hybridNDP: Dynamic Operation Offloading and Cooperative Query Execution in Smart Storage Settings

Christian Knödler
christian.knoedler@
reutlingen-university.de
Data Management Lab,
Reutlingen University

Naeem Ramzan
naeem.ramzan@uws.ac.uk
School of Computing,
Engineering and Physical Sciences,
University of the West of Scotland

Ilia Petrov
ilia.petrov@
reutlingen-university.de
Data Management Lab,
Reutlingen University

ABSTRACT

Modern data-intensive systems perform complex analytical tasks on large datasets that keep growing at superlinear rates. Prevailing system architectures mandate that persistent data is transferred across the whole memory hierarchy to the host to be processed there. Data movement limits the system performance and impacts scalability and resource consumption inversely.

Yet, the emergence of intelligent storage/memory technologies and the ability to offload processing close to data creates new opportunities, as data movement is performed on-device much better performance and lower overall impact on processing. However, to date the decision of which operations to offload has been mostly hard-coded in near-data processing DBMS.

In this paper, we propose hybridNDP in an attempt to automate offloading decisions given an ad hoc query. The core idea is to split queries into host- and on-device processing parts and enable cooperative intervention-free execution. To this end we propose a cost-model to determine potential splits and a cooperative execution model. We evaluate hybridNDP with nKV and the Join-Order Benchmark. Our findings indicate that through the offloading and execution scheme hybridNDP outperforms traditional host-only executions on various queries by up to 4.2×.

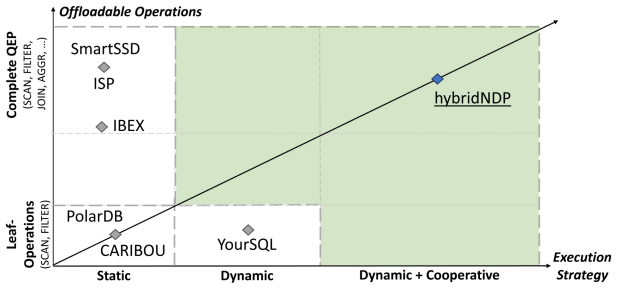

Figure 1: hybridNDP extends the current state of the art by combining in-situ execution of operations with a parallel execution strategy.

1 INTRODUCTION

Motivation. Modern data-intensive systems perform complex analytical tasks on large datasets that keep growing at exponential rates [14, 73]. Besides, datasets have poor data locality [29, 37], which inevitably results in massive data transfers across the whole memory hierarchy. What's more, prevailing DBMS architectures treat storage/memory as *passive* components and employ *processor-centric*, and *data-to-code* designs, mandating that the engine transfers the data to CPUs first, to process it there. Therefore, and because of the low data locality large portions of the dataset are only transferred to the CPU just to be discarded. As a result, such systems become both compute- and data-intensive, yet data movement is slow, increases resource consumption, and impacts system performance and scalability. Ultimately, the degree to which modern DBMS are economical [47, 52] sinks.

Emerging smart (a.k.a. intelligent or computational) storage [25, 57–59, 61, 82], economically combining memory and processing elements on the same device, may offer a practicable solution. Smart storage exhibits higher device-internal bandwidth, parallelism, and the lower on-device latencies [6]. Thus offloading DBMS operations for near-data processing (NDP) close to the physical data location, provably reduces the impact of data transfers, resulting in performance improvements between 3.6× and 15× [18, 23, 29, 32, 37, 68, 81].

Brief state-of-the-art Overview. To date, several NDP-capable DBMS engines have been proposed [2, 8, 13, 18, 26, 29, 30, 32, 34, 36, 51, 70, 71, 74, 77, 80, 81]. Systems such as IBM Netezza [23], Oracle Exadata [71], PolarDB [13] or X-Engine [29], or research prototypes such as [30, 36, 74, 77, 80, 81] can offload operations like `SCAN`s, `SELECTION`s to smart storage. IBEX [80, 81] has demonstrated an extended set of offloadable operations including `GROUP BY` and aggregate functions. ISP [36] furthermore extended the operation-set by offloading `JOIN`s. Lastly, SmartSSD [18] aims at offloading complete QEPs to smart storage.

However, in many cases, the decision of what operations to offload is *hard-coded* [18, 30, 36, 77], i.e. the executor offloads predefined operations *statically* and under all conditions, disregarding the specifics of the query execution plan, device parameters or data properties. In particular, such NDP systems typically hard-code two distinct offloading choices: (a) to offload individual size-reducing and transfer-intensive operations, e.g. `SCAN`s, early selections, early projections, or aggregations [30, 77] that are typically the leaves of a QEP, or (b) to offload whole query execution plans (QEP) [18, 36]. While the former clearly reduces data movement, it is not guaranteed to improve performance nor to exploit the potential for co-execution on the host-engine and smart storage. Whereas the latter may easily overload the weak smart storage compute elements by offloading too much or too complex operations [38]. Noticeably, these are two *opposite extremes* in the problem space of offloading strategies (Fig. 1).

More recently, *dynamic* offloading has been introduced in YourSQL [32] or PolarDB [13] allowing the DBMS to decide what to offload for a given query. Yet, only size-reducing operations, like `SELECTION`s are considered, which represent a conspicuous choice for NDP, but may not necessarily improve performance.

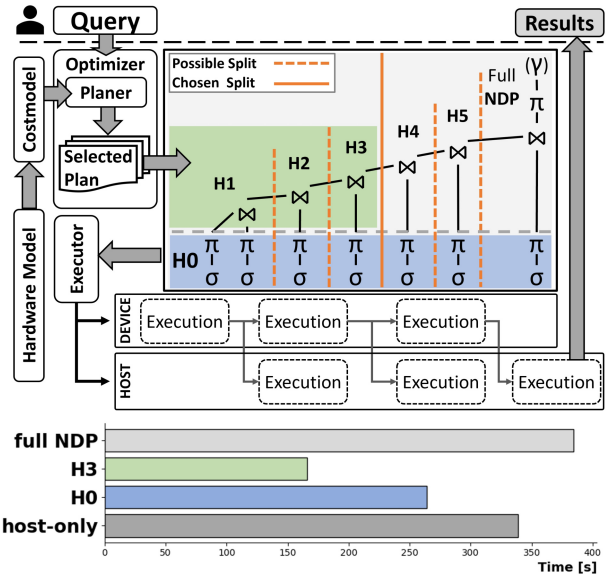

Figure 2: (top) hybridNDP extends the optimizer of a DBMS to calculate possible QEP splits. Out of all possibilities the optimizer chooses a split strategy and deploys the NDP- and host-parts to the on-device and host execution engines. (bottom) Confoundingly, offloading a QEP-split (`H3`) involving compute-intensive, non-size-reducing operations performs best compared to host-only execution (host-only), offloading the obvious size-reducing operations (`H0`) or the NDP execution of the whole query (full NDP).

Unfortunately large parts of the offloading space remain uncovered (Fig. 1). To be practical NDP-DBMS must be able to make *automated* offloading decisions, without hard-coding or optimizer hints. In this paper we pursue the following problems.

Problems. Provided that current offloading strategies covered corner-cases, intermediary strategies may yield better performance. However, they mandate splitting the QEP into an on-device portion involving transfer-intensive, but also non-obvious compute-intensive operations like `JOIN` or `GROUP BY`, while the host-engine handles operations suited for host-processing. This intuition rises the question of how to compute such QEP-splits.

After deploying both QEP portions, the host- and on-device engines will ideally execute simultaneously and overlappingly. Therefore, the second question is how to achieve such *cooperative execution* and prevent host-engine from staling until the results of the offloaded operation arrive and vice versa.

hybridNDP. We present an approach called hybridNDP in an attempt to address the above problems. The *main idea* is to split a QEP into a NDP-side and a host-side parts (called partial query execution plans – PQEP). This way both the NDP- and host-PQEP can execute in parallel and without additional interaction except for transferring intermediary or final results. hybridNDP extends the cost-model of the query optimizer to compute potential split-points, estimate their relative costs and choose a split-point with low overall costs.

However, in complex queries the NDP-PQEP may grow large and comprise multiple resource-intensive (compute, on-device memory) operations such as multi-table `JOIN`s or `GROUP BY`. To this end, hybridNDP introduces a generalized resource-model of smart storage hardware to improve split-point estimation.

hybridNDP investigates non-obvious cases, beyond the current state-of-the-art of offloading transfer-intensive size-reducing operations such as `SCAN`s, `SELECTION`s, early-selection or -projection. To our surprise, we find that offloading PQEPs involving (multiple) non-size-reducing and potentially computationally intensive operations such as `JOIN`s may yield better performance, by reducing the intermediary results in deep left plans.

hybridNDP also investigates a cooperative execution model, allowing the on-device engine and the host engine to overlap their executions and avoid waits. The key intuition is that the host execution is slowed down waiting for the on-device engine to produce results, be them intermediary or final, and vice versa the on-device engine idles waiting for a host-side NDP-invocation.

Introductory Experiment. We illustrate the effect of hybridNDP in a preliminary experiment demonstrating the execution alternatives of a complex multi-table join query – Q8.c of the Join-Order Benchmark [41] (full details provided later on in Listing 3, Experiment 6). Offloading the complete query for on-device execution (Fig. 2 - `full NDP`) performs worse than host-only execution (Fig. 2 - `host-only`), which is not surprising given its computationally intensive nature. The obvious case of offloading the transfer-intensive and size-reducing leaf-nodes of the QEP (Fig. 2 - `H0`), improves performance. However, the non-obvious case (Fig. 2 - `H3`) of offloading a PQEP involving multiple computationally intensive `JOIN`s on top of size reducing early selections and early projections indicates the potential for hybridNDP.

To the best our knowledge hybridNDP is the first approach for automated query offloading in NDP settings. Our **contributions** are: (a) We develop a set of cost-models to compute QEP splits into NDP and host-side PQEP for hybridNDP; (b) We develop a cooperative execution model hybridNDP that allows overlapping executions of the host- and the on-device engines. (c) Under the Join-Order Benchmark hybridNDP yields comparable or better performance in 47% of all 113 queries with improvements of up to 4.2× over the host-only-stack. Overall the optimizer chooses a suitable plan in 31.8% of the queries.

Outline. We continue with a brief background (Sect. 2). We provide more details on the cost-model and splitting QEPs in Sect. 3. The cooperative execution of on-device and host-engines is described in Sect. 4. We present the evaluation in Sect. 5 and conclude in Sect. 8.

