Result-Set Management for NDP Operations on Smart Storage [Extended Version]

Tobias Vinçon*, Christian Knödler*, Arthur Bernhardt*, Leonardo Solis-Vasquez†, Lukas Weber†, Andreas Koch†, Ilia Petrov*
†Technische Universität Darmstadt, Embedded Systems and Applications Group,
*Reutlingen University, Data Management Lab

ABSTRACT

Current data-intensive systems suffer 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×.

ACM Reference Format:
Tobias Vinçon*, Christian Knödler*, Arthur Bernhardt*, Leonardo Solis-Vasquez†, Lukas Weber†, Andreas Koch†, Ilia Petrov*. 2022. Result-Set Management for NDP Operations on Smart Storage [Extended Version]. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 appear 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 raw movement data impairs performance. As 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, in-situ storage access is much faster in terms of both bandwidth and latency, compared to device-to-host.

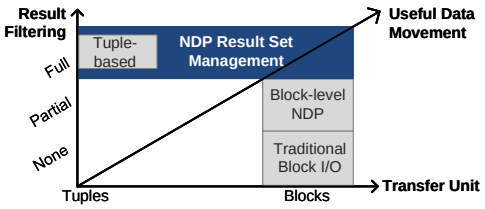

Figure 1: State-of-the-art approaches prioritize for result filtering or I/O throughput. Native NDP Result Handling combines those dimensions and improves overall performance.

State-of-the-art overview. NDP approaches [4, 5, 7, 18, 19] establish the following principles. Firstly, pioneered by IBEX [18, 19], smart storage devices support either *tuple-* or *block-based* access. The former is typically used for the result tuples of an NDP operation. Thus the result transfer units contain only *fully-qualifying* tuples (Fig. 1). The latter is employed for foreground I/O, i.e., any foreground read/write operation accessing raw blocks. Intel's Block-NDP [4] improves the latter by allowing an NDP operation to return, while raw blocks contain *partially-qualifying* data filtering out the rest (Fig. 1). Both are sub-optimal due to the large transfer overheads.

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 those results on the device, whether by a follow-up NDP operation or by the host itself. The latter is very favorable for analytical and data science operations, e.g., k-means with different number of centroids or iterations.

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.

NDP Result-Set Management. In this paper we introduce various result-set management techniques for NDP operations executing on-device. Such in-situ result handling techniques are applicable

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

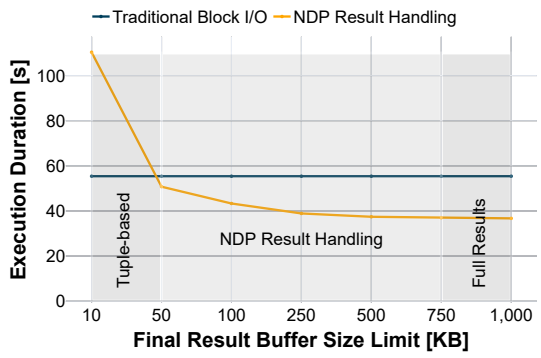

Figure 2: NDP reduces the raw data transferred from device to host, yet its result management impacts performance (blue vs. yellow). With bandwidth-optimized systems, best results are achieved with large densely-packed transfer units (yellow).

to different kinds of NDP operations and yield a *overall reduction of data movement*. These techniques favor not only obvious selections and early projections that are size-reducing, but also `JOIN` and `GROUP BY` that both are I/O intensive and non-size-reducing. We introduce *spilling* techniques for various operators and *local, in-situ materialization* of their results. Such materialization allows for efficient data transfer up to the host, e.g., in a single low-overhead DMA transfer, minimizing the cost of data movement. Furthermore, in-situ materialization enables the *reuse* of the result-set locally for the next NDP operation (yielding hybrid NDP execution models) or for a follow-up operation to be executed later on. Last but not least, they enable *fault-tolerant* NDP executions as individual NDP operations within a pipeline can be seamlessly re-executed upon a failure as their input is persistent and available.

We attempt to quantify the effects in a motivating experiment (Figure 2). We compare the execution of a query with 20% selectivity on a host DBMS-engine that transfers all data to the host to process it there, against an NDP execution of the same query, which performs all raw data transfers in-situ. Thereby, it employs different NDP result-set transfer approaches and varies the result transfer granularity. Clearly, choosing a small transfer granularity (tuple-based) incurs high result data movement overhead and reduces performance, while transferring the full result-set in a single large DMA transfer yields the best performance.

