spread solution for clustered environments, builds the foundation of the nativeNDP framework. Its purpose is to efficiently manage a variety of nodes within a cluster environment. Thereby, stored files are striped across small objects, grouped into placement groups and distributed on these nodes to ensure scalability and high reliability. Its flexible architecture comprises various components and provides interfaces for object, block and file I/O. Internally, exchangeable storage engines are responsible to manage the reads and writes to secondary storage. One of its most recent storage backends is called BlueStore and utilizes RocksDB as an internal KV-Store.

Storage Manager: We replaced the internal KV-Store of BlueStore with our own native storage engine NoFTL-KV [16]. Hereby, hardware characteristics, like in-parallel accessible flash chips of the storage device, are known by NoFTL-KV, which in turn is able to efficiently leverage those. Consequently, the physical location of persisted data is defined by the KV-Store itself rather than any Flash Translation Layer (FTL) of a conventional stack. This opens the opportunity to issue commands directly on the physical locations throughout NoFTL-KV and to streamline low-level interfaces along the entire access path.

NDP Device: Devices are emulated by our own storage-type SCM Simulator, based on [9]. Running as a kernel module it provides the ability to delay read and write request depending on its emulated physical locations by utilizing the accurate kernel timer functions. As a consequence, reads or writes across physical page borders claim respectively multiple I/O latencies. For the experimental evaluation, the simulator is instrumented with realistic storage-type SCM latencies from [11]. Moreover, by its flexible design it allows us to extend it with the necessary NDP interface.

10.3.2 *Interfaces and Abstractions*

The first, most commonly applied interface is the traditional file I/O (Figure 10.2.1). It abstracts the cluster as a large file system, storing its data distributed on multiple nodes. A partitioning and/or replication layer takes care of the internal data placement on various nodes. Instead of the KV-Store the conventional Block I/O is used to issue reads and writes to the NDP Device. This also involves any kind of Flash-Translation-Layer on the device itself to reduce the wear on a single storage cell and consequently ensure longevity of the entire device.

Secondly, a modern object interface offered by RADOS [18] (Figure 10.2.2) can be utilized to put/get objects on the cluster. This abstraction might comprise single or multiple records of a file, or the result set of a pushed down user defined function executed on the respective node. Since the cluster handles data placement, it can transparently execute such algorithms in parallel with the full processing power of the node's servers if the operations are data independent. Within the lower levels, depending on the storage manager, one can either exploit the conventional Block I/O to access the NDP Device or leverage NoFTL-KV's *Native Storage Interface*.

Thirdly, an NDP pushdown necessitates a different kind of interface definition (Figure 10.2.3). The NDP execution of application-specific operations requires open interfaces. These should support NDP of application-specific abstractions such as *DataFrame* for R. Consequently, these interfaces and abstractions mandate flexibility, since

various result types of the application logic on the device must be transferred back to the client. Expensive format conversion along the system stack can be avoided almost entirely. Yet, an extensive toolchain and NDP framework support is required, beginning from the analytical tool to the employed hardware devices in the cluster. Utilizing the processing elements near-storage (e.g. FPGA), the internal, on-device parallelism and bandwidth can be fully leveraged. For instance, [13] projects of up to 50 GB/s, while the workload on slower buses (e.g. PCIe 2.0 $\approx$ 6.4 GB/s) in the system is eased by reducing transfer volumes (i.e. $resultset \ll rawdata$).

10.4 EXPERIMENTAL EVALUATION

To compare the different execution options on the presented system stack and evaluate their bottlenecks, we conduct three experiments aligned to the scenarios of Figure 16.1.

10.4.1 *Datasets and Operations*

To ensure the comparability of the scenarios, datasets and operations are predefined. The datasets are created synthetically as CSV files with random numbers, with varying rows and columns from 1k to 10k. When stored in the KV-Store, each cell of the CSV file is identifiable by an auto-generated key with the structure:

$$[object\_name].[column\_index].[row\_index]$$

Inevitably, this is bloating out the raw file size by approximately 16x-17x but enables to access cells by this unique id. Alternatively, depending on the workload, an arrangement per row or per column is likewise feasible. Table 10.1 summarizes the properties of each dataset for the present experiments.

The operations performed in all experiments is independent of the data distribution and constitutes a typical data science application - calculation of the sum or the average over a given column (Because of the marginal differences only sum is shown further on). The final result set comprises a 32-byte integer value and some additional status data. We leave the implementation of further analytical and/or statistical operations open for future work.

10.4.2 *Experimental Setup*

The server, *nativeNDP* is evaluated on, is equipped with four Intel Xeon x7560 8-core CPUs clocked at 2.26 GHz, 1TB DRAM running Debian 4.9, kernel 4.9.0. The NDP storage device is emulated by our real-time NVM Simulator, extended with an NDP interface and

Table 10.1: Synthetically generated datasets for the experiments. The raw CSV file size is according the Key-Value format bloated out.

Dataset	KV Pairs	CSV Size [MB]	KV Size [MB]	Bloating Ratio
1k/1k	1 000 000	2.8	44	15.9
2k/2k	4 000 000	12	182	15.2
4k/4k	16 000 000	45	738	16.4
6k/6k	36 000 000	101	1 668	16.5
8k/8k	64 000 000	178	2 971	16.7
10k/10k	100 000 000	278	4 649	16.7

functionality. I/O and pushdown operations are handled internally with the storage-type SCM latencies [11].

Since the main target is to evaluate the streamlining of NDP interfaces and abstractions, interferences caused by data distribution or multi-node communication have to be avoided. Therefore, the Ceph cluster is set up with a single object store node. This allows conducting experiments along a clean stack and measuring execution and transfer size for each architectural layer individually.

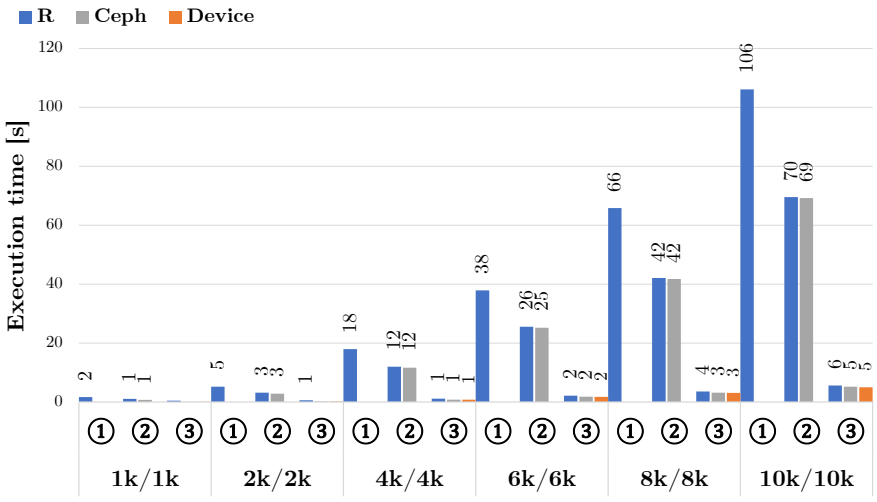


Figure 10.3: Execution time for varying dataset sizes shows the performance impact of data transfers/volume, and the improvement through NDP.

10.4.3 *Experiment 1 – Baseline*

The first experiment utilizes the Ceph cluster in the most common and conventional way - as a file system (Figure 10.2.1). Therefore, the file abstractions, interfaces, and subsequently Block I/O are used to retrieve the entire file. The sum over the 10th column is calculated in R by calling *readCSVDataFrame* of RCeph and caching the resulting DataFrame into the R runtime environment. Here, R's capabilities can be used to filter the DataFrame on the respective column and perform the arithmetic operation.

```
sum <- sum(RCeph::readCSVDataFrame(o_name)[col_id])
```

This experiment defines the baseline for any improvements of nativeNDP. However, it exemplifies multiple drawbacks yielding in a significant performance degradation. Firstly, the entire file has to be read via block I/O, even though only a small portion of it, the 10th column, is necessary to be processed by the operation (Figure 10.5). Secondly, the latency and bandwidth limitations of the network interconnect between the R host and the Ceph cluster, contribute to additional delays to the R processing. The significantly higher transfer size of Host-To-Client, illustrated in Figure 10.5, leads inevitably to a slower request duration. Additionally, as R DataFrames do not support any streaming algorithmic, the processing has to idle until the entire dataset is retrieved from Ceph. Thirdly, additional compute-intensive format conversions along multiple interface boundaries are necessary to create R *DataFrames*, which increase delays even further. For example the "R - parse_time" is 95% of the total time as shown in Figure 10.4. Moreover, such format conversions are directly depending on the data size, which is subsequently affected by the large Host-To-Client transfer size. Lastly, client systems often comprise limited hardware (e.g. notebook or workstation), while typical working sets can range from tens to hundreds of gigabytes. Thus, processing the whole dataset is not always possible without any performance degrading swapping to disk.

These drawbacks lead to a significantly higher total execution time for the calculation in general, as shown in Figure 10.3 (at least 10x).

In total, the baseline experiment results in the lowest performance for all datasets, which is mainly caused by the time spent in transfer and conversion of the CSV object into the R-specific data type *DataFrame* ("R - parse_time" Figure 10.4).

10.4.4 *Experiment 2 – Pushdown Cluster*

For the second experiment, Ceph's advanced object interface is extended to execute a user defined function. It queries the KV-Pairs of the respective dataset from NoFTL-KV of the Storage Manager by filtering

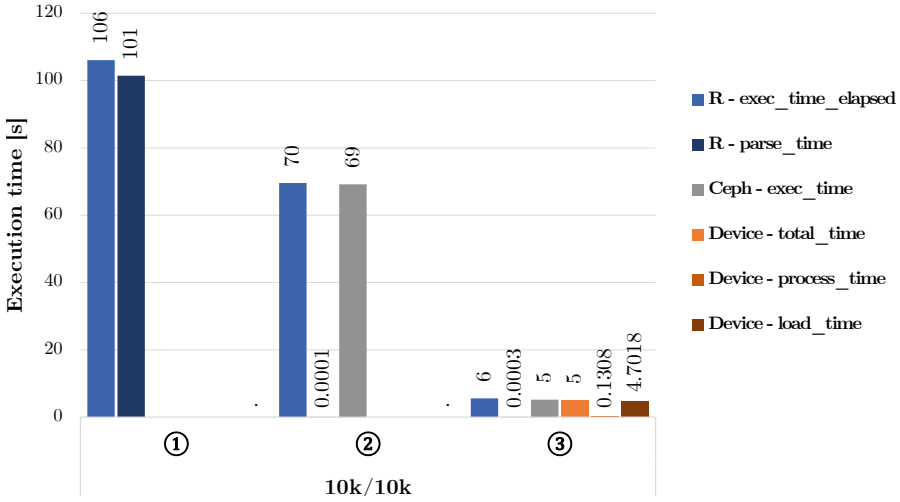


Figure 10.4: A detailed execution time analysis shows the main bottlenecks along the analytical stack.

on the 10th column. Thereby, the retrieved values are cumulated (Figure 10.2.2). In a full-fledged cluster scenario, Ceph will automatically distribute this algorithm on the respective nodes within the cluster and aggregate their results afterwards. Obviously, the result size after the operation pushdown is dramatically smaller than the raw data, which relieves the network and accelerates subsequent expensive data format conversions. Hence, the almost non-existing "R - parse_time" (Figure 10.4) and the respective transfer size from *Host-To-Client* (Figure 10.5). Both result in an overall performance improvement of up to 30% in comparison to the baseline (Figure 10.3).

```
sum <- RCeph::execCmd(o_name, "NDP_CEPH SELECT SUM COLUMN col_id")
```

Nonetheless, the I/O overhead of reading the entire data from the storage subsystem, as shown in Figure 10.5 by *Device-To-Host*, represents a major bottleneck. Therefore, the time spent in format conversions within Ceph increases as well. For the largest dataset, it takes more than 99% of the time. However, it can be avoided by applying NDP.

10.4.5 *Experiment 3 – Pushdown NDP Device*

Our last experiment relies on Near-Data Processing (Figure 10.2.3). Abstractions and interfaces are statically created for the purpose of filtering on a given column and computing sums to enable a device pushdown.

```
sum <- RCeph::execCmd(obj_name,"NDP_DEV SELECT SUM COLUMN col_id")
```

The NDP pushdown leverages the much higher levels of compute and I/O parallelism supported by the on-device processing elements

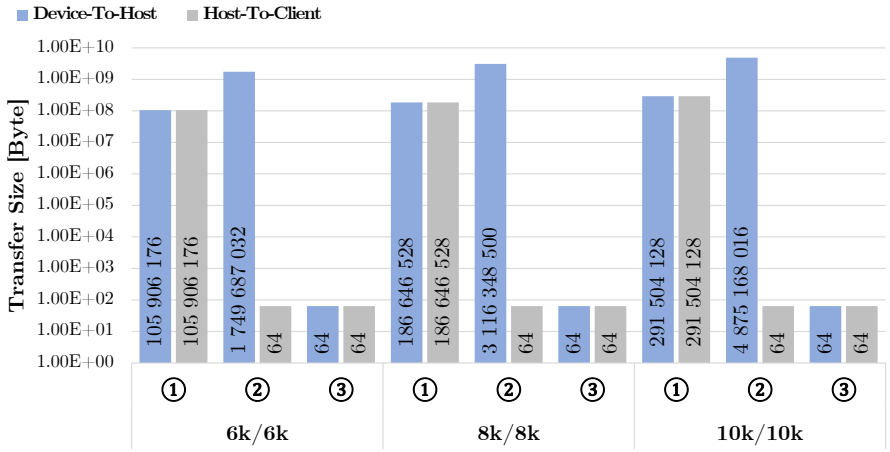


Figure 10.5: Transfer sizes from Device-To-Host and Host-To-Client of varying datasets shows the counteraction of NDP to the *von Neumann bottleneck*

(FPGA, GPU) to compute the sum an order of magnitude faster (Figure 10.3). Thereby, transferring data from the storage chips takes most of the time (Figure 10.4 "Device - load_time"), while the processing is only about 3% of the total time (Figure 10.4 "Device - process_time"). Not only is the network relieved by this early reduction of volume, but also the system-wide number of data transfers is significantly reduced. This is mainly driven by the on-device computation and result size reduction as shown in Figure 10.5. As this is only possible with the application-specific abstractions, a push down command must compulsorily comprise those to apply computation on the device, *in-situ*. In R, for instance, *DataFrame* may be a suitable application-specific abstraction.

10.5 CONCLUSION

We present *nativeNDP*, a NDP approach to effectively pushdown analytical operations to a native storage node of a clustered environment. The evaluation shows improvements of at least 10x over the baseline. Besides the known issues with todays computer architectures, we identify ill-suited interfaces and abstractions along the analytical stack as major drawbacks of current solutions. Moreover, the necessity to push down application-specific abstractions, and data layouts interpretable by the NDP Device is considered a key aspect for a true *in-situ* processing in complex system stacks. To mitigate format conversions along interface boundaries of such stacks, a comprehensive but flexible NDP toolchain is required.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *ASPLOS*. 1998.

[2] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines.

[3] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. "Active disk meets flash." In: *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. - ICS*. ACM Press, 2013, p. 91.

[4] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. "Minerva: Accelerating Data Analysis in Next-Generation SSDs." In: *2013 IEEE 21st Annu. Int. Symp. Field-Programmable Cust. Comput. Mach.* IEEE, Apr. 2013, pp. 9–16.

[5] David DeWitt and Jim Gray. "Parallel Database Systems: The Future of High Performance Database Systems." In: *Commun. ACM* (1992).

[6] Dirk Eddelbuettel. *Seamless R and C++ integration with Rcpp*. Springer, 2013.

[7] Jim Gray and Prashant J. Shenoy. "Rules of Thumb in Data Engineering." In: *Proc. ICDE*. 2000, p. 3.

[8] Boncheol Gu et al. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads." In: *ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.* Vol. 8. 6. IEEE, June 2016, pp. 153–165.

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

[10] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou." In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), pp. 1202–1213.

[11] *ITRS - International Technology Roadmap for Semiconductors Reports*. 2014. URL: http://www.itrs2.net/itrs-reports.html.

[12] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. "A Case for Intelligent Disks (IDISKs)." In: *SIGMOD Rec.* 27.3 (Sept. 1998), pp. 42–52.

[13] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. "In-storage processing of database scans and joins." In: *Inf. Sci. (Ny).* 327 (Jan. 2016), pp. 183–200.

[14] Marco Minutoli, Shannon K Kuntz, Antonino Tumeo, and Peter M Kogge. "Implementing Radix Sort on Emu 1." In: *Work. Near-Data Process.* (2015), pp. 1–6.

[15] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. "Active Storage for Large-Scale Data Mining and Multimedia." In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 62–73.

[16] Tobias Vinçon, Sergey Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. "NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management." In: *EDBT*. 2018.

[17] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-performance Distributed File System." In: *OSDI*. 2006.

[18] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. "RADOS: a scalable, reliable storage service for petabyte-scale storage clusters." In: *PDSW*. 2007.

[19] Louis Woods, Jens Teubner, and Gustavo Alonso. "Less watts, more performance." In: *Proc. 2013 Int. Conf. Manag. data - SIGMOD*. New York, New York, USA: ACM Press, 2013, p. 1073.

Part III

ON-DEVICE NAVIGATION AND DATA INTERPRETATION

11

ON THE NECESSITY OF EXPLICIT CROSS-LAYER DATA FORMATS IN NEAR-DATA PROCESSING SYSTEMS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems"* by Tobias Vinçon, Arthur Bernhardt, Lukas Weber, Andreas Koch and Ilia Petrov in 2020 IEEE 36th International Conference on Data Engineering (ICDE). The contribution of the author of this thesis is summarized as follows.

» As the corresponding and leading author, Tobias Vinçon was in charge of evaluating the foundational principles of explicit cross-layer data formats for NDP. Therefore, he provided the foundational concepts and extended the NoFTL-KV implementation with NDP Parsers and Accessors of an exemplary image processing application. The experimental evaluation on the COSMOS+ hardware was set up and executed by him. The majority of the manuscript's text was written by him with extensive support from the co-authors Arthur Bernhard, Lukas Weber, Andreas Koch and Ilia Petrov. «

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.

11.1 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 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.

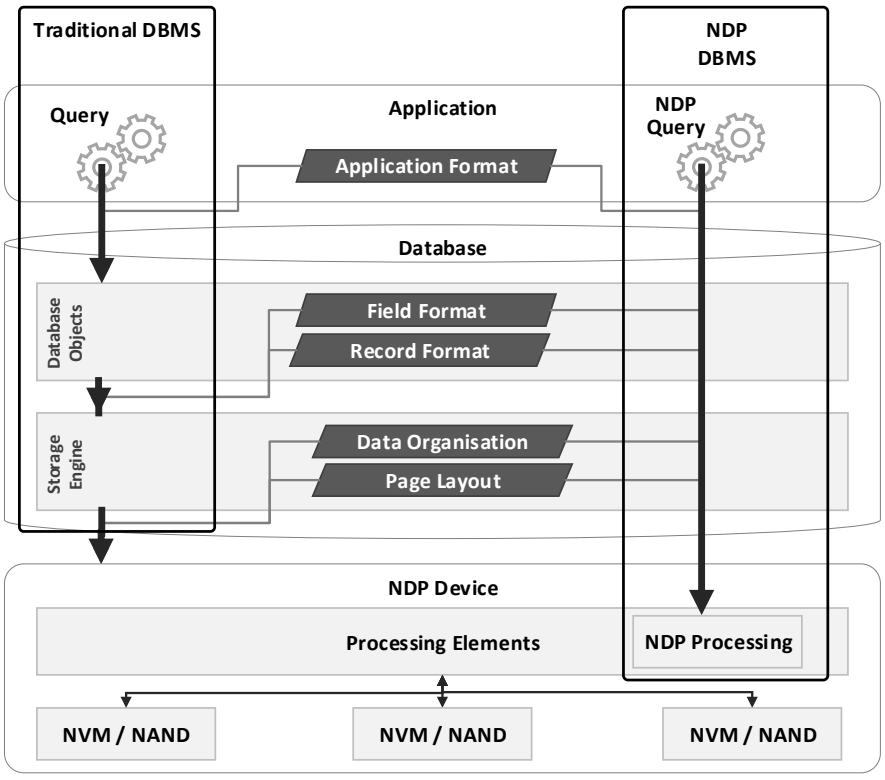


Figure 11.1: In contrast to classical queries, NDP operations must have all necessary format and layout information to execute the respective operations in-situ without host interaction.

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 16.1 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 16.1). 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 15.2 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 11.3. We conclude with Section 13.5 and discuss related work in Section 11.5.

11.2 CONCEPTIONAL BACKGROUND

11.2.1 *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 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.

11.2.2 *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 (Figure 11.2).

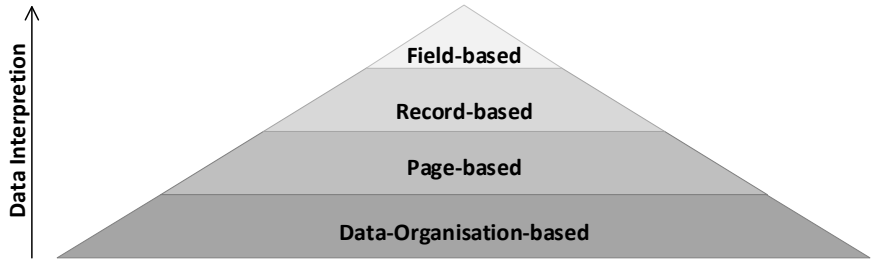


Figure 11.2: Different NDP operation types build upon each other during the execution.

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 [25]. 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 11.2.3. 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 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 11.2.3).

11.2.3 *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 11.3).

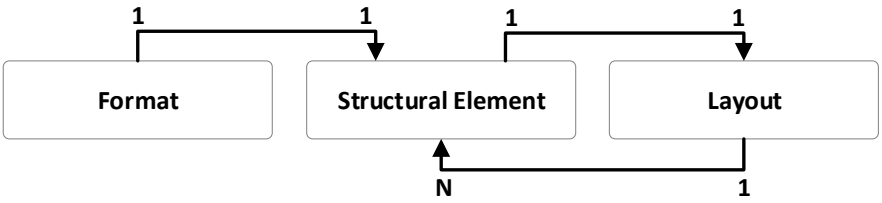


Figure 11.3: Formats describe the properties of a single structural element, while layouts define the arrangement of multiple subordinate elements.

11.2.3.1 *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).

11.2.3.2 *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 11.4 – 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*.

11.2.4 *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 wide-spread *Physical Storage Organisations* and list examples for *Format Parsers* and *Layout Accessors*. As a running example Figure 11.4 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).

11.2.4.1 *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 natural sort-order (see Figure 11.4).

11.2.4.2 *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 11.4 (and Figure 11.5) shows the Record Layout and Format and the necessary information for a Record-based NDP operation.

11.2.4.3 *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 spread representatives are the N-ary Storage Model (NSM) [21], the Decomposition Storage Model (DSM) [5], and, the hybrid between those, the Partition Attributes Across (PAX) [2].

```
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`)
);
```

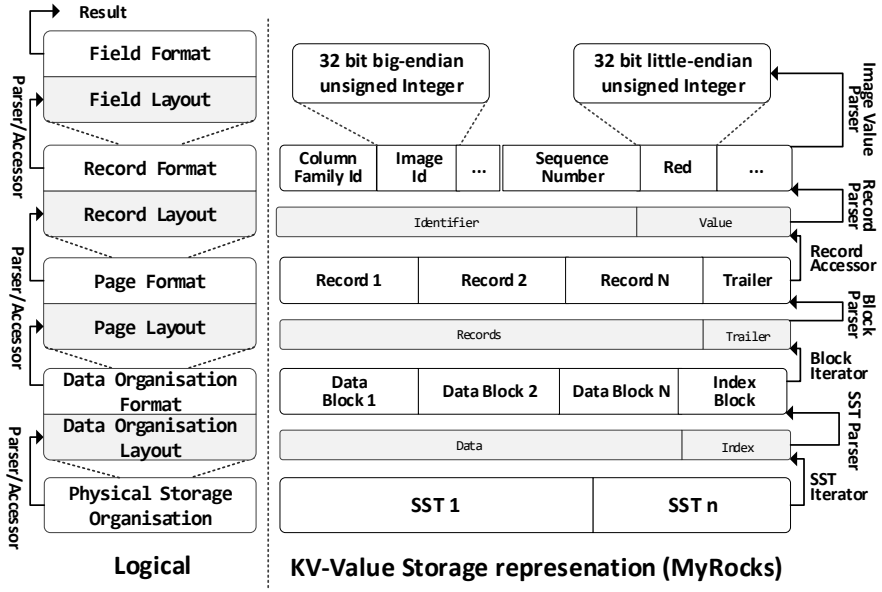


Figure 11.4: (Left) logical data organisation and nested definitions with formats and layouts. (Right) record Format and Layout of the simple NDP-ImageProcessor divide fields in identifier and value for a simple table definition executed in MyRocks.

The difference is the arrangement of records within the space of a classical page as shown in Figure 11.5. 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. IPA [11] and IPA-IDX [10] optimize for byte-level writes and write-amplification.

11.2.4.4 *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 Section 11.2.2), the organization of the under-

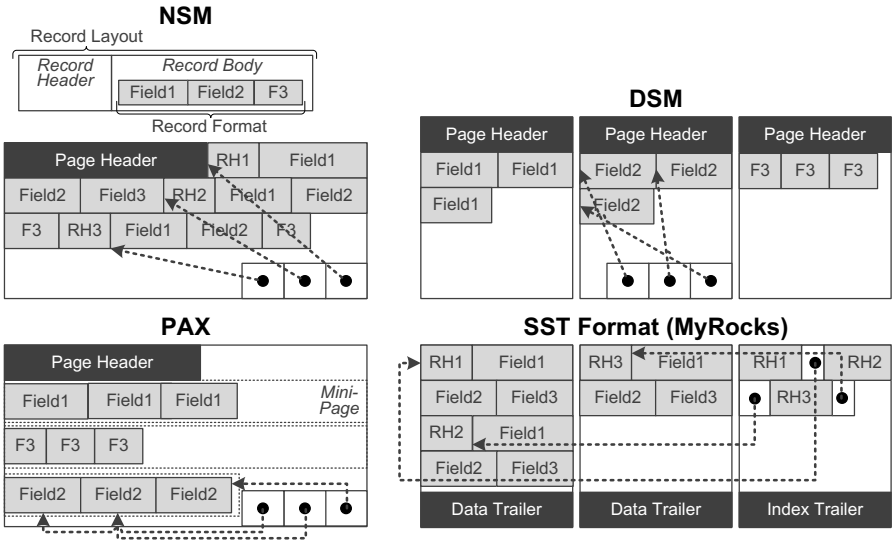


Figure 11.5: Examples of different DBMS page layouts and the SST Layout, widely used in KV-Stores like RocksDB.

lying 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.

11.3 PUSHING DOWN OPERATIONS WITH FORMAT

11.3.1 *The ImageProcessor*

After motivating the necessity of format pushdown from the conceptional perspective, we introduce a simple NDP-ImageProcessor. It uses NoFTL-KV [25], 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 11.4. 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.

Figure 11.6 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

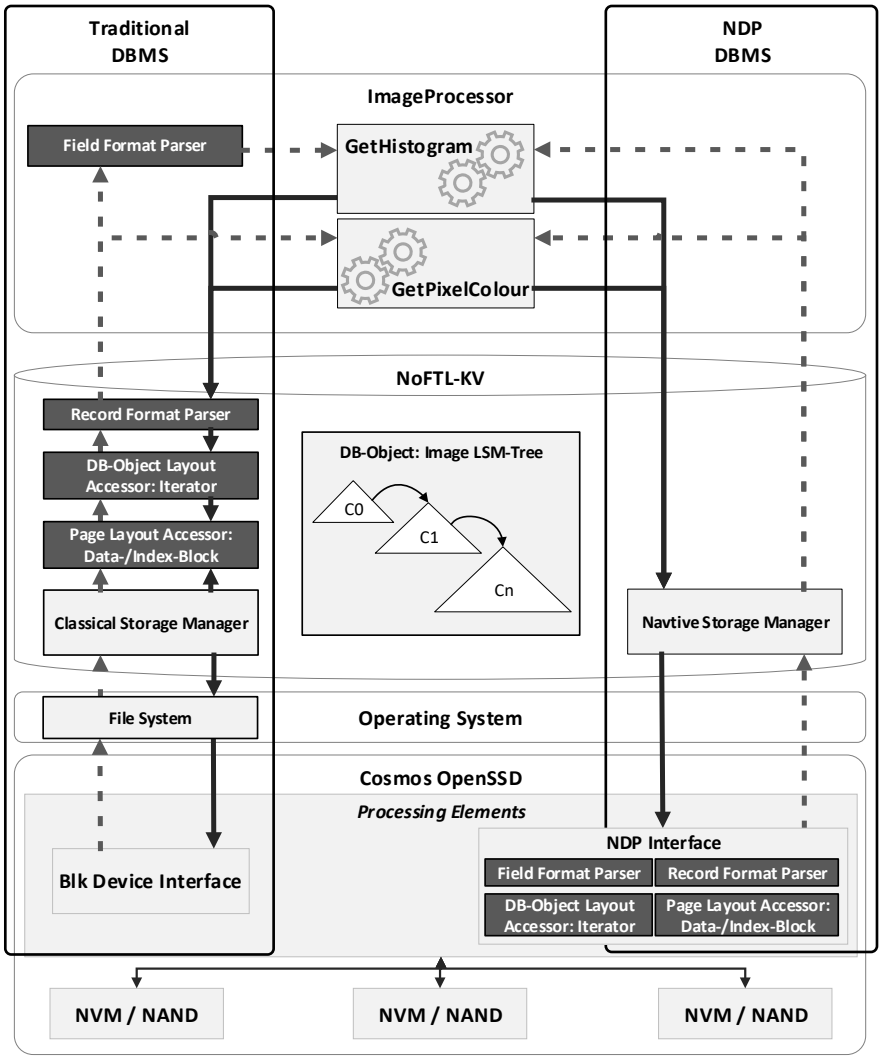


Figure 11.6: The simple NDP-ImageProcessor application runs on top of NoFTL-KV, which is based on the pluggable storage engine of MyRocks, and operates either with conventional query requests via Block I/O to the Cosmos OpenSSD, or utilises the Native Storage Manager to issue NDP calls on the device. 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.

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.

11.3.2 *Testbed*

For the evaluation the system stack shown in Figure 11.6 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 13.4) via a four lane PCIe 3.0 bus. The COSMOS platform [6] 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.

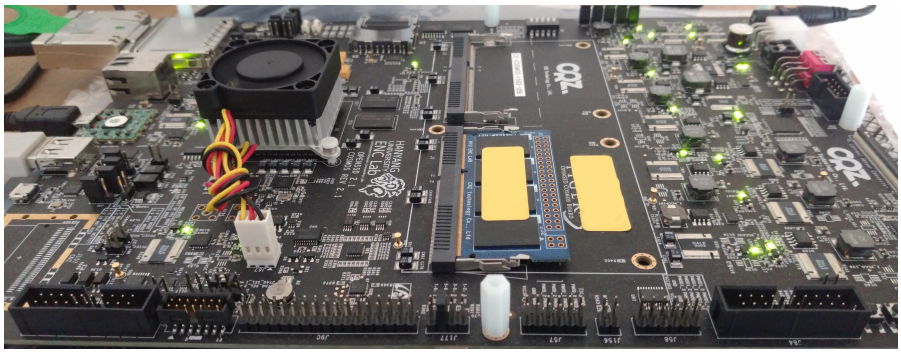


Figure 11.7: The Cosmos OpenSSD board resembles a classical enterprise SSD connected via PCIe Gen3 x4 to the host. It comprises a Zynq 7000 SoC with an FPGA and a dual-core ARM, 1GB RAM and a 512GB NAND Flash module.

11.3.3 *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.

11.3.3.1 *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 13.7.a), and once with caches for the index enabled (Figure 13.7.b).