2 BACKGROUND

2.1 Overview of nKV

hybridNDP (Fig. 3) is part of nKV [74, 77, 78], which is an NDP-based KV-store on top of RocksDB, and is exposed as a MySQL storage engine by means of MyRocks [21]. Main motivation behind employing nKV as a platform for hybridNDP is to reduce the LSM read-amplification through NDP even further. The write-amplification has been addressed at an earlier stage by NoFTL-KV [75]. nKV offers several abstractions to offload NDP-operations. nKV utilizes multi-level LSM-trees [48] whereas C_0 being an in-memory skiplist-based MemTable, and $C_1 ... C_n$ organized as Sorted String Tables (SSTs) on persistent storage.

Shared State. To efficiently process DB operations in-situ without any host-interaction (in an intervention-free manner), nKV employs a small *shared-state* to host all modifications that have not been flushed to persistent storage. For each DB-object modifications are accumulated in an in-memory MemTable (C_0) and passed alongside the NDP invocation to smart storage. This way nKV can generate a transactionally consistent snapshot of the

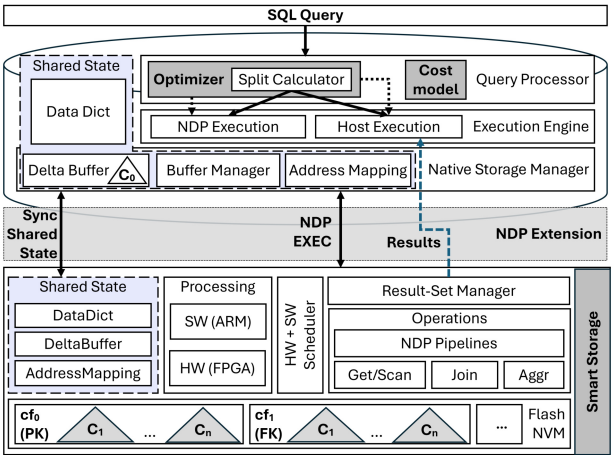

Figure 3: Architectural design of hybridNDP over nKV.

whole database on device, provide transactional guarantees for the NDP-execution (update-aware NDP [78]) and autonomous on-device processing. Furthermore, information about physical placements, with respect to the address-mapping table, for all involved database objects (SSTs) on $C_1 ... C_n$ are sent along with the NDP invocation.

Processing. Computations are placed on the on-device heterogeneous compute elements (such as ARM-cores, FPGA-units) [77] in order to process the offloaded operations. During processing the required data is loaded from flash, processed and the intermediate results are stored in cache-buffers inside the operation hierarchy or persisted on-device [76]. Once the processing of the first block is completed, the results are fetched by the host system and can be further processed or returned to the caller. In the meantime the processing of the next block can be started and results are stored in a round-robin way in several buffer-slots.

Offloadable operations in nKV. nKV currently supports the following offloadable operation types: `SCAN`, `JOIN`, `SELECTION`, `PROJECTION`, `GROUP BY` and aggregation functions. `SCAN`s are offloaded together with filter conditions, as well as any projection fields, if specified. A `JOIN` operation carries multiple join-conditions based on the given attributes of both input tables. nKV supports several basic on-device join-algorithms. Classical `NLJ` [69] (Nested Loop Join) , `BNLJ` (Block Nested Loop Join) and Grace Hash-Join (`GHJ`)[69] are available. However, since they are potentially inefficient in presence of indices, a BNLIJ (Block Nested Loop Index Join) is available to make use of primary and secondary indices. Furthermore, `GROUP BY` and aggregation functions such as `SUM`, `AVG`, `MIN`/`MAX` are supported in-situ, which allows nKV to execute basic, but complete NDP pipelines.

2.2 Overview RocksDB/MyRocks

Key-value stores use straightforward data structures where each distinct key is paired with a corresponding value. Well-known examples of key-value stores include Redis [63], Amazon DynamoDB [3], Apache Cassandra [4], and LevelDB [24] / RocksDB [20]. These systems are well-suited for applications that demand fast read and write operations over large datasets. Unlike relational databases, which support complex queries and joins, key-value stores excel at delivering high performance by providing direct access to data via key-value lookups. This makes them particularly suitable for real-time analytics and distributed systems, where low latency and high throughput are essential.

RocksDB. RocksDB is a high-performance, embeddable, persistent key-value store optimized for fast storage devices like SSDs and high-speed disk drives. Initially developed by Meta, it is based on LevelDB [24], but designed to support highly concurrent access and offer improved performance. RocksDB is highly configurable, allowing users to fine-tune performance and storage characteristics to meet the specific demands of their applications. Its ability to manage large volumes of data with minimal latency makes it a popular choice for applications ranging from embedded systems to large-scale web infrastructures.

Column Families. In RocksDB, column families provide a way to logically partition data within a single database instance. Each column family can be individually configured with its own set of options, allowing for targeted optimizations based on the specific data it holds. This flexibility enables applications to efficiently manage different types of data and access patterns within the same database. For example, one column family could be parametrized for read-heavy workloads, while another is optimized for write-intensive operations. This separation enhances performance and scalability by allowing precise data management policies tailored to the needs of each DB-object. Thus, different DB-objects are placed in separate column families.

LSM Trees. Unlike traditional data structures that update data in-place, LSM-trees [56] are designed as an out-of-place update mechanism to handle the high update and insertion rates seen in modern workloads, while also providing query capabilities.

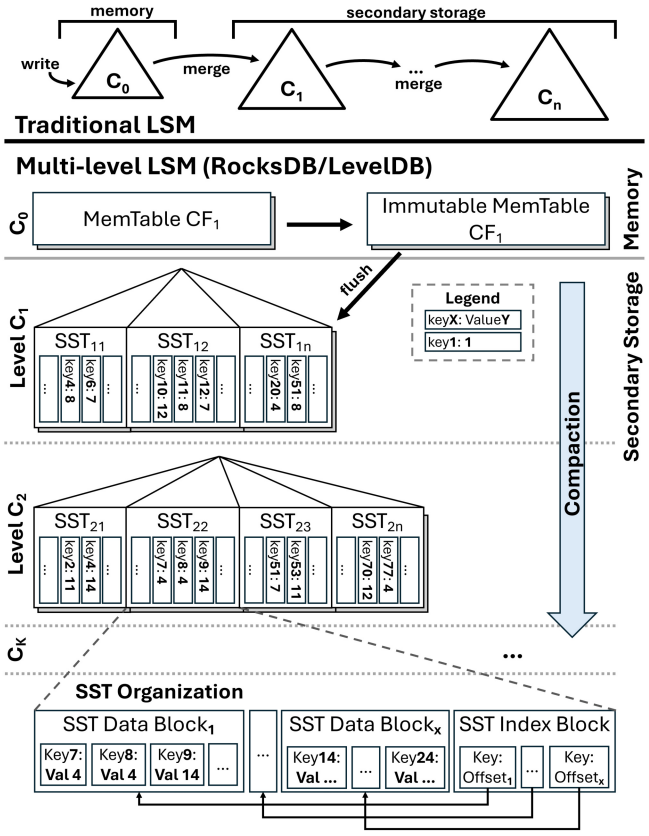

Figure 4: Conceptual organization of the multi-level LSM-Trees in RocksDB/LevelDB.

Classical LSM-trees [56] consist of several B-tree-like index components (C_0 to C_K, Fig. 4), which are stored in new locations and maintain a constant size ratio $r = |C_{i+1}|/|C_i|, i \in [0, K)$. When an insert or update operation is performed, it initially affects the C_0 component, which resides in memory. Once C_0 exceeds its size-threshold, it is flushed to disk and merged with the C_1 component. These merging processes gradually move data from C_0 to C_K, removing outdated key-value pairs and reclaiming space. Modern LSM-trees [48] are multi-level structures.

nKV [74, 77, 78] and RocksDB [20] introduce a separate LSM-tree for each column family, isolating the access patterns of different DB-objects. Changes to an LSM-tree are first placed in the main memory component C_0, which in RocksDB consists of a set of *MemTables*. These MemTables are implemented as memory-efficient data structures like SkipLists. Once a MemTable reaches its size limit, it becomes *immutable*, and a new MemTable is created to store further modifications. The immutable MemTables are eventually converted into *Sorted String Tables (SSTs)* and flushed to secondary storage (Fig. 4), where each LSM-tree component from C_1 to C_K contains multiple SSTs. The key-value pairs within the SSTs are stored in sorted *data blocks*, while a preceding *index block* holds key-offset pairs pointing to the data blocks, creating a sparse index. This index block reduces the complexity of accessing key-value pairs within an SST.

When flushing to C_1, no merge is performed for performance reasons, which can result in overlapping key-value ranges in the SSTs (e.g., SST_{11}-SST_{1n}, C_1, Fig.4). Compactions—merge operations to lower layers ($C_2 \dots C_K$)—either take SSTs from the previous level or combine them with SSTs from the target layer, depending on the strategy (e.g., tiered or leveled). During these compactions, all key-value pairs from the input SSTs are sorted, outdated entries are removed, and the results are stored in new SSTs at the target level. As a result, key ranges in SSTs below C_1 no longer overlap. However, keys can still appear on multiple levels with different values (e.g., $Key4$ or $Key51$) to reflect the temporal distribution of updates to a key-value record. For instance, $Key51$ may have been updated multiple times: the most recent version is in C_1, rendering the version in C_2 obsolete.

To *retrieve* a key-value record by key, the $GET(key)$ operation first searches the MemTables and immutable MemTables in C_0. If the key is not found, the system reads the index block of one or more SSTs in C_1, since SSTs may overlap at this level but not in lower levels ($C_2 \dots C_K$). Using the key-offset pair from the index block, the system identifies the data block that might contain the key, which is then read from secondary storage. If the key is still not found, the same process is repeated for layers $C_2 \dots C_K$. Due to the compaction process and data organization, a key can only exist in a single SST per level at this point. *Range scans* with or without key predicates work similarly but are more complex and are supported by additional internal structures like fence pointers. For example, $SCAN([Key51, Key70])$ traverses all levels, retrieving $Key51$ from C_1, and *Keys* 55, 70 from C_2.

Yet, if a scan involves *value predicates*, such as $SCAN(0 \leq Val \leq 6)$, the only available option is to iterate over the entire dataset. This leads to a substantial increase in I/O operations and a significant opportunity for improvement through NDP.

