Understanding and Optimizing Database Pushdown on Disaggregated Storage

Hua Zhang
University of Wisconsin-Madison
Madison, WI, USA
Beihang University
Beijing, China

Xiao Li
University of Wisconsin-Madison
Madison, WI, USA
Beihang University
Beijing, China

Yuebin Bai
Beihang University
Beijing, China

Ming Liu
University of Wisconsin-Madison
Madison, WI, USA

Abstract

Database pushdown is a widely adopted technique under compute-storage disaggregation. The rising network and I/O speeds, coupled with stagnated compute and memory subsystems of a disaggregated storage architecture in the past decade, render state-of-the-art policy-driven pushdown designs ineffective. This is because the query performance bottleneck has shifted from network and I/O to compute, where computing power at the storage layer becomes scarce.

This paper rethinks pushdown database design via a systematic characterization and identifies three root causes, i.e., table structure agnostic, lower interference tolerance, and lack of operator scheduling. Based on the gathered insights, we build **TapDB**, a new pushdown database that targets emerging storage disaggregation. Driven by two key ideas (i.e., *lazy evaluation* and *trading network and I/O for compute*), TapDB introduces four new mechanisms: a table-aware operator cost estimator based on in-situ meta-learning and cardinality estimation, an admission control scheme to limit execution concurrency, a ballooning-based DRAM-SSD hybrid table, and a critical path-driven operator scheduler. Our prototype shows 1.3–2.3× speedups compared with prior solutions when running SSB and TPCH benchmarks.

CCS Concepts: • **Information systems** → **Database management system engines**; • **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → *Software system structures.*

Keywords: Computation pushdown, Storage disaggregation, Query optimization

ACM Reference Format:
Hua Zhang, Xiao Li, Yuebin Bai, and Ming Liu. 2026. Understanding and Optimizing Database Pushdown on Disaggregated Storage. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790243

ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790243

1 Introduction

Computation pushdown has regained attention with the widespread adoption of storage disaggregation in modern data centers [44, 65, 85, 93]. By offloading selected SQL operators to storage backends, pushdown can reduce network traffic and improve query performance, as demonstrated in both commercial systems [2, 26, 35, 39, 110, 111] and academic prototypes [114, 117, 120]. However, hardware trends over the past decade fundamentally change the cost balance of pushdown. Our analysis (§2.2) shows that network bandwidth and storage I/O throughput have improved by orders of magnitude, while CPU and memory performance on storage nodes have advanced only marginally. This leads to dramatically higher network and I/O density per core, making CPU resources increasingly scarce at the storage layer.

Such hardware trends contradict the fundamental assumptions of pushdown database designs, rendering them ineffective. Existing solutions (heuristic-based [39, 116, 117, 120] and cost-driven [15, 89, 106, 110]) view the operator pushdown as a policy question, i.e., relying on offline characterizations or simplified runtime estimates and outsourcing an operator to the storage layer when reduced data transfers yield performance improvements. The recent adaptive pushdown [118], like the cost-driven strategy, adjusts query execution based on the storage node load, but still uses static cost models and ignores compute-layer processing when pushback. Traditional pushdown methods become inadequate in modern storage architectures due to two shifts: the bottleneck moving from network/I/O to compute, and increased resource contention at the storage layer. Through systematic analysis (§3), we identify three root causes: the amplified impact of table structures under high I/O speeds, reduced tolerance to compute variability in new storage nodes, and the critical yet previously overlooked role of the operator scheduler.

Building on these insights, we identify three core challenges: (a) *unknown table structures* before execution, (b) *unpredictable runtime resources*, and (c) *uncertain memory*

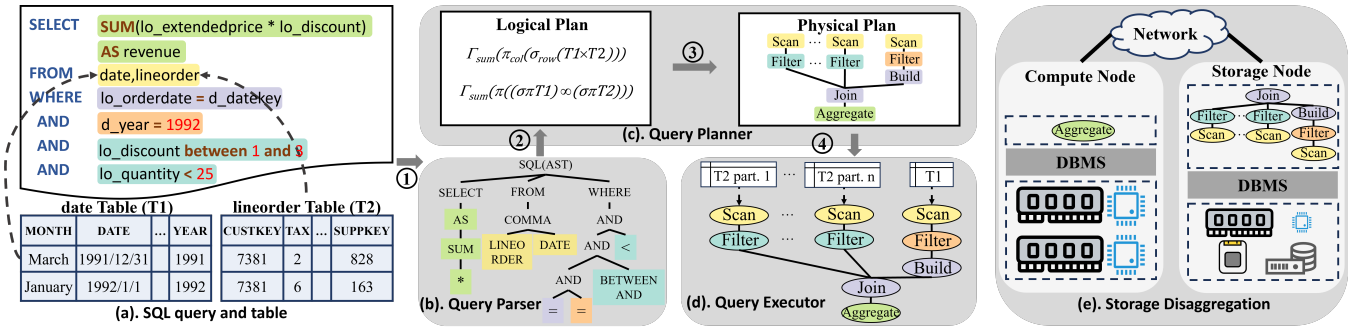


Figure 1. The workflow of an OLAP query (SSB Q1.1) running on a DBMS with computation pushdown enabled. (a) presents the input query and data tables. (b)–(d) depict three execution phases of a database engine. (e) shows how database operators are scheduled over local compute nodes (e.g., *aggregate*) and remote storage nodes (e.g., *build*, *join*, *filter*, and *scan*).