The **contributions** of this paper include the following:

- 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-size reducing operations.
- 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).
- 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.

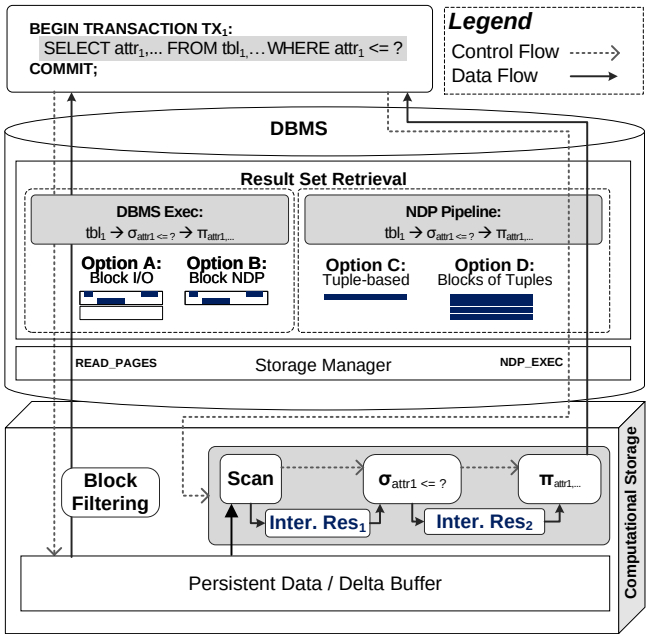

Figure 3: Different ways of transferring results from the device to host exist. While Block I/O transfers all raw pages, on-device block filtering reduces data transfers. Pushing down entire pipelines necessitates intermediate and final result management, spanning tuple-based and blocks of tuples.

We move on to present the related work (Sect. 2) and introduce NDP pipelines. Then, we discuss in-situ result-set materialization (Sect. 3) and system design (Sect. 4). Their performance impact is experimentally evaluated in Sect. 5.

2 RELATED WORK

The first proposed approaches of Near-Data Processing such as *Database machines* [6] or *Active Disk/IDISK* [1, 10, 13] occur already in the late 1980s. With proprietary magnetic/mechanical storage hardware, they achieved to execute small processing tasks near the storage. With the advances in the semiconductor industry, numerous systems [3, 4, 7–9, 14–16, 18–20] were established. While most of them focus on the hardware aspects of NDP, only a few give insights about their result management. IBEX [18, 19] exploits a tuple-based interface for executing joins on-device to ease the integration into the volcano-style MySQL database. A completely different approach is pursued by BlockNDP [4], which sets up on the interface of traditional filesystems and enables filtering on block-level granularity. While both approaches have their benefits for their given system architecture, in this paper, we propose a general concept for result-set handling on smart storage.

2.1 Result-Set Management

We now discuss the different types of result handling techniques in existing solutions, also depicted in Figure 3.

Block I/O. The classical approach is to use traditional I/O to read the physical pages that belong to a certain database object (Figure 3,

Option A), which is also often referred to as Block I/O [18, 19]. The drawback of this approach is that much of the data stored on those pages is either not required for the processing entirely or is discarded during the processing, i.e., by a selection criteria or projection. Consequently, the storage system bus is not used at its full potential even though its bandwidth might be fully exhausted.

Block-level NDP. One approach to avoid reading pages that are discarded later by the execution engine, is to discard those already in-storage. Thus, introducing in-situ processing on block-level granularity (Figure 3, Option B) allows filtering the physical pages according to some criteria and transfer only the matching (e.g., BlockNDP [4, 5]). As a consequence, the bus system bandwidth is only spent for reading pages containing some relevant data. However, physical pages usually still comprise multiple tuples, sometimes even in different arrangements that constitute data unnecessary for the execution engine. Hence, the bus is still not fully leveraged efficiently.