Secondary Indices. Secondary indices are a standard feature in MyRocks/RocksDB. However, most other persistent key/value stores do not support them, as this would contradict the general concept and access patterns, which boil down to key-based accesses over the primary LSM-tree. In RocksDB they are maintained as separate column families and, therefore, result in separate LSM-Trees. Both primary and secondary LSM-Tree employ the same structure, but differ in the record-format. A key in the secondary index combines its primary key, which is the secondary key in the primary index, with the key of the primary index and keeps the value for metadata. Lookups on the secondary index table require the DBMS to first perform a lookup inside the secondary LSM-Tree and extract the target primary-key, before performing another lookup in the primary LSM-Tree for each matching secondary key.

Use of Bloom-Filters and Fence-Pointers. Bloom-filters and Min-Max filters (also known as fence-pointers) are used by nKV's host-engine as in standard MyRocks/RocksDB to exclude as much SST-files outside of the predicate range as possible. Currently the NDP-engine makes no use of Bloom-filters as they have been already probed on the host side. However, this may change in future with more powerful smart storage devices.

3 COST-MODEL FOR hybridNDP

We now describe the details on the cost-model decision that determines a suitable host/NDP plan, and possible QEP splits.

Certain conditions must be met in order to offload PQEPs containing multiple JOIN operations, otherwise hybridNDP resorts to a traditional host-only strategy:

- Each table in a query is managed by a compatible engine.
- The smart storage device must be mounted in NDP-mode.

To calculate a suitable hybridNDP plan and QEP-splits, we utilize the parameters given in Table 1. For cardinality estimation, we rely on the standard MySQL techniques, which in case of MyRocks, are collected out of index samples to generate statistics (Table 1, System Variables).

3.1 Hardware model.

As different hardware cannot be compared directly to each other since not all of actual hardware properties can be taken into account, we introduce an abstract *hardware-model*. It comprises the characteristics (Table 2) of the internal/external flash properties, the performance of the embedded compute elements, the available memory, and the smart storage interconnect (e.g., PCIe).

Calculation of Hardware Parameter Settings. To determine the parameter-set for the hardware model, we create a special hardware profiling benchmark that measures the basic characteristics of the underlying smart storage device, which are then translated into the parameter values in Table 2. The profiler is an on-device micro-benchmark designed to be executed before the DBMS startup. In a single run it determines the static values for the hardware model, which are placed in the DBMS parameter file. The CPU and memory characteristics are determined based on a series of *memcpy-* operations across various buffers, and performing a set of floating point operations, whereas the flash performance is determined based on a mix of random read and write. The speed of the interconnect is determined by performing handshake-like data transfers of different sizes.

3.2 Cost calculation.

A basic approach to calculate the cost (c_{total}) of individual QEP's combines the *scan*, *cpu* and *transfer* costs (eq. (1)).

$$c_{total} = c_{scan} + c_{cpu} + c_{trans} \tag{1}$$

Table 1: Parameters of the cost-model

Variable	Definition
Cost-Calculation Variables – unit: costs	
c_{table}	Total access costs for a table
c_{scan}	Total scan costs for a single table
c_{cpu}	Total CPU costs for a single table
c_{trans}	Total transfer costs for a single table
c_{node}	Total cost of a node inside a QEP with all subnodes
c_{total}	Total cost of the QEP
cf_{pcie}	Cost function to calculate the costs of the PCIe
Intermediate Calculation Variables	
$calc_{sel}$	The selectivity for a table – unit: percentage
$calc_{frt}$	Estimated flash read costs utilizing the HW-Model
$calc_{pcf}$	Projection cost impact factor utilizing the HW-Model
$calc_{tvb}$	Transfer volume in bytes
$node_{ren}$	Resulting number of records for the current node
$node_{brc}$	Costs of buffer management of the current node utilizing the HW-Model
$node_{pbn}$	Number of projected bytes for current node
User / Configuration Variables	
usr_{rec}	Row Evaluation Cost
System Variables	
tbl_{ren}	Number of table records, matching the conditions
tbl_{sea}	Costs of accessing table-data provided by the underlying storage engine
tbl_{pfn}	Number of projection fields of the current table
tbl_{tfn}	Number of total fields of the current table
tbl_{pbn}	Number of projection bytes of the current table
tbl_{tbn}	Number of total bytes of this table
tbl_{nbs}	Block size of the current table/node – unit: bytes

Table 2: Parameters of the HW-model

Variable	Description
FLASH	
ndp_hw_{FCF}	Flash clock frequency (device)
$host\_hw_{FCF}$	Flash clock frequency (host)
hw_{FSW}	Flash weighting for hybrid-idx calculation
CPU	
hw_{CME}	Efficiency of CPU **memcpy**-operations
hw_{CCF}	CPU clock frequency
hw_{CCN}	Number of CPU cores
MEMORY	
hw_{MSH}	Memory size of host system
hw_{MSS}	Memory size of selections on device
hw_{MSJ}	Memory size of joins on device
ndp_hw_{MSW}	Memory weighting for hybrid-idx calculation
INTERCONNECT	
hw_{IPL}	Number of PCIe lanes
hw_{IPV}	PCIe version

However, it is applicable only to single-table queries, but not optimal for multi-join queries, especially on different hardware. To this end, we extend the cost calculation to respect hardware capabilities of the *hardware-model* in the following paragraphs.

Scan. The scan cost (eq. (2)) is computed by adding the table access costs tbl_{sea} and flash-access overhead $calc_{frt}$. To this end, we combine the number of flash pages to be read with a hardware factor that calculates the cost of accessing a single page utilizing flash frequency $host\_hw_{FCF}$, ndp_hw_{FCF} and the cost-factor resulting from the hardware profiler. The resulting costs vary depending on the estimated number of records (tbl_{ren}) and are applied to the host- and NDP-QEPs. Noticeably, the internal smart storage bandwidth is typically higher than the external, and therefore the host- and NDP-QEP scan costs differ.

$$c_{scan} = tbl_{sea} + calc_{frt} \tag{2}$$

CPU. The CPU cost calculation (eq. (3)), involves the records the current table to be evaluated (tbl_{ren}) and the amount of bytes required for projection ($node_{pbn}$). The former results from the total number of table records combined with the estimated selectivity $calc_{sel}$, multiplied by the evaluation cost per record (usr_{rec}). The latter (projection bytes) comprises the total table size in bytes tbl_{tbn}, the number of projection attributes tbl_{pfn} (and their bytes $node_{pbn}$), and the amount of processed bytes per record. Each is normed with a compute factor $calc_{pcf}$ taken from the hardware model and extends the CPU cost calculation. To account for the lower computational power of smart storage devices, hardware-model introduces a cost-factor for crucial operations, e.g. hw_{CME} - *memcpy*.

$$c_{cpu} = tbl_{ren} \cdot usr_{rec} \cdot node_{pbn} \cdot calc_{pcf} \tag{3}$$

Transfer. Reducing data movement is a major goal for NDP. Both, host and device cost calculations have to estimate the amount of data to be transferred from smart storage to the host. The transfer cost of a single table (c_{trans}), involves the transfer volume ($calc_{tbn}$) divided in blocks ($calc_{tvb}$), as well as the device-to-host latency (cf_{pcie}). In turn, the transfer volume is the product of the estimated number of records ($tbl_{ren} \cdot cacl_{sel}$) and the number of attributes (tbl_{pbn}) in the table in bytes. In case of NDP, the transfer volume is significantly smaller due to size-reducing techniques like early selection and early projection. Therefore the cost model respects only the attributes (tbl_{pbn}) and records, required on the host. Due to a lack of cardinality estimation, host and NDP plans utilize the number records estimated through the selectivity. The transfer-time of single blocks is part of the hardware model, which provides a cost-function (cf_{pcie}) representing the transfer speed of the underlying PCIe connection (eq. (4)) based on PCIe properties such as bandwidth, lane count, line encoding, step-speed, and version. Equations (5) and (6) compute the transfer costs for a host and on-device processing, respectively, whereas the latter are potentially lower.

$$c_{trans} = \frac{(calc_{sel} \cdot tbl_{ren} \cdot tbl_{pbn}) \cdot cf_{pcie}(hw_{IPV}, hw_{IPL})}{tbl_{nbs}} \tag{4}$$

$$calc_{tvb} = calc_{sel} \cdot tbl_{ren} \cdot tbl_{pbn} \tag{5}$$

$$calc_{tvb} = calc_{sel} \cdot tbl_{ren} \cdot tbl_{tbn} \tag{6}$$

Join. For multi-table queries, the optimizer estimates the best access path for each table, which comprises the scan (c_{scan}) and the CPU (c_{cpu}) costs. Furthermore, available indices and their suitability are considered. The resulting cost for the current table is combined with the subsequent table, for which best access path is estimated likewise. The cumulative costs for each further table are compared to other possible join orders. However, since data transfer occur only at the end of the processing with NDP/hybrid, the transfer costs (c_{trans}) are also pending at the end, depending on their selectivity (eq. (7) and equation (8)), whereas in the host scenario all tables are transferred.

$$c_{trans} = \frac{node_{ren} \cdot node_{pbn}}{tbl_{nbs}} \cdot cf_{pcie}(hw_{IPV}, hw_{IPL}) \tag{7}$$

$$c_{total} = c_{node-1} + node_{ren} \cdot usr_{rec} + node_{brc} + c_{trans} \tag{8}$$

The optimizer opts host-only or NDP-only execution based on the total QEP costs of both. Furthermore, suitable indices

are selected and in case of NDP an execution pipeline for on-device processing is built. As part of the join-order calculation hybridNDP also selects appropriate join-types.

3.3 Calculation of split-points.

In addition to the final QEP, which provides an adequate join-sort-order, hybridNDP optimizes the QEP by estimating a suitable split-position. QEP splitting mandates two preconditions: (a) the QEP must comprise at least 2 tables; and (b) the amount of data to be transferred must be close to the maximum transfer volume per command in order to exploit the high on-device bandwidth.

The main idea behind plan splitting is to compute a target cost c_target, and determine the split with closest cost to c_target out of all potential plan splits. hybridNDP computes c_target eq. (12) – (9) based on the host-to-device (in %) performance-ratio for CPU $split_{cpu}$ (eq. (9)) and memory $split_{mem}$ (eq. (10) and eq. (11)), with the goal of maximizing on-device resource utilization, but avoiding overload.