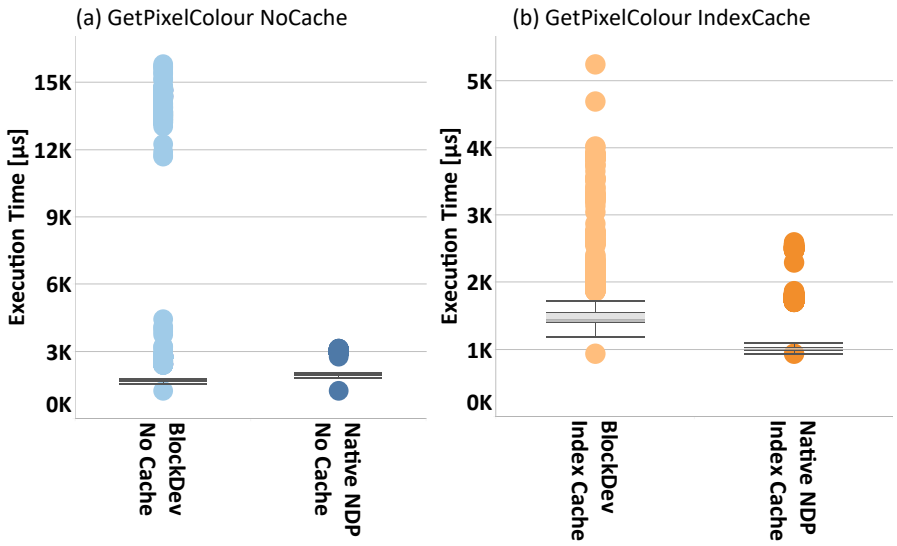


Figure 11.8: NDP calls exhibit robust performance, in general. In absence of any on-device/NDP caches, the traditional stack executes slightly faster, due to non-deactivatable caches in the operating system. However, if small on-device index caches are enabled, NDP's performance improves around 33% against 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.

11.3.3.2 *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.

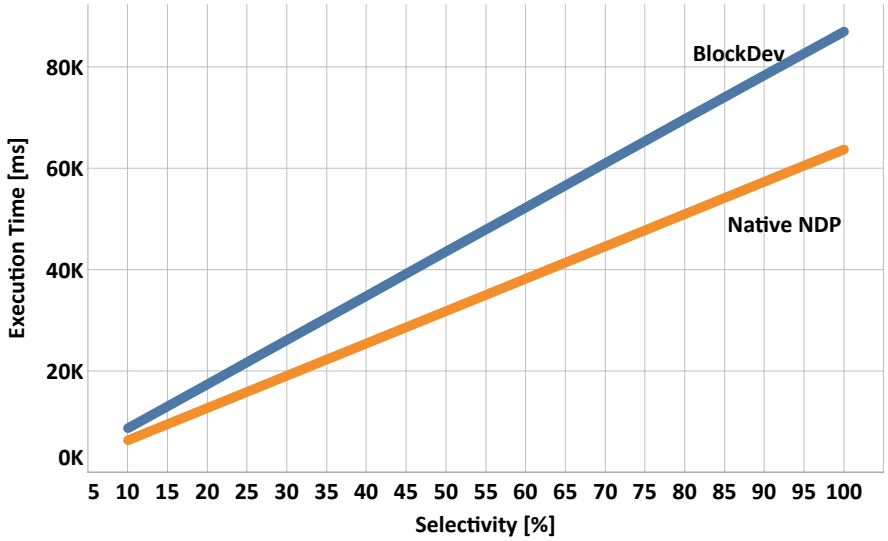


Figure 11.9: The performance of both stacks increases linearly for the given data set size across a range of selectivities. However, due to optimizations exploiting Cosmos's hardware properties improve the performance of NDP by about 27%.

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, the performance can continuously be improved by approximately 27%.

11.4 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 1/4 of the host CPU.

11.5 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 [21] and DSM [5] date back to at least the 80s. Yet, also recently, new variations were proposed like PAX [2], BLU [22] of DB2, or DataBlocks of HyPer [19]. Some layouts even make use of the hardware properties of Flash like Delta Records in [11].

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, 23] from the late 1990s.

With the advent of Flash technologies and reconfigurable processing elements Smart SSDs [8, 15, 24] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [26]. Biscuit [9] is a proposal for a general NDP framework. JAFAR [3, 27] 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 [7, 17]. Kanzi [12], Caribou [13] and BlueDBM [20] 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.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D Hill, and Marios Skounakis. "Weaving Relations for Cache Performance." In: *Proc. VLDB 01* (2001).

[3] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[4] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: ed. by A. R. Hurson, L. L. Miller, and S. H. Pakzad. 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[5] George P Copeland and Setrag N Khoshafian. "A decomposition storage model." In: *In Proc. SIGMOD 1985* (1985).

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

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

[8] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD*. 2013.

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

[10] Sergey Hardock, Andreas Koch, Tobias Vinçon, and Ilia Petrov. "IPA-IDX: In-Place Appends for B-Tree Indices." In: *Proc. DaMoN*. 2019. ISBN: 9781450368018.

[11] Sergey Hardock, Ilia Petrov, Robert Gottstein, and Alejandro Buchmann. "From In-Place Updates to In-Place Appends." In: *Proc. SIGMOD '17*. 2017.

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

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

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

[15] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST*. May 2013.

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

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

[18] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. "In-storage Processing of Database Scans and Joins." In: *Inf. Sci.* (2016).

[19] Harald Lang, Tobias Mühlbauer, Florian Funke, and et al. "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation." In: *Proc. SIGMOD 16*. 2016.

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

[21] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 2003.

[22] Vijayshankar Raman, Gopi Attaluri, and Ronald Barber. "DB2 with BLU Acceleration: So much more than just a column store." In: *Proc. VLDB 13* (2013).

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

[24] Sudharsan Seshadri, Steven Swanson, and et al. "Willow: A User-Programmable SSD." In: *USENIX, OSDI* (2014), pp. 67–80.

[25] 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.

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

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

12

NKV: NEAR-DATA PROCESSING WITH KV-STORES ON NATIVE COMPUTATIONAL STORAGE

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"nKV: Near-Data Processing with KV-Stores on Native Computational Storage"* by Tobias Vinçon, Lukas Weber, Arthur Bernhardt, Andreas Koch and Ilia Petrov in 2020 16th International Workshop on Data Management on New Hardware (DaMoN). The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and co-leading author, Tobias Vinçon was responsible for designing and implementing the nKV system including the NDP invocation of software-based parsers and accessors for the GET, SCAN and BC operations. Lukas Weber complemented the NDP functionality with hardware-based processing elements and integrated those into the COSMOS+ architecture. The experimental evaluation and the manuscript's text was a joint work of Tobias Vinçon and Lukas Weber with extensive support from the other co-authors Arthur Bernhardt, Andreas Koch and Ilia Petrov. «*

ABSTRACT

Massive data transfers in modern key/value stores resulting from low data-locality and data-to-code system design hurt their performance and scalability. Near-data processing (NDP) designs represent a feasible solution, which although not new, have yet to see widespread use.

In this paper we introduce nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. On the one hand, nKV can directly control the data and computation placement on the underlying storage hardware. On the other hand, nKV propagates the data formats and layouts to the storage device where, software and hardware parsers and accessors are implemented. Both allow NDP operations to execute in host-intervention-free manner, directly on physical addresses and thus better utilize the underlying hardware. Our performance evaluation is based on executing traditional KV operations (*GET*, *SCAN*) and on complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance on real hardware – the *COSMOS+* platform [7].

12.1 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 bandwidth in combination with poor data locality, but also result from traditional (*data-to-code*) system architectures.

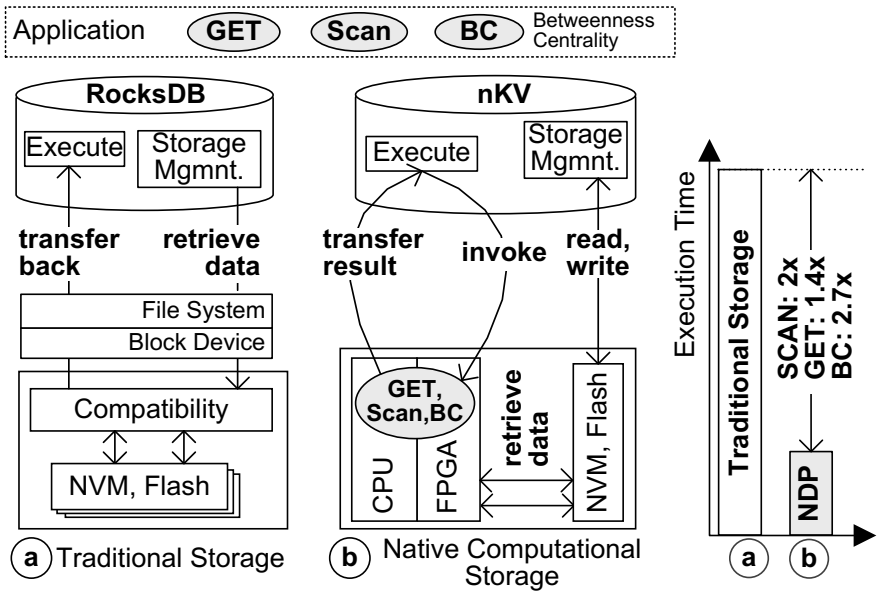


Figure 12.1: KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution. In other words, operations are executed as close as possible to the physical data location, utilizing the much better on-device I/O performance. NDP leverages several trends. Firstly, hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device – so called NDP-capable *computational storage*. As a result, even commodity storage devices nowadays, have compute resources that can be effectively used for NDP, but are executing compatibility firmware (to traditional storage) instead. Secondly, with semiconductor storage technologies (NVM/Flash), the *device-internal* bandwidth, parallelism, and latencies are significantly better than the external ones (device-to-host). Both lift major limitations of prior approaches like ActiveDisks [1, 24] or Database Machines [5].

Wide-spread, high-performance persistent key-value stores like LevelDB or RocksDB [10] tend to rely on a traditional layered-storage stack (Figure 16.1). It simplifies their architecture, allows for more

flexibility and eases data management and administration. However, layers within the DBMS (e.g. Storage Manager or access methods), but also underlying the file- and operating system encapsulate information and functionality necessary for the successful utilization of NDP techniques. *Firstly*, NDP operations executed on-device require the physical address ranges of the data to be processed. In traditional storage, address information is scattered along the layers of the storage stack (DBMS, File System, OS) and hidden behind layers of abstraction (Figure 16.1). *Secondly*, NDP-operations need to navigate through and interpret the physical data on-device. To this end data formats and layout accessors are necessary on device. However, *data format definitions* are only available within the DBMS or sometimes within the application on top. Moreover, data layouts (page or record) and traversal methods for the data organization (files or LSM-trees) are typically hard coded in the DBMS and thus not available on device.

To address the above, in this paper, we present nKV, which is a key/-value store utilizing *native computational storage* and *near-data processing* (Figure 16.1). nKV eliminates intermediary layers along the I/O stack (e.g. file system) and operates directly on NVM/Flash storage. nKV directly controls the physical data placement on chips and channels, which is critical for utilizing the on-device I/O properties and compute parallelism. Furthermore, nKV can execute various operations such as *GET* or *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores and as hardware-software NDP (HW/SW-NDP) using corresponding FPGA-based accelerators. The necessary FPGA hardware is built in the form of simple processing elements that can be used to offload certain tasks from the ARM-cores. Under nKV we target *host-intervention-free* NDP-executions, i.e. the NDP-device has the complete address information, can interpret the *data format* and access the data in-situ without host interaction. To reduce data transfers nKV also employs novel *ResultSet*-transfer modes. nKV is resource efficient as it eliminates compatibility layers and utilizes freed compute resources for NDP. nKV performs 1.4×-2× better than RocksDB: *GET latency* – 1.4×; *SCAN* – 2×; *BC* execution time – 2.7×.

This paper is organized as follows. In the next section we describe the data organization of RocksDB and the challenges it poses to NDP. In Section 13.2 we describe the architecture of nKV and how those NDP-challenges are addressed in terms of interface extensions (Section 12.3.1), in-situ data processing (Section 12.3.2), as well as operations and algorithms (Section 12.3.3). The architecture of the underlying NDP hardware accelerators is described in Section 12.5. We discuss the experimental results in Section 12.6 and conclude in Section 13.5.

12.2 BACKGROUND

In contrast to traditional data organizations, where data is updated in-place, LSM-trees [22] have been proposed as an out-of-place update structure to tackle the sustained update and insertion rates of modern workloads and provide query capabilities at the same time. Classical LSM-trees [22] comprise multiple B-Tree-structured index components (C_0 to C_K, Fig. 12.2) that are stored in new locations and have constant size ratios $r = |C_{i+1}|/|C_i|, i \in [0, K)$. An insert or update operation hits the C_0 component that is located in memory. Once it reaches a size threshold, it is flushed to disk and is merged with the C_1 component. The merge processes gradually move data from C_0 to C_K, purge outdated KV-Pairs, reclaim space and indirectly ensure hot-cold data separation.

nKV builds on RocksDB [10], which introduces one independent LSM-Tree per column family to separate the access characteristics of different database objects. Modern LSM-Tree variants (surveyed in [20]) are multi-levelled. Modifications to an LSM-Tree are first placed in the main memory component C_0, which comprises a set of *MemTables* in RocksDB. These are realized as memory-efficient data structures such as SkipLists. Whenever a MemTable reaches a given size limit, it becomes *immutable* and a new one is created to accommodate further modifications. Later on, immutable MemTables are transformed into *Sorted String Tables (SST)* and flushed to the secondary storage (Fig. 12.2), whereas each LSM-tree component $C_1..C_K$ comprises multiple SSTs. Thereby, the contained key-value pairs are placed into multiple *data blocks* in sort order of the key. Furthermore, an *index block* that comprises key-offset pairs pointing to each data block (a sparse index) is prepended. Index blocks reduce the access complexity to key-value pairs within the SST.

During the flush to C_1 no merge occurs for performance reasons. Consequently, overlapping key-value ranges of SSTs can occur (consider SST_{12}-SST_{1n}, C_1, Fig.12.2). Merge steps to underlying layers $C_2 \ldots C_K$, called compactions, take either SSTs only on the level above or combine them with SSTs on the target layer, based on the given strategy (e.g. tiered or levelled). Either way, all KV pairs of the input SSTs are sorted, out-dated entries are pruned, and the results are stored in new SSTs on the target level (see dotted box, Fig. 12.2). Hence, key ranges in SSTs bellow C_1 do not overlap anymore. Yet, keys may appear on multiple levels with different values (consider `Key11` or `Key70`), *to account for the temporal distribution of updates to a given key-value record*. For instance `Key70` has been updated multiple times: `Key70` on C_1 is the most recent record and its existence invalidates `Key70` on C_2 and C_3.

To *retrieve* a key-value record based on the key, the `GET(key)` first traverses the MemTables and the immutable MemTables on C_0. If the

respective key is not found, the index block of one or more SSTs in C_1 has to be read (as SSTs may overlap on C_1, but not on C_2...C_K). By parsing the key-offset pair, the data block, which might contain the key, can be identified and also has to be read from secondary storage.

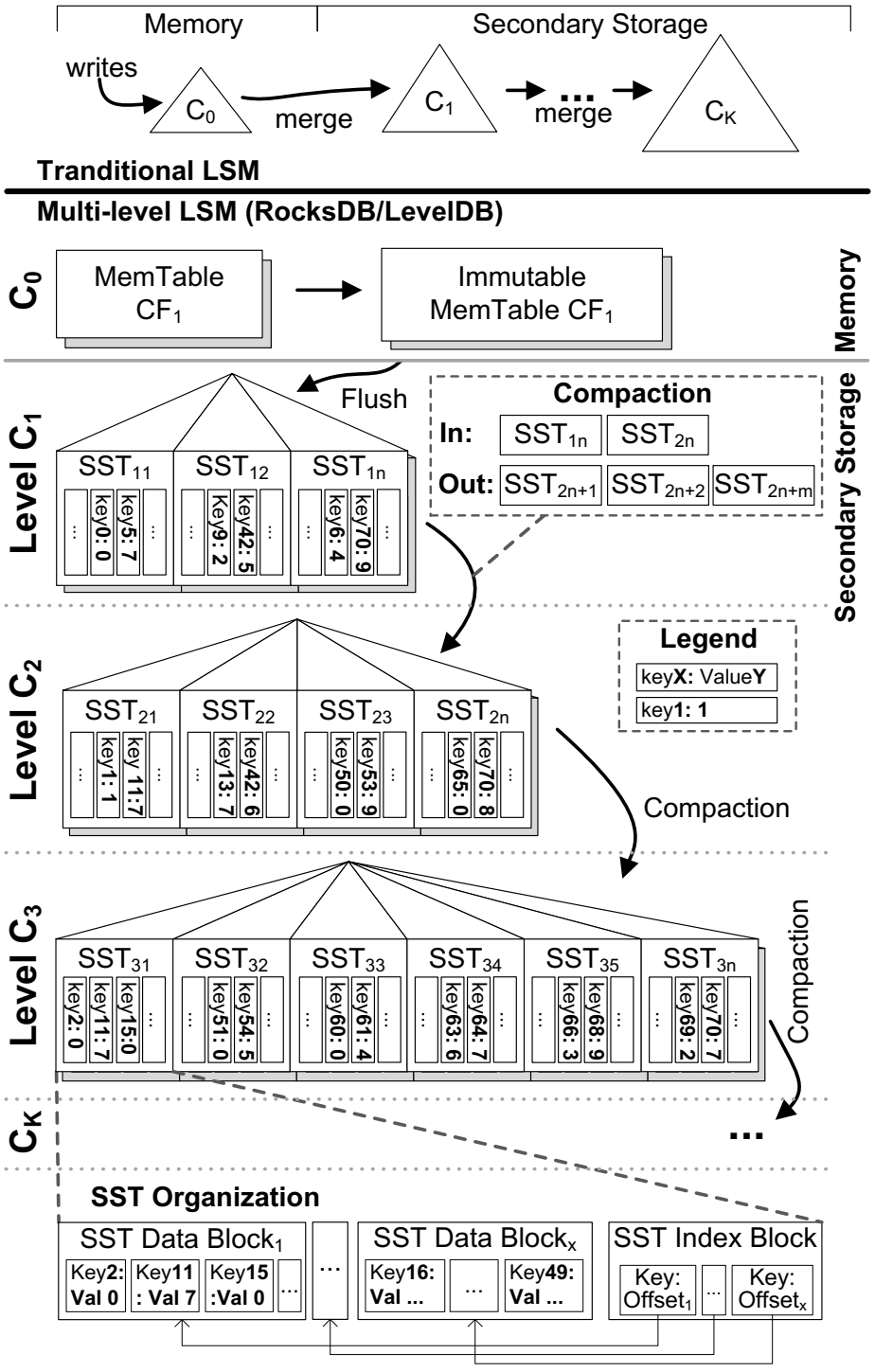


Figure 12.2: Conceptual organization of the multi-level LSM-Trees in RocksDB/LevelDB.

If the key is still not found, layers (C_2...C_K) have to be traversed similarly. Due to the data organization and the compaction process, a key can now reside only in a single SST per level. *Range scans* with or without key predicates behave similarly, but are more complex and are supported by other internal structures (like fence pointers). Consider SCAN([Key68, Key70]), which traverses all levels and retrieves Key70 from C_1, and Key69 and Key68 from C_3.

However, if a scan involves *value predicates*, e.g. SCAN($0 \leqslant Val \leqslant 7$), the only option is to iterate over the entire dataset, yielding a significant increase of I/O transfers, which in turn has enormous potential to be improved via NDP.

12.3 ARCHITECTURE OF NKV

Native computational storage. One of the underlying design principles behind nKV is that native storage enables efficient NDP (Figures 16.1 and 13.2). In this sense nKV extends [28]. *Native storage* is storage that is operated without intermediary/compatibility layers of abstraction along the critical I/O path, and is directly controlled by the database. This means that nKV can directly operate on NVM/Flash storage using physical addresses and thus can precisely control physical placement of SST data. It is this physical placement that allows utilizing the on-device I/O bandwidth and the FPGA's compute parallelism.

nKV physically places *SST data blocks* and *SST index blocks* on different *LUNs* and *Channels* (see Figures 12.2 and13.3). This allows for reaching the internal bandwidth (Table 12.2) by requesting index and data blocks asynchronously and utilizing processing parallelism of FPGA-based processing elements (PEs). Besides, individual levels of the LSM-Tree are physically separated on different chips and LUNs to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device, reducing demand pressure on the bus.

Furthermore, nKV operates directly with physical addresses, to access (read or write) precisely the physical pages that are needed. This, in turn, is essential for reducing read- and write-amplification.

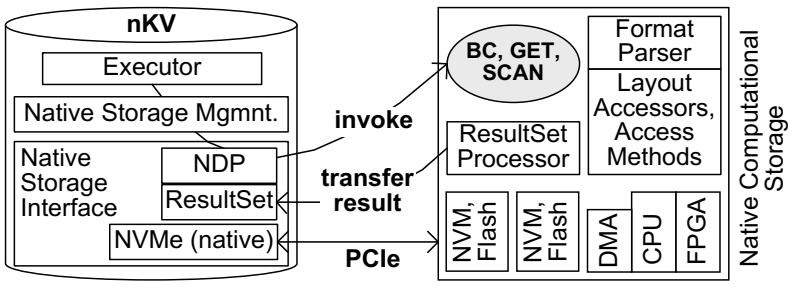


Figure 12.3: Architecture of nKV.

Moreover, it inherently avoids costly host round-trips for logical-to-physical address translation. Native storage eliminates these *information hiding* effects incurred through layers of abstraction and thus simplifies the NDP-operations. Hence, native storage leads to leaner NDP-functionality.

Computation Placement. By using native computational storage, nKV can directly place computations on the heterogeneous on-device compute elements, such as ARM-cores or FPGA-based processing elements. nKV can execute various operations such *GET*, *SCAN* or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM-cores or with hardware support from the FPGA (cf. Section 12.5). The experimental evaluation indicates that some NDP operations such as *NDP_GET(key)* perform best on the ARM-cores, while other operations like *NDP_SCAN(value_condition)*, benefiting from parallelism, perform best on the FPGA. For its NDP-operations nKV utilizes *hardware/software co-design* to handle the proper separation of concerns and achieve best performance.

12.3.1 *NDP Interface Extensions*

NVMe support. nKV has a dedicated high-performance *user-space* and *in-DBMS* NVMe layer (Figure 13.2). It is very lean and tightly integrated with the rest of nKV. The *native NVMe* integration can control multiple NVMe submission and completion queues either through dedicated threads or through the transactional context. Moreover, it reduces the I/O overhead as it allows the seamless creation of I/O and NDP tasks, the precise allocation of transfer buffers for the DMA engine, and prioritizable placement within the NVMe submission queues. The deep database integration additionally avoids expensive synchronization between user- and kernel-space, and shortens the I/O paths even further as no drivers are involved along the critical access paths. Internally, the native storage command set is translated to specific NVMe I/O and NDP tasks. Although these resemble the standardized NVMe commands, they define a new category - n over NVMe. In nKV, they can be scheduled either for *synchronous* or for *asynchronous* execution.

Command set. Besides the classical native storage interface, nKV introduces NDP-Extensions [28] in terms of a generic *NDP_EXEC()* command. It takes the following parameters, among others:

(i) *Command Identifier* – Unique identifier of the NDP function;
(ii) the *SearchKey* or *SearchKeyRange(s)* – for GET or SCAN;
(iii) the *ResultsSet Size*;
(iv) *AddressMappings* – these are ranges of physical addresses, where the physical data to be processed is located;

(v) *Min/Max Keys* – RocksDB supports a type of zone map range filter that can be used on device to skip processing some index/-data blocks;
(vi) *Miscellaneous* – command specific parameters such as data format definitions.

nKV composes the NVMe command based on the given parameters, current state and address information, and its transactional context. After placing it in the NVMe submission queue and DMA transferring the parameters to the device, the NDP command is then executed. The result set is handled by the ResultSet processor, which also observes the execution status. Finally the ResultSet is transferred to nKV by the DMA engine. An NDP-operation can invoke multiple low-level NDP commands synchronously or asynchronously.

12.3.2 *In-situ Data Access and Interpretation*

Under nKV, the NDP-device can interpret the *data format* and *access* the data without *host intervention* (synchronization with the host) [29]. To this end, nKV extracts definitions of the *Key- and Value-formats*, and passes them as input parameters to the NDP-commands. Moreover, the *data format* such as the *Key- and Value-formats* can be automatically extracted from the DB-catalogue (*system-defined*) or can be defined by the *application*.

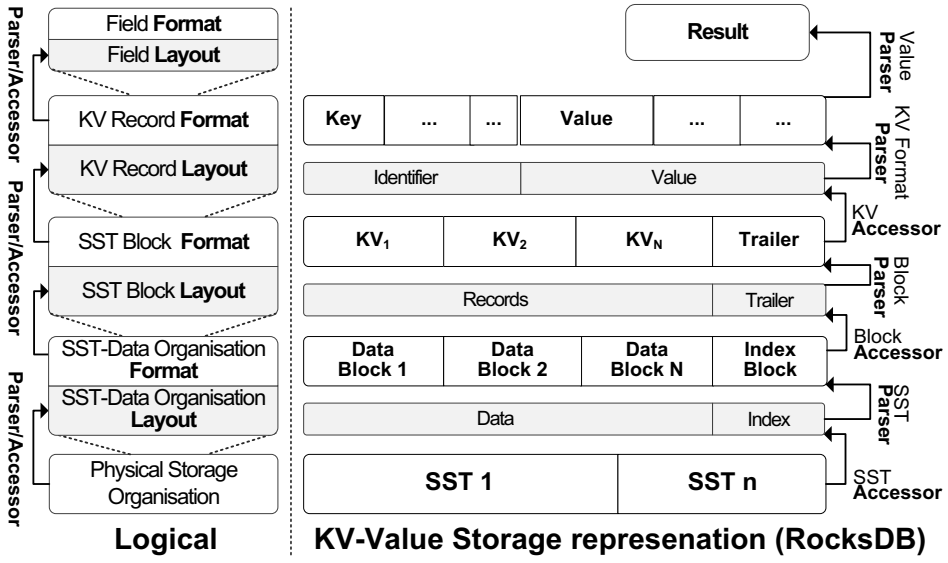


Figure 12.4: In-situ access and data interpretation in nKV, based on layout accessors and format parsers.

nKV employs a thin on-device *NDP-infrastructure* layer that supports the execution and simplifies the development of NDP-operations (Figures 13.2). It comprises *data format parsers* and *accessors* that are implemented in both *software* and *hardware* (Figure 13.3). The in-situ

accessors traverse and extract the contained sub-entities of the persistent data. Whereas, the in-situ *data format parsers* process the *layout* of each persistent entity, and extract the sub-entities by invoking further accessors (Figure 13.3).

KV-Stores like LevelDB or RocksDB [10] organize the persistent LSM-Tree data into so called *String Sorted Tables (SST)* – see Sect. 15.2. To process a *GET(key)* request, for instance, nKV first identifies the respective *SST* and invokes an *NDP_GET* command with the corresponding physical address ranges, the respective *Key- and Value-formats* as well as further parameters. First, the *SST layout accessor* is invoked to access the data and the index blocks. Subsequently, the index block parser is activated to interpret the data and verify if the *key* is present and extract its offset. If this is the case, data block accessor and parser are invoked to extract the *Key/Value entry*. In case of an *NDP_SCAN(key_val_condition)* operation, the KV accessor is subsequently invoked to extract it, followed by a *field* parser to extract its value and verify the *condition*. This way, nKV extends scans in classical KV-Stores. Typically, they are only able to process filter criteria on key-embedded attributes, but not filter predicates involving the value.

12.3.3 *Operations and Algorithms*

Lookup. KV-Stores offer fast (low-latency, high parallelism) retrieval of a value, based on its key, through the *GET(key)* operation. In nKV, this operation first performs a lookup within main-memory components (MemTables, Fig. 12.2) regardless of execution model.

If the search key is not found, the lookup will proceed scanning the deeper LSM-Tree levels by processing their indices first and eventually their associated data blocks. Both, index and data block scanning are I/Os intensive in the traditional stack (Figure 16.1), while with NDP, these can be performed efficiently on-device.

Scan. As mentioned in Section 12.3.2, Scans can be performed either on Key- or Value-embedded attributes. The index blocks of the LSM-Tree might be leveraged to navigate to necessary data blocks for key-attribute scans, similar to the Lookup operation. However, there is no auxiliary structure to accelerate scans on value-embedded attributes. Either way, multiple data blocks have to be examined depending on the selectivity of the filter-predicate. Consequently, Scans usually result in a high amount of data transfers, which NDP can significantly reduce.

UDF: Betweenness Centrality. Many applications involve more complex algorithms. Such user-defined functions (i.e. Betweenness Centrality) can also be supported by nKV. The specific algorithm implemented within nKV relies on [6] and measures the degree, to which nodes stand among each other. The logic involves shortest-path searches over

the given nodes and therefore results in a variety of lookups and scans involving random and sequential I/O.

12.3.4 *Data Consistency, Database Maintenance and NDP*

In parallel to the execution of NDP functionality, nKV allows the processing of database maintenance e.g. compactions. Such parallel operations might result in new data or even changes to the LSM-Tree. Yet, as nKV's NDP-operations are executed on a snapshot of the physical data, concurrent modifications do not effect its consistency. This can be ensured by firstly, the underlying mechanism of Copy-on-Write (CoW), secondly the precise placement through the native storage interface, and thirdly, the direct control of the physical GC by nKV. Moreover, nKV requires no on-device bad-block re-mapping like other native storage management solutions [4], since bad-block management and wear-leveling are handled directly within the database engine [23]. Thus native storage management [23] leverages the the above issues by DBMS managed physical-to-logical address mapping and data placement.

12.3.5 *Result Set Handling*

Unnecessary data transfers may occur depending on how the result of an NDP-operation is transferred back. Therefore, ResultSet management additionally helps to avoid unnecessary stalls of processing resources. nKV aims to reduce the data transfer overhead caused by a Volcano-style *record-at-a-time* model. Instead it aims to bulk-transfer the *ResultSet*. The former is simpler, but leads to more frequent shorter burst transfers causing bus overhead. The latter results in fewer, but longer bursts leveraging the throughput-optimized PCIe.

Each NDP call in nKV defines a maximum ResultSet size as a parameter. The NDP ResultSet-Processor (Figure 13.2) allocates on-device resources for it: either in DRAM, or if the contention is high, it allocates a temporary region on Flash. As long as the actual result size does not exceed the predefined one, the ResultSet is sent back as bulk DMA-transfer, to leverage the full performance of the DMA engine. Alternatively, it may be pipelined to the next NDP-operation. Furthermore, nKV has a built-in extension mechanism that in the worst case may preemptively request more physical space from the DBMS, as it manages the logical-to-physical address mapping [23].

12.4 HARDWARE-ARCHITECTURE

The *COSMOS+* platform [7] is a PCIe-based extension-card. It contains all the required hardware-modules to function as a regular NVMe-based SSD. It can be fitted with up to 2 DIMM-extensions containing

Flash modules. The available Toggle-NAND Flash-extensions can be configured in SLC or MLC mode. In this work, they are configured as SLC with 16 dies organized in two channels.

The main computational engine of the *COSMOS+* platform is a Xilinx Zynq-7000 SoC (XC7Z045-3FFG900) that combines two 667 MHz ARM Cortex-A9 cores with an FPGA (Figure 12.5). In the The *COSMOS+* platform [7], the FPGA-portion is used to implement the required SSD-infrastructure. This infrastructure is made up of two separate domains: The first one is responsible for accessing the flash memory. It comprises one or many Tiger4-controllers with corresponding low-level flash interfaces. For each channel, a distinct Tiger4-controller, as well as a low-level interface, is required.

The second domain contains primarily an NVMe-Core, which allows access from the host to the device via the NVMe interface. The NVMe-Core also wraps the actual low-level PCIe-interface.