Tuple-based. With NDP it becomes viable to execute whole portions of a query execution plan, comprising multiple operators on-device. Such *NDP pipelines* comprise multiple database operators (e.g. selection, projection, joins, grouping and aggregation, etc.) with the goal of reducing the overall data movement up to the host. Noticeably, NDP pipelines produce *intermediary* and *final* results, both of which need to be carefully managed, because of the different nature of the operations (e.g. size-reducing or not, pipeline-able or not). For both intermediary and final results, one approach is to send them tuple per tuple (Figure 3, Option C), similar to a volcano-style execution engine. This clearly eases the integration into volcano-style DBMS and also allows for transferring only relevant data to either the next operator (intermediary result) or the host (final result). Unfortunately, this also entails a heavy communication overhead, especially with state-of-the-art bandwidth-optimized bus systems (PCIe) and protocols (NVMe).

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 properly, we propose an approach that can batch multiple result tuples (intermediary or final) into transfer units/blocks. In our approach, the size of these units can be configured from very small, to achieve a tuple-based behaviour, up to very large, to align with present bus or system optimized settings. 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.

3 IN-SITU MATERIALIZATION

The above 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 consume them in *a deferred* manner. It processes the results later on (see *consumption mode*, below) or even reuses them multiple times (see *reuse semantics*, below). 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 [12] of the DBMS.

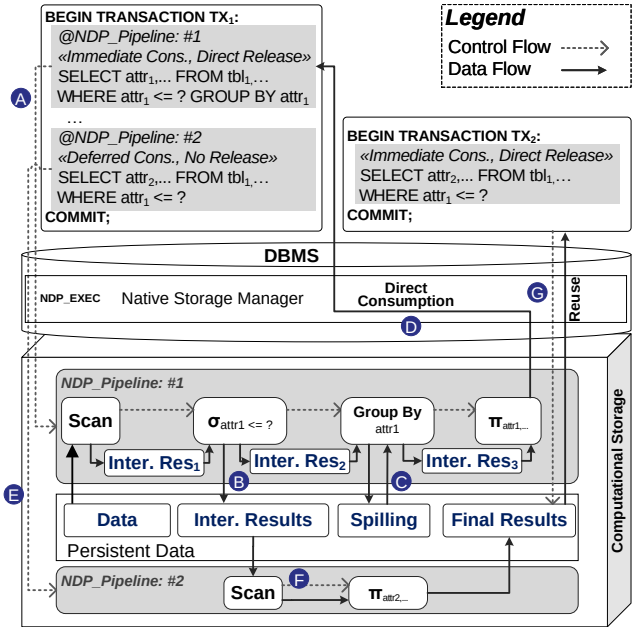

Figure 4: NDP Pipelines can materialize intermediate and final results in-situ, e.g., for reuse in further processing stages or to overcome resource constraints of NDP-Devices.

Consumption Mode. NDP pipelines necessitate different result consumption modes. As shown in Figure 4.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 4.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 4.B). The latter can be issued completely asynchronously. Secondly, we allow for result-*spilling* (Figure 4.C). It is applicable to operators such as a hashtable-based `GROUP BY` or an `HASH JOIN` implementations that exceed the on-device memory limits. These limits can be easily reached, as especially 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 4.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 point in time from the host or from another NDP pipeline.

Parsers and Accessors. The native NDP approach in nKV [15, 16] is based on the concept of in-situ data interpretation. To this end, NDP *parsers* and *accessors* have been proposed [15, 17] 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 4.F, where the *scan* in *NDP_Pipeline #2* can be optimized to consume the

intermediary results from *NDP_Pipeline #1*. To handle interpretation of intermediary results, we extended the parsers and accessors [15, 17] to cope with the different record formats of intermediary results and interpret intermediary data 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 4.G) or processing (Figure 4.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 [15, 16] is based on the concept of native storage [12]. In essence, native storage [12] 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 roundtrip to the host.

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.

4 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 [15, 16], 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, which allows host-device interactions to take place, while several interconnected and distributed state machines facilitate the NDP processing on-device (Figure 5).