Next, hybridNDP determines the intermediate costs for all tables in the plan. These are then added cumulatively to each other, starting with the table with the lowest cost and adding the next higher cost. The addition of every further table marks a potential split-point, which is marked H0 through Hn. The cumulative cost at every split-point is denoted as c_{node}. For example, consider Fig. 5, which shows the cumulative costs c_{node} (y-axis) along a fictitious QEP with 5 tables and their position in the plan, i.e., *split-point* H0 – *split-point* H4, x-axis). Clearly H4 (Fig. 5.①) marks the total cost of the QEP involving all tables.

Finally, to determine a suitable split-position in the QEP, hybridNDP selects the split point with the smallest absolute distance to c_target (Fig. 5.③). Up to a certain threshold, the optimizer opts for on-device processing. However, as the amount of data increases, the device's capabilities become insufficient and the optimizer tends to opt for host-only processing. Notably, in multi-join queries involving a large number of tables, are often computationally- and memory-intensive. Thus, hybridNDP will potentially offload smaller portions of the QEP, i.e., pick an early split-position (more on the left Fig. 5), to remain economical and avoid over-consuming the weak smart storage.

$$split_{cpu} = \frac{100 \cdot (ndp\_hw_{FCF} \cdot hw_{FSW})}{(host\_hw_{FCF} \cdot hw_{FSW})} \tag{9}$$

$$split_{dev} = (tbl_N \cdot hw_{MSS} + tbl_{N-1} \cdot hw_{MSJ}) \tag{10}$$

$$split_{mem} = \frac{100 \cdot (split_{dev} \cdot ndp\_hw_{MSW})}{(hw_{MSH} \cdot ndp\_hw_{MSW})} \tag{11}$$

$$c_{target} = \frac{c_{total} \cdot (split_{cpu} + split_{mem})}{(2 \cdot 100)} \tag{12}$$

3.4 Example of query plan splitting.

To clarify the procedure of plan-splitting, we provide an example of using JOB Q1.a (Listing 1). It consists of 5 tables (lines 2-6) linked by 4 JOIN. The query meets the preconditions, and the optimizer identifies the split points H0 to H3 (Fig. 6 (diamonds)):

- H0: Offloading only leaf-nodes (tables ct, it, mi_idx, t, mc) to smart storage while the join operations remain on the host.
- H1: Offloading the leaf-nodes (tables ct, it) including the join ($tbl\_ct \bowtie tbl\_it$) to device and handover the join results to the host for further processing.
- H2-Hn: Offloading leaf-nodes, as well as 2-n joins to device, but executing the remainder of the plan on the host.

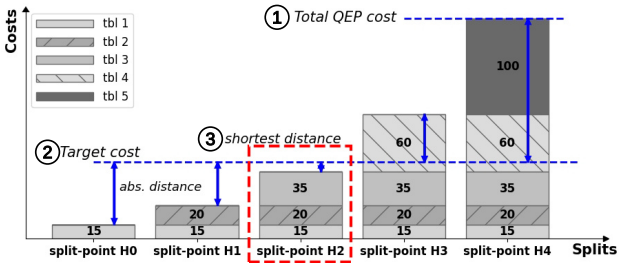

Figure 5: Split-point calculation for a QEP with 5 tables: The total QEP cost ① is calculated and compared against the target cost ② which is computed using the HW-model. Finally the split-point with the shortest distance ③ is depicted as QEP-split-point.

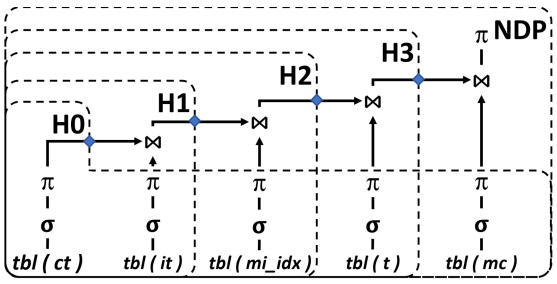

Figure 6: Overview of possible split-points for JOB Q1.a. Out of 5 tables (ct, it, mi_idx, t, mc) from the QEP, the split points (H0-H3 diamonds) are identified resulting in different PQEP plans for device (dotted container) and host (remaining).

The optimizer utilizes the cost model to find an appropriate split point for the workload distribution. Once selected, data dictionary information is used to define the necessary parameters, predicates, and buffers for the transfer between device and host which are added to the NDP command. As a result for H2, tables (ct, it and mi_idx) and joins ($tbl\_ct \bowtie tbl\_it$, $tbl\_it \bowtie tbl\_mi\_idx$) are used for device execution. Insight: Splitting the QEP allows the optimizer to consider additional optimizations on the NDP-PQEP specific to the device (see Sect. 4.2).

Listing 1: JOB Q1.a

```
1  SELECT mc.note, t.title, t.production_year
2  FROM company_type AS ct, info_type AS it,
3       movie_info_idx AS mi_idx, title AS t
4       movie_companies AS mc,
5  WHERE ct.kind = 'production␣companies'
6   AND it.info = 'top␣250␣rank'
7   AND mc.note NOT LIKE
8       '%(as␣Metro-Goldwyn-Mayer␣Pictures)%'
9   AND (mc.note LIKE '%(co-production)%'
10       OR mc.note LIKE '%(presents)%')
11  AND ct.id = mc.company_type_id
12  AND t.id = mc.movie_id
13  AND t.id = mi_idx.movie_id
14  AND mc.movie_id = mi_idx.movie_id
15  AND it.id = mi_idx.info_type_id;
```

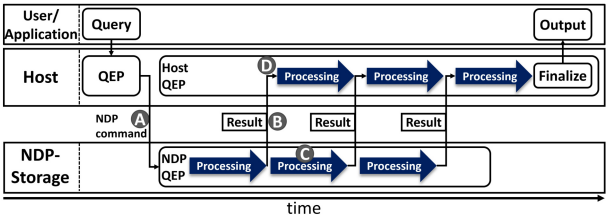

Figure 7: Parallel processing and interaction between the host and the device. (A) Initial NDP-invocation including all relevant information for the device execution; (B) Providing the intermediate results and hand them over to the host system for further processing; (C) the smart storage continues processing of the next intermediate-result-set independently of a host invocation; (D) mapping the incoming data to host-internal structures for further processing

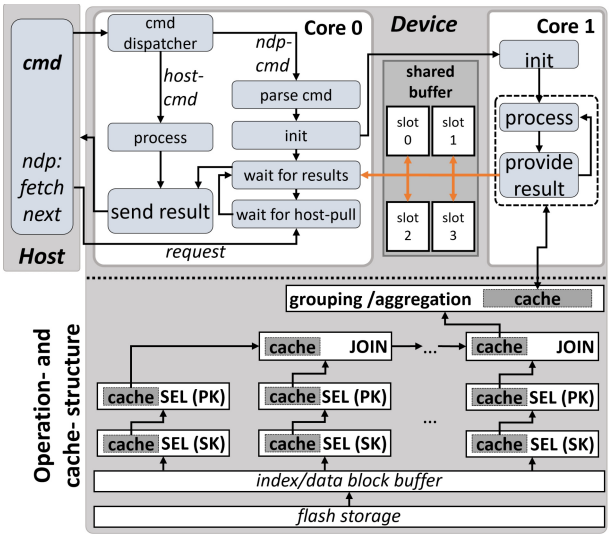

Figure 8: Device internal NDP-command processing through a management-core (Core 0) and a dedicated execution-core (Core 1)

4 COOPERATIVE EXECUTION MODEL

We now describe the cooperative execution model and the handling of multiple devices with their own PQEP based on the example shown in Listing 1 and Fig. 6.

4.1 Host processing

As soon as the nKV optimizer provides a finalized QEP and partial QEPs the executor performs an NDP-invocation for the NDP-PQEP. We extended the initial command of nKV, which already contains information about the logical / physical data placement and the additional information about used indices, among others (see Fig. 7.(A)). Since the host can start processing when the first intermediate results are available, it waits until they are ready on device. After fetching and transferring the intermediate result set from device to a dedicated memory-area on the host (see Fig. 7.(B)), the host system loads the PQEP of the host system and performs a lookup for the correct entry-point to place the data. In case there are further join operations the data will be mapped and added to the internal join-buffer-structures. Next, the processing of the current intermediate result is triggered on the host side (see Fig. 7.(D)) while the device autonomously creates the next portion of intermediate results (see Fig. 7.(C)). If smart storage provides the intermediate results faster than the host-engine can consume them, the remaining PQEP and the remaining buffers for the intermediate results on device are exhausted, the smart storage stalls and waits for the host-engine. Vice versa, the host execution will stall until the next portion of results is generated.
Insight: Choosing a late-split in the QEP which offloads the higher computational effort to smart storage, which could lead to long initial execution times (i.e., the time that takes to process the first intermediate results), which in turn increases the total query execution time. On the other hand, offloading low computational effort to the device, tends towards a traditional data-to-code architecture and therefore, the advantages of NDP processing cannot be exploited.

4.2 On-Device processing

Hardware Architecture. To efficiently handle the execution on smart storage, we rely on a dual-core *COSMOS+* platform. The first core is dedicated to receiving and handling incoming requests. It works as a relay for the NDP-execution and passes the incoming NDP-request to the second core, but processes host read/write I/O directly. Furthermore, it keeps track of the intermediate result-set provided by the second core and transfer them to the host on demand. The second core is solely dedicated to NDP-processing (Fig. 8). After receiving the invocation with all necessary information from the 1st core, it will process the NDP-request until it is finished or interrupted. If no more result-buffer-slots are available, the core halts until a free buffer is queued up. Figure 8 (top) illustrates the core-to-core interaction.
NDP operation structure. The NDP-pipeline typically comprises multiple operations. Noticeably, an NDP-pipeline aims at reducing data movement and achieving suitable NDP execution on-device and can include typical pipeline-breakers such as `GROUP BY` or `JOINs`, rather avoid them. On device, we employ a volcano-execution model, which means that the results of the underlying operation(s) are used as an input to the upper-level operation as shown in Fig. 8 (bottom). Therefore, intermediate result buffering (*cache*) is required to store the already processed results of each operation, to avoid the necessity of holding complete tables in-memory. Each operation process the input data as long as it's own buffer is capable of storing another record before handing over to the next operation. The final operation stores it's results in buffer-slots where they remain until they are fetched by the host for further processing (Fig. 8 - shared buffer). Notably, as soon as the first intermediate result-set is stored inside the result-buffers it is fetched by the host-system and executed in parallel. Along the on-device pipeline, multiple buffers exist and are allocated to each stage during processing:

- **block-buffer**: Holds the raw pages read from the flash storage.
- **selection primary-key/secondary-key cache**: Stores the resulting data of the selection after filtering (selectivity) and attribute-projection (projectivity),
- **join cache**: Stores either the resulting record, after attribute-filtering, or a pointer to the underlying caches.
- **group-cache**: If grouping is part of the processing, the group-cache holds the hashed grouping-values with it's data or pointer to the actual location in the underlying structure.
- **shared-buffer** The shared buffer stores on multiple slots the final processed records. This allows processing remaining work

without waiting for the host to fetch the result - as long as there are free buffer-slots.