Both of these domains are running at different clock-frequencies. While the flash domain uses a 100 MHz clock, the NVMe-Core is running at 250 MHz. When planning to extend this platform, the following aspects are relevant:

1) The amount of resources on the FPGA-portion (PL) of the SoC is limited. While the platform can be fitted with more flash-DIMMs, this also requires more flash controllers. This in turn impacts the resource requirements. In this work, one flash-DIMM is used with 2 flash controllers. While this limits the available flash memory and the corresponding parallelism, it also frees up resources for different use (i.e. computational processing elements).

2) Since the different domains are running at different clock-frequencies, the extensions should be able to run at the same clock-frequencies. In the case of the *COSMOS+* platform, this is not a huge problem, since most hardware-accelerators can run at 100 MHz and can therefore reside in the flash-domain.

A simplified view of the architecture is depicted in Figure 12.5. It also includes the nKV hardware *extensions* described in this work (Section 12.5).

12.5 HARDWARE-ACCELERATION

Using the baseline architecture (Figure 12.5), specific processing elements (PEs) are implemented, allowing to move computation from software running on the ARM-cores to the programmable logic of the SoC, potentially improving latency, throughput, and available parallelism. The PEs are written in Chisel3 [3] using a relatively simple architecture that can be subdivided into four distinct domains (cf. Figure 12.6):

The control-domain consists of a register file, holding a number of control registers, which are accessible using an AXI4-Lite interface.

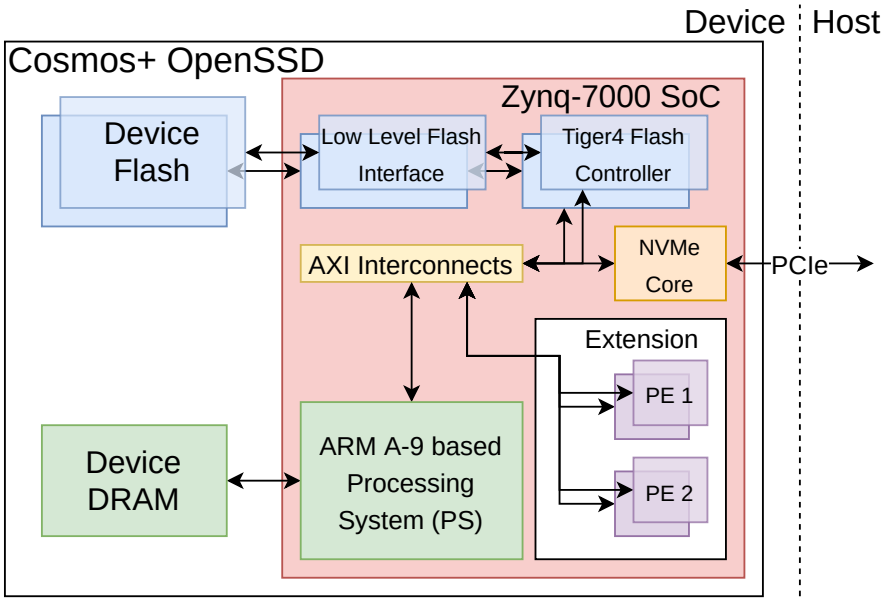


Figure 12.5: A simplified view of the architecture of the *COSMOS+* OpenSSD [7], including the proposed extension.

The corresponding addresses are mapped into the address space of the processing system (PS), so that the ARM-cores can read and write these registers and thereby control and configure the PE. The control registers hold the required parameters for the functionality provided by the processing elements (e.g. the memory addresses of the input and output). In addition, the signaling to the ARM-cores is also done using these registers. One register indicates whether the PE is busy, while another can be used trigger the execution.

The memory-domain contains a load and a store unit. These are connected to the PSs DRAM-interface, allowing the PE to access the device DRAM. Both the load and the store unit perform data transfers in chunks of 32 KB, which corresponds to the size of a single data-block in our RocksDB-configuration. The transfers are performed using AXI4 bursts to maximize throughput. The data-width of the AXI4-Bus is 64 bits and the AXI burst length is 16. Generally, longer bursts allow higher throughput, due to the sequential access pattern. Unfortunately, the Zynq-7000 family only supports a maximum burst length of 16.

The accessor-domain is responsible for converting the different data granularities (64 bit words vs tuples) between the memory and the computational domain. The tuple input buffer will buffer incoming 64 bit data words from the load unit, until a complete tuple is available. This will then be passed to the filtering unit. Similarly, the tuple output buffer will receive a resulting tuple and split it into words of 64 bits to allow transfer to memory via the store unit. In this context, a tuple corresponds to a key-value pair (kv-pair).

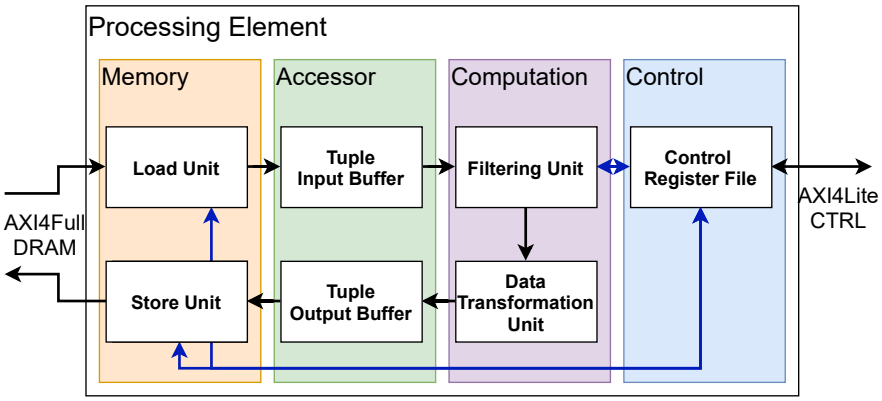


Figure 12.6: The overall Microarchitecture of the proposed Parser Processing Elements.

The computational domain is comprised of two distinct modules. The first one is the filtering unit, which accepts single kv-pairs as input. Depending on the control registers, the filtering unit will then pass on matching kv-pairs to the data transformation, while non-matching ones are discarded. In the current implementation, the filtering unit can be configured to apply a single predicate on a kv-pair. This is done using three parameters: the column selector ($i \in [0, n-1]$, where n is the number of distinct data-fields), the compare operator ($op \in \{nop, =, \neq, >, \geqslant, <, \leqslant\}$) and a reference value (c) to compare against. Considering a kv-pair $t = (t_0, t_1, t_2, \ldots, t_{n-1})$, the following expression is evaluated: $r = t_i \ op \ c$. If r is true, the kv-pair will be passed on, else it will be discarded.

The last module transforms the data into the required output format. This corresponds to a projection of tuples and allows the automatic removal of RocksDB-metadata or unnecessary tuple elements. The transformed tuple is passed back to the accessor-domain, to be stored back to the device DRAM. The complete microarchitecture of the PEs is also depicted in Figure 12.6.

Building atop of the baseline architecture of COSMOS+ (Figure 12.5), we developed an extended architecture, which contains additional processing elements. In particular, we built two different processing elements for the specfic evaluation dataset (the database-of-research-papers): One for the data of the paper themselves (paper-PE), and another one for the data of the references (ref-PE). Initial experiments showed, that the paper-PE can process a 32 KB block of data faster than the two Tiger4s controllers are able to provide it (due to the flash latency). Thus, we employ a single paper-PE in the final architecture. For the handling of the paper references in the database, much more data has to be processed multiple times. In this case, the flash latency becomes less relevant, since the reference data is cached in the on-device DRAM and does not have to be fetched from flash memory each time. Thus, the architecture can keep multiple ref-PEs busy. To

keep the interconnects balanced, we opted for seven ref-PEs, yielding a total of eight PEs (including the single paper-PE). Generally, it would be possible to increase the number of PEs, but all active PEs compete for access to the on-device DRAM. Thus, there will be a point of diminishing returns considering overall throughput as soon as the full memory bandwidth is saturated. Instead of replicating PEs for more throughput, it might be more reasonable to use multiple different PEs to increase flexibility of the hardware acceleration.

Table 12.1: FPGA-Resource Utilization of the Baseline and extended Architectures, including hierarchical utilizations of relevant sub-modules.

	Slices		BRAM		DSPs	
	abs.	%	abs.	%	abs	%
Baseline	14544	26.61	78	14.31	0	0
Tiger4	8174	5.81	15	2.75	0	0
NVMe-Core	4312	7.89	29	5.32	0	0
LL Flash	475	0.87	5	0.92	0	0
Extended	35667	65.26	101	18.53	0	0
paper-PE	33103	15.14	0	0.0	0	0
ref-PE	4012	1.84	0	0.0	0	0
Available	54650	100.00	545	100.00	900	100

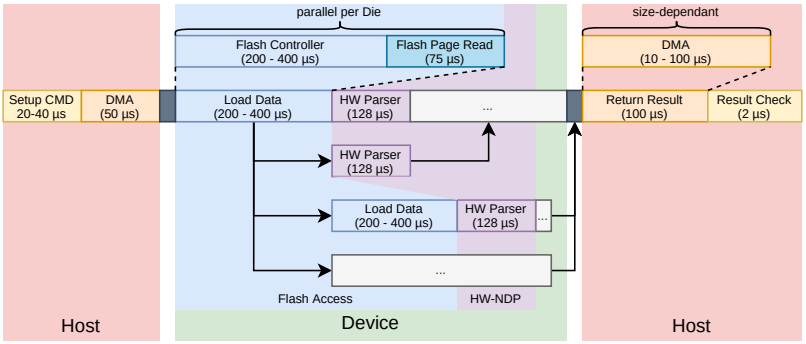


Figure 12.7: Break-Down of Execution Times within the NDP Stack with HW support.

The baseline and extended hardware designs were synthesized and implemented using Xilinx Vivado v2019.1. The resulting resource utilizations are reported in Table 12.1, both in terms of absolute numbers, as well as relative to the resources available on the Zynq 7045 chip. The baseline results indicate that the Zynq has many spare resources. Even though small, a large fraction of its hardware resources are unused. The main reason for this lies in the flash-configuration. For a platform with two flash-DIMMs and the full parallelism, eight flash controllers and flash interfaces are needed. In our design, we only use

one flash-DIMM with two controllers/interfaces, which vastly reduces the resource-requirements.

These free resources are then leveraged by our extended architecture to offload computations from the ARM-core to the FPGA. In doing so the hardware accessors and format parsers can be instantiated multiple times. In fact, nKV uses two different kinds of parsers with up to seven instances.

Another interesting point is the vast difference in resource requirements between the paper-PE and the ref-PE. The reason for this lies in the different sizes of the parsed kv-pairs. The kv-pairs processed by the paper-PE are 136 bytes each, while the kv-pairs processed by the ref-PE are merely 36 bytes each. Apart from the data-size, the number of distinct data fields also plays an important role, due to the number of required comparators.

Finally, there is a complete lack of DSP utilization. DSPs are hard-wired special-function slices which provide fast arithmetic and logical operations, that are typically relevant in the context of digital signal processing. For our work, DSPs could be interesting for arithmetic comparisons as well as pattern recognition.

For future extensions of this work, the above could be exploited to reduce Slice-utilization, or to implement more complex functionality within the PEs.

12.6 EVALUATION

For the evaluation, the *COSMOS+* board [7] (see Figure 12.5) is attached over PCIe 2.0 x8 as an NVMe block device supported by Greedy FTL to realize traditional storage. The host server is equipped with a 3.4 GHz Intel i5, 4GB RAM and runs Debian 4.9 with ext3. We configure both RocksDB [10] and COSMOS+ [7] with a 5MB cache. COSMOS+ is directly mapped into the userspace and controlled by *native NVMe*.

We evaluate nKV on a *2.4GB* research paper graph dataset from Microsoft Academic Graph [27]. It comprises approx. *48 million* Key/Value-pairs: *3.8M* papers, *40M* references, *18K* venues, and *4.2M* authors. For each experiment, we report the average execution times of three cold test runs. The baseline for our experiments is RocksDB using block-device storage (*Blk*) on top of *GreedyFTL* and *ext3*. Performance results of GET(key), SCAN(value_predicate) and BC are reported for three different stacks: *Blk* as baseline, software NDP (*NDP:SW*) on the ARM and FPGA-assisted NDP (*NDP:SW+HW*).

12.6.1 *Low-level Flash Properties*

Physical data placement and the on-device Flash characteristics play an essential role in nKV. The following Table 12.2 shows the on-device

latency and bandwidth, measured by directly issuing page reads to the Flash Management Unit. The level of parallelism is controlled by data placement on either different Channels, LUNs or both. While a single page-read takes approx. 300 µs, careful data placement on Channels and LUNs reduces latency down to 94 µs with full parallelism (Table 12.2). However, an upper limit of around 217 MB/s can be observed for sequential and random workloads, which is due to the bus limitations of COSMOS+.

Table 12.2: Flash Latencies and Bandwidth (BW) of the *COSMOS+ OpenSSD* for different levels of parallelism.

	Pages	**Parallelism**	**Duration per Page [—s]**	
	1	1 Ch. 1 LUN	300.00	
	4	2 Ch. 2 LUN	113.50	
	8	2 Ch. 4 LUN	*94.12*	
Access	**Pages**	**Parallelism**	**BW [MB/s]**	**IOPS**
Random	1500	1 Ch. 1 LUN	52	3000
	1500	2 Ch. 1 LUN	102	6000
	1500	1 Ch. 8 LUN	108	6000
	1500	*2 Ch. 8 LUN*	*213*	*13000*
Seq.	*640*	*2 Ch. 8 LUN*	*217*	*13000*

12.6.2 *Experiment 1: Lean Native Stack*

One important conceptional characteristic of NDP with nKV is the removal of traditional compatibility layers to simplify the access stack. To verify this property, we execute a *GET(key)* command. We compare the results of our baseline (Blk) against nKV with software NDP (NDP:SW), and nKV with Parsers-PE support (NDP:SW+HW) – Figure 12.8.

nKV utilizes the native data placement and **improves the round-trip time by 1.4**× due to *native NVMe* and mapping the device into its userspace. This reduces the execution time from 7.9 ms to 5.7 ms, as shown in Figure 12.8. Interestingly, there is no benefit from hardware PEs since the gains from concurrent executions are limited due to the sequential nature of first reading and then processing Flash data.

12.6.3 *Experiment 2: Data Transfer Reduction*

While the relatively simple GET-operation does not benefit from the hardware PEs, this changes for bigger and more complex operations like the SCAN. In addition to the vastly reduced latency, the use of NDP has additional effect on the overall system. While all three implementations read similar amounts of data from flash (492.3 MB ±

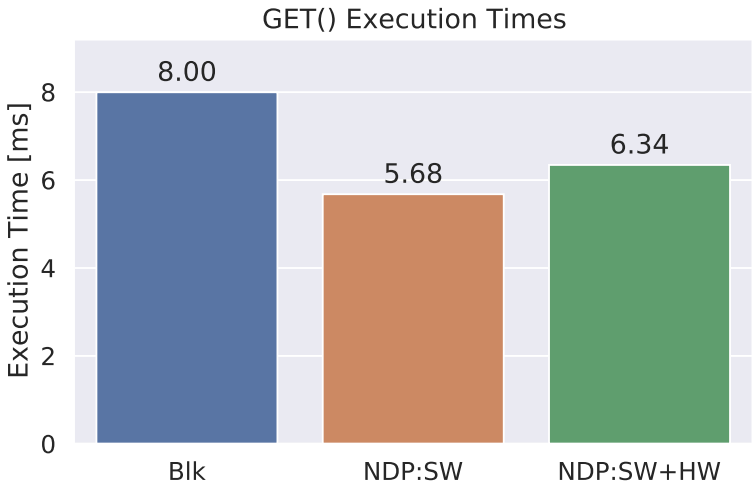


Figure 12.8: GET execution times for Blk, NDP:SW and NDP:SW+HW.

0.2 MB due to caching), the required DMA data transfers vary. For NDP-operations, a single DMA transfer is required to push down the additional parameters. We draw the following conclusions. Firstly, efficient ResultSet handling reduces the transfer overhead by employing large bulk DMA transfers. Secondly, due to on-device filtering the amount of data to be transferred also decreases depending on the predicate selectivity.

The execution time is reduced by **more than 2x** (from 6.95 s to 3.35 s) as shown in Figure 12.9.

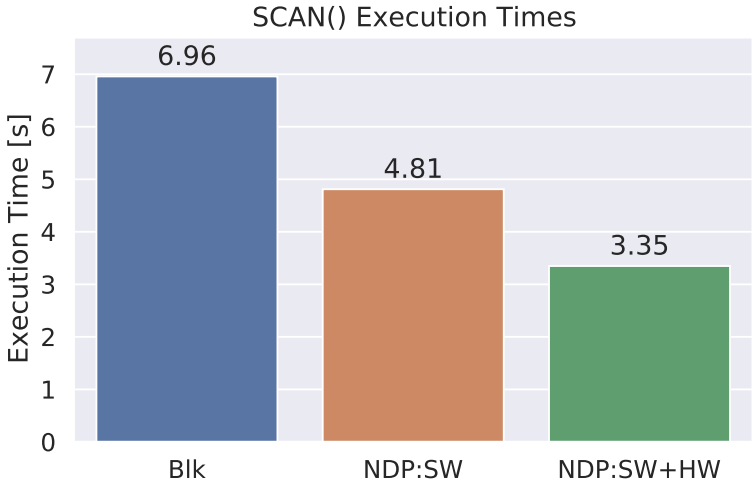


Figure 12.9: SCAN execution times for Blk, NDP:SW and NDP:SW+HW.

12.6.4 *Experiment 3: Native Computational Storage*

Native Computational Storage plays an essential role for nKV. Especially with complex graph analysis operations like Betweenness Centrality (BC) the concepts of native data placement, flash parallelism and computation placement can be leveraged to improve the performance. An execution on a smaller graph, benefits the software implementation. For a large number of edges, the complexity is high and multiple HW Parsers can be utilized to improve performance. With a total of 7 HW Parsers nKV achieves **2.7x speed-up** for 2.037.755 edges (Figure 12.10).

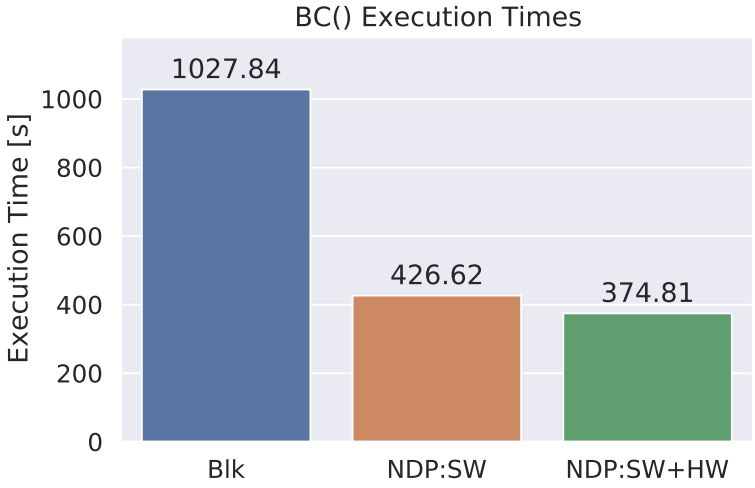


Figure 12.10: Betweenness centrality (BC) execution times for Blk, NDP:SW and NDP:SW+HW.

12.6.5 *Experiment 4: Execution Parallelism*

In large graph processing the number of applied HW Parsers is important to balance between FPGA utilization, memory bandwidth limitations and performance. nKV allows configure the number of HW Parsers individually for each NDP operation. In Figure 12.11 BC is executed with 2.037.755 edges using a different number of ref-PEs. Clearly, increasing the number of PEs per operation, yields better speed-ups. Using seven PEs instead of three PEs results in a **speed-up of 1.25x**. While the data suggests that more parsers are better, it is important to note that all instances compete for DRAM accesses. Thus, adding more parsers will yield diminishing returns due to memory contention and increased randomness in the memory access patterns.

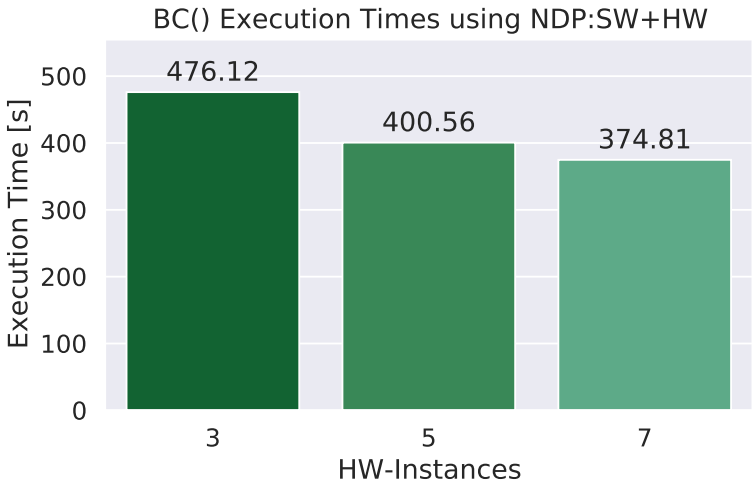


Figure 12.11: Betweenness centrality (BC) execution times for for NDP:SW+HW using 3, 5 and 7 instances of the ref-PE hardware parser.

12.7 RELATED WORK

The Near-Data Processing is deeply rooted in *database machines* [5] developed in the 1970s-80s or Active Disk/IDISK [1, 17, 25] from the late 1990s. Besides dependence on proprietary and costly hardware, the I/O bandwidth and parallelism are claimed to be the limiting factors. While not surprising, given the characteristics of magnetic/mechanical storage combined with Amdahl's balanced systems law [11], this conclusion needs to be revised. Storage devices built with modern semi-conductor storage technologies (NVM, Flash) are offering high raw bandwidths with high levels of parallelism *on-device*.

With the advent of Flash technologies and reconfigurable processing elements, Smart SSDs [9, 16, 26] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [30]. Biscuit [12] is a timely proposal for a general NDP framework. JAFAR [2, 31] is one of the first systems to target NDP for DBMS (column-store) use, whereas [15, 19] target joins besides scans. The use of NDP in the realm of KV-Stores has been investigated in [8, 18]. Kanzi [13], Caribou [14] and BlueDBM [21] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on persistent KV-Stores and NDP focusses on *bandwidth* optimizations. NoFTL-KV [28] addresses the problem of *write-amplification*. The NDP extensions demonstrated by nKV target the *read-amplification*, *latency improvements* and *computational storage*.

12.8 CONCLUSION

In this paper we introduced nKV – a key-value store designed for native computational storage and near-data processing. nKV controls physical data placement directly and hence the on-device I/O parallelism. Along the same lines, nKV can place NDP operations on different compute elements on device (ARM or novel FPGA PEs) and also configure the hardware per operation accordingly, e.g. the number of hardware parsers used. Both placement methods impact the performance of NDP operations, GET is faster on the ARM, while SCANs are faster with hardware support. nKV is based on the principle of explicit cross-layer data formats, hence hardware or software layout accessors and format parsers are deployed an can be used for different operations.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS 1998*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. "Chisel: Constructing hardware in a Scala embedded language." In: *Proc. DAC 2012*. 2012.

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

[5] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: ed. by A. R. Hurson, L. L. Miller, and S. H. Pakzad. 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[6] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality." In: *Journal of Mathematical Sociology* (2001).

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

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

[9] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD 2013*. 2013.

[10] Facebook. *RocksDB*. https://github.com/facebook/rocksdb. 2020.

[11] Jim Gray and Prashant J. Shenoy. "Rules of Thumb in Data Engineering." In: *Proc. ICDE 2000*. 2000.

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

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

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

[15] Insoon Jo and et al. et. "YourSQL : A High-Performance Database System Leveraging In-Storage Computing." In: *Proc. VLDB 2016*. 2016.

[16] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST 2013*. May 2013.

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

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

[19] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. "In-storage Processing of Database Scans and Joins." In: *Inf. Sci.* (2016).

[20] Chen Luo and Michael J. Carey. "LSM-based storage techniques: a survey." In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

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

[22] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Inform.* (1996).

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

[24] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. "Active disks for large-scale data processing." In: *Computer (Long. Beach. Calif).* 34.6 (2001), pp. 68–74.

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

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

[27] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. "An Overview of Microsoft Academic Service (MAS) and Applications." In: *Proc. WWW 2015*. 2015.

[28] 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*. 2018.

[29] Tobias Vincon, Arthur Bernhardt, Lukas Weber, Andreas Koch, and Ilia Petrov. "On the Necessity of Explicit Cross-Layer Data Formats in Near-Data Processing Systems." In: *Proc. HardBD @ ICDE 2020*. 2020.

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

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

13

NKV IN ACTION: ACCELERATING KV-STORES ON NATIVE COMPUTATIONAL STORAGE WITH NEAR-DATA PROCESSING

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing"* by Tobias Vinçon, Lukas Weber, Arthur Bernhardt, Andreas Koch and Ilia Petrov in 2020 46th International Conference on Very Large Data Bases (VLDB). The contribution of the author of this thesis is summarized as follows.

» As the corresponding and co-leading author, Tobias Vinçon was responsible for designing and implementing the nKV system including the NDP invocation of software-based parsers and accessors for the GET, SCAN and BC operations. Lukas Weber complemented the NDP functionality with hardware-based processing elements and integrated those into the COSMOS+ architecture. Tobias Vinçon was in charge of designing the paper graph demonstration walk-through and provided an interactive grafical user interface to showcase the performance improvements. With feedback from the other co-authors Lukas Weber, Arthur Bernhardt, Andreas Koch and Ilia Petrov, Tobias Vinçon concluded the text. «

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) designs represent a feasible solution, which although not new, has yet to see widespread use.

In this paper we demonstrate various NDP alternatives in nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. We showcase the execution of classical operations (*GET*, *SCAN*) and complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance due to NDP. nKV runs on real hardware - the *COSMOS+* platform.

13.1 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 bandwidth in combination with poor data locality, but also result from traditional (*data-to-code*) system architectures.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution, i.e. as close as possible to the physical data location, utilizing the much better on-device I/O performance. NDP leverages several trends. Firstly, hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device – so called NDP-capable *computational storage*. As a result, even commodity storage devices nowadays have compute resources that can be effectively used for NDP, but are executing compatibility firmware (to traditional storage) instead. Secondly, with semiconductor storage technologies (NVM/Flash) the *device-internal* bandwidth, parallelism, and latencies are significantly better than the external ones (device-to-host). Both lift major limitations of prior approaches like ActiveDisks or Database Machines.

In this paper, we demonstrate nKV, which is a RocksDB-based key/-value store utilizing *native computational storage* and *near-data processing* (Figure 16.1). nKV eliminates intermediary layers along the I/O stack (e.g. file system) and operates directly on NVM/Flash storage. nKV directly controls the physical data placement on chips and channels, which is critical for utilizing the on-device I/O properties and compute parallelism. Furthermore, nKV can execute access operations

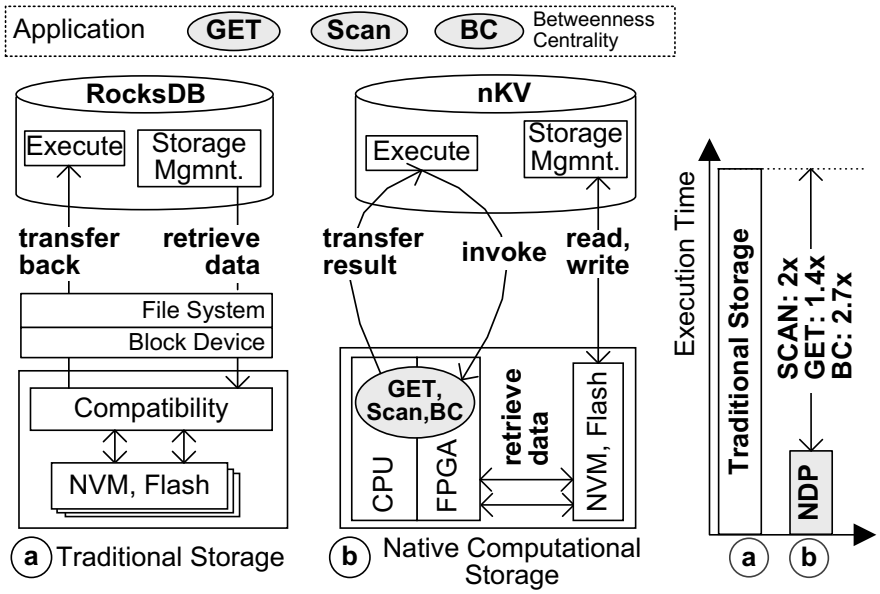


Figure 13.1: KV-Store transferring data along a traditional I/O stack (a); and (b) nKV executing operations in-situ on native computational storage.

like *GET* or *SCAN*, or more complex graph processing algorithms such as *Betweenness Centrality* as *software NDP* on the ARM cores or with FPGA hardware support (NDP:HW+SW). Under nKV we target *intervention-free* NDP-execution, i.e. the NDP-device has the complete address information, can interpret the *data format* and access the data in-situ (without any *host interaction*). To reduce data transfers nKV also employs novel *ResultSet*-transfer modes. nKV is resource efficient as it eliminates compatibility layers and utilizes freed compute resources for NDP.

We demonstrate nKV for the use-case of a database of research papers, and on a *2.4GB* graph dataset with *48 million* KV-pairs. Our demonstration scenarios involve interacting with the paper DB, browsing and analyzing it: (a) *analysis scenario (BC)*: verifies if the 10-year best paper award was awarded the most prominent paper from 10 years ago and offers some unexpected insights; (b) *Latency-based (GET)*: we let the audience pick a paper from the BC ResultSet and display its details; (c) *Bandwidth-based (SCAN)*: we retrieve other papers from same Venue/Author/Year. nKV performs 1.4×-2× better than RocksDB: *GET latency* – 1.4×; *SCAN bandwidth* – 2×; *Betweenness Centrality* – 2.7×.

13.2 ARCHITECTURE OF NKV

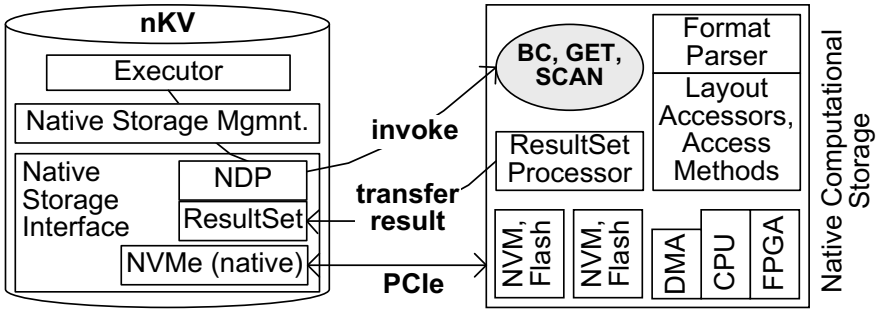


Figure 13.2: Architecture of nKV.

Native computational storage. To make efficient use of the on-device I/O and compute resources nKV extends [14] and employs *native storage* (Figures 16.1 and 13.2). This way it eliminates intermediary layers along the critical I/O path like the file system, and can operate directly on NVM/Flash storage using physical addresses. nKV can therefore precisely control physical placement of SST data, which is critical for utilizing the on-device I/O properties and compute parallelism. nKV physically places *SST data blocks* and *SST index blocks* on different *LUNs* and *Channels* to utilize the on-device parallelism and lower the processing latency (see Figure 13.3). *Native storage* is essential for reducing read- and write-amplification, and also for executing NDP-operations avoiding information hiding through these

layers of abstraction. Furthermore, individual levels of the LSM-Tree can be physically separated on different chips and LUNs to improve I/O throughput and parallelism since I/O-heavy compaction jobs do not block the entire device and relieve the bus. In addition, native storage allows the realization of lean NDP-functionality.