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

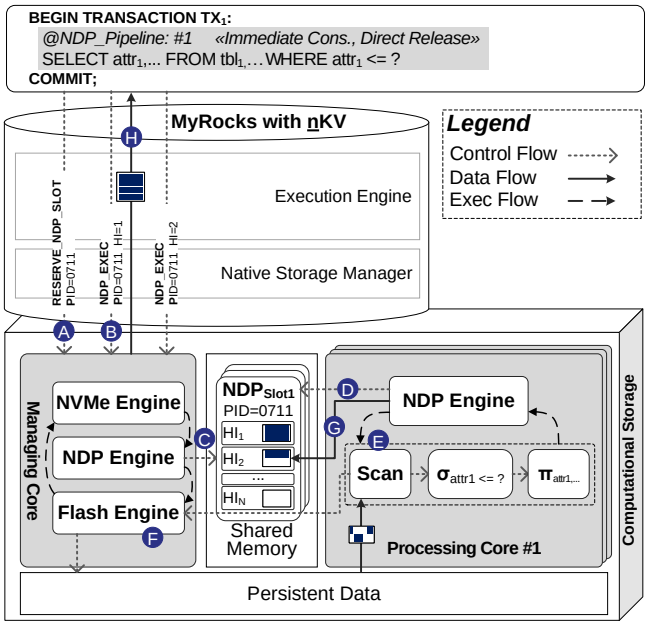

Figure 5: The system design relies on a lean communication protocol and multiple interconnected state machines, as well as shared memory to manage results of NDP pipelines.

device must allocate sufficient resources for the planned command. Therefore, the host can reserve an NDP Slot on-device (Figure 5.A) which is then assigned to a given processing id (PID). Subsequently, an NDP invocation is performed by means of NDP_EXEC as an NVMe command (Figure 5.B). Besides all relevant parameters for the execution, the command 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 NVMe command is returned with the respective results as payload (Figure 5.H). Upon that, the host can repetitively issue further NDP_EXEC 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 NVMe command is returned with a status field indicating the cause. As described in Sect. 3, the native storage manager resolves it, by scheduling GC or by allocating further pages and issuing a follow-up NDP_EXEC 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* (Figure 5). 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 as shown in

Figure 5.A/B/H. The NDP Engine is responsible for allocating the NDP Slot, transferring either information or extracting results from the HIs (Figure 5.C). Its counterpart on the processing cores continuously polls for new HIs of the NDP Slot (Figure 5.D) 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 (Figure 5.G) 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.

5 EXPERIMENTAL EVALUATION

Experimental Setup. We use MyRocks (MySQL 5.6) with nKV [15, 16] as storage manager. The *COSMOS+* board [11] is employed as an NDP-capable storage device and rough equivalent to a consumer-grade NVMe SSD or smart storage device (e.g., Samsung SmartSSD [8]). It comprises a Zynq 4045 SoC with an FPGA, two 667 MHz ARM A9 cores, and an MLC Flash module configured as SLC. 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.

Configurations. 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 [12, 15, 16]. 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.

Dataset and Workload. As dataset we utilize LinkBench [2] 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 employ a simple selection-projection query `SELECT id, type,...FROM nodetable WHERE type <= ?;` and 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 6, blue). By using the NDP stack (Figure 6, 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

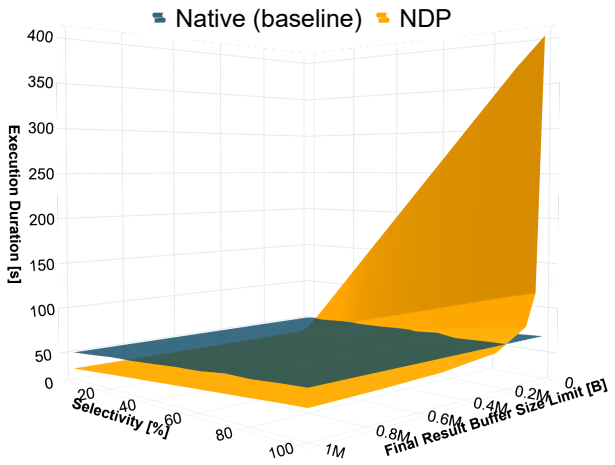

Figure 6: Traditional Block I/O cannot filter data on-device (blue) in contrast to NDP. Yet, transferring results in small granularities (e.g., tuple-based) entails high communication overhead (yellow). NDP wins after the intersection.

filtering the data, as shown with 0% selectivity. With higher selectivities, the amount of data to be transferred increases. Furthermore, the transfer granularity also entails an overhead of handshakes between host and device in the PCIe/NVMe communication. Thus, with *COSMOS+*, the best execution is obtained by transferring large blocks of tuples that improves 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. NDP pipelines can be divided into several processing and result management stages. Our approach foresees a finite state machine that enables a concurrent execution of those stages (interleaved pipelining). Thus, final results can be transferred to host, while the NDP device processes the next batch. In this experiment, we investigate the impact of interleaved pipelining and transfer granularity on the execution duration (Figure 7). We execute the query from Experiment 1 on the *NDP* stack with (yellow) and without (brown) 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, small transfer granularities that entail a high communication overhead benefit from interleaved pipelining, shortening execution durations by up to 30%. Yet, the largest possible *COSMOS+* transfer size (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.