As soon as the root NDP-PQEP operation has calculated the first result-set and stored it in the operation-assigned cache, core1 copies the result - in case a slot is available - to the shared buffers. Thus, core0 can pass the results to the host system insofar a host request is available.

Secondary index handling. The offloaded hybridNDP PQEP may contain join operations, for which there are matching secondary indices. To handle them efficiently, without falling back to scan operations, we added on-device secondary-index processing to smart storage. To the best of our knowledge, we are the first to employ secondary-index as on-device processing within an NDP-DBMS. Within RocksDB secondary index structures are stored as separate LSM-trees. Therefore, each secondary index has its own column family assigned. During preparation of the NDP-command on the host side, the optimizer extracts the additional information and extends the NDP-*selection* with a separate NDP-*early selection*. Thus, the early NDP-*selection* gathers the necessary information about the physical placement of the index and its format. The top level NDP-*selection* operation remains with the primary index and its structures. However, during a seek operation the upper NDP-*selection* requests the values from the underlying selection and seeks the requested records from the primary-keys that are stored along the secondary-key on the secondary-index structures.

Consider the example shown in Fig. 9. The given query performs a join between two simple tables: A and B. The generated plan might include an *indexed-block-nested-loop* as the join-type and a scan operation on the primary index for table A as well as an index-lookup on a secondary index for table B. During processing, the first results of table A are requested via the primary index (1) and the resulting KV 12*F* remains in the result-buffer of selection A until pickup (2). The join operation requests a key-lookup to table B for key 12 (3) where the request is redirected to the secondary selection of table B (4). Thus, the key 12 is looked-up in the secondary index and the resulting primary key 15 is forwarded to the primary selection of table B (5). The primary selection of table B again performs a seek-operation in the primary-index with the key 15 from the secondary index before giving the resulting value *UN* to the join (6) where it is combined with the data *F* from table A and returned to the requester.

Insight: The utilization of secondary indices yields significant performance gains for on-device processing. Although low selectivity profits from scan operations, whereas it may be suboptimal with high selectivities that are predestined for key-lookups.

Cache structure optimization. Due to a limited amount of memory we store the intermediate results as (a) row cache format or (b) pointer cache format. The former copies the resulting records after filtering and projection and stores them completely in the operation result buffer. The latter stores only the memory addresses of the projected attributes in its result buffer. Obviously, storing the complete record requires a lot of memory in the intermediate operations, yet limiting the total amount of tables that can be processed. However, storing only the memory address requires the data to be persistently stored on the lowest level until the processing of the pipeline is finished. In our setup, we switch to a pointer cache setup if multiple tables (> 2) have to be processed, otherwise using the row cache approach.

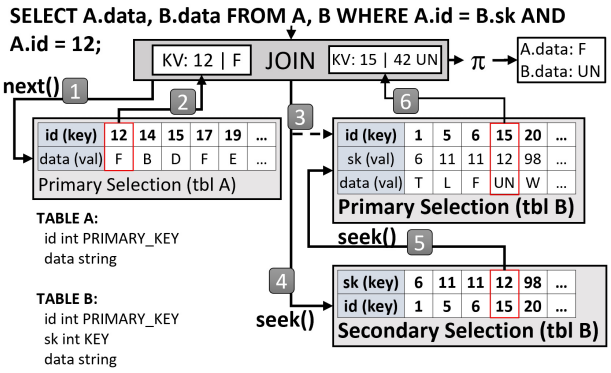

Figure 9: Enabling on-device secondary index processing through retrieving the seek-results from the secondary index/LSM-tree to perform a lookup of the primary-keys in the primary-index.

5 EXPERIMENTAL EVALUATION

Experimental Setup. The experiments are carried out on a server running MySQL 5.6 MyRocks over nKV and Debian 4.9. The server is equipped with a 4× core 3.4 GHz Intel I5 CPU with a 6MB L3 cache and 4GB of RAM. As consumer-class smart storage, we use *COSMOS+* [59], attached over PCIe2.0 x8, containing a Zynq 7045 SoC including an FPGA and two ARM A9 Cores running at 667 MHz. The smart storage has a 1 TB MLC Flash module configured in SLC mode. Furthermore, *COSMOS+* is equipped with 1 GB of DRAM, which is common for consumer-grade SSDs, which typically have 1GB-4GB [66, 67]. For the NDP-engine we make the following memory reservations:

- 20 MB internal systems (e.g. *program code*, *NVMe buffer*, *flash controller structures*),
- 520 MB for temporary storage (e.g. *data-block buffer*, *index-block buffer*, *result-set buffer*), and
- 100 MB for nKV internal structures (*index-block-mapping*, *data-block-mapping*, *data-dictionaries*, *processing-queues for core synchronization*, *operation-states*).

Thus, buffer NDP processing is restricted to approx. 400 MB.

Our experiments are conservative and the experimental platform is designed to approximate economical consumer-level devices (~150-200 €/TB), with relatively weak compute capabilities. To substantiate this claim we run the CoreMark Benchmark [19] on the host and the *COSMOS+* smart storage device. The host achieves a score of 92343 it./sec, while the single ARM core used for NDP processing on *COSMOS+* reaches only 2964 it./sec. Enterprise-class smart storage devices [25, 82] have much more powerful compute capabilities that can be leveraged for data

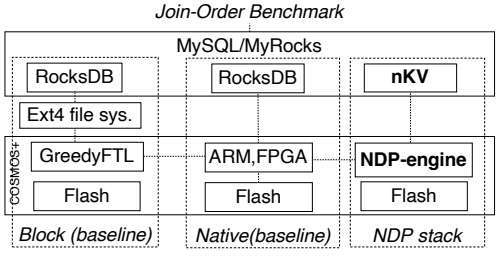

Figure 10: System setup of nKV for different baselines.

processing (e.g., 16 to 24 cores), albeit at higher cost (~500-1000 €/TB). However, in an enterprise scenario we also expect the host-system to have several dozens or hundreds of cores. Thus, the resource ratios will potentially remain the same or change for the benefit of the host.

Baselines. The experimental design is based on the following baselines: BLK, NATIVE and the NDP stack, which is used by hybridNDP and is evaluated against the BLK stack and based on the NATIVE stack, which in turn is used by nKV (Fig. 10).

Block/BLK (Baseline). As the main baseline we use the *block*-stack, which utilizes the traditional file-system stack with all it's abstractions. It is based on a block storage device and configured as block-device with an ext4 file-system. By default, the *block*-stack is utilized by MySQL for query execution by transferring all data from the device to the host system. The *COSMOS+* board runs GreedyFTL with 1 MB DRAM cache to maintain the block-device compatibility.

Native. As a second baseline, *native* eliminate all abstraction layers between the host and the device, which allows direct communication with the device. It mounts the storage directly into the user space of nKV through native NVMe. However, also the *native* stack transfers all data from the device to the host system in order to be processed.

NDP. The *NDP*-stack is based on the *native*-stack and extends the NVMe interface with commands for handling NDP-requests, which allows the execution of operations near to the data storage. I/O requests are managed by one of the ARM-Cores of the device, while the remaining ARM-Core is utilized for dedicated NDP-pipeline processing. In contrast to the host system, which has a relatively large amount of memory, the on-device memory is limited, which may prevent tables from fitting in memory. With respect to the number of tables to be processed on-device, the sizes of the buffers along the NDP-pipeline (selection, join, grouping) are adjusted until a certain performance threshold is reached. The outcome is a 17 MB buffer assignment for each selection through the primary index and another 17 MB for a secondary index. For each *block-nested-loop-*/ and *indexed-block-nested-loop*-join we assign 7 MB. This setup allows at most 12 tables with secondary index or up to 17 tables without secondary index to be processed in an operation-pipelined manner during a single NDP-call. However, smaller buffer sizes affect the on-device performance, due to more frequent buffer refreshes, depending on the workload. Overall, we determined a buffer size of $\geq$ 512KB reasonable for a *BNL*-join, whereas a *BNLI*-join is less affected.

Workloads. The workload is based on the Join-Order Benchmark [41]. Due to restrictions in the current implementation we slightly modify JOB to use fixed-sized byte-lengths for character-based values by employing string padding or trimming longer values. We also took the 4 byte alignment of the *COSMOS+* board into account. Therefore, the length of each value relies on the number of bytes that indicate the length of the value ($totalbytes = length - length - bytes$). For the integer-based values we always use a 4 byte integer for simplicity. Noticeably, we neither determine nor inject the optimal selectivities for the respective query in contrast to [41], which decreases the level of accuracy of the QEP-split decision. Optimal selectivities would result in much more accurate split predictions.

The data-set comprises around 74 million records in total, distributed over 21 tables resulting in nearly 16 GB of data including 6 GB for indices. The largest tables comprise nearly 50% of the total records, resulting in nearly 4 GB of data, including indices. Most of the tables have multiple secondary indices.

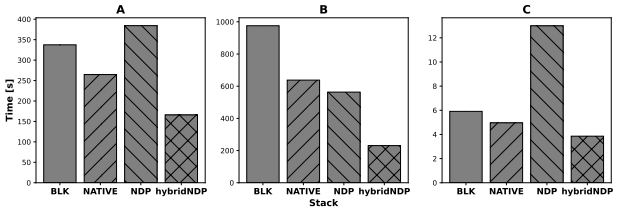

Figure 11: JOB queries (A) 8c, (B) 17b and (C) 32b, executed on the BLK, NATIVE, NDP and hybridNDP stacks. Notably, hybridNDP outperforms all baselines, whereas full-NDP is sub-optimal for (A) and (C), but on par for NATIVE (B).