NDP Interface Extensions. nKV defines NDP-Extensions to the native storage interface. Furthermore, nKV has a dedicated high-performance *in-DBMS* NVMe layer (Figure 13.2). It does not rely on an NVMe kernel driver, but is deeply integrated in the DBMS and hence runs in *user-space*. The *native NVMe* integration reduces the I/O overhead, as it avoids expensive switches between user and kernel space (drivers), and shortens the I/O even further, as no drivers are needed.

Computation Placement. By using native computational storage, nKV can place computations directly on the heterogeneous on-device compute elements, such as ARM CPUs or the FPGA. nKV can execute various operations such as *GET* or *SCAN*, or more complex graph processing algorithms like *Betweenness Centrality* as *software NDP* on the ARM cores, or with *hardware* support from the FPGA. nKV demonstrates that hardware implementations alone cannot reach the best performance as pure software implementations do not. For its NDP-operations nKV utilizes *hardware/software co-design* to handle the proper separation of concerns and achieve best performance.

ResultSet Handling. nKV aims to bulk-transfer the *ResultSet* of an NDP-Operation to avoid the data transfer overhead caused by a *record-at-a-time* model. Thus nKV materializes the ResultSet, partially or fully, depending on the NDP operation. It is then DMA-transferred with multiples of the COSMOS+'s DMA-engine transfer unit (4KB).

In-situ data access and interpretation. Under nKV the NDP-device can interpret the *data format* and *access* the data without *host intervention*. To this end, nKV extracts definitions of the *Key- and Value-formats*. These are then passed as input parameters to NDP-commands. Moreover, the *data format* such as the *Key- and Value-formats* can be automatically extracted from the DB-catalogue (*system-defined*), or can be defined by the *application*.

nKV employs a thin on-device *NDP-infrastructure* layer that supports the execution and simplifies the development of NDP-operations (Figures 13.2). It comprises *data format parsers* and *accessors* that are implemented in both *software* and *hardware* (Figure 13.3). The in-situ *accessors* are used used to traverse and extract the contained sub-entities of the persistent data. Whereas, the in-situ *data format parsers* process the *layout* of each persistent entity, and extract the sub-entities by invoking further accessors (Figure 13.3).

KV-Stores like LevelDB or RocksDB organize the persistent LSM-Tree data in to so called *Sorted String Tables (SST)*. To process a *GET(key)* request, for instance, nKV first identifies the respective *SST* and invokes an *NDP_GET()* command with the physical address ranges (of

these SSTs), the respective *Key- and Value-formats* as well as further parameters. First, the *SST layout accessor* is invoked to access data and index blocks. Subsequently, the index block parser is invoked to interpret the data, verify if the *key* is present, and extract its location. If this is the case, the data block accessor and parser are invoked to extract the *Key/Value entry*. In case of an *NDP_SCAN(key_val_condition)* operation, the KV accessor is subsequently invoked to extract it, followed by a *field* parser to extract its value and verify the *condition*.

13.3 DEMONSTRATION WALK-THROUGH

Demo Setup. The demonstration setup comprises a desktop PC as host equipped 3.4 GHz Intel I5 CPU, 4 GB RAM, connected to *COSMOS+* via NVMe over PCIe (Figure 13.4). The COSMOS+ [2] has a Zynq 7045 SoC with an FPGA, two 667 MHz ARM A9 CPU Cores and an MLC Flash module configured as SLC. We configure both RocksDB and COSMOS+ with 5 MB cache.

We demonstrate nKV on the use-case of a database of research papers, and on a *2.4GB* graph dataset including *48 million* Key/Value-pairs. These comprise approx. *3.8M* papers, *40M* references, *18K* venues, and *4.2M* authors. *BC* operates on a graph with varying number of relevant edges: from *2.5K* to *2 million*. The audience will browse and analyze the paper set using a GUI (Figures 13.5), triggering different operations on the paper graph in different scenarios.

13.3.1 *Walk-Through*

1. Complex Graph Analysis – *BC*. The demo starts by letting the audience pick a *DB conference venue* and an *year* (e.g., *VLDB, 2000*). Subsequently, nKV executes *Betweenness Centrality* to determine the most prominent paper from that year. The audience can then verify if that paper had indeed been awarded the *10-year best paper award* ten years later. Expect some unexpected(!) insights.

Under the hood, nKV executes a complex NDP operation pipeline, comprising a SCAN and BC. Based on the audience selection, nKV first filters out the relevant papers and references by running a *SCAN* and applying *val_condition* on the *values* of all paper KV-pairs. This is only possible since the data formats are available in-situ, and the *format parses* and *layout accessors* execute on-device. The intermediary result is materialized on-device, which is essential for such NDP-pipelines. Subsequently, BC is executed on the intermediary results. nKV switch between software NDP or software/hardware NDP. We demonstrate how the hardware accessors and parsers can be instantiated multiple times, and run in parallel on the FPGA yielding best results.

Observation: nKV executes NDP-pipelines and complex operations in-situ. Given the high parallelism and compute intensity, NDP:SW+HW yields best results.

2. Latency – *GET*. After the *BC* analysis, the audience can interactively pick a paper from the BC ResultSet and display its details.

Under the hood, the NDP execution of GET is performed in SW and in NDP:SW+HW. Since only a single NDP_GET() is executed at a time, nKV utilizes native data placement, but not the on-device parallelism.

Observation: Latency-critical operations are 1.4× faster and best results are achieved with NDP:SW, closely followed by NDP:SW+HW (Figure 13.7).

3. Bandwidth – *SCAN*. After the audience has been presented the details of a paper (previous scenario), they can opt for retrieving other papers from the same Venue/Author/Year.

Under the hood, this results in an NDP *SCAN(value_condi- tion)*. The operation is performed with different selectivities and different result set sizes, based on the audience selection (Figure 13.8a). Importantly, the selection condition is on the value, which requires NDP format parsers and layout accessors to be evaluated in-situ. Conversely, the *Blk* RocksDB stack transfers the entire data to the host, to interpret the values there, apply the *val_condition*, and eventually discard most of the data. Figure 13.8b shows the extra read volume transferred by the *Blk* to perform the same SCAN.

Observation: Bandwidth-critical scan and selection operations require I/O bandwidth and high hardware parallelism. Hence, NDP:SW+HW is best and outperforms the traditional stack by 2×.

4. Parallelism and Native Computational Storage Last but not least we execute BC again, however this time we demonstrate the effect of *configurable parallelism* in native computational storage, whenever nKV executes a complex operation (Figure 13.6b).

nKV can configure the degree of parallelism required by each NDP-operation. While the amount of compute parallelism is limited for NDP:SW, as there are few ARM cores, the same does not apply to the FPGA. As described in Section 13.2, there can be *multiple parallel instances* of the hardware accessors and parsers on the FPGA. These are relatively space-efficient, as 16 instances fit even into the small Zynq 7045 FPGA. Interestingly, operating with the maximum available parallelism does not always yield the the best results (Figure 13.6c).

Observation: nKV can employ the FPGA for NDP:SW+HW, increasing the level of computational storage parallelism. However, this capability only translates into performance benefits for *complex* operations.

13.4 RELATED WORK

The Near-Data Processing approach is deeply rooted in well-known techniques such as *database machines* or Active Disk/IDISK. With the advent of Flash technologies and reconfigurable processing elements Smart SSDs [4, 8, 13] were proposed. An FPGA-based intelligent storage engine for databases is introduced with IBEX [15]. JAFAR [1, 16] is one of the first systems to target NDP for DBMS (column-store) use, whereas [7, 10] target joins besides scans. Recently, Samsung announced its KV-SSD [11]. The use of NDP in the realm of KV-Stores has been investigated in [3, 9]. Kanzi [5], Caribou [6] and BlueDBM [12] are RDMA-based distributed KV-Stores investigating node-local operations.

Much of the prior work on persistent KV-Stores and NDP focuses on *bandwidth* optimizations. NoFTL-KV [14] addresses the problem of *write-amplification*. The NDP extensions demonstrated by nKV target the *read-amplification*, *latency improvements* and *computational storage*.

13.5 CONCLUSION

We demonstrate nKV, which is a key/value store utilizing *native computational storage* and *near-data processing*. We showcase the execution of classical operations (*GET*, *SCAN*) and complex graph-processing algorithms (*Betweenness Centrality*) in-situ, with 1.4×-2.7× better performance due to NDP. nKV runs on real hardware - the *COSMOS+* platform. nKV utilizes the the available I/O and compute parallelism on the native computational storage through direct data and operation placement. Complex operations (BC, SCAN) benefit from it, whereas others (GET) benefit from software NDP.

REFERENCES

[1] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

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

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

[4] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD*. 2013.

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

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

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

[8] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST*. 2013.

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

[10] Sungchan Kim, Sang-Won Lee, Bongki Moon, and et al. "In-storage Processing of Database Scans and Joins." In: *Inf. Sci.* (2016).

[11] *KV-SSD*. https://github.com/OpenMPDK/KVSSD. Samsung.

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

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

[14] 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.

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

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

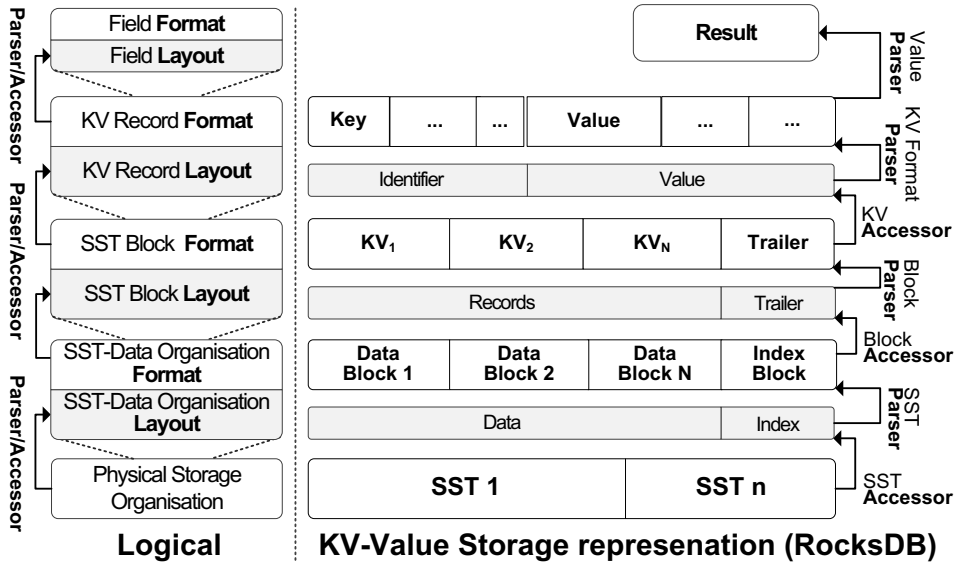


Figure 13.3: In-situ access and data interpretation in nKV, based on layout accessors and format parsers.

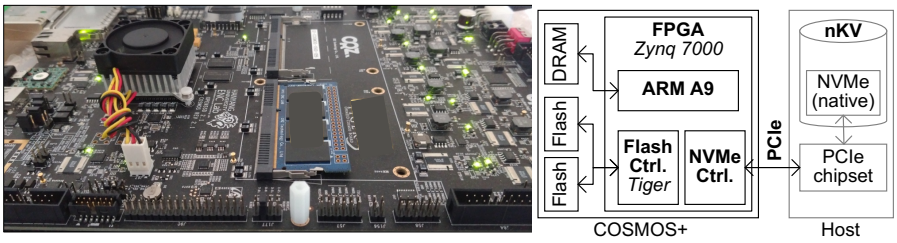


Figure 13.4: COSMOS+ and the Demonstration Setup.

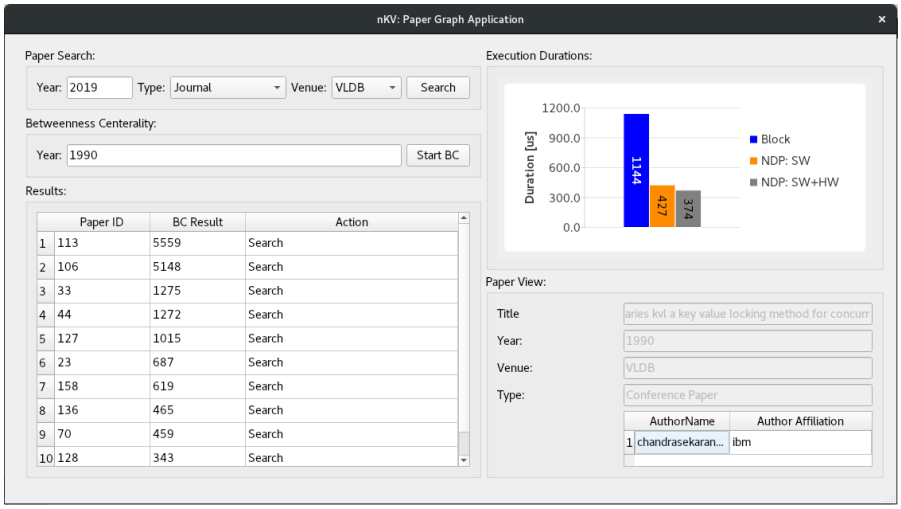


Figure 13.5: Interactive GUI.

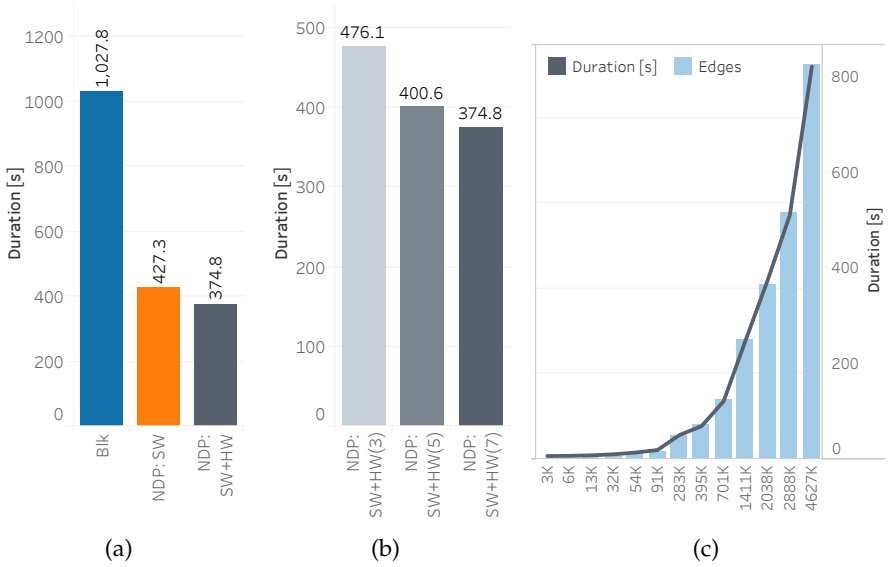


Figure 13.6: Betweenness Centrality: (a) BC on different stacks; (b) BC with different levels of parallelism; (c) BC execution time vs number of relevant edges (complexity).

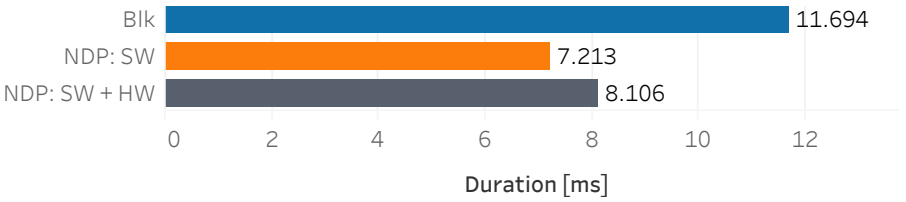


Figure 13.7: GET Latencies on different stacks.

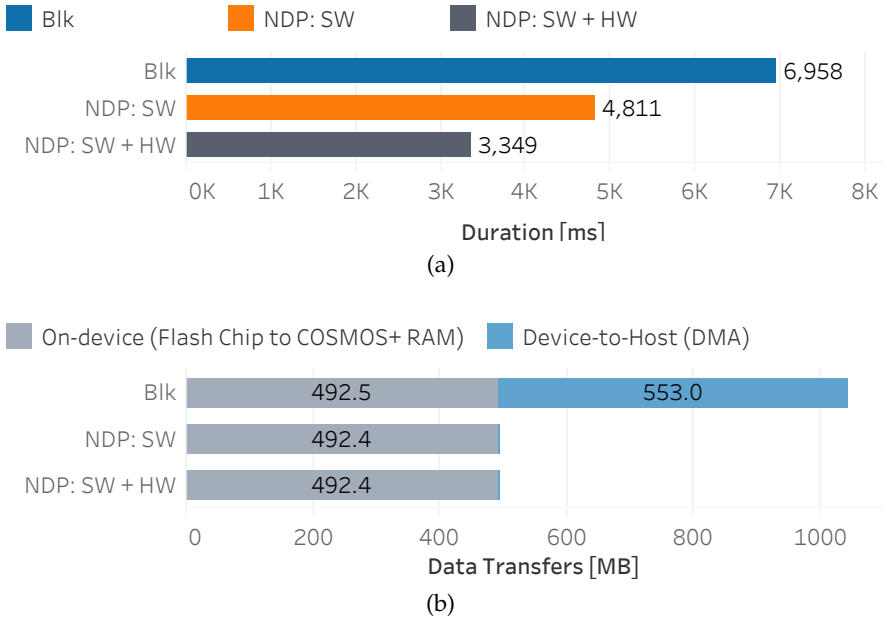


Figure 13.8: SCAN performance:(13.8a) SCAN on different stacks; (13.8b) Data Transfer Volume

14

A FRAMEWORK FOR THE AUTOMATIC GENERATION OF FPGA-BASED NEAR-DATA PROCESSING ACCELERATORS IN SMART STORAGE SYSTEMS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"A Framework for the Automatic Generation of FPGA-based Near-Data Processing Accelerators in Smart Storage Systems"* by Tobias Vinçon, Lukas Weber, Lukas Sommer, Leonardo Solis-Vasquez, Christian Knödler, Arthur Bernhardt, Andreas Koch and Ilia Petrov in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding author, Tobias Vinçon provided with nKV, the NDP invocation, and software-based orchestration of processing elements, essentail parts for the exerimental evaluation. Lukas Weber and Lukas Sommer extented those with a framework for automatic generation of FPGA-based processing elements and evaluated this against the static baseline. The manuscript's text was joint work of Tobias Vinçon, Lukas Weber and Lukas Sommer with feedback from Leonardo Solis-Vasquez, Christian Knödler, Arthur Bernhardt, Andreas Koch and Ilia Petrov. «*

ABSTRACT

Near-Data Processing is a promising approach to overcome the limitations of slow I/O interfaces in the quest to analyze the ever-growing amount of data stored in database systems. Next to CPUs, FPGAs will play an important role for the realization of functional units operating close to data stored in non-volatile memories such as Flash.

It is essential that the NDP-device understands formats and layouts of the persistent data, to perform operations in-situ. To this end, carefully optimized format parsers and layout accessors are needed. However, designing such FPGA-based Near-Data Processing accelerators requires significant effort and expertise. To make FPGA-based Near-Data Processing accessible to non-FPGA experts, we will present a framework for the automatic generation of FPGA-based accelerators capable of data filtering and transformation for key-value stores based on simple data-format specifications.

The evaluation shows that our framework is able to generate accelerators that are almost identical in performance compared to the

manually optimized designs of prior work, while requiring little to no FPGA-specific knowledge and additionally providing improved flexibility and more powerful functionality.

14.1 INTRODUCTION

The rate at which new data is produced and stored every day has constantly been increasing in recent years, and with the advent of the internet-of-things (IoT), this trend will continue in the foreseeable future. A substantial amount of the data produced every day is stored in database systems, such as key-value stores (KV-store). Of course, this data is not write-only: To make sense (and gain value) out of the stored data, it needs to be analyzed, ever more so now in the golden age of Big Data and Machine Learning.

Data analytics has been limited by slow I/O interfaces to the attached storage devices such as non-volatile memory (NVM). This severely hampered the processing of stored data. An interesting approach to overcome this limitation is *Near-Data Processing* (NDP): Instead of moving huge amounts of data from storage via the I/O-bottleneck to the CPU for analysis, which will eventually yield a result typically much smaller in size than the input data, Near-Data Processing places the computation much closer to the data. With hardware vendors being able to economically *integrate* processing units with non-volatile memories on a single chip or board, Near-Data Processing can help to overcome the limitations on data analytics imposed by slow I/O interfaces.

One example for a Near-Data Processing system for key-value stores was presented by Vinçon et al. in [18, 19]: By combining what they refer to as *Native Computational Storage*, which removes unnecessary abstraction layers and unifies information about data format and layout in a single layer with NDP capabilities, they were able to demonstrate speedups of up-to factor 2.7x for real-world data analysis. For their approach, they did not only use standard CPUs, but also leveraged the computational power and parallelism of FPGAs. However, the FPGA-based NDP processing elements (PEs) in their work were hand-crafted, requiring significant development effort and expertise.

In addition, not only do data storage formats *evolve* over time, but the specific data representation requirements in the actual NDP-operations also tend to change over time. Hand-crafting highly optimized NDP-accelerators becomes impractical in such scenarios, which may include data analytics on big data sets, or evolving feature vectors in machine learning.

In this work, based on the nKV architecture [18], we present a framework to *automatically* generate FPGA-based NDP accelerators from data format specifications. The generated PEs are able to filter and transform data from key-value stores, based on user-specified

filter predicates and transformation rules. The PEs are integrated in a system-on-chip (SoC) architecture for the Cosmos+ OpenSSD platform [17].

In the evaluation, we compare the performance of the automatically generated accelerators with hand-crafted designs and assess the impact of the data format on the hardware footprint of the generated accelerators.

14.2 MOTIVATION

In general, the development of hardware accelerators for specific applications is a tedious task that requires knowledge of the application domain, as well as expertise in accelerator development and device specifics. Typically, using the database specification, a corresponding hardware accelerator will be implemented using some form of Hardware Description Language (HDL) such as Verilog or VHDL. As soon as the accelerator design is finished, a suitable software interface has to be implemented. Depending on the target platform, this interface may vary. For the OpenSSD Cosmos+, the HW/SW interface has to be developed as device firmware, which is executed on the ARM-cores of the device. Since the architecture and the accelerator design impact how the accelerator is controlled, it is necessary to consider both when developing the software interface. As soon as the software interface is implemented, all of the components can be integrated. In this stage, the firmware is adapted to use the software interface to access the accelerator. Lastly, the hardware design (including the accelerator) has to be synthesized into a bitstream, which is used to program the FPGA-portion of the Zynq-7000 SoC on the Cosmos+. After compilation and synthesis has finished, the accelerated system can be deployed and used.

A major problem of this toolflow is the required cross-domain knowledge. Especially the PE development requires experience with hardware development, as well as a good knowledge of the target platform. Additionally, HW-SW dependencies exist, which makes it impractical to develop the software interface without a finalized accelerator design.

In this work, we aim to implement a framework which allows the *automatic generation* of the accelerator design by composing fixed architecture templates. These templates allow for the concurrent generation of the software interface. The merit of this approach is twofold: First, hardware development expertise is no longer required. The proposed framework is usable without any knowledge about hardware development or HDLs. Instead, the required information is provided to the tools in a simple C-style syntax. Additionally, the dependency between the accelerator design and the interface development is removed, allowing an overall faster development cycle.

14.3 NEAR-DATA PROCESSING BACKGROUND

14.3.1 *Background: Key-Value Stores*

In this work, we focus on Near-Data Processing for wide-spread, high-performance Key-Value (KV) Stores, in particular RocksDB [8]. In order to provide querying capabilities in combination with high sustained insert and update rates, modern KV-Stores often use out-of-place update approaches such as Log-Structured Merge-Trees (LSM-Tree) [14].

An LSM-tree employs multiple components $C_0 ... C_k$. All insertions and updates are performed on the first component C_0, typically located in memory. After C_0 reaches a defined size threshold, its content, i.e., the insertions and updates, is flushed to persistent memory and merged with component C_1. Over time, the merges will gradually move data from C_0 to C_k to ensure a separation of hot and cold data. During each merge process, outdated key-value pairs are purged and their space is reclaimed.

RocksDB uses LSM trees in a *multi-leveled* variant [13]. The component C_0 comprises multiple MemTables and is located in *volatile* memory, while the remaining components $C_1 ... C_k$ reside in *persistent* memory (e.g., Flash). Whereas the MemTables in C_0 are typically implemented using a memory-efficient structure such as skip-lists, the data is transformed into the so-called *Sorted String Tables* (SST) format during the flush from C_0 to the persistent component C_1. Each component $C_1 ... C_k$ in persistent memory comprises multiple SSTs. Each SST in turn is composed by an *index block* and a number of *data blocks*. The key-value pairs are stored in the data-blocks in key-sorted order.

During the merge process, as part of the LSM tree algorithm, the SSTs are *compacted*, i.e., outdated entries are pruned. Nevertheless, as the compaction process only happens as part of the merge process, multiple key-value pairs for the same key can be present on different levels of the LSM tree hierarchy. For example, a more recent key-value pair k, v' in component C_2 supersedes a pair k, v in component C_5. For performance, no compaction takes place during the flush from component C_0 to component C_1.

Access operations to the key-value store, such as GET or SCAN require to traverse multiple index blocks, starting at the MemTables in component C_0. Assuming that the key is not present there, *all* index blocks of every SST from C_1 need to be traversed (remember that no compaction is performed during the flush, thus multiple pairs for a key can be present in C_1), followed by traversing a single index block in the remaining index components $C_2 ... C_k$. SCAN operations with a value predicate (e.g., $\mathrm{SCAN}(0 < Value < 42)$) even require traversal of the *entire* data-set.

The NDP PEs generated by our tool-flow operate on SST files using parallelized NDP operations for faster access.

14.3.2 *nKV: Near-Data Processing Architecture*

The NDP PEs developed in this work is based on the nKV Near-Data Processing architecture developed by Vinçon et al. [18]. Their architecture and custom key-value store exploits two key insights: First, while intermediate layers and abstraction such as block devices and file systems simplify the architecture and implementation of key-value stores such as RocksDB, they also introduce inefficiencies and complicate the implementation of true near-data processing. For NDP to be effective, it needs to operate directly on the physical addresses of the data in the key-value store. Therefore, nKV uses native computational storage, i.e., the intermediate abstraction layers along the critical I/O path have been removed and nKV directly operates on Flash storage, using physical addresses.

Having control over the physical placement of data also allows nKV to optimize the *placement* of data. By distributing data on independent Flash channels and LUNs, nKV facilitates parallel access and processing of data [18]. Moreover, keeping the data of different LSM-tree index components separated on different Flash chips, avoids blocking of the entire bus by compaction jobs taking place as part of the LSM-tree merge.

The second important insight that underlies nKV is the fact that placing the computation closer to the data can significantly reduce the amount of data transferred, and consequently speed-up access. Many KV-store operations, such as the SCAN-operation on value predicates explained in the previous section, are very I/O-intensive, requiring much more data to be moved from storage to the processor than what is required for the final result of the operation. Using Near-Data Processing, i.e., placing the computation much closer to the data, does not only reduce the I/O complexity of the operations, but also allows for higher degrees of parallelism, as the device-internal bandwidth of storage devices (e.g., parallel access to different Flash channels) is typically much higher than the bandwidth of the I/O interface to the processor. In a similar fashion, NDP also achieves much shorter latencies.

In general, KV-Stores employ concrete data formats defined by either the application on by the database object itself (e.g. table), when applied as a DBMS storage engine, the data catalog. The nKV architecture exploits on-device data access and allows for data format interpretation in-situ. While information about the layout and format of data is scattered and encapsulated across multiple abstraction layers in classical KV-stores, nKV removes these layers and introduce on-device infrastructure for data format parsers and accessors in both

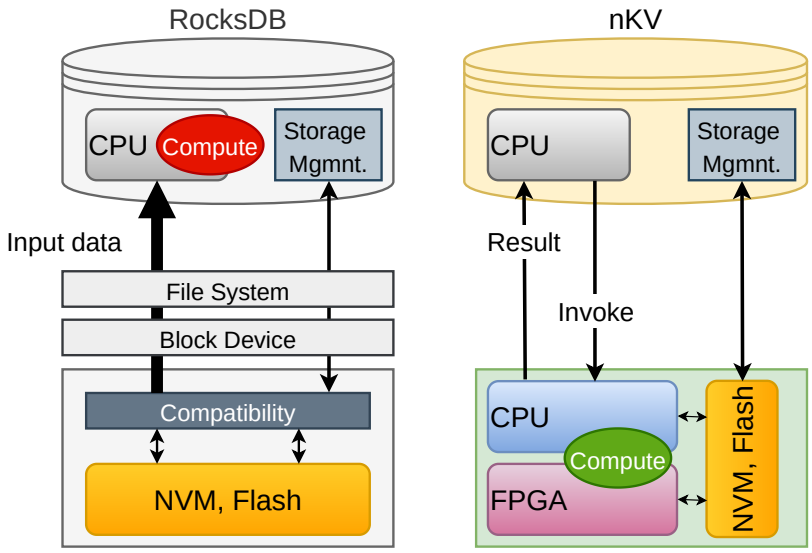


Figure 14.1: Comparison of traditional KV-store and the nKV-architecture with native computational storage and Near-Data Processing.

soft- and hardware. The infrastructure operates on the SST format and allows interpretation of the data format and data access *without host intervention*.

The difference between the nKV architecture, with its native computational storage and use of NDP, and traditional KV-store setups, such as RocksDB, can be seen in 14.1: While RocksDB has to retrieve large amounts of data from the storage through intermediate layers to perform the requested operation on the host CPU, the nKV architecture can leverage the full device-internal bandwidth of the Flash and perform the requested operation on-device, eventually transferring only the much smaller result set back to the host.

While the prior FPGA-based NDP PEs for nKV were designed manually, this work will target the existing nKV architecture, and provide an *automated* tool-flow for generating FPGA-based hardware accelerators for NDP operations.

14.4 NEAR-DATA PROCESSING ACCELERATOR GENERATION

Our implementation targets the Cosmos+ OpenSSD platform [17], which features a Xilinx Zynq-7000 SoC (XC7Z045). Additionally, the Cosmos+ offers two kinds of memory: Fast but volatile DRAM, and slow but persistent Flash memory.

The Cosmos+ baseline architecture enables it to be used as a "plain" NVMe SSD. To this end, the programmable logic (PL) of the Zynq SoC is used to implement an NVMe interface as well as controllers for the the attached Flash memory. Specifically, the Tiger4 Flash memory controller is used [17]. This baseline architecture is extended with FPGA-based NDP processing elements (PEs) in [18], which supports

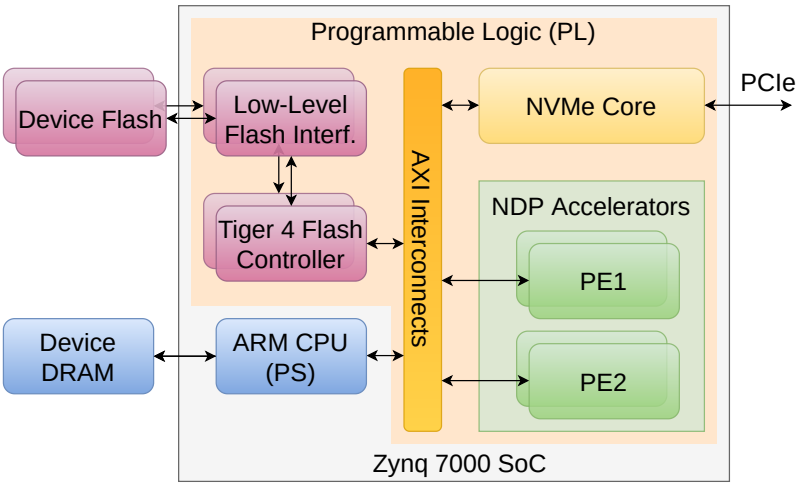


Figure 14.2: Overall system architecture based on the Cosmos+ OpenSSD platform, extend with FPGA-based NDP accelerators.

hardware/software co-execution for NDP in conjunction with the Zynq ARM cores. While [18] uses manually developed PEs, in this work we will introduce a way to automatically generate them from abstract specifications .