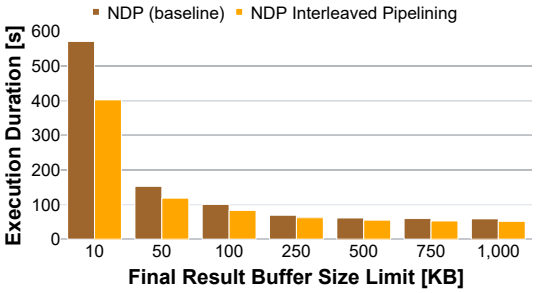

Figure 7: Using on-device state machines enables interleaved pipelining that improves execution significantly.

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 8. 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 (brown). We vary the selectivity to determine the impact of the final result size on the materialization cost.

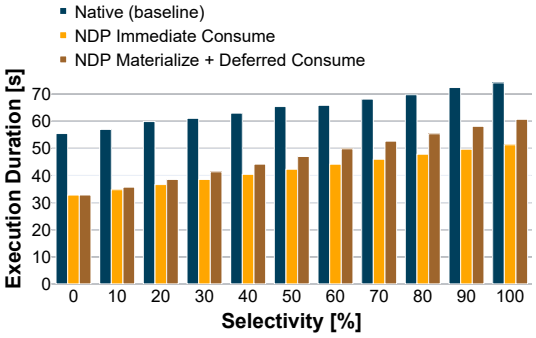

Figure 8: Final result materialization and deferred consumption (brown) entail a small overhead over immediate consumption (yellow) and outperform the baseline (blue).

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.

Experiment 4: The reuse of in-situ materialized results has marginal costs and amortize the 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 9 on a logarithmic scale.

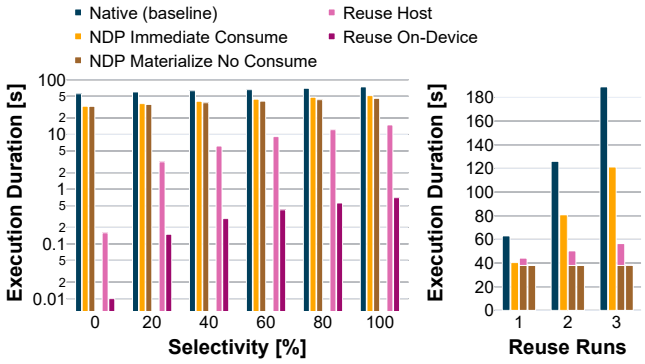

Figure 9: Reuse of materialized results improves the host (magenta) and on-device (dark red) performance significantly.

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 9 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 materialization of results. 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.

6 CONCLUSION

In this paper, we introduce novel NDP result-set management techniques to reduce the performance impact of result-set data transfers and enable reuse and better NDP execution modes. Based on the observation that the space on modern smart storage is cheap and fast, with low-latency and high-bandwidth data transfers, we introduce *in-situ operator spilling*, *in-situ materialization*, and on-device *reuse techniques*. These allow for richer *NDP pipelines* involving non-size-reducing operators and reuse. Our evaluation indicates that *in-situ reuse* improves performance by up to 95% for host consumption and 73-400x for on-device consumption, *materialization* results in 45%, while the optimal transfer granularity yields 27%.

Acknowledgments. This work has been partially supported by *BMBF PANDAS – 01IS18081C/D*; *DFG neoDBMS – 419942270*; *HAW Prom*, *MWK, Ba.-Wü., Germany.*

REFERENCES

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

[2] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proc. SIGMOD*. 12 pages.

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

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

[5] Antonio Barbalace and Jaeyoung Do. 2021. 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.

[6] Haran Boral and David J. DeWitt. 1989. Parallel Architectures for Database Systems. In *Database Machines*, A. R. Hurson, L. L. Miller, and S. H. Pakzad (Eds.). Springer Berlin Heidelberg, Chapter Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines, 11–28.

[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *Proc. FAST*. 29–41.

[8] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs. *Proc. SIGMOD* (2013), 1221. https://doi.org/10.1145/2463676.2465295

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

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

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

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

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

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

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

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

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

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

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

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