For the detail evaluation, we rely on the queries 8c, 17b and 32b of the JOB since they query different numbers of tables and various combinations of primary-/secondary indices, which allows hybridNDP to employ `BNL`-, as well as `BNLI`-joins including utilization of secondary indices. `BNL`-join builds a hash table in the buffer [69], its use is preferred over our grace hash join and enforced for a fair comparison. Furthermore, Q8.c (Listing 3) has millions of intermediate results in each processing step reaching the buffer limitations. Q8.c is also a good example for optimal host/device co-processing.

Experiment 1: hybridNDP enables automated offloading and fills the gap between sub-optimal NDP and host-only decisions. Our opening experiment demonstrates the gains through hybridNDP relative to the baselines based on JOB queries 8c, 17b, 32b (Fig. 11). The depicted queries contain various numbers of tables to be joined, and vary the number of indices used, as well as several possible variants of conditions on nearly all data-types.

Figure 11 shows the execution time of three different JOB queries for each stack. While hybridNDP shows a significant performance improvement due to it's parallel processing, it can be seen with query (A) and query (C) that the runtime for a complete NDP execution is significantly higher than the baselines due to the high CPU compute intensity. Even though, queries like (B) are favorable for NDP due to the high selectivity in the early stages of the QEP, NDP still gets outperformed by hybridNDP due to the higher compute power that is required in the later stage of the QEP where a higher amount of data has to be compared and processed. Furthermore, a split of the QEP for (B) in a later stage guarantees low waiting times on both systems since the intermediate results in the early stages are low (see Table 3) and can be processed efficiently on-device, while the host evaluates the produced results against the last table. In this case, the host engine incurs extra but hidden costs for transferring the rest of the tables. Notably, the optimizer does not incur any significant additional effort for the NDP cost calculation as well as for the split calculation. We measured an average of 13 μs for query 17b in the vanilla MyRocks as well as for all presented stacks.

Experiment 2: hybridNDP improves the execution time of various queries. We continue our evaluation by executing all 113 JOB queries (33 queries with their individual number of sub-queries a-*n*) on the *block*-stack as well as offloading the leaf-nodes (`H0`), the full NDP-execution (*NDP*) and each intermediate hybrid splits *H1* to H_x as shown in Fig. 12. Throughout all executions, the same on-device parameter configuration has been used. Overall, hybridNDP outperforms (Fig. 12, green), or is on par (yellow) with the baselines in nearly 47% of all cases. The performance gain varies due to the executed query and it's chosen QEP from the

Table 3: Correlation of intermediate results and execution times for JOB - Q17.b

stack	# records processed on device	exec. time [s]	stack	# records processed on device	exec. time [s]
BLK	–	975	H3	148.552	637
NATIVE	–	637	H4	148.552	632
H0	1	638	H5	*7.796.926*	*230*
H1	41.840	637	NDP	52.307	563
H2	41.840	637			

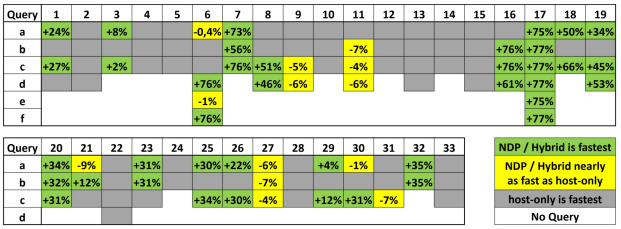

Figure 12: Performance of host-only, hybridNDP and full NDP execution under JOB. The columns represent the 33 JOB-query groups, each with several specific queries (rows). hybridNDP may outperform the host-only execution (green) or be on par with it (yellow), for which we report the performance improvement in percent. Not all query-groups contain queries from a-f (white).

MySQL-optimizer. Notably, only 1.7% of the query execution is favorable for a complete NDP-execution, and in 7% of the queries the offloading of only the leaf-nodes gains the best performance, the remaining queries are favored for a hybrid execution.

Insight: Despite conditions with higher in-situ computational effort, early data-size reduction has a higher impact on the query execution time. An optimal split of the QEP favors the parallel processing and therefore reduces the execution time. Unfavorable QEP-splits may prolong run-times.

Experiment 3: The cost model used by hybridNDP calculates suitable plans. So far, we showed the effectiveness of the partial offloading of queries to the device. Next we investigate into the quality of offloading decisions and the accuracy with which the optimizer estimates QEP-splits for hybridNDP given by our cost-model (see Fig. 13). In this case, we rely on the complete set of JOB queries and compare the decision of the optimizer with the results presented in Exp. 2. Without any user interaction, the optimizer predicts the best suitable hybrid-execution in 20,35%

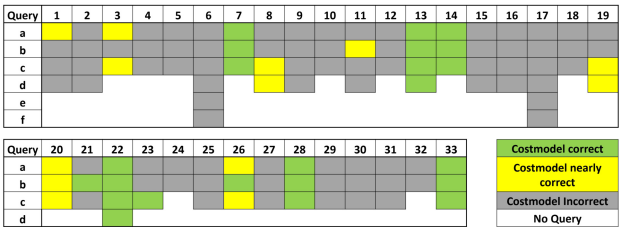

Figure 13: hybridNDP decision under JOB for all 33 query groups (columns) with their specific sub-queries as rows. hybridNDP, estimates the best suitable execution strategy (green) or targeting nearly the optimal (yellow), but misestimates the best suitable plan (gray).

and an acceptable hybrid-execution in another 11,50% of the queries. Overall, the cost model chooses a suitable plan in 31.8% of the queries of the JOB. These results are acceptable, compared to accuracy of PostgreSQL-optimizer, which yields a median error of 38% for all JOB queries [41] under perfect cardinalates. The cost model relies on the selectivity estimation performed by the DBMS, which limits the accuracy especially in cases of complex queries with multiple tables (cascading effect). In contrast to [41] we do not determine and inject optimal selectivities.

Experiment 4: Queries without utilization of indices are optimal for NDP operations even for non-obvious cases. We now relax prior assumptions and consider queries, favorable for smart storage. Given a query comprising two tables (Listing 2) joined on non-indexed columns (line 2), but shrinking the total amount of data based on a primary key column (*id* - line 3). The smart storage configuration is the same as in Exp. 2 and the table data is again taken from JOB. Table `movie_keyword` consists out of 4.5 million records and table `movie_link` has up to 30k records which result, by executing the query, in a total dataset size of around 8.5 million records with full-projection on both tables. The query is executed on the baselines and the *NDP*-stack as shown in Fig. 14.B.

Listing 2: Query joining 2 tables on non-indexed columns

```
1  SELECT * FROM movie_keyword, movie_link
2  WHERE movie_link.id <= 10000 AND
3  movie_keyword.movie_id = movie_link.movie_id;
```

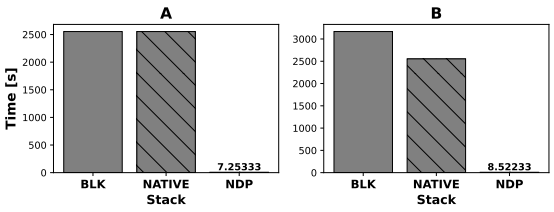

Figure 14: The *NDP*-stack outperforms the baselines due to the early-selection and early-projection with an on-device BNL-join for (A) limited and (B) full projection queries.

The *NDP*-stack clearly outperforms the baselines for the full- and limited projection despite the non-size-reducing join operation, which highlights the already existing strength of *NDP*.

Insight: While host-only processing mandates transferring data from storage to the host system to process it there, NDP-processing can directly access the requested data from the flash and filter out non-matching results in-situ.

Experiment 5: In-situ index processing increases the offloading potential for complex queries. So far we considered the on-device execution without secondary-index neither on device nor on the host-engine. Now we investigate the impact of in-situ indices on NDP-join-processing, considering the queries from Exp. 4, with *block-nested-loop*-join (NDP BNL) and *indexed-block-nested-loop*-join (NDP BNLI). Fig. 15 shows the results. We observe that the *block-nested-loop*-join, which does not utilize indices, is now a bottleneck for the *NDP*-processing. However, the *BNLI*-join is on par with the host-engine despite the higher CPU processing power on the host-side.

Insight: The current data-flow system architectures [42] avoid indices and target bandwidth utilization instead. On the upside they have simpler system designs and leverage the ample bandwidth in NDP settings. This experiment shows that if indices are

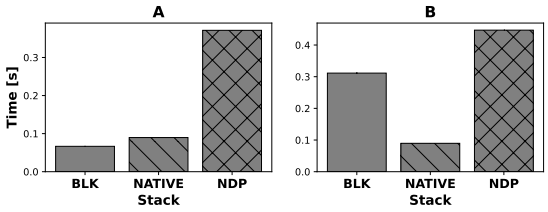

Figure 15: In-situ index utilization leverages the on-device performance and enables the device to outperform (in case of small projection - A) or compete (in case of full projection - B) compared to the host.

available, they are beneficial for NDP. Without the in-situ index processing the runtime of non-obvious NDP-operations may not keep up with host-only processing, which makes the efficient in-situ index handling relevant to real systems.

Experiment 6: Optimal timing for host/device co-processing is crucial for fast cooperative-execution. We continue our evaluation by varying the position at which hybridNDP splits the QEP host-PQEP and NDP-PQEP to investigate the performance impact. To this end, we consider JOB-query 8c (see Listing 3). The query processes seven tables, which results in nine possible execution strategies for hybridNDP (*block*-only, `H0` through `H6` as the hybrid options and *NDP*-only). The given query is executed for all stacks by forcing the cost-model to split the QEP at the respective position. The results are shown in Fig. 16.

We observe different execution times depending on the position of the split in the QEP. On the one hand a splits `H0` to `H2` tend to shift most of the compute power to the host system. On the other hand splits `H4`, `H5` or *NDP*-only assigns most of the processing effort to smart storage. With `H3` an optimal split of the computational work is performed.