When adding FPGA-based PEs, a balance between Flash parallelism and compute parallelism has to be struck, since both the Flash memory controllers and the PEs compete for FPGA resources on the reconfigurable portion of the Zynq-7000. In this work, we use a single Flash DIMM and two separate Flash controllers for the Flash memory. The resulting system architecture is shown in 14.2.

To reduce the implementation complexity, the PEs are not directly coupled to the Flash memory. Instead, the data is first buffered in DRAM, and the results are also initially collected in DRAM. While this might seem counter-intuitive, this detour does not have significant negative performance impact due to two issues: First, the overall Flash bandwidth achievable using two Tiger4 controllers is only about 200 MB/s. Second, most of the data will be accessed multiple times, and thus profits from being stored in faster DRAM (compared to the relatively slow Flash memory).

14.4.1 *NDP Accelerator Architecture Template*

While the concrete functionality of the accelerators is automatically generated to match the specified filtering and data transformations, all accelerators use the same *architectural template* as a basis. This template, which is also depicted in 14.3, comprises four main components. The first component, the control component (14.3.a) is simply a register file, which is mapped into the memory space of the on-chip ARM core. The registers can then be used for communication between CPU and PE.

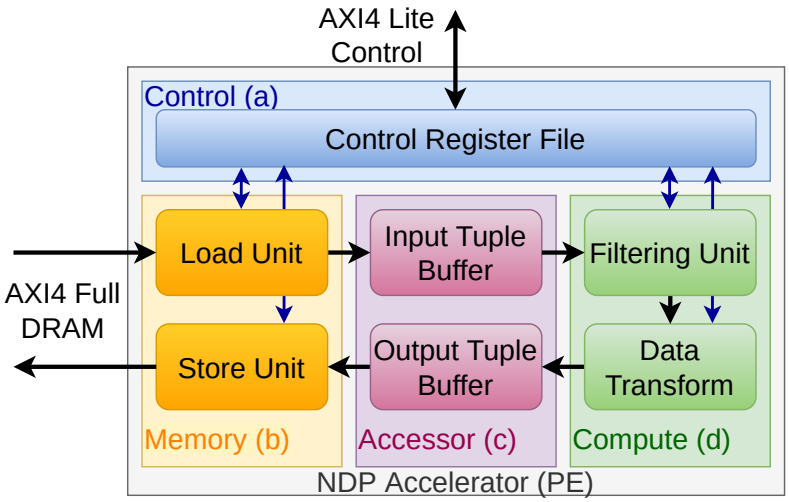


Figure 14.3: Architectural template used by the generated NDP accelerators.

The second component, marked (b), of the template is concerned with loading and storing data from/to memory. As described in the previous section, the PEs do not have direct access to the Flash memory. Instead, the input data is loaded from the DRAM via the corresponding AXI4 interface provided by the Zynq PS. The loading and processing of data takes place at a granularity of 32KB blocks.

The two tuple buffers in the accessor component, marked (c), are responsible for converting between the native bit-size of the memory interface (64 bit on Zynq-7000), and the actual size of a tuple in the KV-store (i.e., a key-value pair).

The computation component, marked (d) in 14.3, consists of two main functional units: The filtering unit will discard any tuple that does not match a user-specified predicate. Predicates can evaluate elements of the key, as well as the value and, in contrast to prior work **??**, can also be defined across *multiple* columns. This is achieved by the option of chaining multiple filtering units, each evaluating a single predicate. The number of filtering stages is configurable, and the framework will automatically generate the required logic.

The second functional unit is the data transform unit, which transforms the tuples that passed the filter, as defined by the user. Example for transforms include discarding RocksDB meta-data, or unnecessary columns. Both units, the filtering unit as well as the data transformation unit, are generated automatically, as described in the next section.

14.4.2 *Automatic Generation of NDP Accelerators*

In general, the underlying abstraction of most contemporary databases is *structured* application data. An example for this structuring are relational databases, that impose a *database scheme* on all of the stored

```
/* @autogen define parser Point3DTo2D with
chunksize = 32, input = Point3D, output = Point2D,
mapping = {output.x = input.y, output.y = input.z }
*/
typedef struct { uint32_t x, y, z; } Point3D;
typedef struct { uint32_t x, y;    } Point2D;
```

Figure 14.4: Example Code showing how a PE is defined for automatic generation. The generated PE will automatically transform data from the `Point3D`-type to `Point2D`-type, discarding the field `x`. Additionally, the `Point3D`-structs can be filtered using predicates on all of the present fields (`x`, `y` and `z`).

data. As an alternative to relational databases, key-value stores employ a less structured way of storing data. While key-value stores typically do not enforce a structure, most applications still use structured data. Thus, the application might use string-based key-value stores to store the binary data, maintaining the application-level structuring of the data outside the KV-store. The application would then use an internal record-based datatype (e.g. `structs`), and transform this data into a corresponding key-value pair. The resulting key-value pair obviously has the same structure as the underlying `struct`.

For our automatic generation, we have to assume that the data is structured, as we would not be able to interpret the value data for filtering or other processing otherwise. Typically, an application will use data-classes or structs to represent this structure. By interpreting these type-definitions, our tools can generate the matching hardware NDP units for the specified data structures. In our framework, we rely on C-inspired type-definitions, as well as annotations for the specification of the PEs. This allows the database engineer to reuse his application code for the generation of PEs. An example for the specification of a PE is given in 14.4.

From the parsed type-definitions and annotations, an internal representation of these types is built. This internal representation is limited to data-types that are suitable for hardware-processing. Specifically, integers and single/double precision floating point types are supported. In addition to these primitive types, it is also possible to work with (nested) arrays and (nested) structs. For byte-arrays, it is also possible to flag them as string-data using a `prefix` annotation. If the annotation is given, the corresponding byte-array will be split into a `prefix` that is handled as a regular field, while the rest of the byte-array is not used for predicate-evaluation. The reason for this lies in the potential sizes of strings, which makes them very hard to process in hardware.

For example, the output of the Tuple Input Buffer is just a sequence of bits containing the complete data of the corresponding struct. With the information gathered by the contextual analysis, these bits can

be interpreted. For example, consider a `struct Point` which encodes the coordinates x, y and z (all 32 bit integers) of a point in three-dimensional space. The hardware now knows, that the first 32 bits encode x, while the second 32 bits encode y, etc. Using this information, it is now possible to filter points that lie behind a certain threshold (filtering), or project the 3D-data into a two-dimensional space (data transformation).

Contextual Analysis As described previously, the contextual analysis phase of our tools is responsible for computing the data-layouts from the parsed representations of the type-definitions. To simplify this process, the contextual analysis performs multiple transformations on the struct data-type. The input to the contextual analysis are trees representing the struct-types. Each node describes a different part of the overall structs, with leaf nodes representing actual primitive types (e.g., integers), while regular nodes can be nested structs or arrays. In the first step, arrays that are annotated to represent strings are transformed into structs, which contain a `prefix`-field followed by an array, which contains the rest of the string (`postfix`). After strings are resolved in this manner, the next step removes arrays completely from the tree, by flattening them into structs with a corresponding sequence of scalar element fields. In essence, an array `uint_32t [2]` becomes the struct `struct {uint_32t elem_0, elem_1;}`. Since the data layout is identical for both, this scalarization simplifies the following steps. In a final step, the contextual analysis determines the largest *relevant* field. Relevant fields are those that can be used for filtering predicates. In our case, this includes all primitive fields *except* string-postfixes. Using the size of the largest field, the contextual analysis then determines, whether other fields have to be padded. The padding ensures that all relevant fields can be processed in a single comparator unit.

Memory Interface The memory interface contains a Load- and a Store-Unit, both having access to the PS-DRAM via a shared AXI4 Full interface. In contrast to [18], we opted for more flexible units. Vinçon et al. rely on fully static units that always load and store *complete* data blocks (32 KByte). While this keeps the hardware footprint minimal, it is not very efficient with regard to the use of memory bandwidth. Due to the Data Transformation step, which often removes elements such as metadata from the tuples, the output is almost always smaller than 32 KByte. As memory contention is a major bottleneck, reducing the number of memory accesses will improve the performance. In our work, the Load- and Store-Unit can be configured (using the Control Register File) to store variable amounts of data, thereby reducing unnecessary memory accesses and memory contention.

Tuple Buffers The Tuple Buffers transform the unstructured data retrieved from memory into processable structured data, and back again for storage. To do this, a buffer is used to group the incoming stream of 64 bit words, until one or more complete tuples are

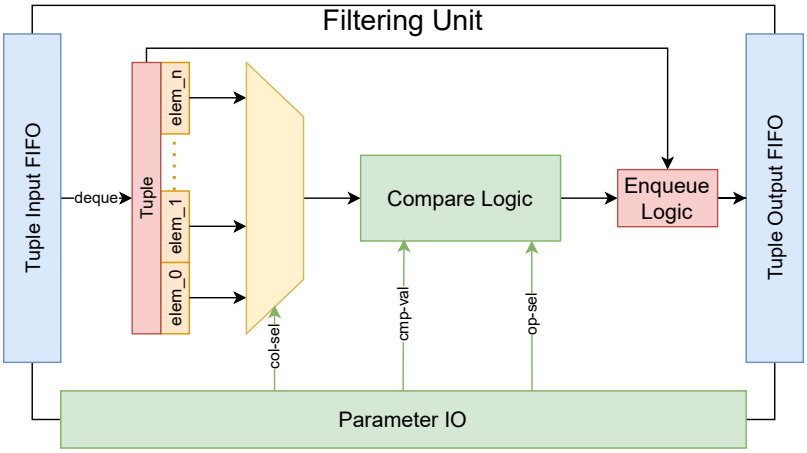


Figure 14.5: Internal structure of the Filtering Unit.

available. According to the padding and type information gathered by the contextual analysis phase, this word is split into a vector of correspondingly padded words. A second vector contains all of the disregarded string-postfixes. The string-postfixes are carried along the computations, but cannot be accessed. The Output Buffer reverses the transformation of the Input Buffer, so that the result can be stored back by the Store Unit.

Filtering Unit This module provides the selection-functionality on the incoming stream of tuples. To do this, hardware is generated that allows the comparison of tuple-members against a given reference-value using a set of compare-operations. An important extension over the work presented in [18] is the fact that the set of operators can be easily extended in our toolflow. Each operation is represented using a function mapping two data-words to a boolean value, which in turn is used to determine, whether a tuple is filtered out. Using a user-defined set of operations or the pre-defined standard set of operations ($\neq$, ==, >, >=, <, <=, nop), the Compare Unit is generated. Since our toolflow relies on the Chisel3-framework [3] for the implementation of the actual hardware, this also enables flexibility. For example, the framework supports interfacing to Verilog and VHDL, which in turn allows addition of custom compare-operations. A schematic view of the filtering unit is shown in 14.5.

The input and output are FIFOs. In each cycle, a present tuple is dequeued from the input FIFO and one of its elements is selected using a multiplexer. This element is used as input to the Compare Unit which also uses the `compare_value` and `operator_select` to determine the exact operation to perform. The resulting signal is used to determine, whether the current tuple is to be enqueued into the output queue. A very important advantage of this architecture is the chainability. Due to the clear interface, this unit can be chained multiple times to

allow the evaluation of multiple predicates in a pipeline, which was not possible with the architecture in [18].

Data Transformation Unit The Data Transformation Unit is automatically generated from the given struct-types. Both input and output are tuple-FIFOs. During the generation of the Data Transformation Unit, the framework will automatically match each (nested) field of the output-struct to the appropriate (if any) field of the input-struct. Using this mapping of input- to output-fields, hardware will be generated that implements this transformation. In general, there are three cases: 1) When the input and output are of the same struct-type, tuples are simply passed through. 2) If the output-struct contains *only* (nested) fields that are also present in the input-struct, the mapping is automatically derived. 3) If the output struct-type contains (nested) fields that are not present on the input, the user has to specify which (nested) input-field is to be used. While this is very flexible, it also requires user interaction in the form of corresponding annotations. An example for this is shown in 14.4 with the `mapping`-key. Using this key, it is defined that `y` and `z` are used for the projection into 2-d space. Without a mapping, the toolflow would default to the second case and use `x` and `y` for the projection.

Composition All of the described modules are then composed into a PE. Due to their latency-insensitive design, the corresponding interfaces can be directly wired-up. Additionally, all modules are automatically connected to their respective control registers. The control register file is automatically configured to provide the required number of registers.

14.4.3 *Automatic Generation of the Software Interface*

In addition to automatically generating the PEs for performing the NDP operations, we also added a tool pass, which automatically generates a *software-interface* for controlling the PEs. The reasoning behind this is to allow a database-engineer to use the PEs without any additional knowledge about how they work and how they are controlled.

Using the information about the Control Register File and the behavior of the PEs, we generate the software-interface bottom-up: First, we generate compiler-macros for encoding the different addresses. From these macros, we built simple software-functions for accessing the different control registers. In a final step, we use these access-functions to built more complex functionality, such as synchronous and asynchronous filtering functions using one or multiple of the filtering stages. For debugging-purposes, functions are generated for printing the state of the PE and for outputting the corresponding data-types. All generated functions are collected in a single header-only library

```
/** Control Register Addresses. */
#define START 0
#define BUSY 4
[...]
#define FILTER_OP_0 60
#define CYCLE_COUNTER 64
/** Generated  Functions */
uint32_t filter_sync(...) {...}
uint32_t filter_async(...) {...}
void wait_until_done(...) {...}
```

Figure 14.6: Snippet from the generated software-interface that can be used to interact with the PEs.

file, which can then be added to the project by the database-engineer in order to exploit the PEs.

An example-snippet of the generated header-only library file is given in 14.6.

14.5 EVALUATION

We will first compare our automatically generated PEs against the hand-crafted units used in [18]. Since [18] has already shown that the NDP approach outperforms the typical non-NDP approach, we will omit this discussion. Then, we will examine the hardware utilizations of the generated PEs and determine their usability on the OpenSSD Cosmos+ SSD platform. All hardware-syntheses are run targeting the Xilinx Zynq-7000 SoC (XC7Z045). In all designs, the Flash controllers and processing elements are clocked at a frequency of 100 MHz, while the NVMe-Core is clocked at 250 MHz, which is in line with the original baseline. While a higher frequency could improve the performance of the PEs, the main bottleneck in this architecture is the available Flash bandwidth.

Performance For the performance evaluation, we use the same benchmarks as in [18]. They work on a sample dataset for a publication reference graph. The nodes of the graph are papers published in journals and conferences. The edges of the graph are references between those papers. Overall, the dataset is comprised of 3,775,161 Paper-Entries and 40,128,663 references between them. For the evaluation, we run GET- and SCAN-operations using the same software-NDP baseline as well as the adapted algorithm, which uses the corresponding PEs. Note that for both operations the execution is implemented in a hybrid way, where the software executes a very general algorithm and exploits the hardware whenever datablocks have to be filtered or transformed.

The resulting NDP-runtimes for GET are shown in 14.7 (a). Note that both the NDP hardware and software runtimes we report for

GET are slightly slower (ca. 10%) than those given in [18]. This is due to updated firmware for the COSMOS+ board, which traded some performance for higher reliability. As described in [18], it also makes sense that the GET-operation does not profit greatly from hardware support, since it is sequential and the configuration-overhead (i.e., writing control registers) of accelerators is too high to make an overall difference. Even though, the GET-operation does not improve, the automatically generated PEs are similar in performance in comparison to the ones used by [18].

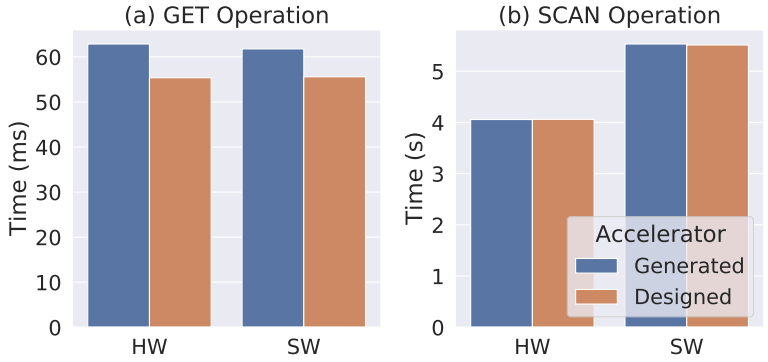


Figure 14.7: Execution times of the GET and SCAN operations, comparing our work to the work provided in [18]. For both Operations execution is executed with HW-acceleration (HW) and without (SW).

The SCAN operation has much longer runtimes, making the minor firmware-induced timing variations between [18] and our measurements negligible. As in [18], the hardware-accelerated NDP SCAN is faster than the software version. The performance of our generated accelerator is on par with the manually optimized one as shown in 14.7 (b). Using the generated PEs slightly increases the runtime by 0.018 seconds from 5.512 seconds to 5.530 seconds.

An additional extension of our work is the possibility to generate PEs featuring multiple filtering stages. Using multiple pipelined filtering stages allows the implementation of more complex NDP-functionality. Moreover, due to the use of elastic pipelines, additional filtering stages will only add very small increases to the overall execution times. Since the filtering stages are able to process a tuple per cycle, the increase in latency of additional filtering stages will be marginal. Especially for compute-bound tasks, this would give the hardware accelerators an edge over the use of the on-device ARM-cores.

Hardware Utilization We generated accelerators that provide the same filtering and transformation functionality as [18] and compare our hardware utilization against theirs. Specifically, we use 1 paper-PE to process the nodes in the graph and 7 ref-PEs to process the edges. Since [18] only reports slices for the PEs, we limit our comparison to

slices as well. Please note that each of our generated accelerators also uses a single BRAM slice, which was not the case for the custom built processing elements of [18].

Table 14.1: FPGA Resource Utilization of the PEs used in [18] and our work. The design contains the complete COSMOS+ OpenSSD platform as well as 1 paper-PE and 7 ref-PEs.

	Slice Util. (abs.)		Slice Util.(%)	
	[18]	Our Work	[18]	Our Work
Overall	40821	41934	74.70	76.73
paper-PE	9480	14348	17.35	26.25
ref-PE	1277	1446	1.41	2.65
Available	54650	54650	100.00	100.00

14.1 shows the corresponding utilization results. It is noteworthy that for both of the PE-types, the resource utilization has grown. Some of this can be attributed to the improved Load- & Store units, which have become more flexible. Specifically, instead of always processing blocks of a certain size, our infrastructure can be configured to load only partial data blocks. Analogously, the Store-Unit can be configured to write back partial blocks. Since the Data Transformation will typically strip data away, this reduces the overall amount of data read and written, which in turn reduces memory contention. Also, note that the overall increase is *less* than expected, considering the size increases of the individual PEs. This is due to a more efficient use of interconnects in our refined architecture template.

We also evaluated the amount of hardware required for multi-staged filtering, as well as for different tuple sizes. For the first part, we take a closer look at the correlation between tuple-sizes and required hardware. For this part of the evaluation, we rely on out-of-context synthesis. In out-of-context syntheses, only a selected part (in our case the PE) is synthesized without the rest of the surrounding architecture. The resulting utilizations represent the amount of logic resources required *without* very dense packing. For the generation of the PEs, we used a number of different input formats that feature tuple sizes ranging from 64 bits up to 1024 bits. For of these sizes, we specified a `struct` with the corresponding number of `uint32_t` and `uint8_t` values. Input and output types are identical and mapped automatically. For each size, we generate a PE that is able to compute on the complete tuple (at the granularity of 32-bit fields) and another PE, where half of the data is discarded using string-prefixes.

The results are shown in 14.8. An interesting observation is the fact that for smaller PEs, the use of string-prefixing yields a higher slice-requirement. At a first glance, this would make the prefixing

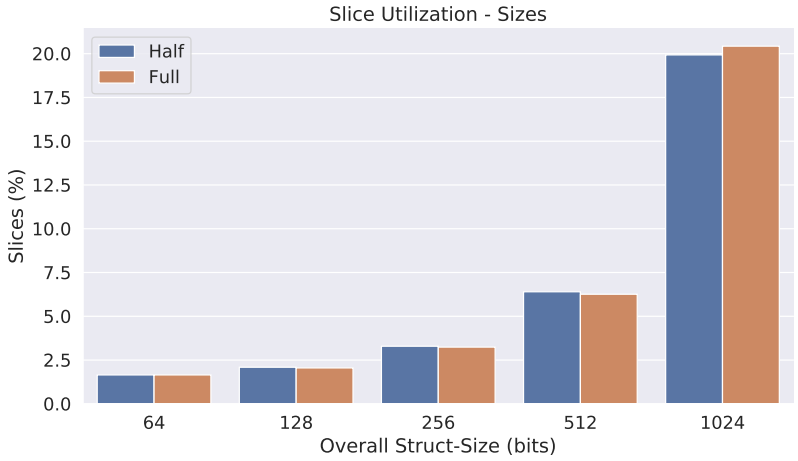


Figure 14.8: Out-of-Context Slice Utilization of generated PEs in correlation to the size of the processed tuples. Half refers to accelerators using the prefixing, whereas Full refers to the ones using all data.

irrelevant. To understand why prefixing is still necessary in some cases, we have to consider that the critical part of our hardware is the Filtering Unit with the compare operations at its core. In 14.9, all fields have a width of 32 bit, which means that the corresponding compare-operators are also 32 bit operators. For the 1024 bit struct, the corresponding string-data would have an overall size of 512 bits. A full-width compare unit would vastly increase the amount of required hardware. Thus it is still reasonable to use the prefixing.

Lastly, we take a closer look at the multi-stage feature and the resulting hardware-requirements. For this part of the evaluation, we reuse the same data-formats as in the previous step, but focusing on 256 bit structs only. For both (with and without string-prefixes), we built accelerators with up to 5 filtering stages for more complex predicates. Of these, especially the 2-staged ones are interesting, since they could be used to implement RANGE_SCANs. Again, the utilization results were obtained using out-of-context synthesis.

Looking at the results shown in 14.9, we can see an almost linear correlation between the number of stages and the slice requirement. Additionally, we observe that the increase per additional stage is small compared to the overhead incurred by the fixed part of the template (Load/Store Unit, Tuple Buffers). This implies that multi-stage filtering incurs only minor additional cost, while offering a lot more flexibility.

14.6 RELATED WORK

The first approaches for Near-Data Processing, moving computation closer to the data date back to as early as the 1970s. However, approaches such as database machines [5] or ActiveDisk [1, 12, 15] were

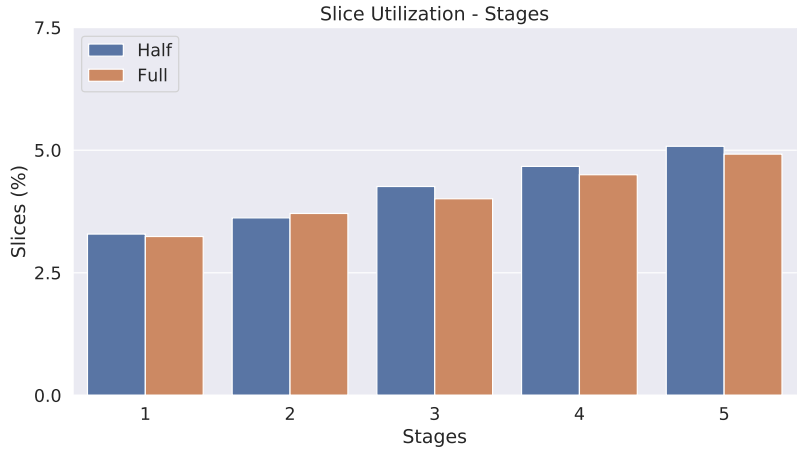


Figure 14.9: Out-of-Context Slice Utilization (in percent) of generated PEs in correlation to the number of filtering stages. Additional stages increase resource requirement in a linear fashion, but provide more flexibility. The use of string-prefixing (Half) has only minor impact.

severely limited by the I/O-limitations and memory bandwidth of mechanical hard-drives.

Only after the wide-spread availability of modern non-volatile storage solutions, e.g., Flash-based SSDs, significant advances in the performance of Near-Data Processing systems became possible. Approaches such as SmartSSD [7, 11, 16] exploit the much higher I/O-bandwidth of modern storage devices as, for example, provided by parallel, independent Flash-channels. JAFAR [2, 20] was one of the first systems focusing on Near-Data Processing for DBMS. Biscuit [10] was another approach targeting NDP for DBMS, namely MySQL. In contrast to our work, they only employed the ARM-based CPUs found in commodity SSD hardware for software-based Near-Data Processing, but also identified the lack of a usable framework for programming NDP PEs as an important issue. Our framework allows to automatically generate FPGA-based Filtering and Data Transformation units from simple user-input. It thus offers a solution to make FPGA-based NDP acceleration accessible to non-FPGA experts.

With their HRL architecture [9], Gao et al. present a new hardware architecture targeting NDP that combines fine-grained reconfigurable regions, as found on FPGAs, with coarse-grained regions as common in Coarse-Grained Reconfigurable Arrays (CGRA). Their overall system architecture combines this accelerator with DRAM in an Hybrid Memory Cube (HMC), but does not include non-volatile memories.

Architectural challenges and other considerations on how to integrate FPGAs into Near-Data Processing architectures were discussed by Dhar et al. [6] and Becher et al. [4]. While Dhar et al. envisioned an architecture featuring Flash storage and a combination of FPGA and High-Bandwidth Memory (HBM), with the FPGA processing

data cached in HBM, the ReProVide architecture proposed by Becher et al. uses a combination of an ARM CPU and an FPGA, similar to our approach. In the multiple dynamically reconfigurable regions of the FPGA, different pre-synthesized NDP PEs can be used. However, these accelerators must be hand-crafted and cannot be generated automatically.

14.7 CONCLUSION & OUTLOOK

In this work we have developed a framework for the automatic generation of FPGA-based accelerators for the use with Near-Data Processing applications. Our evaluation shows that our automatically generated accelerators provide almost identical performance compared to a setup with hand-crafted hardware accelerators. This is worthwhile, since our approach effectively removes the need for custom hardware development and lowers the entry barrier for hardware-accelerated databases. Moreover, our multi-staged filtering approach enables more powerful computations with minimal overhead.

While filtering and transformation of data are wide-spread use-cases that can easily be realized using our framework, more computational and analytical tasks could also be performed using this architecture. In future work, we will investigate, how we can leverage the data-parallelism of the architecture to perform more compute-intensive tasks. Using our architecture, it is possible to access and process all tuple-elements in parallel, which could offer great potential for faster analysis of the processed data.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS 1998*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[3] J. Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language." In: *Proc. DAC 2012*. 2012.

[4] Andreas Becher et al. "Integration of FPGAs in Database Management Systems: Challenges and Opportunities." en. In: *Datenbank-Spektrum* 18.3 (Nov. 2018). ISSN: 1610-1995. (Visited on 08/22/2020).

[5] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

[6] Ashutosh Dhar et al. "Near-Memory and In-Storage FPGA Acceleration for Emerging Cognitive Computing Workloads." In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2019.

[7] Jaeyoung Do, J. Patel, D. DeWitt, and et. al et. "Query Processing on Smart SSDs: Opportunities and Challenges." In: *Proc. SIGMOD 2013*. 2013.

[8] Facebook. *RocksDB*. https://github.com/facebook/rocksdb. 2020.

[9] Mingyu Gao and Christos Kozyrakis. "HRL: Efficient and flexible reconfigurable logic for near-data processing." In: *2016 IEEE Intl. Symp. on High Performance Computer Architecture (HPCA)*. 2016.

[10] Boncheol Gu et al. "Biscuit: a framework for near-data processing of big data workloads." In: *ACM SIGARCH Computer Architecture News* (June 2016). ISSN: 0163-5964. (Visited on 08/21/2020).

[11] Yangwook Kang, Yang-suk Kee, and et al. "Enabling cost-effective data processing with smart SSD." In: *Proc MSST 2013*. May 2013.

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

[13] Chen Luo and Michael J. Carey. "LSM-based storage techniques: a survey." In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

[14] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Inform.* (1996).

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

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

[17] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. "Cosmos+ OpenSSD: A NVMe-based Open Source SSD Platform." In: *Flash Memory Summit* (2016).

[18] Tobias Vinçon et al. "NKV: Near-Data Processing with KV-Stores on Native Computational Storage." In: *Proc. 16th International Workshop on Data Management on New Hardware*. Portland, Oregon: ACM, 2020. ISBN: 9781450380249.

[19] Lukas Weber et al. "On the necessity of explicit cross-layer data formats in near-data processing systems." In: *Distributed and Parallel Databases* (2021). DOI: https://doi.org/10.1007/s10619-021-07328-z.

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

Part IV

NDP OFFLOADING MODELS

15

NEAR-DATA PROCESSING IN DATABASE SYSTEMS ON NATIVE COMPUTATIONAL STORAGE UNDER HTAP WORKLOADS

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads"* by Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhard, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch and Ilia Petrov in 2023 49th International Conference on Very Large Data Bases (VLDB). The contribution of the author of this thesis is summarized as follows.

» As the corresponding and leading author, Tobias Vinçon was resposible for the conceptional details and implementation of the shared state propagation in nKV. He contributed the NDP interface, parsers and accessors, the orchestation of software and hardware-based NDP as well as NDP pipelines and operations which were utilized in the experimental evaluation. Thereby, Leonardo Solis-Vasquez extended the COSMOS+ architecture with mutli-core functionality. Arthur Bernhard was in charge of providing the neoDMBS implementation in cooperation with Sajjad Tamimi and Florian Stock. Extending Linkbench with HTAP functionality was contributed by Christian Knödler. The manuscript's text was created in joint work by Tobias Vinçon and Ilia Petrov with feedback from all authors including Andreas Koch. «

ABSTRACT

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

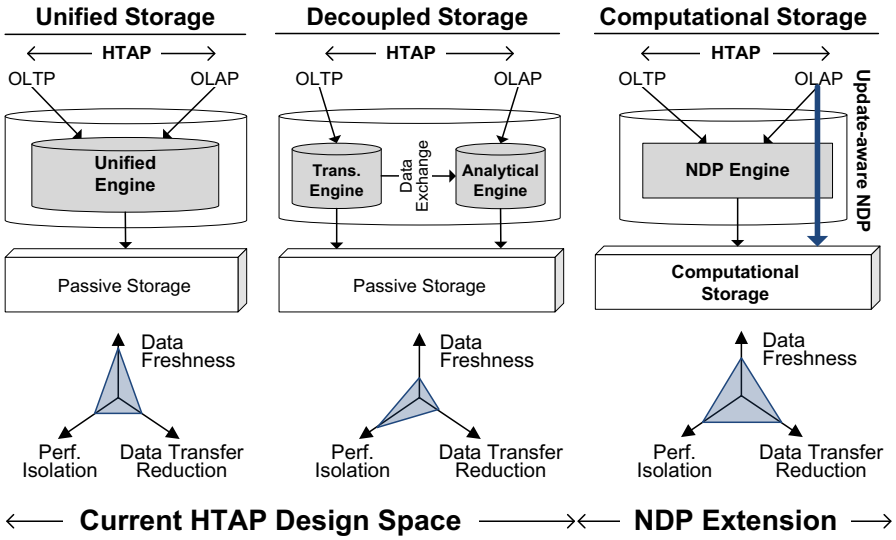


Figure 15.1: State-of-the-art HTAP architectures can be divided into unified and decoupled storage systems [58] and optimize either on data freshness or performance isolation. NDP and computational storage allow tackling both dimensions, while reducing cold data transfers for better performance.

15.1 INTRODUCTION

Modern data-intensive systems run Hybrid Transactional and Analytical Processing (HTAP) workloads combining long-running analytical queries (OLAP) as well as frequent and low-latency update transactions (OLTP) on the same dataset and even on the same system [53]. Such hybrid systems operate with continuous update rates on a hot portion of a large dataset, while performing complex analytical tasks on both the hot and the much larger cold part of the dataset. Consequently, large data transfers of cold data occur that are partly due to poor data locality, but also due to traditional (*data-to-code*) system architectures. Such transfers entail non-robust performance, scalability issues, and poor resource efficiency.

Near-Data Processing (NDP) is a *code-to-data* paradigm targeting in-situ operation execution, i.e. as close as possible to the physical data location. NDP leverages the trend towards *smart/computational storage* as hardware manufacturers can fabricate *combinations of storage and compute* elements economically, and package them within the same device. Furthermore, with semiconductor storage (NVM/Flash), the *device-internal* bandwidth, parallelism, and latencies are much better than the external ones (device-to-host). Both trends lift major drawbacks of prior approaches like ActiveDisks [1, 34, 59] or Database Machines [14], such as bandwidth limitation and expensive proprietary hardware. Interestingly, even commodity devices nowadays come

with compute hardware used for running backwards compatibility firmware not for data processing.

Based on their storage design, two types of HTAP architectures can be distinguished [58]: unified and decoupled storage (Fig. 15.1). The former executes the OLTP and OLAP operations on the same dataset based on snapshotting and multi-versioning techniques, and is optimal for analytical processing on latest data. The latter separates the OLTP and the OLAP sub-system. It trades higher (but amortizable) OLAP response times, data freshness, selective access, workload adaptability, for higher OLTP throughput.

Problem 1: HTAP architectures cause transfer of cold data. In state-of-the-art large-memory settings, the *working dataset* fits in memory, yet the *complete dataset* is much larger (cold, historic data) and available on persistent storage. For instance, Umbra [51] is a novel system, representative of the new class of hybrid in-memory/SSD-based high-performance DBMS. HTAP workloads tend to process/analyze cold data, which is generally not in memory. Existing HTAP architectures (Fig. 15.1) assume passive storage, and thus OLAP processing entails large transfers of cold data. This impacts the system performance, and limits its scalability and resource efficiency (as shown in a motivating experiment – Fig. 15.2 – discussed later on). A key observation of this paper is that NDP naturally fits in the HTAP problem space, as NDP allows in-situ operations to process cold data without *moving* it to the host. NDP enables *intervention-free* execution, where the DBMS can continue processing, after asynchronously offloading NDP operations and delegating their execution to computational storage.

Problem 2: NDP necessitates transactional consistency. Despite all its advantages, NDP is currently utilized solely in read-only settings. Yet, in update-intensive HTAP settings, the most recent modifications of OLTP-style transactions are only available in the large DBMS memory [21], likely scattered across different data structures. However, analytical NDP operations from OLAP transactions, offloaded to computational storage, require that most recent data in-situ, alongside the cold persistent dataset, to achieve *consistency* and *freshness* guarantees. These properties are necessary since NDP operations execute in the context of the invoking transaction. The question of how the most recent data can be collected and propagated to smart storage, and how consistency and freshness can be ensured is still considered open [21].

Update-aware NDP. In this paper, we propose a *snapshot-based*, *update-aware NDP architecture* for computational storage and HTAP workloads. The *core idea* is to define a small shared state that accumulates modifications to main-memory data and DBMS state. Noticeably, the *shared state* is the only delta between the working set in the large DBMS memory, containing the most recent updates, and the much larger, but colder and complete dataset on computational storage. The shared state is regularly flushed to computational storage, whenever

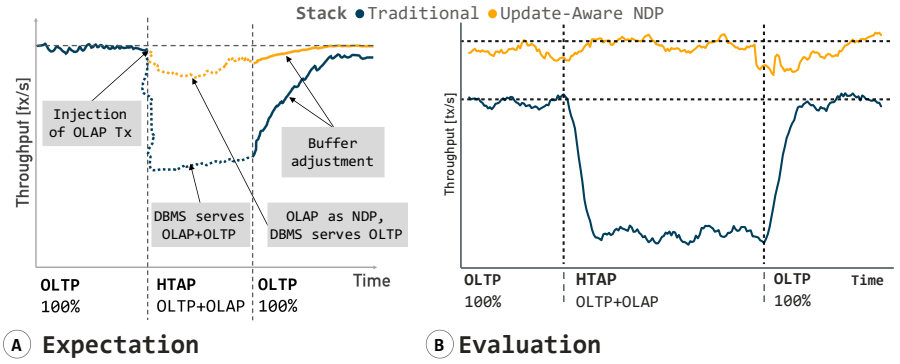


Figure 15.2: Expectation: Ⓐ traditional systems suffer the impact of data transfers after an OLAP query injection. Update-aware NDP executes OLAP operations in-situ, with robust performance. Our evaluation Ⓑ confirms this behaviour.

it reaches a pre-defined limit, but most importantly it is propagated as part of every NDP invocation. Thus, at the point of invocation the computational storage attains a complete and consistent *snapshot*, and the read-only NDP operation can execute with consistency guarantees. Moreover, the in-situ execution is asynchronous and free from DBMS/host intervention, as the computational storage now has a complete snapshot of the entire dataset. Although our architecture is aligned with disaggregated storage architectures [17, 45, 70], here we target in-situ processing.

We demonstrate the impact of NDP in HTAP settings with a motivating experiment (Fig. 15.2) in MySQL/MyRocks as traditional stack and MyRocks over nKV [69] as an NDP stack. In the first phase, both systems run LinkBench [5] as an OLTP workload. In the second HTAP phase, we inject an analytical OLAP operation (Betweenness Centrality, performing a graph analysis) in parallel to OLTP. Upon its completion, we switch back to pure OLTP in the last phase. The clear performance drop with the traditional stack during the HTAP phase is due to excessive cold data transfers. Noticeably, the OLTP throughput remains unchanged with the NDP stack, because of the intervention-free in-situ OLAP execution.

Our contributions are:

- We propose an *update-aware NDP architecture* that utilizes a small *shared state* to create a consistent snapshot on computational storage and execute the analytical NDP operation in-situ against it. Such NDP executions have snapshot transactional guarantees.
- We propose new NDP execution models. NDP storage can execute operations asynchronously in an *intervention-free* manner. Currently, we focus on read-only, analytical NDP operations.
- We describe how intermediary and final results can be handled in-situ to reduce data-transfers.

- A case study shows the implementation in two different systems: a key-value store (nKV), and a multi-version DBMS (neoDBMS). The evaluation is performed on real hardware - *COSMOS+* [18].

This paper is organized as follows. The next section provides necessary background and discusses related work. In Sect. 15.3 we go into the details of the proposed update-aware NDP architecture and describe the concepts behind its execution model. The experimental evaluation is discussed in Sect. 15.4 and we conclude with Sect. 15.5.

15.2 BACKGROUND AND RELATED WORK

We now discuss the current state-of-the-art HTAP systems from the industry [25, 26, 42, 43, 57] and academia [2, 4, 6, 7, 15, 28, 35, 39, 44, 48, 50, 58, 62] and classify their approaches into the HTAP design space. Our goal is to present: (a) the relevant background on the HTAP design space and the extension with NDP, (b) an overview of important aspects in regard to NDP and today's systems support [3, 8, 9, 20, 21, 29, 31–33, 36, 49, 63, 64, 73, 74, 76], as well as (c) a brief outline of the native storage concepts as they form the foundation of the present system architecture.

15.2.1 *HTAP Workload and Systems*

Today's database systems persist and operate on large and ever-increasing amounts of data. However, the processing no longer involves only OLTP-style workloads, operating on a small but hot portion of the entire data. Moreover, real-time analytical queries (OLAP), often with very complex algorithms, operate on the cold data from the storage tier as well as the freshest updates from the OLTP workload. The combined workload, termed Hybrid Transactional and Analytical Processing (HTAP), extends the problem space of database architectures with the following aspects [15, 48, 58]:

- *Data Freshness.* For the analytical portion of the HTAP workload, the given system architecture should aim for having the most recent version of data resulting from updates performed in the OLTP workload. Therefore, fast propagation of these updates to the analytical snapshot is required, and optimally avoids any performance drops for the transactional workload.
- *Data Consistency.* Regardless of the data freshness, the entire system must ensure transactional guarantees for its transactions so that transactional and analytical queries have a consistent view on the data. Various mechanisms have been proposed and applied in databases to construct the so-called required snapshot. Two of the most prominent are the Copy-on-Write (CoW) approach and Multi-Version Concurrency Control (MVCC). The first ensures the visibility of older versions by creating a copy for modifications. The

second creates a new version for every modified record and extends it with the current timestamp.

- *Data Transfers.* Independently of the separation of the analytical from the transactional engine, data transfers from the storage tier to the host processing units account for a large part of the overall performance. They result from: (a) HTAP processing of cold- and hot-data likewise, and (b) the cold data being much larger than the hot data and thus, causing buffer pollution and finally high eviction rates. Overall, the result is limited scalability, bandwidth boundness, and performance loss.
- *Performance Isolation.* DBMS are often used by business-critical applications, for which robust performance in terms of latency and throughput is essential. Interference between OLTP and OLAP workload must be prevented, in particular for hybrid scenarios that run both of them concurrently [48].
- *Memory Pollution.* While OLTP workloads operate on the hot data (working set) that fits in state-of-the-art large main memories, the major portion for OLAP processing is the cold data that exceeds the memory capacity. Hence, OLAP scans in hybrid scenarios inevitably entail buffer pollution in case the transactional and analytical engines share the same buffers, e.g. database buffers or OS page cache and device caches. Even though the buffer size is usually defined to be larger than the hot portion of the data set, the analytical queries have to fetch large parts of the cold data into the buffer, and thus cause evictions of the hot data.

Current state-of-the-art architectures proposed for HTAP scenarios can be classified into two major categories [58]. Firstly, Unified Storage Systems build snapshots for every occurrence of an analytical query. Consequently, this kind of system operates on the freshest data and is optimal for in-memory OLAP processing. Widely-known systems of this category are HyPer [35], Caldera [4], DB2 BLU [57] or SAP HANA [25]. Secondly, Decoupled Storage Systems continuously transfer modifications from the transactional engine to the separate analytical engine. Thus, workload optimizations and performance isolation can be introduced at the cost of data freshness. BatchDB [48], SQL Server [43] or Oracle's Dual Format [42] are representatives for this category.

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

compute placement, as today's devices often come with multiple heterogeneous processing capabilities.

15.2.2 *Near-Data Processing*

Early approaches of Near-Data Processing date back to the 1980s-90s. *Database machines* [14] or *Active Disk/IDISK* [1, 34, 59] introduced proprietary magnetic/mechanical storage hardware. However, manufacturing costs combined with the low bandwidth and parallelism became limiting factors. With the advance in the semiconductor industry, Flash technologies and reconfigurable processing elements arose, and Smart SSDs [21, 63] were proposed. Since then, a variety of specific database and generic NDP frameworks were introduced such as IBEX [73, 74], Minerva [20], Willow [63], BlueDBM [49], JAFAR [8, 76], Kanzi [31], ISP [38], YourSQL [33], Biscuit [29], PapyrusKV [36], DoppioDB [3, 64], Caribou [32], Batched Writes [22], BlockNDP [9], Umbra [51], PolarDB [17] or nKV [68, 69]. Besides avoiding costly data transfers between host and device, each NDP approach optimizes for specific characteristics.

Storage Properties. By moving the execution closer to the storage, the opportunity to intensively leverage the hardware properties of storage technologies emerged. Flash, NVM, and HBM are widely utilized in NDP approaches due to their extremely parallel interfaces. Thus, significantly higher on-device bandwidths can be achieved in contrast to communications with the host. Indeed, [37] makes the case for 50 GB/s device-internal versus 6.4 GB/s device-to-host bandwidth. This is due to the physical organisation of semiconductor storage devices, which involves multiple chips, connected over independent channels to the on-device processing element. The chip-level bandwidth increases with chip density, which in turn increases due to modern 3D stacking technologies.

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

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

large scans. Depending on aspects such as the actual operation, workload, and underlying storage technology, the computation placement decision can vary and hardware can be configured individually for each NDP invocation.

Disaggregation and shared-storage architectures. The proposed NDP architecture is aligned to current disaggregated storage architectures [17, 45, 70]. These aim at elasticity and pushdown of database operations in the storage layer that decouples the CPU resources on the compute nodes from those of the storage nodes. The present work aims at NDP, which is a distinct subset of that problem space targeting in-situ processing.

15.2.3 *Native Storage*

Under native storage [54], the DBMS operates directly on the physical storage without intermediary layers of abstraction. Consequently, functionality like address mappings or garbage collection, that appears multiple times along the I/O path of traditional system stacks, can be combined and deeply integrated into the DBMS. This benefits not only the workload-aware scheduling, but also enables leveraging the hardware characteristics. Especially with the advent of Flash as general-purpose storage in today's data centres, the throughput is highly dependant on the utilization of parallel I/O units, e.g. channels and LUNs [68]. Thus, native storage also establishes novel storage abstractions like Regions and Groups [30] that can adapt to the workload at runtime per database object, and improve throughput, latency, and reduce write-amplification [67]. Other approaches, not yet as deeply integrated into the database as native storage, are pursued by [13, 55].

15.2.4 *Update-aware NDP Systems*

The components of the update-aware NDP architecture (Sect. 15.3) are generic, aligned with existing architectures of modern DBMS, and are easy to integrate. Throughout this paper, we focus on two systems: LSM-tree KV-Stores and multi-version DBMS.

Firstly, we employ nKV [68], a KV-store based on RocksDB, which can be exposed as a MySQL storage engine by means of MyRocks (*MyRocks over nKV*). It is a single-versioned, Copy-on-Write system, supporting `Repeatable Read` as the highest isolation level. The underlying data organisation is based on multi-level LSM-Trees [47], with C_0 being an in-memory skiplist-based MemTable, and $C_1 ... C_n$ organized as Sorted String Tables (SSTs) on persistent storage. The latter comprise data blocks with the actual KV-Pairs, and an additional index structure referencing these. As shown in Figure 15.3, each active transaction is assigned a separate *write batch* that contains transaction-local modifications before commit. Thus, transaction reads

are first issued against their batch, before querying the MemTables or the persistent data. Modifications of a transaction are invisible to other transactions as they go to separate WriteBatches. Considering the example of Figure 15.3, a snapshot taken during TX_4 comprises only $key\_a = 11$ and $key\_b = 2$.

Secondly, we introduce *neoDBMS* [10], as a multi-version NDP-DBMS based on PostgreSQL. neoDBMS stores all updates as physically materialized version records (Fig. 15.3). As such they are identified by an implicit *RecordID* (<*PageNr,SlotNr*>). The version records of each tuple form a version chain organized as a singly-linked list in a New-to-Old (N2O) manner, where every version has a forwards reference to its predecessor [27]. The invalidation of a version is handled implicitly by the presence of a successor version. All version records in a chain have the same *virtual id* (VID, e.g. t_a or t_b) as they belong to the same tuple. To mark the entry-point of a chain, neoDBMS introduces a VID_{Map} containing the RecordID of the latest version of each tuple. The N2O organisation yields fast visibility checks, especially for fresh data. In addition to the *VID* every physical record contains a transactional creation timestamp, unique for each version chain. These are utilized by the version visibility check to construct a transactionally consistent snapshot. For instance, to construct the snapshot between TX_3 and TX_4 (Fig. 15.3) the version chain is traversed to determine the first visible version of each tuple to a transaction, e.g. $t_a.v_3$ and $t_b.v_1$ to a transaction starting at the time of the snapshot.

15.3 UPDATE-AWARE NDP ARCHITECTURE

We begin with an overview of the components of the proposed architecture (Fig. 15.4) and elaborate on them in the sections to follow.

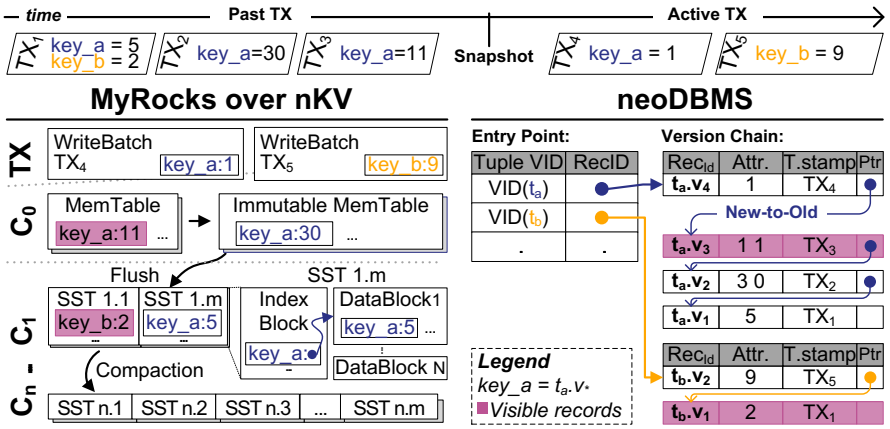


Figure 15.3: Storage organization under nKV and neoDBMS.

15.3.1 *Shared State and NDP Execution Model*

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

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

Shared State. DBMS usually maintain the freshest data, mapping tables, status or system information, in the large and fast main memory of the host system. Modifications or newly inserted records are scattered across the database address space (Fig. 15.5.A) and remain there, until they get evicted. Beyond actual records, modifications spill across various auxiliary structures, such as status and mapping tables, e.g. logical-to-physical address mapping. However, NDP operations require all of the latest data and state, to ensure transactional guarantees. In the words of Do et al. [21]: *"If there is a copy of the data in the buffer pool that is more current than the data in the SSD, pushing the query processing to the SSD may not be feasible."*

In the update-aware NDP architecture (Fig. 15.5.B), we accumulate the modifications to all of those structures in an incremental way and place them together in shadow data structures that are collectively referred to as the *shared state*. As a result, the original data is left unmodified in the large memory of the DBMS. The shared state is small and configurable in the range of a few hundred KB to a few MB at most and can be propagated at low overhead.

The *Delta-Buffer* is a key element of the shared state. It accumulates modifications as replacement records. Thus, records in the delta-buffer

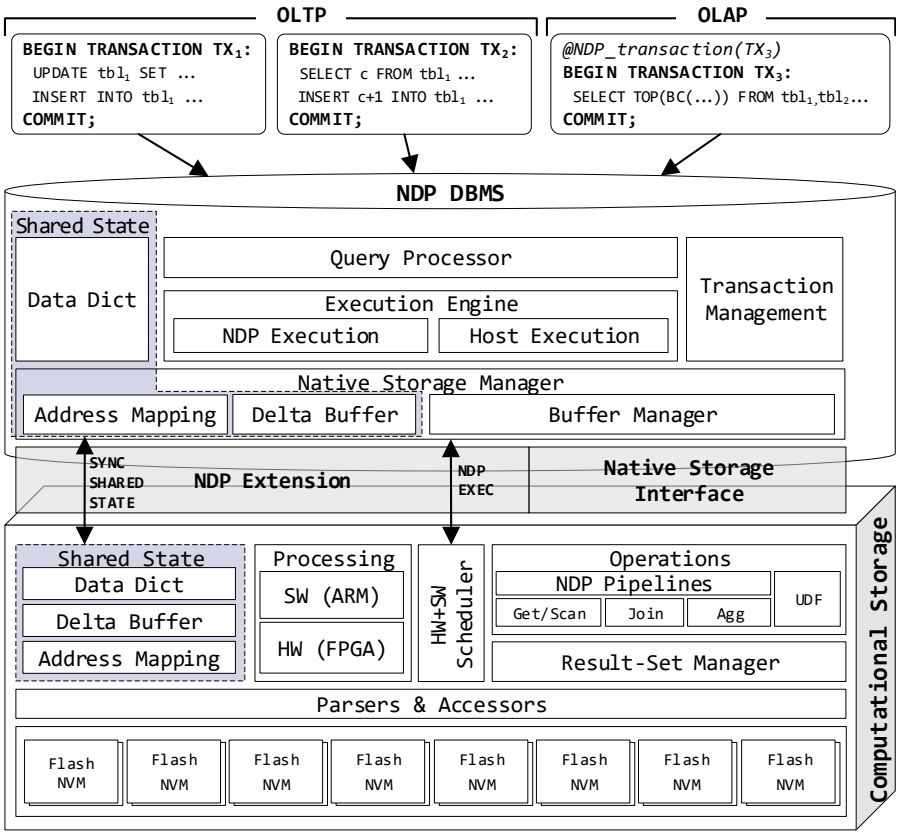


Figure 15.4: Update-aware NDP enables a transactionally consistent in-situ processing of OLAP operations.

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

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

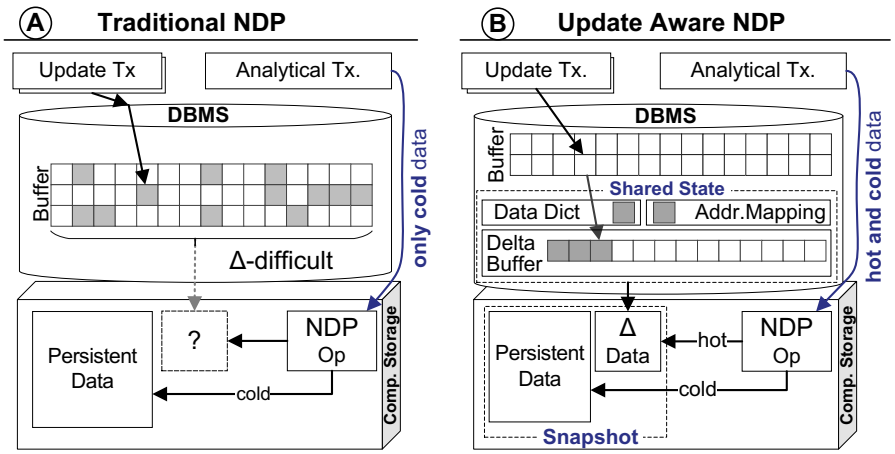


Figure 15.5: Modifications are gathered in a small in-memory shared state and propagated to computational storage, for consistent snapshot-based hot and cold data processing.

Thus, every DB-object can be assigned to a separate and individually-sized delta-buffer, to account for different levels of hotness and update patterns.

The concept of shared state is *aligned* with existing architectures of modern DBMS and is easy to integrate. For instance, it resembles the staging area in modern main-memory DBMS, such as SAP HANA [65], the delta storage in multi-version DBMS [75] or the batch-write transactional buffer in RocksDB.

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

The second mode is *Pass Along & Cache*: At the time of an NDP invocation, the shared state is snapshotted and, together with the list of transactions currently in-flight, propagated to storage as part of the

NDP invocation. The state is merely cached on-device for the duration of the call, and released/garbage-collected upon its completion. This is possible, since the max. shared state size can be configured to be smaller than memory limits of the NDP device.

In this mode, applying the shared state to persistent storage is difficult in the general case since (a) it contains possible modifications of the invoking transaction, yet it is unknown whether it will commit; and (b) its space utilization may be low and incur overhead to successive space management operations (compaction, garbage collection). With the shared state in place, the NDP operation executes in a shared-nothing manner without any intervention or synchronization with the host. Thus, the DBMS and device can operate independently and only synchronize at the end.

15.3.2 *NDP Transaction Management*

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

Any transaction containing NDP operations is called *NDP transaction* (annotated as `@NDP_Transaction`). Transactional consistency mandates that the NDP operations from an NDP transaction must only process modifications by transactions committed prior to its beginning, while ignoring modifications from concurrent transactions other than their own. The issue at hand is that, at the time of the NDP invocation, it is unknown whether concurrent transactions will commit or abort, and thus, which records should be processed. We tackle this by executing the NDP operation against a transactional snapshot created for it in-situ (described below). Noticeably, the snapshot construction and the execution are intervention-free, since inside the shared state, a list of the in-flight transactions is propagated alongside the NDP invocation, and is thus available on the device. This approach works well in the widespread *general MVCC case* [12], where the snapshot comprises only the latest committed version records prior to NDP transaction beginning. For example, NDP operations from transaction TX_3NDP (Fig. 15.6.A) can only operate on data from TX_1, ignoring modifications from TX_2.

Transaction Scheduling. Depending on the DBMS design and the CC flavor, modifications from concurrent transactions might be visible. For instance, MySQL/MyRocks [24] mandates that the visible record is the latest committed ahead of the NDP invocation (snapshot creation), rather than the NDP transaction start. Thus, modifications from TX_2 might be relevant to TX_3NDP (Fig. 15.6.B), but will not be present on device. Propagating them is difficult and unscalable. To this end, we

propose a *transaction admission* mechanism for transactions with NDP operations (Fig. 15.6). Whenever such NDP-transaction arrives (e.g. TX_3NDP), it is assigned a transactional timestamp, as usual. However, its admission is delayed until after the completion of all transactions that were active when it arrived. The delay is typically very short (2ms in our setup), since OLTP transactions are fast relative to slow NDP/OLAP operations. Currently, we allow a single NDP invocation at a time. At the time the NDP-tx. is admitted for execution, no CC anomalies occur, since modifications from TX_2, but also from TX_4 are ignored (Fig. 15.6.C). Hence, TX_3NDP has at least snapshot-isolation guarantees.

Transactional Guarantees. We now analyze how update-aware NDP supports transactional guarantees for read-only operations in two transactional scenarios (Fig. 15.7). In particular, update-aware NDP within nKV is bound to MyRocks' highest isolation level `Repeatable Read` and its MVCC implementation, and can avoid Dirty Read and Non-Repeatable Read anomalies. Since NDP transactions wait for all other active OLTP transactions to complete (commit or rollback) before a snapshot of the current state is taken and the pushdown to the device is issued (see Fig. 15.7.A), it is ensured that the shared state and the delta-buffer only include transactionally consistent data for all previously started transactions. In case another transaction is started right after the NDP transaction, but before the NDP invocation (see Fig. 15.7.B), MyRocks stores all writes of this transaction in a separate *WriteBatch* (Sect. 15.2.4), ensuring that those updates will not be available to other transactions until its commit. Even if the transaction commits during the pushdown execution, the changes will not be present on device, as they have not been propagated with the NDP_EXEC call.

In-situ Snapshot Construction. In-situ snapshot creation is DBMS specific and can be realized with (a) a Copy-on-Write mechanism (MyRocks over nKV); as well as with (b) visibility-checking in multi-versioned DBMS. In Copy-on-Write based systems such as nKV, this snapshot is usually identified via a snapshot identifier, e.g. a sequence number (see Fig. 15.8). Records with a newer identifier are simply skipped during processing. This is possible because the write batching mechanism ensures that the delta-buffer only committed data. For

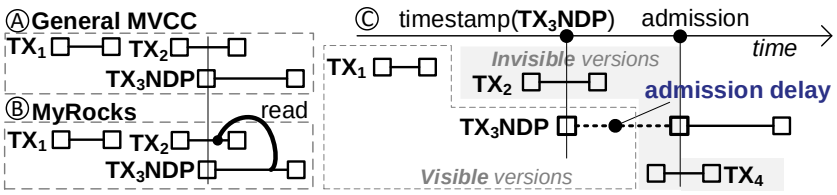


Figure 15.6: NDP transactions are delayed until concurrent OLTP tx. complete ensuring an intervention-free execution.

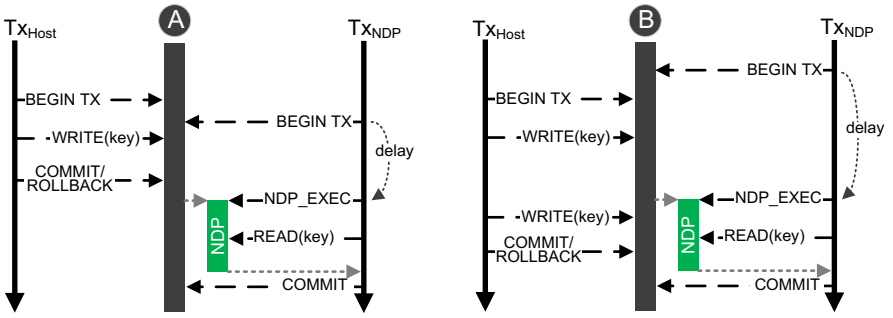


Figure 15.7: Update-aware NDP offers transactional guarantees depending on the integrated database.

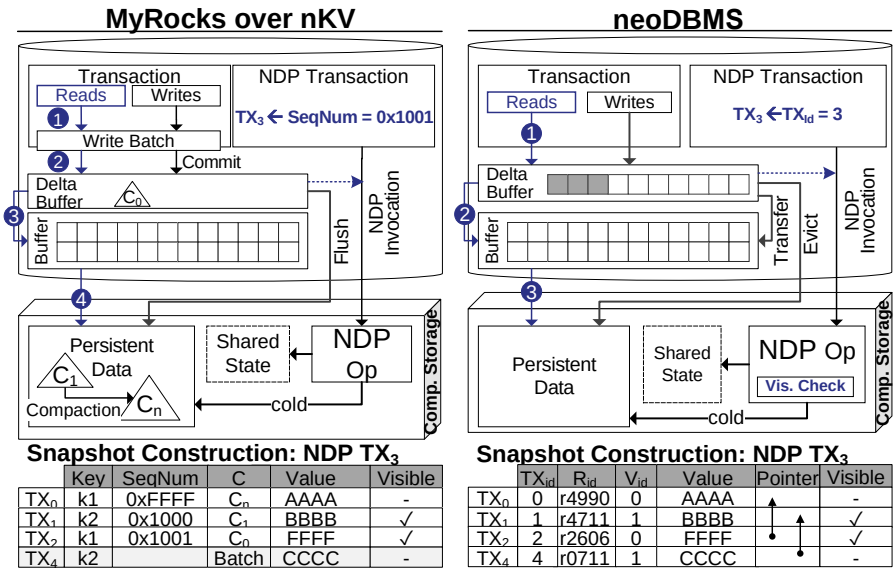


Figure 15.8: In-situ snapshot creation is DBMS specific.

instance, under nKV (Fig. 15.8), the NDP transaction and invocation get a sequence number of `0x1001`, and the in-situ snapshot comprises keys `k1` and `k2`, as `k1` with sequence number `0xFFFF` is skipped due to its lower component level.

In a multi-version system like neoDBMS, an in-situ visibility checking is performed. To this end, the shared state also comprises the version chain information and the list of in-flight transactions at the time of the NDP invocation. Given the invoking *transaction id*, the visibility check can now traverse the version chain backwards starting from the VID_{MAP} entry point (Fig. 15.4) to find the version, visible to the NDP transaction. We utilize newest-to-oldest order and thereby can ensure fast visibility checks, especially for fresh data [27]. For instance, neoDBMS (Fig. 15.8) will only construct an in-situ snapshot for $\text{TX}_{\text{id}} = 3$ comprising version records `r4711` for VID/tuple `1` (as `r0711` as higher creation timestamp) and `r2606` for VID/tuple `0` (as its creation timestamp is the highest $\leqslant \text{TX}_3$).

15.3.3 *NDP Interface*

To enable an efficient pushdown of NDP commands, the lean interface definition of native storage [68] is extended. It builds upon NVMe, yet as a user-space module to avoid high user-/kernel-space switching overhead. Our native NVMe leverages SPDK [66].

Interface Design. Native storage [54] allows operating directly on physical memory, without any intermediary layers, by means of read-/write/erase commands. This interface is extended by a command that transmits the current shared state via the NVMe payload to the device. Furthermore, an NDP_EXEC command extension sends parameter sets to device and can trigger a variety of executable functions (see Sect. 15.3.6). Its parameter set includes: (1) the shared state, and (2) the operation-specific parameters. Moreover, metadata and schema information are also included, i.e. column families and their respective data formats, number of LSM levels, assignment of SST per level, and many more.

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

15.3.4 *Parsers and Accessors*

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

Layout accessors exist for every element of the persistent data layout and help to navigate through the binary data organization and to access sub-elements. For instance, for a given key (Fig. 15.9), accessors allow navigating through the index block of an SST to the physical location of a record within a data block. Accessors are simple to realize, with a microarchitecture resembling load units. They can be instantiated multiple times to increase the parallelism.