Listing 3: Query 8c of the JOB.

```
SELECT a1.name, t.title
FROM aka_name AS a1, company_name AS cn,
     cast_info AS ci, movie_companies AS mc,
     name AS n1, role_type AS rt, title AS t
WHERE cn.country_code = '[us]'
  AND rt.role = 'writer'
  AND a1.person_id = n1.id
  AND n1.id = ci.person_id
  AND ci.movie_id = t.id
  AND t.id = mc.movie_id
  AND mc.company_id = cn.id
  AND ci.role_id = rt.id
  AND a1.person_id = ci.person_id
  AND ci.movie_id = mc.movie_id;
```

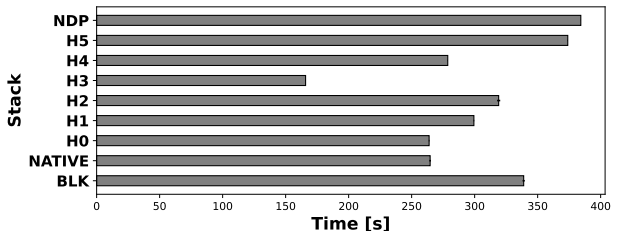

Figure 16: Execution time of Q8.c (JOB) for the host- and NDP-only execution, for different split-positions.

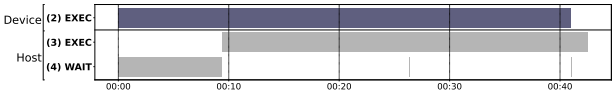

Figure 17: Execution of JOB Q8.d showing overlapping execution offloading 2 tables to smart storage, while executing remainder on the host.

To illustrate an optimal example of the parallel-processing flow we take a closer look at query 8d of the JOB, which is structurally identical to 8c, however, the `rt.role` condition targets `'costume designer'` instead of `'writer'`. Within query 8d the most suitable split position is `H3`. As shown in Fig. 17 once the *NDP*-command is offloaded, processing of the first buffer is initiated. However, during this time, the host system waits, which limits the performance in the order of the offloaded computational effort. As soon as the first intermediate result is processed and transferred to the host system, both - host and device - work in parallel. As long as neither the host nor the device has to wait for the other part, an optimal parallel processing is performed.

Looking into detail of the execution of Q8.d (JOB) for the optimal parallel processing (Fig. 17 - `H2`), we observe that the host has (despite the initial on-device execution) nearly no waiting time (see table 4 (left)) since the on-device processing provides the next results just-in-time. The on-device execution instead has a full workload nearly throughout the total processing time. Table 4 (right) shows a detailed breakdown of the on-device execution, highlighting the expensive *memcmp* operation. Insight: Early QEP splitting requires the host to process most of the query, while late splitting moves it to smart storage. Both could prolong runtimes, which makes it important to find a suitable split-point.

6 RELATED WORK

First approaches of offloading operations to computational devices ([16], [64]) showed a beneficial approach to reduce data-transfers between the computational device and the host system. Current systems confirm their validity and improve the implementation in modern database management systems. Several state-of-the-art systems, like [1, 13, 15, 30, 37, 40, 51, 77, 80], showed a beneficial approach to reduce data-transfers between smart storage and the host-engine, however, the optimization-research space is barely targeted. The main focus of these approaches base on the offloading single operations (leaf-nodes - e.g. [32]), simple but complete pipelines to be executed on device

Table 4: Detailed evaluation of host (left) and device (right) processing distribution JOB - Q8.d, `H2`

Host		*Device*	
Stage	**duration [ms] / [%]**	**Operation**	**duration [%]**
NDP setup (command)	121 / ~0.0%	memcmp	45,6
Wait (initial device exec.)	9.350.407 / ~21.81%	compare internal keys	13,2
Wait (2nd, 3rd device exec.)	5.667 / ~0.01%	seek index block	5,36
Result transfer	124 / ~0.0%	selection processing	4,56
Processing	33.507.442 / ~78,17%	seek data block	2,89
		flash load	2,43
		other	25,96

or decide at a coarse-grained level. COPRAO [7] in approach for NDP optimization, closest to our work. However, it extends traditional optimization rules and cost models to account for the specific on-device hardware capabilities, enabling hardware-aware query optimization. It also refines strategies to handle dynamic hardware changes during query execution. Additionally, COPRAO does not rely on using general resource abstractions as seen in cost-model based strategies, instead takes a hardware-conscious, rule-driven approach tailored for FPGA-accelerated environments. With ORCA [5], the optimization is even purged from the DBMS itself and outsourced to a better Optimizer than the implemented MySQL-Optimizer. hybridNDP optimizes on the software-side before the actual execution. Based on [38] which decides at a coarse-grained level (for a whole query) how to execute the given query, hybridNDP is able to analyze each query and splits a QEP into multiple parts to offload only the beneficial operations to the device. The offloading of partial QEPs is not only beneficial for obvious cases (data-intensive operations), but also for sequences of multiple joins which is, to our knowledge, the first published approach. The presented approach leads to an interesting execution-mode: interleaved execution.

Offloading in heterogeneous systems. Offloading and query optimization in GPU databases have been extensively studied to exploit GPU parallelism for accelerating database operations. Early works [72] introduced relational query co-processing on GPUs, focusing on accelerating spatial selection and joins. Full query processing systems followed [11, 27, 33, 50, 60, 84]. Various approaches aim to balance computation across heterogeneous resources [28, 39, 79, 83], with many employing cost-model-based offloading strategies to determine the optimal distribution [11, 27, 39, 83]. Other systems rely on data locality [10, 33], load balancing [50, 79], or architectural differences [60, 62, 84] for offloading decisions.

Recently, Data Processing Unit (DPU) in SmartNICs [12, 49, 53–55] have become available and are being increasingly deployed in datacenter infrastructures. For example, they are used to reduce the so called datacenter tax [31], improve disaggregated storage [85], data compression [44], and serverless tasks [46], offload distributed file system [35]. The problem of making automated SQL offloading decisions has been discussed in [22, 43].

7 DISCUSSION

In heterogeneous systems, involving different types of accelerators (e.g. GPUs, SmartNICs, FPGAs) or smart storage/memory (e.g. SmartSSDs, PIM systems like UPMEM or Samsung PIM), the offloading problem (determining what functionality to offload) is not specific to smart storage, which we address in this work. On GPUs different offloading strategies exist like cost based [9, 11, 27], or via load balancing [39, 50, 83] in order to avoid scheduling work on a less suitable processing element, and optimize parallel processing [65]. hybridNDP proposes a cost model including a HW-model which can be extended to adapt to different types compute element. However, several extensions in the cost model and hybridNDP have to be addressed like data transfers to the accelerator in order to compute correct execution cost, or selecting operations optimal for device processing to fully utilize its compute resources.

The general goal in offloading is both to reduce data movement and exploit the available (heterogeneous) computational capabilities of the acceleration device (GPU, FPGA, smart storage, DPU) through appropriate data-operation co-placement. However, depending on the type of acceleration device and its capabilities the balance between the compute and data movement targets may vary. For example, the computational capabilities of smart storage/memory are inversely impacted by economical constraints, as these are fabricated and sold under a commodity model. Thus the resulting relatively weak computational capabilities of consumer-grade SmartSSDs (~150-200 €/TB, like the on used in this work) shift the balance towards reduction of data movement and utilization of internal I/O characteristics, whereas enterprise-class Smart-Storage devices [25, 82] (~500-1000 €/TB) have much more powerful compute capabilities that can be leveraged for more computationally-intensive data processing. At the other side of the spectrum are acceleration devices such as GPUs (e.g., ~181 000 €/TB, NVIDIA A100 with 94GB HBM) or DPUs (e.g., ~26 000 €/TB, Bluefield-3 SmartNIC, with 16 ARM cores), where the balance stresses mainly leveraging the compute capabilities, to reduce of data movement.

Given their relatively weak computational capabilities, the goal of offloading work on consumer-class SmartSSDs, like the on used in this work, is exploitation of their good internal I/O characteristics, such as high bandwidth and parallelism, and low latencies, to reduce the movement of large and cold persistent data. Noticeably, the reduction of data movement, yields lower resource contention on the host and improves performance and scalability, the host CPU utilization and the energy consumption per unit of useful work. Prior work has demonstrated offloading strategies on mitigation of the impact of SSD garbage collection on performance and longevity [75, 77, 78], improvement of compactions [17, 45] in LSM KV-stores, MVCC visibility checking [8]. Furthermore, prior work has shown how operation offloading leverages the on-device computational resources to reduce data movement. In this work, we propose an approach for automating offloading decisions, leveraging I/O and compute resources.

8 CONCLUSIONS

In this paper we introduce hybridNDP as an approach automating offloading decisions in Near-Data Processing (NDP) DBMS. At present NDP-DBMS tend to hard-code such decisions ignoring the specifics of the given query, selectivities, or the presence of indices. Furthermore, either only data-intensive and size reducing operations (leaves in a QEP), or whole query execution plans tend to be offloaded, both of which deliver may sub-optimal results.

hybridNDP follows a different avenue and splits the QEP into host and NDP partial query execution plans. Surprisingly, we find that partial QEP may lead to much better results than the current extreme offloading strategies. To this end, hybridNDP introduces a cost-model and a hardware-model of smart storage to determine QEP-splits in an automated manner. Our evaluation under the JOB [41] benchmark indicates that in ~32% of JOB the queries the optimizer chooses a suitable or better plan for NDP.

Furthermore, hybridNDP introduces a cooperative execution model, that allows the host- and the NDP-engine to execute overlappingly and minimize waiting. As a result hybridNDP yields comparable or better performance in 47% of all 113 queries with improvements of up to 4.2× over the host-only execution.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for the valuable comments. This work has been partially supported by *DFG neoDBMS.2 – 419942270* and *BMBF PANDAS – 01IS18081C/D.*

REFERENCES

[1] I. F. Adams, J. Keys, and M. P. Mesnier. Respecting the block interface - computational storage using virtual objects. In *Proc. FAST 2019*, 2019.

[2] G. Alonso, T. Roscoe, D. Cock, M. Ewaida, K. Kara, D. Korolija, D. Sidler, and Z. Wang. Tackling hardware/software co-design from a database perspective. In *Proc. CIDR*, 2020.

[3] Amazon. Amazon DynamoDB. https://aws.amazon.com/de/dynamodb/, 2024.

[4] Apache. Apache cassandra. https://cassandra.apache.org/, 2024.

[5] W. Y. K. E. G. P.- L. C. S. Arunprasad P. Marathe, Shu Lin. Integrating the Orca Optimizer into MySQL. In *Adv. Database Technol. - EDBT*, volume 2022-March, pages 511–523, 2022.

[6] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[7] L. Beena Gopalakrishnan Nair and K. Meyer-Wegener. COPRAO: A Capability Aware Query Optimizer for reconfigurable Near Data Processors. In *Proc. 37th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2021, Chania, Greece, April 19-22, 2021*, pages 54–59. IEEE, 2021.