Format parsers. While accessors handle in-situ navigation, format parsers are required to *extract* persistent binary elements (records,

values), interpret them semantically, and allow for further processing, mathematical operations, or comparisons. We actually distinguish field, record, and page formats and layouts for this purpose. For instance, in MyRocks, each element of the LSM-Tree based data organization (Fig. 15.9, right side) corresponds to a specific parser and accessor. The index block is interpreted according to its format, and the physical page pointers to the data blocks are extracted. Similarly, the data block is processed by the respective parsers and accessors to obtain the actual records, which themselves contain elements such as (a) an identifier, including a *column_family_id*, all *primary key* fields, the sequence number, and the key/value type; and (b) the actual value formatted according to the DDL definition.

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

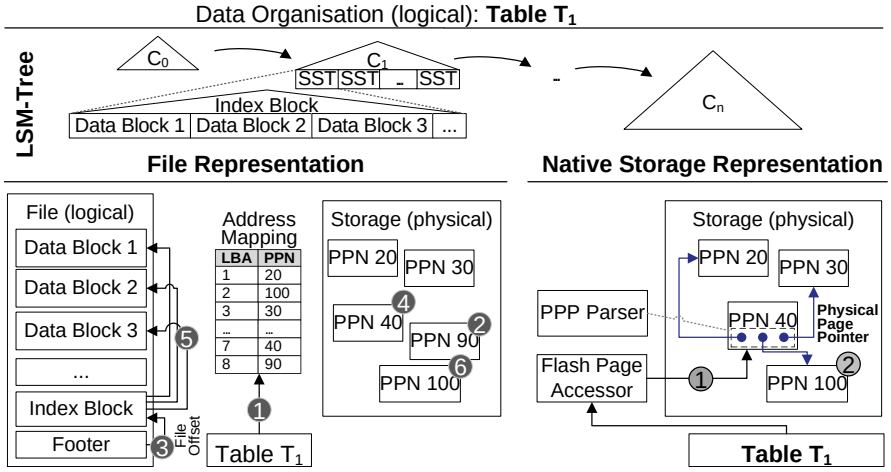


Figure 15.9: Physical Page Pointers eliminate the overhead of logical-to-physical address translation in file-based designs. The numbers indicate the necessary navigation steps.

Physical Page Pointers. Under native storage [54], the DBMS has direct control over the physical storage and manages the logical-to-physical address mapping. However, NDP-executions also necessitate address information in-situ, for on-device address resolution and intervention-free execution. The propagation of this information incurs high synchronization overhead. For instance, the size of the page-level address-mapping can be as large as 1 GB for 1 TB of storage. To this end, we introduce *Physical Page Pointers* (PPP, Fig. 15.9) that complement parsers and accessors, such that any reference within the persistent dataset is based on a PPP. They are designed for append-based storage (e.g. with LSM-Trees [47, 52], or Partitioned B-Trees [60]), since persistent data is immutable and is only modified by DBMS-controlled storage maintenance (i.e. garbage collection, compaction).

PPPs eliminate the overhead of in-situ address resolution and address-mapping synchronization. The latter is still maintained, but only within the DBMS.

For example, the index block of an SST utilizes PPP parsers and accessors to refer to the data blocks (Fig. 15.9). In contrast, traditional DBMS mostly use files and offsets within them, which require on-device address-mapping for in-situ navigation. To process an SST file (Fig. 15.9), the DBMS extracts the address mappings for the file (1), loads (2) and processes (3) the index block (4). For each index block entry, the DBMS resolves (5) the address for the data block and (6) loads it.

15.3.5 *Software and Hardware-based NDP*

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

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

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

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

15.3.6 *NDP Pipelines and Operations*

Even though the actual operation execution is not the primary focus of this paper, we describe how the proposed architecture handles the execution of sequences of NDP operations. With computational

storage, we propose a *hybrid execution model*, combining pipelined block-at-a-time [56] and materialized execution strategies.

Inspired by [77], the operations in a demand-pull pipeline are split into operation execution groups. *Block-at-a-time (BaT)* execution [56] (formerly termed *vectorization* [77] – not to be confused with SIMD-vectorization) is achieved by embedding a buffering phase between any two execution groups. All the operators in a pipeline are connected through a record-at-a-time interface. The output of an operator within a group is passed on-the-fly to the next one, while the buffering stage, caches records internally until a buffer budget is reached. Once full, the next execution group can pull the buffered records over the same record-at-a-time interface. This mode leverages the device-internal memory hierarchy and heterogeneous processing elements, as the buffer stage can be placed in the device DRAM cache. Nonetheless, the buffer/cache budget remains a key limiting factor.

To this end, an operator can *materialize* intermediary or final results on the device (Fig. 15.10), and the next operator can operate on the materialized data (more details in Sect. 15.3.7). Local materialization allows: (a) the creation of complex NDP-pipelines, possibly with size-reducing operators at the end; (b) in-situ handling of non-size-reducing operations like joins or grouping, (c) the reduction of data transfers to host and more efficient DMA handling.

Currently, nKV supports the following NDP operations. *Get* retrieves the *value* for a given *key* and benefits from in-situ execution with low-latency [68, 69]. *Scan.* Both key and value filter-scans can benefit from parallel in-situ executions [68, 69]. Furthermore, our format parsers realize *projection*. Depending on the query, the query planner embeds it as an *early projection* [40] in the initial pipeline stages to reduce the size of the result set. Furthermore, we support *Joins* such as Block Nested Loop Join and Grace Hash Join that spill intermediate partition results to the computational storage. This is especially advantageous as joins are non-size-reducing. Finally nKV also supports hash table-based *GROUP BY* and *aggregation*.

Betweenness Centrality (BC) is a UDF used as an analytical HTAP operation. It performs a classical analysis on social graph data, and measures the degree to which nodes stand among each other. Our BC implementation in nKV is inspired by [16], and utilises the Node and Link tables of LinkBench [5] as graph representation. The logic, outlined in Algorithm [16, 68], sequentially scans the nodes with the NodeTableParser. In case the type of the node complies to the given search criteria, its neighbours are looked up via the LinkTableParser and distances are calculated recursively. Finally, the BC results are calculated according to the original algorithm in [16]. Overall, BC yields a random and sequential I/O mix.

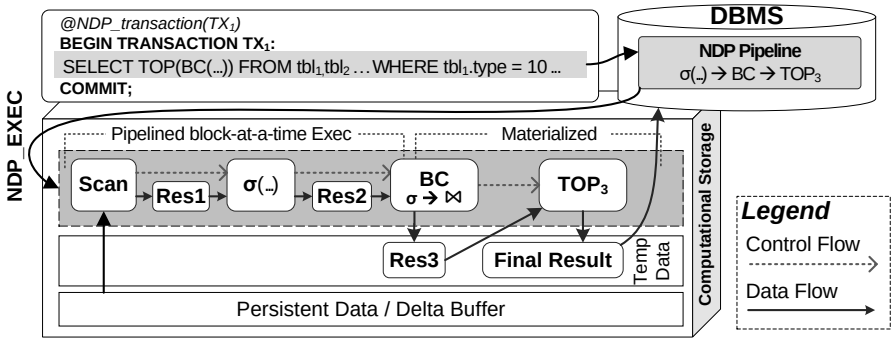


Figure 15.10: NDP-pipelines can be executed on the device, for faster processing or reduction of data transfers. Result set materialization is viable on computational storage.

15.3.7 *Result-Set Handling*

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

A key insight is that, computational storage offers fast local memory (BRAM, fast HBM or DRAM) as well as ample and cheap storage. Furthermore, a native storage DBMS can exclusively allocate and control on-device memory, allowing in-situ executions to materialize intermediary or final results there (see Sect. 15.3.7). The update-aware NDP architecture allows non-size-reducing operations, such as joins or grouping, to materialize their results in-situ to reduce data transfers (Fig. 15.10), while the next operator in a pipeline can operate on the materialized data.

Planning and execution. The planner estimates the upper bounds of the sizes of intermediary and final results along an NDP-pipeline. If the estimate exceeds the buffer stage memory (BRAM, HBM), the *BaT* execution group ends, a *materialization stage* is injected in the NDP-pipeline, and another execution group begins.

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

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

15.4 EXPERIMENTAL EVALUATION

Experimental Setup. The experiments are conducted on two different system stacks (Fig. 15.11). The first, *MyRocks over nKV*, is based on MyRocks with nKV [68, 69] as storage manager and is used if not mentioned otherwise. The host is running Debian 4.9 OS and is equipped with a 3.4 GHz clocked Intel i5 CPU and 4 GB RAM. The *COSMOS+* board [18] is attached over PCIe Gen 2.0 ×8 and comprises a Zynq 7045 SoC with an FPGA, two 667 MHz ARM A9 Cores, and an MLC Flash module configured as SLC. *COSMOS+* is roughly equivalent to a consumer NVMe SSD or smart storage device (e.g. Samsung SmartSSD [21]) in terms of price and resources. The concrete configuration depends on the evaluation stack. MyRocks (MySQL 5.6) is configured with `Repeatable Read` as `Serializable` is not supported. Unless mentioned otherwise, the memory footprint is set to 7.5% of the dataset size (incl. 400 MB block buffer), and the mutable *memtables* are configured to 32 MB.

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

Baselines. We evaluate update-aware *NDP* against two baselines (Fig. 15.11): the *block* and the *native stacks* under nKV and neoDBMS.

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

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

NDP. Both nKV and neoDBMS introduce the concept of *native NDP* and build on top of *native*. This allows offloading the OLAP processing to the device where most of the data is already located, while the OLTP workload fetches the required data to the host on-demand. In MyRocks over nKV, one ARM core of the *COSMOS+* exclusively handles foreground I/O, while the other one performs the NDP/OLAP execution. Thus, NDP execution on the *COSMOS+* is limited to a single execution at a time, while the host benefits from its 4 core CPU. On-device, 200 MB DRAM are reserved as a hashtable-based block buffer for reading pages that can be used by the OLAP operations. neoDBMS relies on 16 RISC-V [61] processors on the FPGA that are operated via the TaPaSCo framework [41].

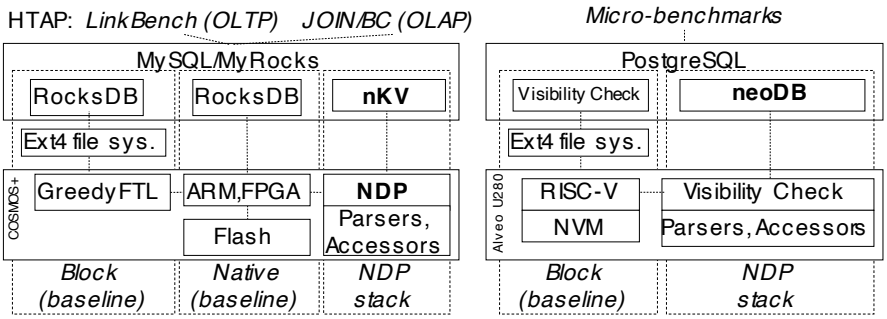


Figure 15.11: System Setup.

Workload. The workload is based on an HTAP-extended version of LinkBench [5] (if not specified otherwise). LinkBench [5] represents a social graph that is larger than the database memory. The graph is frequently updated by the OLTP-style transactional workload of LinkBench [5]. In addition, we introduce new analytical workload portions, performing graph analysis with either BC or `JOIN/GROUP BY` queries (see Sect. 15.3.6).

The initial dataset comprises a graph with 10M nodes and 20 GB of data. The workload is controlled by several parameters described below.

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

Experiment 1: Update-aware NDP enables transactionally consistent NDP executions of OLAP operations in presence of OLTP updates in HTAP systems, without performance drops. We open with a general experiment, demonstrating that with NDP as part of the HTAP

design space, analytical queries are executed without degrading the performance of the concurrent transactional workload, while analytical queries operate on the freshest data. To conduct this experiment, the HTAP workload is configured with $\text{OLAP}_{\text{PAUSE}} = 100\text{s}$ and $\text{OLTP}_{\text{SKEW}} = 40\%$.

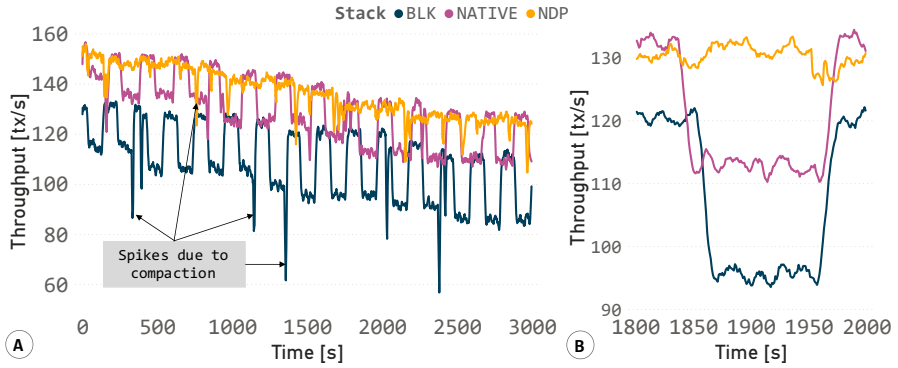


Figure 15.12: (A) LinkBench with HTAP extension is executed on the Block, Native, and NDP Stack. The throughput drops during OLAP queries due to increased I/O and the related buffer pollution. (B) Enlarged detail of one drop.

Figure 16.3.A shows the OLTP throughput over time for all stacks. The *native* and *block* baselines exhibit significant performance drops whenever an OLAP query is injected. These are due to the increased number of read I/Os, as the cold data for the OLAP execution must be fetched from storage. In contrast, no buffer misses occur in the NDP stack due to the in-situ OLAP execution (Fig. 16.3.B).

Several aspects need to be considered. Firstly, OLAP processing incurs significant buffer pollution, as hot OLTP working set pages are evicted to make room for cold data. Even after the completion of OLAP processing and a workload switch back to OLTP, it takes time for the buffer to recover and retain the hot OLTP working dataset in memory (Fig. 16.3.B). We investigate this effect in a further experiment by varying the buffer size (Fig. 15.13). Clearly, the larger the DBMS buffer, the longer the adjustment time upon a workload change. Secondly, *NDP* and *native* have higher throughput (tx/s) compared to the *block* stack baseline, due to the leaner I/O stack. Lastly, each stack exhibits regular and sharp performance drops. These relate to compactions and flushes of the LSM-tree, and explain the gradual performance degradation over time (Fig. 16.3.A). Overall, the OLTP throughput of *NDP* is 30% better than *block* in the HTAP phase, and 12% better than *block* during the OLTP phase.

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

Experiment 2: Update-aware NDP is intervention-free, yielding robust and resource-efficient performance. Now we investigate the

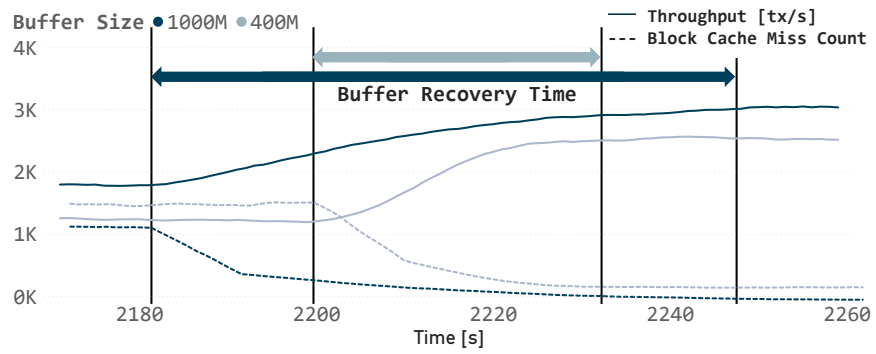


Figure 15.13: After an OLAP query, the cache misses (dashed) decrease as the buffer retains the hot dataset. The buffer recovery time (solid arrows) for the OLTP throughput (solid) to reach the original level depends on the buffer size.

hypothesis that with *intervention-free* NDP in HTAP settings, in-situ OLAP processing does not impact host-side OLTP processing, yielding better CPU utilization and robust performance. The experiment (Fig. 15.14) sets the HTAP phase so that the time between two successive OLAP requests is $\text{OLAP}_{\text{PAUSE}} = 1000\text{s}$. We report the host CPU utilization, for host-only HTAP (*native* baseline), and for *NDP* OLAP-execution with concurrent host OLTP.

We observe significant drops in CPU utilization (Fig. 15.14), during the OLAP phase under the *native* stack. These are due to CPU stalls, while waiting for I/O to fetch cold data from storage for host-only HTAP processing. With *NDP*, these drops are minimized, as OLAP processing is offloaded to computational storage, and the in-situ execution is asynchronous and intervention-free. Therefore, the free host CPU resources are utilized for concurrent OLTP processing, as the working OLTP dataset typically fits in memory. Moreover, *NDP* leverages storage device resources that would otherwise remain idle. In particular, we utilize both *COSMOS+* ARM cores, the FPGA, and exploit the full Flash parallelism. In addition, *intervention-free* NDP translates into robust transactional throughput, as shown in the pre-

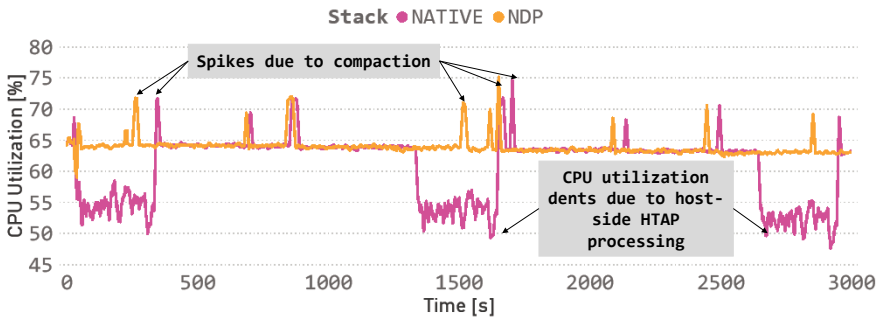


Figure 15.14: OLAP processing on the host, degrades CPU performance due to I/O wait time. NDP yields robust utilization of host resources, by leveraging on-device capabilities.

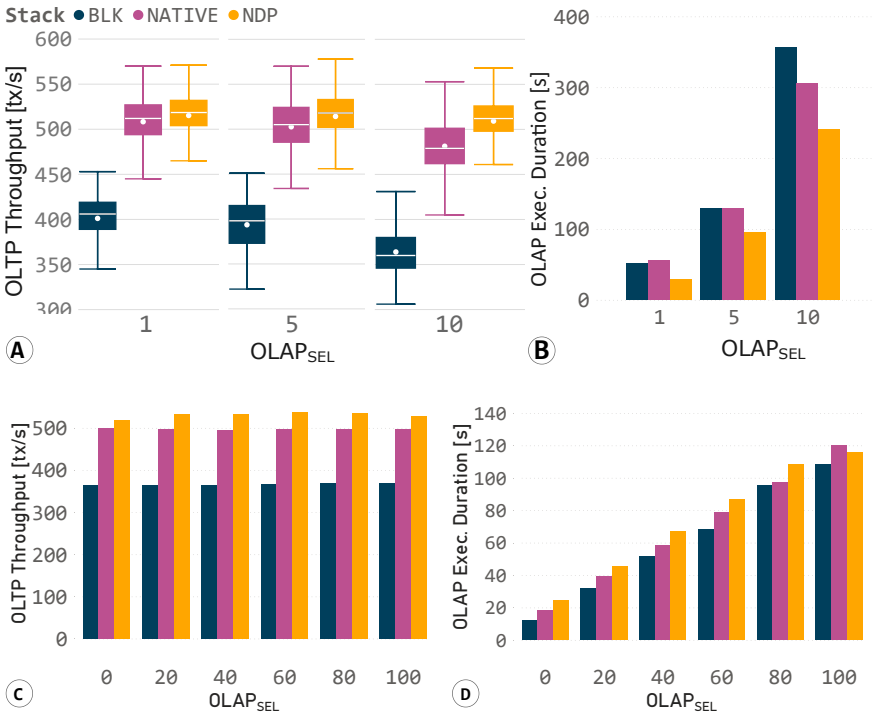


Figure 15.15: Executing BC as OLAP workload avoids dropping OLTP throughput (A), with increasing selectivities and OLAP runtimes (B). NDP outperforms native/block under OLTP, (C) although JOIN/GROUP BY queries are slower on-device (D).

vious experiment (Fig. 16.3). *Insight.* Intervention-free NDP frees up host resources, making them available to other tasks.

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

First, we consider `Selection/BC/TOP` to show how NDP dampens the effect of varying selectivity on OLAP executions. Notably, these are *size-reducing* operations. To this end, we vary OLAP_{SEL}, which determines the number of `NODE` table records that BC is processing. Thus, higher OLAP_{SEL} yields higher OLAP read-intensity and more data transfers, as well as more nodes to be processed and longer OLAP runtimes (Fig. 15.15.B). Given the HTAP workload, Figure 15.15.A shows the throughput of a frequent concurrent OLTP transaction `GetLinkList`, with varying OLAP_{SEL}. With increasing OLAP_{SEL} (Fig. 15.15.A), the average OLTP throughput decreases and its variance expands under the *block* and *native* stacks. This is due to the increasing OLAP duration, which causes more data transfers and a larger performance drop as observed in Experiment 1. *Insight.* With *NDP*, the throughput remains stable with varying OLAP_{SEL}.

Second, we consider `JOIN/GROUP BY/AGGREGATION` pipeline to show that NDP can handle *non size-reducing* operations, because of the hy-

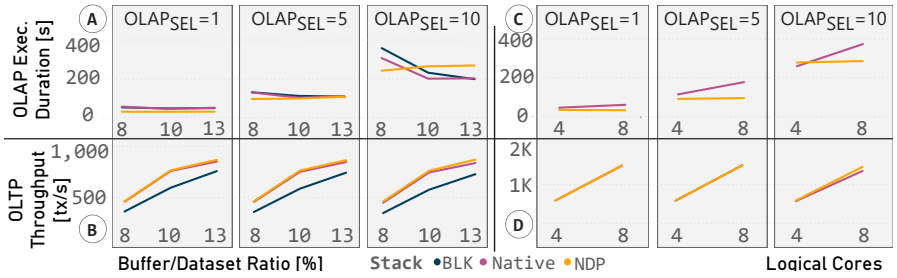


Figure 15.16: System performance behaviour with larger host memory footprints (A,B) and more logical cores (C,D).

brid execution model. We use the query: `SELECT n.type, SUM(c.count) FROM node n JOIN count c ON n.id = c.id WHERE n.type <= ? GROUP BY n.type;`. NDP and host plans resort to a BNLJ, while the on-device we resort to hash-based grouping, which does not spill to flash in this query. Again we vary the selectivity OLAP$_{SEL}$. Fig. 15.15.D shows increasing OLAP execution times with higher selectivities. Noticeably, NDP OLAP becomes compute-bound due to the slow on-device ARMs. Nonetheless, we achieve better overall HTAP performance due to intervention-free NDP.

Third, we vary the host resources for `Selection/BC/TOP`. In the first step, we increase the block buffer (Fig. 15.16.A/B), varying memory footprint from 8% (1.6 GB) to 13% (2.6 GB) of the dataset size. State-of-the-art approaches [19, 23, 46] aim at 10%. More host memory, yields better OLTP throughput (Fig. 15.16.A) and OLAP times (Fig. 15.16.B) under all stacks. With larger OLAP$_{SEL}$ and more memory, the OLAP gap between *BLK* and *NDP* shrinks, as larger memories shorten the OLAP performance drop length (Fig. 16.3) by reducing the buffer pollution.

Next, we attach COSMOS+ to another host with a more powerful CPU. More logical cores improve OLTP performance (Fig. 15.16.C), but entail higher buffer pollution that slows down OLAP queries on *native* (Fig. 15.16.D) due to increased buffer contention, while NDP OLAP remains unaffected due to intervention-free NDP. In fact, the higher the host parallelism, the higher the potential improvement through update-aware NDP, due to better relative OLAP execution times. NDP preserves its advantage, whenever both cores and memory are increased in a lockstep, since the buffer pollution caused by better OLTP (more cores) counters the positive OLAP impact (more memory).

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

Figure 16.5.A clearly shows that the *native* stack baseline outperforms the *block* under all settings. This is due to the leaner I/O stack, reducing the amount of data to be read and transferred to host, and due to the advanced native NVMe storage manager that reduces I/O latencies (Fig. 16.5.B/C). Yet, *NDP* improves OLAP runtime by 52% over *native* and 48% over *block* for a lower number of neighbours (5K, 10K). With more neighbours, the number of nearest neighbour searches within the analytical operation rises, as does the number of nodes to be revisited by the algorithm as well. This behaviour benefits vastly from large buffers, which is a major constraint on commodity computational storage devices, given the limited *COSMOS+* DRAM capacity. Thus, buffer misses on the device entail more Flash reads under *NDP* relative to *native* and *block* (see Fig. 16.5.B). Regardless of these limitations, Figure 16.5.C clearly indicates that the device-to-host data transfers can be reduced significantly.

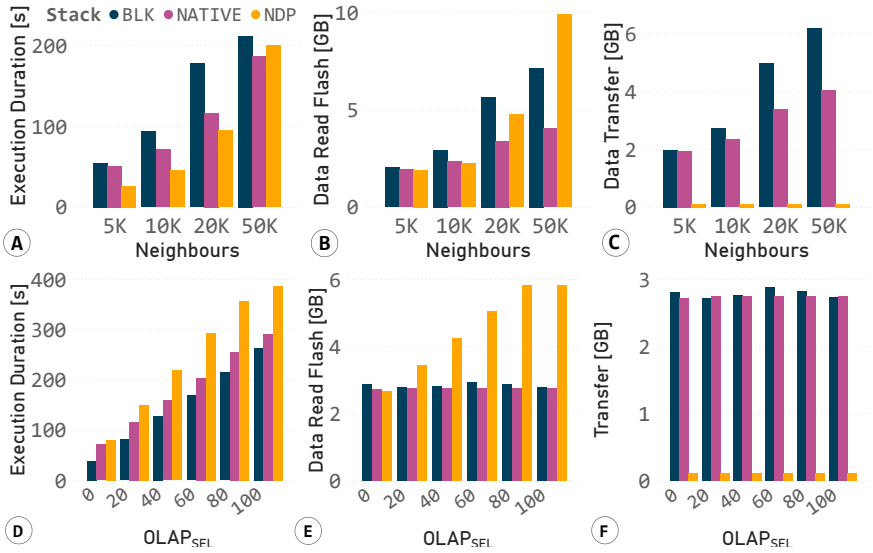


Figure 15.17: (A) Processing BC, Native and NDP outperform Block. (B) More neighbours yield more NDP I/O. (C) Yet, host-device data transfers are reduced significantly. Similar effects are visible with JOIN/GROUP BY/AGGR query (D,E,F).

Figure 16.5.D shows the execution time of the OLAP query from Exp. 3 under the same conditions. The low NDP performance is due to the NDP BNL-JOIN compute-boundness, but also due to its I/O intensity (Fig. 16.5.E), which grows as expected, due to the small on-device join buffer. Nonetheless, the whole NDP-pipeline is size-reducing, keeping the number host-transfers *low* (Fig. 16.5.F).

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

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

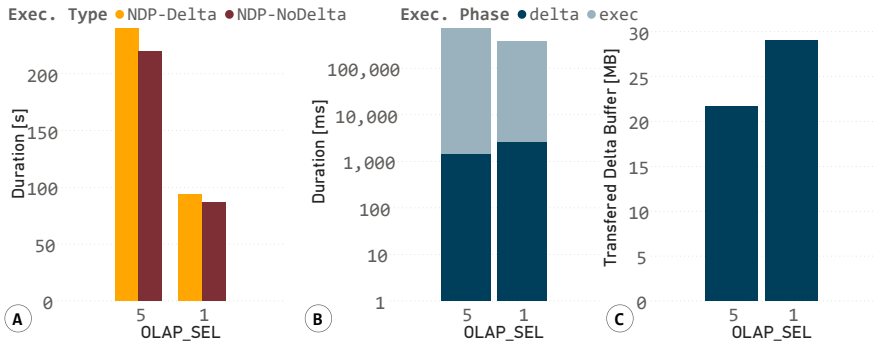


Figure 15.18: (A) NDP operates on fresh data with low overheads as (B) transfer times are low due to (C) small delta-buffer sizes.

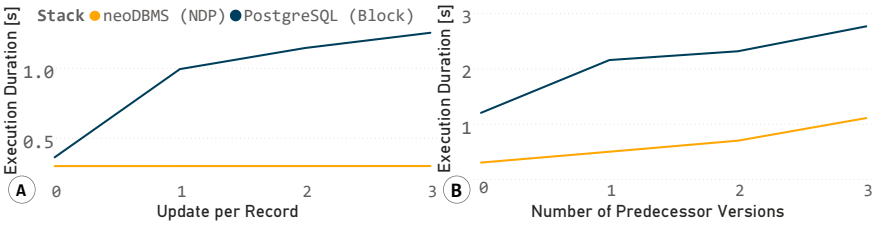


Figure 15.19: (A) The most recent tuple-version is retrieved with overhead due to N2O in neoDBMS. (B) Accessing predecessor versions is sped up by leveraging the on-device parallelism.

computational storage. We now quantify this overhead by executing the HTAP experiments with and without shared state transfers. We vary OLAP_{SEL} to achieve different shared state sizes. Figure 15.18.A shows the average OLAP execution duration in both settings. This execution time is broken down (Fig. 15.18.B) into shared state transfer time to device, and the subsequent processing time on a logarithmic scale. Figure 15.18.C shows the average size of the shared state for those breakdowns.

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

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

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

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

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

In a follow-up micro-experiment, we start each T_R, but leave it open, while all T_U commit, thus increasing the number of versions to four. Now compute `SUM(ol_amount)` for each reading transaction T_R (Fig. 15.19.B) in-situ and on the host. Figure 15.19.B shows the overhead of creating a snapshot at different points in time, and traversing the version chain to different predecessors. The in-situ snapshot creation time increases with the number of versions per tuple to be skipped. Nonetheless, the in-situ creation is 2× faster.

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

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

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

15.5 CONCLUSIONS AND FUTURE WORK

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

Future Work. Offloading modifying NDP operations is an important challenge for reducing data transfers. They require synchronisation and invalidation mechanisms for disaggregated memory environments for transactional consistency [11]. Furthermore, efficient NDP logging space management techniques are necessary.

REFERENCES

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In: *Proc. ASPLOS*. San Jose, California, USA, 1998. ISBN: 1-58113-107-0.

[2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. "H2O." In: *Proc. SIGMOD*. 2014, pp. 1103–1114. ISBN: 9781450323765. DOI: 10.1145/2588555.2610502. URL: http://15721.courses.cs.cmu.edu/spring2018/papers/10-storage/h2o.pdf%20http://dl.acm.org/citation.cfm?doid=2588555.2610502.

[3] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. "Tackling Hardware/Software co-design from a database perspective." In: *Proc. CIDR*. 2020.

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

[5] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. "LinkBench: A Database Benchmark Based on the Facebook Social Graph." In: *Proc. SIGMOD*. 2013. ISBN: 978-1-4503-2037-5.

[6] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. "Janus: A Hybrid Scalable Multi-Representation Cloud Datastore." In: *IEEE Trans. Knowl. Data Eng.* 30.4 (2018), pp. 689–702. ISSN: 10414347. DOI: 10.1109/TKDE.2017.2773607.

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. "Bridging the archipelago between row-stores and column-stores for hybrid workloads." In: *Proc. SIGMOD*. Vol. 26-June-20. 2016, pp. 583–598. ISBN: 9781450335317. DOI: 10.1145/2882903.2915231. URL: http://dx.doi.org/10.1145/2882903.2915231.

[8] Oreoluwatomiwa O. Babarinsa and Stratos Idreos. "JAFAR : Near-Data Processing for Databases." In: 2015.

[9] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. "BlockNDP: Block-storage near data processing." In: *Proc. Middlew.* 2020, pp. 8–15. ISBN: 9781450382014. DOI: 10.1145/3429357.3430519. URL: https://doi.org/10.1145/3429357.3430519.

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

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

[12] Philip A. Bernstein and Nathan Goodman. "Concurrency Control in Distributed Database Systems." In: *ACM Comput. Surv.* 13.2 (June 1981), pp. 185–221.

[13] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem." In: 2017.

[14] Haran Boral and David J. DeWitt. "Parallel Architectures for Database Systems." In: *Database Machines*. Ed. by A. R. Hurson, L. L. Miller, and S. H. Pakzad. Springer Berlin Heidelberg, 1989. Chap. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, pp. 11–28. ISBN: 0-8186-8838-6.

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

[16] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality." In: *Journal of Mathematical Sociology* (2001).

[17] Wei Cao et al. "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database." In: *Proc. FAST*. 2020, pp. 29–41.

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

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

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

[21] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. "Query processing on smart SSDs." In: *Proc. SIGMOD* (2013), p. 1221. ISSN: 07308078. DOI: 10.1145/2463676.2465295. URL: http://dl.acm.org/citation.cfm?doid=2463676.2465295.

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

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

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

[25] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview." In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33. URL: http://dblp.uni-trier.de/db/journals/debu/debu35.html%7B%5C#%7DFarberMLGMRD12%7B%5C%%7D5Cnhttp://sites.computer.org/debull/A12mar/issue1.htm.

[26] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. "Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads." In: *Proc. VLDB Endow.* Vol. 8. 12. 2015, pp. 1716–1727. DOI: 10.14778/2824032.2824069.

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

[28] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. "HYRISE-A main memory hybrid storage engine." In: *Proc. VLDB Endow.* 4.2 (2010), pp. 105–116. ISSN: 21508097. DOI: 10.14778/1921071.1921077.

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

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

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

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

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

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

[35] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *Proc. ICDE*. 2011, pp. 195–206. ISBN: 9781424489589. DOI: 10.1109/ICDE.2011.5767867.

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

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

[38] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. "In-storage processing of database scans and joins." In: *Inf. Sci.* 327 (Jan. 2016), pp. 183–200. ISSN: 00200255. DOI: 10.1016/j.ins.2015.07.056. URL: http://dx.doi.org/10.1016/j.ins.2015.07.056%20https://linkinghub.elsevier.com/retrieve/pii/S0020025515006003.

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

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

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

[42] Tirthankar Lahiri et al. "Oracle Database In-Memory: A dual format in-memory database." In: *Proc. - Int. Conf. Data Eng.* 2015-May (2015), pp. 1253–1258. ISSN: 10844627. DOI: 10.1109/ICDE.2015.7113373.

[43] Per Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. "Real-time analytical processing with SQL server." In: *Proc. VLDB Endow.* Vol. 8. 12. 2015, pp. 1740–1751. DOI: 10.14778/2824032.2824071.

[44] Juchang Lee, Wook Shin Han, Hyoung Jun Na, Chang Gyoo Park, Kyu Hwan Kim, Deok Hoe Kim, Joo Yeon Lee, Sang Kyun Cha, and Seung Hyun Moon. "Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases." In: *VLDB J.* 27.3 (2018), pp. 421–444. ISSN: 0949877X. DOI: 10.1007/s00778-018-0503-z.

[45] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. "Disaggregated Data Centers: Challenges and Trade-offs." In: *IEEE Communications Magazine* 58.2 (2020), pp. 20–26. DOI: 10.1109/MCOM.001.1900612.

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

[47] Chen Luo and Michael J. Carey. "LSM-based storage techniques: a survey." In: *The VLDB Journal* 29.1 (2020), pp. 393–418.

[48] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. "BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications." In: *Proc. SIGMOD*. Vol. Part F1277. 2017, pp. 37–50. ISBN: 9781450341974. DOI: 10.1145/3035918.3035959. URL: http://dx.doi.org/10.1145/3035918.3035959.

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

[50] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "ScyPer: a hybrid OLTP&OLAP distributed main memory database system for scalable real-time analytics." In: *Datenbanksysteme für Business, Technologie und Web (BTW) 2044*. Ed. by Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 499–502.

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

[52] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)." In: *Acta Inform.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903.

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

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

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

[56] Orestis Polychroniou and Kenneth A. Ross. "Towards Practical Vectorized Analytical Query Engines." In: DaMoN'19. 2019.

[57] Vijayshankar Raman, Gopi Attaluri, and Ronald Barber. "DB2 with BLU Acceleration: So much more than just a column store." In: *Proc. VLDB* 6.11 (2013), pp. 1080–1091. ISSN: 2150-8097. DOI: 10.14778/2536222.2536233. URL: https://researcher.watson.ibm.com/researcher/files/us-ipandis/vldb13db2blu.pdf%20http://dl.acm.org/citation.cfm?id=2536233.

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

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

[60] Christian Riegger, Tobias Vinçon, Robert Gottstein, and Ilia Petrov. "MV-PBT: Multi-version indexing for large datasets and HTap workloads." In: *Adv. Database Technol. - EDBT*. Vol. 2020-March. 2020, pp. 217–228. ISBN: 9783893180837.

[61] *RISC-V*. https://riscv.org/.

[62] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. "L-Store: A real-time OLTP and OLAP system." In: *Proc. EDBT*. Vol. 2018-March. 2018, pp. 540–551. ISBN: 9783893180783. DOI: 10.5441/002/edbt.2018.65. arXiv: 1601.04084.

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

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

[65] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth." In: *Proc. SIGMOD*. Scottsdale, Arizona, USA, 2012. ISBN: 978-1-4503-1247-9.

[66] *SPDK*. https://spdk.io.

[67] 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.

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

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

[70] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. "Building An Elastic Query Engine on Disaggregated Storage." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 449–462. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/vuppalapati.

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

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

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

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

[75] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An Empirical Evaluation of In-memory Multi-version Concurrency Control." In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. ISSN: 2150-8097.

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

[77] Jingren Zhou and Kenneth A. Ross. "Buffering Database Operations for Enhanced Instruction Cache Performance." In: *Proc. SIGMOD 2004*. SIGMOD '04. Paris, France, 2004, pp. 191–202. ISBN: 1581138598.

Part V

NDP EXECUTION AND RESULT-SET MANAGEMENT

16

RESULT-SET MANAGEMENT FOR NDP OPERATIONS ON SMART STORAGE

BIBLIOGRAPHIC INFORMATION

The content of this chapter has previously been published in the work *"Result-Set Management for NDP Operations on Smart Storage"* by Tobias Vinçon, Christian Knödler, Arthur Bernhard, Leonardo Solis-Vasquez, Lukas Weber, Andreas Koch and Ilia Petrov in 2022 18th International Workshop on Data Management on New Hardware (DaMoN). The contribution of the author of this thesis is summarized as follows.

> *» As the corresponding and leading author, Tobias Vinçon was in charge of designing the concepts for near-data processing result-set management. Ilia Petrov supported him in structuring and embedding these concepts. Moreover, Tobias Vinçon integrated the concepts in nKV and provided the testbed with COSMOS+ as real hardware. Likewise, the manuscript's text was created by him with valuable feedback from all authors including Christian Knödler, Arthur Bernhard, Leonardo Solis-Vasquez, Lukas Werber and Andreas Koch. «*

ABSTRACT

Current data-intensive systems suffer from scalability as they transfer massive amounts of data to the host DBMS to process it there. Novel near-data processing (NDP) DBMS architectures and smart storage can provably reduce the impact of raw data movement. However, transferring the result-set of an NDP operation may increase the data movement, and thus, the performance overhead. In this paper, we introduce a set of *in-situ* NDP result-set management techniques, such as *spilling*, *materialization*, and *reuse*. Our evaluation indicates a performance improvement of 1.13× to 400×.

16.1 INTRODUCTION

Regardless of the increasing data sizes and the evolution of storage technology, modern DBMS employ traditional *data-to-code* architectures. They require growing amounts of data to be transferred to the DBMS host to be filtered and processed there. *Data movement* turns into a performance and scalability limitation, as it consumes scarce bandwidth and increases resource and energy consumption. The advent of *intelligent storage* and disaggregated memory enables Near-Data

Processing (NDP) architectures and *code-to-data* paradigms that target execution of DB-operations close to where data is physically stored. To this end, NDP can leverage the higher device-internal bandwidth, parallelism and especially faster storage for data processing and filtering. Yet, not only raw-data movement impairs performance. Since there are different types of NDP operations (e.g., size-reducing but also non-size-reducing ones) and different execution modes, result-set management for NDP operations looms as an important factor.

The **core intuition** of this paper is that NDP necessitates result-set management techniques. This observation is governed by the following trends. Firstly, modern intelligent storage is capable of managing result-sets, both intermediary and final. Different NDP operations need it due to their potentially non-size-reducing nature or their execution mode. Secondly, storage (like Flash or NVM) is cheap and abundant as these technologies offer high density. Lastly, access to in-situ storage is much faster than that of device-to-host in terms of both bandwidth and latency.

State-of-the-art overview. NDP approaches [2–4, 10, 11] establish the following principles. Firstly, pioneered by IBEX [10, 11], smart storage devices support either *tuple-* or *block-based* access. The former is typically used for the result tuples of an NDP operation (*Tuple-based NDP* or *Blocks of Tuples NDP*). Thus, the result transfer units contain only *fully-qualifying* tuples (Figure 16.1). The latter is employed for foreground I/O, i.e., any read/write operation accessing raw blocks (*Block I/O*). Intel's Block-NDP [2] improves the latter by allowing an NDP operation to only return raw blocks containing *partially-qualifying* filter criteria (*Block-level NDP.*). Both are sub-optimal due to the large transfer overheads (Figure 16.1).

Secondly, qualifying tuples or blocks are transferred up to the host *immediately*, i.e., as soon as they are produced. Depending on either the selectivity or the NDP operation itself, the *immediate* result-set transfer mode may cause significant overhead. Furthermore, it may preclude employing optimizations such as a single large low-overhead DMA transfer, utilizing the full I/O bandwidth. The immediate host transfer precludes a *reuse* of results on the device, whether by a follow-up NDP operation or by the host itself.

In a nutshell, while current tuple-based approaches reduce the overall volume by transferring the precise result tuples, they may not attain the best performance due to transfer overhead and low bandwidth utilization. Conversely, block-based approaches may utilize the full I/O bandwidth, but incur a performance penalty by transferring more data. Noticeably, none of them allows for reuse.

The **contributions** of this paper include the following. (a) We introduce *in-situ result-set materialization* that enables combining arbitrary NDP operations into *NDP pipelines*. NDP pipelines that reduce the overall data transfer to the host even though they may contain non-

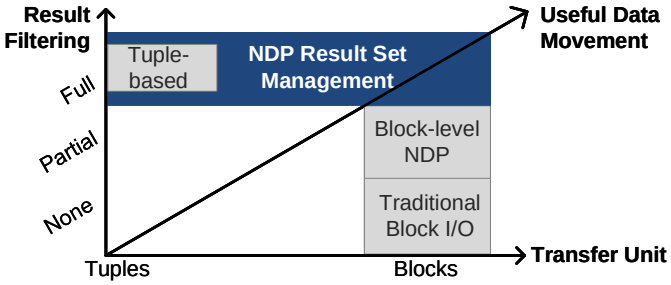


Figure 16.1: State-of-the-art approaches prioritize for result filtering or I/O throughput. Native NDP Result Handling combines those dimensions and improves overall performance.

size-reducing operations. (b) We also introduce on-device *spilling* of data to persistent storage (e.g., Flash), by which NDP operations are viable even on resource-constrained intelligent storage devices (e.g., especially in memory). (c) Furthermore, we introduce the reuse of results materialized in-situ in further processing without significant overhead. Additionally, this reuse enables fault tolerance e.g., in complex pipelines.
Next, we discuss in-situ result-set materialization (Sect. 16.2) and the system design (Sect. 16.3). The performance is evaluated in Sect. 16.4.

16.2 IN-SITU MATERIALIZATION

Prior mentioned approaches treat results only as *transient* data, while their consumption happens *immediately* after their generation. Our approach introduces the ability to *(fully) materialize* them, as well as to consume them in *a deferred* manner. It processes the results later on (see *Consumption Mode*) or even reuses them multiple times (see *Reuse Semantics*). In general, materialization can be achieved for both, final and intermediary results. This also requires space on the computational storage device, which is abundant and cheap. Space allocation is performed for each NDP invocation by the Native Storage Manager [6] of the database system.
Consumption Mode. NDP pipelines necessitate different result consumption modes. As shown in Figure 16.2.A, *NDP_Pipeline #1*, TX_1 is annotated with an *immediate consumption*. Hence, it treats the operation's input data and its result as *transient*, and relies on pipelining. Given an immediate consumption, the final results are transferred back to the host (Figure 16.2.D) as soon as a result unit (e.g., a result tuple or a block of result tuples) is produced.

In this paper we introduce two additional alternatives. Firstly, we allow for *in-situ materialization* of intermediary results for either follow-up NDP operations or upcoming NDP pipelines (Figure 16.2.B). The latter can be issued completely asynchronously. Secondly, we allow for result-*spilling* (Figure 16.2.C). It is applicable to operators such

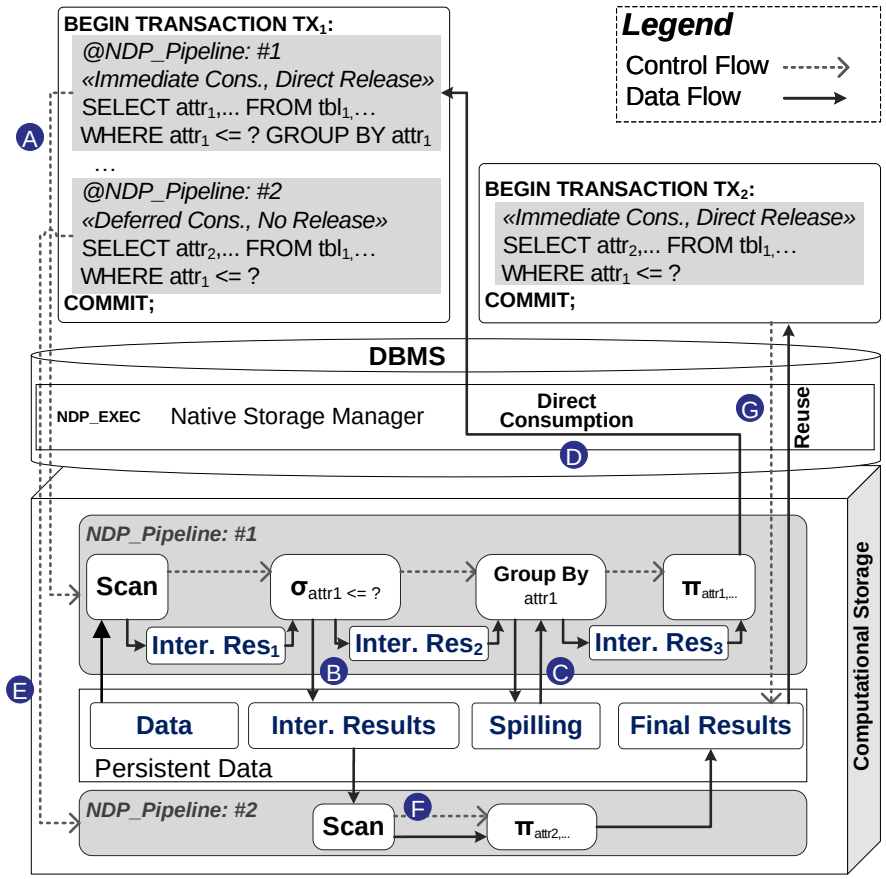


Figure 16.2: NDP Pipelines can materialize intermediate and final results in-situ, e.g., for reuse in further processings.

as a hashtable-based `GROUP BY` or an `HASH JOIN` implementations that exceed on-device memory limits. These limits can be easily reached, especially as consumer-grade NDP devices have constrained hardware resources.

With materialization in place, NDP pipelines can also be instrumented with a *deferred consumption* mode as depicted in Figure 16.2.E, *NDP_Pipeline #2*. Thereby, the final results are not transferred back to the host immediately, but rather stored on the persistent storage for consumption at later on by the host or another NDP pipeline.

Parsers and Accessors. The native NDP approach in nKV [7, 8] is based on the concept of in-situ data interpretation. To this end, NDP *parsers* and *accessors* have been proposed [7, 9] to handle data from the base tables. However, database operations in an NDP pipeline typically consume intermediary results from previous stages, for which no suitable parsers and accessors exist. Consider for example, Figure 16.2.F, where the *scan* in *NDP_Pipeline #2* can be optimized to consume the intermediary results from *NDP_Pipeline #1*. To handle the interpretation of intermediary results, we extended the parsers and accessors [7, 9] to cope with record formats of intermediary results and interpret them on-device to avoid data movement.

Reuse Semantics. Whenever a result (intermediary or final) is materialized, its data is available for consumption until its address space is released. Hence, multiple queries can reuse the data by either consuming it from the computational storage device (Figure 16.2.G) or processing (Figure 16.2.F). By releasing the data, their allocated storage location is flagged for garbage collection and will be erased with its next execution.

Space management, allocation and planning. nKV [7, 8] is based on the concept of native storage [6]. In essence, native storage [6] mandates that the DBMS operates directly on the physical storage avoiding intermediary layers, i.e., a file system or on-device translation layers. As a result, functionality like address mappings or garbage collection is deeply integrated in nKV.

Planning and allocation. The *planner* estimates the upper bounds of the sizes of intermediary and final results along an NDP-pipeline. If the estimate exceeds a predefined buffer size, then a *materialization* or *spilling* stage is injected in the NDP-pipeline.

Depending on the size estimation, the planner and the storage manager employ an allocation strategy that targets fast levels of the on-device memory hierarchy first, e.g., on-device DRAM. If insufficient, a materialization and spilling to persistent storage is planned. In this sense, every materialization stage is assigned an exclusive physical address range by the native storage manager as the DBMS controls the address mapping. If the space proves insufficient, the execution stalls and the computational storage requests more space from the DBMS in an extra host-roundtrip.

Space management and garbage collection. nKV controls storage directly, manages logical-to-physical address mapping, and schedules the garbage collection (GC). It allocates and exclusively assigns physical address ranges to each pipeline and its materialization or spilling stages. Thus, nKV ensures that other transactions, pipelines or NDP operations do not overlap in the same storage space. nKV preserves these address ranges for the duration of the execution until the completion of the invoking transaction or the reuse phase. Only then, nKV's storage manager marks them for GC and performs an asynchronous GC call, which is executed as an NDP operation.

16.3 SYSTEM DESIGN

To investigate the previously described aspects of result-set management and in-situ materialization, we integrated those concepts into MyRocks. As storage manager, we use nKV [7, 8], an NDP-capable KV-Store based on RocksDB that already supports a native storage interface towards computational storage devices. Moreover, we define a communication protocol on top of NVMe to enable host-device interactions, while several interconnected and distributed state machines facilitate NDP processing on-device.

Communication Protocol. Our proposed communication between the host and device is kept lean to avoid any unnecessary data transfers and host-device roundtrips. Prior to any processing, the device must allocate sufficient resources for the planned command. Therefore, the host can reserve an NDP Slot on-, which is then assigned to a given processing id (PID). Subsequently, an NDP invocation is performed. It includes pre-allocated physical pages for either in-situ materialization or spilling, as well as a monotonically increasing host interaction id (HI), ensuring a total order of all upcoming interactions. From this point onward, the processing will be fully managed by the device itself and executed without any intervention with the database engine. Whenever a block of tuples (as the final result-set) exceeds its limits, the associated command is returned with the respective results as payload. Upon that, the host can repetitively issue further commands until all results are retrieved and the NDP pipeline reached its completion. The NDP Slot is automatically returned afterwards. Upon an error during processing or in the event of insufficient resources (e.g., pre-allocated physical pages), the command is returned with a status field indicating the cause. As described in Sect. 16.2, the native storage manager resolves it, by scheduling GC or by allocating further pages and issuing a follow-up commands with the respective action, i.e., by passing new free page addresses to the device.

On-Device State Machines. In general, the processing elements, e.g., cores, on the intelligent storage device can be subdivided in a single *managing core* and multiple *processing cores*. Thereby, several state

machines, interconnected via a shared memory, run simultaneously on each core to perform certain functionalities. The managing core runs the NVMe Engine and interacts with the host via the previously described protocol. The NDP Engine is responsible for allocating the NDP Slot, transferring either information or extracting results from the HIs. Its counterpart on the processing cores continuously polls for new HIs of the NDP Slot before executing the NDP pipeline. During execution, persistent data is requested via flash reads towards the flash engine located on the managing core. Result tuples are placed into blocks of the respective HI before they are returned to the host. Thus, the NDP engine can continue running on the processing cores, while existing results are transmitted up to the host in parallel by the managing core. This way interleaved pipelining is achieved.

16.4 EXPERIMENTAL EVALUATION

Experimental Setup. We use MyRocks (MySQL 5.6) with nKV [7, 8] as storage manager. The *COSMOS+* board [5] is employed as an NDP-capable storage device and rough equivalent to a consumer-grade NVMe SSD or smart storage device. It comprises a Zynq 4045 SoC with an FPGA, two 667 MHz ARM A9 cores, and Flash storage. The board is connected via PCIe 2.0 ×8 to a host with a 3.4 GHz clocked Intel i5 CPU and 4 GB of RAM, running Debian 4.9. The maximum transfer size per NVMe DMA request is limited to 1 MB, due to the NVMe engine of *COSMOS+*. Therefore, this is also our largest result transfer unit.

As a baseline for the evaluation, we use nKV with native storage, but without NDP (*Native*). It eliminates file system and block-device layers, and allows for leveraging the physical properties of the underlying storage with native storage management [6–8]. The results are compared to the *NDP* configuration which utilizes one ARM core exclusively, as managing core and performing host-device communication and interacting with the flash controller. The other ARM core is used as processing core and is dedicated to NDP pipeline processing. It uses on-device 200 MB DRAM as block buffer and 32 kB intermediary result buffers between pipeline stages if not mentioned otherwise. MyRocks is configured to have a memory footprint of around 10% of the data set size, including a block buffer of 1.4 GB. As dataset we utilize LinkBench [1] configured with 10M Nodes and 20 GB of data. Queries are always issued after a cold start to avoid measuring unintended effects from caching and to ensure consistent results.

Experiment 1: Efficient NDP result handling can reduce not only the data to be transferred from device to host, but also improve execution duration by batching multiple results in larger transfer units. In our first experiment, we investigate the influence of the result transfer granularity on the execution duration. To this end, we em-

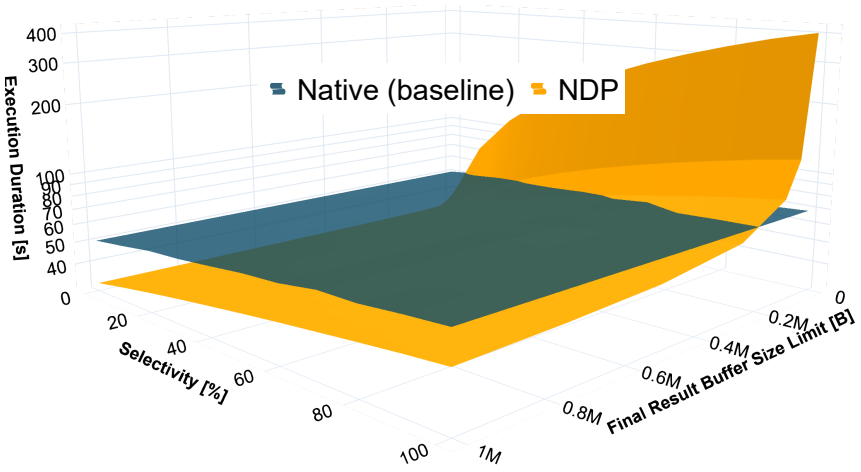


Figure 16.3: Traditional Block I/O cannot filter data on-device (blue) in contrast to NDP. Yet, transferring results in small granularities (e.g., tuple-based) entails high communication overhead (yellow). NDP wins after the intersection.

ploy a simple selection-projection query: `SELECT id, type,...FROM nodetable WHERE type <= ?`. We vary the selectivity to increase the result-set size and the amount of data to be transferred. As a baseline, we report the execution time for classical block-based I/O with the *Native* stack (Figure 16.3, blue). By using the NDP stack (Figure 16.3, yellow), we continuously increase the granularity of the data transfer unit from 1 kB (simulating tuple-based) to the limit of 1 MB (blocks of tuples).

NDP reduces the device-to-host data transfers to the final results of the given query. The only remaining cost is for reading and filtering the data, as shown with 0% selectivity. With higher selectivities, the amount of data to be transferred increases. Furthermore, the transfer granularity entails an overhead of handshakes between host and device in the PCIe/NVMe communication. Thus, with *COSMOS+*, the best execution duration is obtained by transferring large blocks of tuples, improving the performance by up to 27%.

Insights. NDP result management is capable to adapt to and optimize for the given underlying storage link technology. Therefore, it is necessary to adjust the granularity of transfer units accordingly by either sending tuple-per-tuple or by batching multiple result tuples into blocks of specific sizes.

Experiment 2: Concurrent execution of processing and final result transfers as pipeline stages improves performance. In this experiment, we investigate the impact of interleaved pipelining and transfer granularity on the execution duration (Figure 16.4.a). We execute the query from Experiment 1 on the *NDP* stack with and without interleaved pipelining and vary the granularity of transfer units. In general, the performance with interleaved pipelining is significantly faster than processing and transferring results in a sequential order. Particularly,

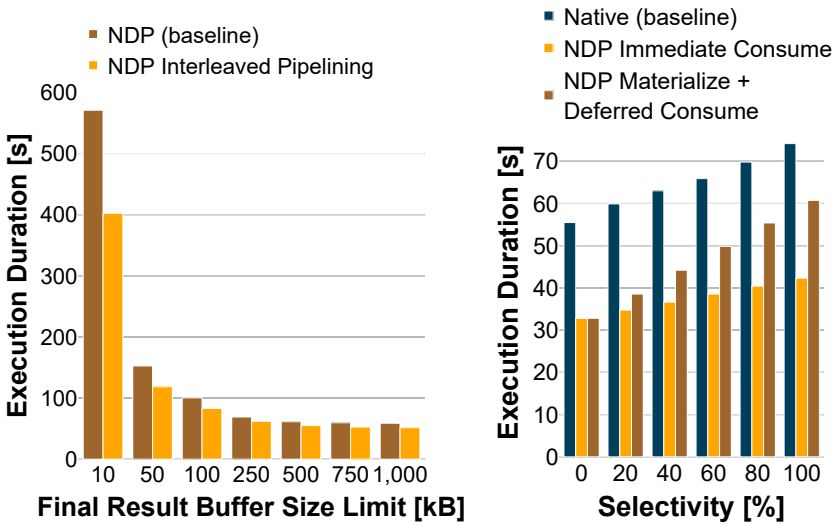


Figure 16.4: a) On-device interleaved pipelining improves execution significantly. b) Final result materialization and deferred consumption (brown) entail a small overhead over immediate consumption (yellow) and outperform the baseline (blue).

small transfer granularities, entailing a high communication overhead, benefit from interleaved pipelining, shortening execution durations by up to 30%. Yet, the largest possible transfer unit (1 MB) improves performance by 13%.

Insights. Interleaved pipelining enables result-set transfers while further processing is executed concurrently. Thereby, it efficiently conceals the communication overhead entailed by smaller transfer units, benefits larger transfer blocks, and shortens host processing delays. Other approaches are bound to the standard block granularity, while nKV can vary it.

Experiment 3: Result materialization can be achieved without a significant execution runtime overhead in NDP pipelines. Next, we investigate the costs of materializing final results of an NDP pipeline in Figure 16.4.b. Again, we execute the query of Experiment 1 on the *Native* (blue) and the *NDP* (yellow) stack without materialization as baselines. The same query is repeated as NDP pipeline that materializes its final results on Flash and immediately retrieves those via classical I/O. We vary the selectivity to determine the impact of the final result size on the materialization cost. Executing the query as NDP pipelines outperforms the *Native* baseline by up to 40%, despite materialization, even for higher selectivities. In fact, materialization costs largely depend on the final result-set size, and thus, add up 4% to 20% on the original execution time. However, this increase also includes the final result-set retrieval from Flash.

Insights. NDP pipelines allow to materialize their results on device without high execution overheads. Thereby, the cost for materialization increases with higher selectivities and result sizes, while still outperforming the *Native* baseline.

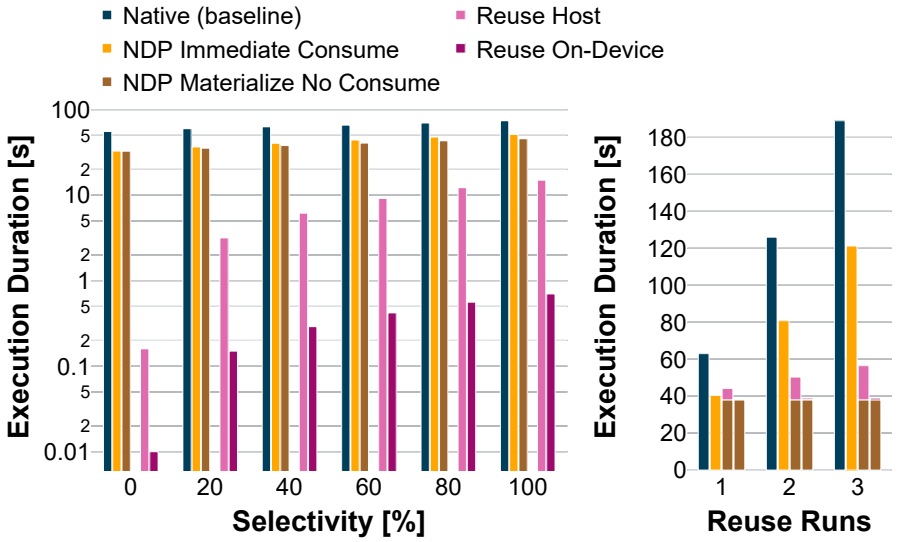


Figure 16.5: Reuse of materialized results improves the host (magenta) and on-device (dark red) performance significantly.

Experiment 4: The reuse of in-situ materialized results has marginal costs and amortizes those materialization costs already after the second consumption. Last but not least, we investigate the reuse of in-situ materialized results. In particular, we focus on the materialization of Experiment 3 and extend it with the costs of NDP result materialization *without* consumption (brown), result *reuse* on the host (magenta), and NDP result *reuse* on-device (red) as shown in Figure 16.5 on a logarithmic scale.

Since NDP result materialization without consumption (brown) does not require retrieving the result data after persisting it to Flash, it shortens the execution duration by up to 12% compared to *NDP* immediate consumption (yellow), and by up to 45% compared to *Native*, depending on the selectivity and the respective result-set size. However, consuming it in a deferred manner will add up the costs for either *reuse on host* or *reuse on-device*, and thus, will be marginally slower than NDP immediate consumption, while still outperforming the *Native* baseline. However, the full potential of reusing materialized data develops by the second execution (Figure 16.5 right). While *reuse on host* has significantly lower duration (up to 95%, compared to NDP immediate consumption), *reuse on-device* can speed up the consumption even further by 73× to 400× over NDP immediate consumption, since reading previously-filtered data leverages the full Flash parallelism of *COSMOS+*. This is especially useful for iterative (e.g., k-means) or follow-up NDP operations.

Insights. NDP pipelines enable an efficient and flexible result materialization. They can be consumed either immediately or deferred. Moreover, the materialized data can be reused multiple times on the host but also on-device with significantly lower execution times.

REFERENCES

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. "LinkBench: A Database Benchmark Based on the Facebook Social Graph." In: *Proc. SIGMOD*. 2013. ISBN: 978-1-4503-2037-5.

[2] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. "BlockNDP: Block-storage near data processing." In: *Proc. Middlew.* 2020, pp. 8–15. ISBN: 9781450382014. DOI: 10.1145/3429357.3430519. URL: https://doi.org/10.1145/3429357.3430519.

[3] Antonio Barbalace and Jaeyoung Do. "Computational Storage: Where Are We Today?" In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[4] Wei Cao et al. "POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database." In: *Proc. FAST*. 2020, pp. 29–41.

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

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

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

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

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

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

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