[8] A. Bernhardt, S. Tamimi, F. Stock, C. Heinz, C. K. Tobias Vinçon, A. Koch, and I. Petrov. neoDBMS: In-situ snapshots for multi-version dbms on native computational storage. *Proc. ICDE*, 2022.

[9] S. Bress, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Inf. Syst.*, 38(8):1084–1096, 2013.

[10] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1891–1906, New York, NY, USA, 2016. Association for Computing Machinery.

[11] S. Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank Spektrum 14*, page 199–209, 2014.

[12] Broadcom. Stingray PS225 SmartNIC. https://octopart.com/datasheet/ps225+-+dual-port+25gbe+pcie+ethernet+smartnic-avago-94766085, 2018.

[13] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, F. Zhu, and T. Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *Proc. FAST*, pages 29–41, 2020.

[14] C. Corp. Cisco global cloud index: Forecast and methodology, 2016-21.

[15] A. De, M. Gokhale, S. Swanson, and e. al. Minerva: Accelerating data analysis in next-generation ssds. In *Proc. FCCM*, 2013.

[16] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[17] C. Ding, J. Zhou, J. Wan, Y. Xiong, S. Li, S. Chen, H. Liu, L. Tang, L. Zhan, K. Lu, and P. Xu. Dcomp: Efficient offload of lsm-tree compaction with data processing units. In *Proc. ICPP*, page 233–243, New York, NY, USA, 2023.

[18] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs. *Proc. SIGMOD*, page 1221, 2013.

[19] EEMBC. CoreMark CPU-, MCU-Benchmark. https://www.eembc.org/coremark/, 2025.

[20] Facebook. Rocksdb. https://github.com/facebook/rocksdb, 2020.

[21] Facebook Inc., MyRocks. Transaction isolation in myrocks. https://github.com/facebook/mysql-5.6/wiki/Transaction-Isolation, 2021.

[22] F. Faghih, T. Ziegler, Z. István, and C. Binnig. Smartnics in the cloud: The why, what and how of in-network processing for data-intensive applications. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, page 556–560, New York, NY, USA, 2024. Association for Computing Machinery.

[23] P. Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbooks*, 2011.

[24] S. Ghemawat and J. Dean. LevelDB. Open-Source Implementation. https://github.com/google/leveldb, 2022.

[25] P. GmbH. prodesign HAWK Versal VC1902 Acceleration Card. https://www.prodesign-fpga-acceleration.com/products/prodesign-hawk-vc1902-acceleration-card, 2023.

[26] B. Gu, A. S. Yoon, and e. al. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proc. ISCA*, jun 2016.

[27] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), Dec. 2009.

[28] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[29] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 651–665, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent Distributed Storage. In *Proc. VLDB*, 2017.

[31] H. Ji, M. Mansi, Y. Sun, Y. Yuan, J. Huang, R. Kuper, M. M. Swift, and N. S. Kim. STYX: Exploiting SmartNIC capability to reduce datacenter memory tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 619–633, Boston, MA, July 2023. USENIX Association.

[32] I. Jo, D.-h. Bae, and e. al. YourSQL : A High-Performance Database System Leveraging In-Storage Computing. In *Proc. VLDB*, 2016.

[33] T. Karnagel, D. Habich, and W. Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10(7):733–744, Mar. 2017.

[34] J. Kim and et al. Papyruskv: A high-performance parallel key-value store for distributed nvm architectures. In *Proc. SC*, 2017.

[35] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[36] S. Kim, H. Oh, and e. al. In-storage processing of database scans and joins. *Inf. Sci. 2016.*

[37] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon. In-storage processing of database scans and joins. *Inf. Sci.*, 327:183–200, jan 2016.

[38] C. Knoedler, T. Vincon, A. Bernhardt, L. Weber, L. Solis-Vasquez, I. Petrov, and A. Koch. A cost model for ndp-aware query optimization for kv-stores. *Proc. DAMON*, 2021.

[39] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 555–569, New York, NY, USA, 2016. Association for Computing Machinery.

[40] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram. Summarizer. In *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture - MICRO-50 '17*, pages 219–231. ACM Press, 2017.

[41] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, Nov. 2015.

[42] A. Lerner and G. Alonso. Data flow architectures for data processing on modern hardware. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5511–5522, 2024.

[43] A. Lerner, C. Binnig, P. Cudré-Mauroux, R. Hussein, M. Jasny, T. Jepsen, D. R. K. Ports, L. Thostrup, and T. Ziegler. Databases on modern networks: A decade of research that now comes into practice. *Proc. VLDB Endow.*, 16(12):3894–3897, Aug. 2023.

[44] Y. Li, A. Kashyap, W. Chen, Y. Guo, and X. Lu. Accelerating lossy and lossless compression on emerging bluefield dpu architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 373–385, 2024.

[45] M. Lim, J. Jung, and D. Shin. Lsm-tree compaction acceleration using in-storage processing. In *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–3, 2021.

[46] G. Liu, L. Zhao, Y. Li, Z. Duan, S. Chen, Y. Hu, Z. Su, and W. Qu. Fuyao: Dpu-enabled direct data transfer for serverless computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 431–447, New York, NY, USA, 2024. Association for Computing Machinery.

[47] D. Lomet. Cost/performance in modern data stores. In *Proc. DAMON'18.*

[48] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[49] Marvell. Marvell LiquidIOTM III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf, 2024.

[50] S. Meraji, B. Schiefer, L. Pham, L. Chu, P. Kokosielis, A. Storm, W. Young, C. Ge, G. Ng, and K. Kanagaratnam. Towards a hybrid design for fast query processing in db2 with blu acceleration using graphical processing units: A technology demonstration. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1951–1960, New York, NY, USA, 2016. Association for Computing Machinery.

[51] S.-w. J. Ming, Arvind, and et al. BlueDBM: An Appliance for Big Data Analytics. *Proc. ISCA*, 2015.

[52] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[53] NVIDIA. NVIDIA Mellanox BlueField Data Processing Unit. https://network.nvidia.com/sites/default/files/doc-2020/pb-bluefield-dpu.pdf, 2020.

[54] NVIDIA. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2023.

[55] NVIDIA. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2024.

[56] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inform.*, 1996.

[57] OpenSSD. Daisy openssd fpga platform. https://www.crz-tech.com/crz/article/daisy/, 2023.

[58] OpenSSD. Daisy plus openssd fpga platform. https://www.crz-tech.com/crz/article/DaisyPlus/, 2023.

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

[60] H. Pirk, S. Manegold, and M. Kersten. Waste not… efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 508–519, 2014.

[61] R. Pitchumani and Y.-S. Kee. Hybrid data reliability for emerging key-value storage devices. In *Proc. FAST*, FAST'20, page 309–322, 2020.

[62] D. P. Raja Appuswamy, Manos Karpathiotakis and A. Ailamaki. The case for heterogeneous htap. 2017.

[63] redis.io. Redis. Open-Source Implementation. https://github.com/redis/redis, 2024.
[64] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer (Long. Beach. Calif).*, 34(6):68–74, 2001.
[65] V. Rosenfeld, S. Breß, and V. Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Comput. Surv.*, 55(1), Jan. 2022.
[66] Samsung. 990 pro nvme™ m.2 ssd. https://www.samsung.com/de/memory-storage/nvme-ssd/990-pro-4tb-nvme-pcie-gen-4-mz-v9p4t0bw/, 2024.
[67] Samsung. Samsung 850 pro sata iii 2.5zoll ssd. https://www.samsung.com/de/memory-storage/sata-ssd/850-pro-sata-3-2-5-inch-ssd-1tb-mz-7ke1t0bw/, 2024.
[68] S. Seshadri, S. Swanson, and et al. Willow: A User-Programmable SSD. *USENIX, OSDI*, 2014.
[69] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3), sep 1986.
[70] D. Sidler, Z. Istvan, M. Owaida, K. Kara, and G. Alonso. Doppiodb: A hardware accelerated database. In *Proc. SIGMOD*, 2017.
[71] M. Subramaniam, J. Loaiza, and T. Shetler. A technical overview of the oracle exadata database machine and exadata storage server. white paper. oracle corp. https://www.oracle.com/technetwork/database/exadata/exadata-dbmachine-x4-twp-2076451.pdf, 2013.
[72] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 455–466, New York, NY, USA, 2003. Association for Computing Machinery.
[73] A. Szalay and J. Gray. 2020 computing: Science in an exponential world. *Nature*, 440:413–414, Mar. 2006.
[74] T. Vinçon, C. Knödler, L. Solis-Vasquez, A. Bernhardt, S. Tamimi, L. Weber, F. Stock, A. Koch, and I. Petrov. Near-data processing in database systems on native computational storage under htap workloads. *Proc. VLDB Endow.*, 15(10):1991–2004, jun 2022.
[75] 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.
[76] T. Vinçon, C. Knödler, A. Bernhardt, L. S.-Vasquez, L. Weber, A. Koch, and I. Petrov. Result-set management for ndp operations on smart storage. In *Proc. DAMON*, 2022.
[77] T. Vincon, L. Weber, A. Bernhardt, A. Koch, and I. Petrov. nKV: Near-Data Processing with KV-Stores on Native Computational Storage. In *Proc. DaMoN*, 2020.
[78] T. Vincon, L. Weber, A. Bernhardt, C. Riegger, S. Hardock, C. Knoedler, F. Stock, L. Solis-Vasquez, S. Tamimi, A. Koch, and I. Petrov. nKV in Action: Accelerating kv-stores on native computational storage with near-data processing. *PVLDB*, 12, 2020.
[79] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *Proc. VLDB Endow.*, 5(11):1543–1554, July 2012.
[80] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB*, 2014.
[81] L. Woods, J. Teubner, and G. Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In *Proc. SIGMOD*, 2013.
[82] Xilinx. SmartSSD computational storage drive. https://www.xilinx.com/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf, 2021.
[83] K. Zhang, J. Hu, B. He, and B. Hua. Dido: Dynamic pipelines for in-memory key-value stores on coupled cpu-gpu architectures. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 671–682, 2017.
[84] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.
[85] Q. Zhang, P. A. Bernstein, B. Chandramouli, J. Hu, and Y. Zheng. Dds: Dpu-optimized disaggregated storage. *Proc. VLDB Endow.*, 17(11):3304–3317, Aug. 2024.