demands of concurrent operators. These render prior offline and static approaches ineffective in modern storage appliances. To address them, we design and implement a new pushdown database, dubbed **TapDB**. TapDB is driven by two ideas: *lazy evaluation* and *trading network and I/O for compute*, which (a) makes the informed pushdown decision based on the runtime system capability and (b) guards the pushdown operator to achieve non-degraded performance without inference. TapDB introduces four techniques: (1) table-aware operator cost estimation based on in-situ meta-learning and cardinality estimation (§4.2); (2) an admission control scheme by dynamically adjusting the learned cost to ensure pushdown operators stay within the interference-free window (§4.3); (3) a DRAM-SSD hybrid table based on the memory ballooning technique that implicitly extends the memory interference-tolerance region and storage-layer memory capacity (§4.4); (4) a critical path-driven operator scheduler that allocates the limited computing resources based on the operator's running urgency (§4.5).

We prototyped TapDB atop the latest FPDB [1] and evaluated it over four generations of disaggregated storage nodes. Using SSB and TPCH benchmarks, TapDB outperforms prior solutions by 1.3–2.3× under different data scales. TapDB makes informed database operator pushdown decisions based on the actual communication-computation trade-off at runtime, mitigates the performance interference, and sparingly schedules co-located operators to maximize performance. We further examine the effectiveness of individual mechanisms, study different impacting factors, analyze the system scalability, and quantify the performance overheads. TapDB is available at https://github.com/netlab-wisconsin/TapDB.

2 Background

2.1 Database Pushdown

Computation pushdown, running SQL operators close to the disk storage, is a well-known database optimization technique. It was originally proposed and developed for database machines in the 1970s, such as the Intelligent Database Machine [109] and Grace [41]. More recently, with the growing popularity of cloud-native databases [2, 26, 33, 39, 110] and storage disaggregation architecture, database pushdown has regained significant interest, and is used for OLAP query acceleration, networking traffic reduction, and client-side CPU cycle savings. Figure 1 depicts the workflow of an OLAP SQL query and how database operators are pushed down to the storage layer. We take the SSB [19] Q1.1 as an example (Figure 1-a) and assume two database tables are partitioned and distributed across several disaggregated storage backends.

- **Phase 1: Query Preprocessing.** The SQL parser analyzes and validates the input query, producing an abstract syntax tree that captures its syntactic structure (Figure 1-b).
- **Phase 2: Query Planning.** The planner [5, 12, 16, 17] produces a logical plan and applies standard optimizations. It then generates physical plans [9, 18] and selects one via cost- or rule-based optimization [27]. Pushdown can be decided during planning or applied to a finalized physical plan [74]. A physical plan is modeled as a Directed Acyclic Graph (DAG) of operators and data flows (Figure 1-c).
- **Phase 3: Query Execution.** An operator executes when its inputs are ready, and it is scheduled on an available engine (Figure 1-d). It first accesses local cached data and fetches from remote storage on cache misses. Based on the pushdown decisions made during planning, operators are scheduled to either compute or storage nodes [2, 26, 33, 39, 110, 117, 120]. In this example, all operators except *aggregate* are pushed down to the storage node (Figure 1-e).

2.2 Disaggregated Storage and Hardware Trends

Storage disaggregation is widely adopted in enterprise and cloud data centers [28, 44, 65, 85, 93], enabling independent scaling and high resource utilization. With high-bandwidth networks and NVMe-oF [14], disaggregated NVMe drives achieve millions of IOPS with tens of microseconds latency.

We examine four generations of disaggregated storage appliances deployed in our cluster over the past decade (Table 1) and identify several hardware trends. Network fabrics

Storage Node	Year	Compute&Memory	Network	Storage	Net Density	I/O Density
Gen1-SA	2010	2× Intel Xeon E5504 (4 cores), 64GB DDR3	8Gb Fibre Channel	FC, SATA3.0 HDD	**1Gbps**	**37.5K**
Gen2-SA	2014	2× Intel Xeon E5-2640v2 (8 cores), 96GB DDR3	25GbE Ethernet	iSCSI, SATA3.0 SSD	**1.6Gbps**	**75K**
Gen3-SA	2018	2× Intel Xeon Gold 6130 (16 cores), 192GB DDR4	100GbE Ethernet/RoCEv2	NVMe-oF, NVMe (Gen3)	**3.2Gbps**	**125K**
Gen4-SA	2022	4× ARM A72 (8 cores), 64GB DDR4	400GbE Ethernet/RoCEv2	NVMe-oF, NVMe (Gen3)	**12.5Gbps**	**500K**

Table 1. Hardware specifications of four generations of storage applications (SA) in our cluster. We report the compute, memory, network, and storage configurations. The network density and I/O density are defined as Gbps per core and IOPS per core, respectively.

evolve from 8Gb Fibre Channel and 25GbE to 100/400GbE (RoCEv2), while storage protocols shift from Fibre Channel [63] and iSCSI [56] to NVMe-over-Fabric [14]. Storage devices advance from SATA HDDs to SSDs and NVMe SSDs, increasing per-drive bandwidth to several GB/s and reducing latency to tens of microseconds. In contrast, CPU core counts grow modestly, with largely stagnant core frequency and per-core memory bandwidth. Recent designs further adopt energy-efficient ARM-based storage targets, similar to trends in public clouds [3, 11, 13]. As a result, network and I/O density (Gbps/IOPS per core) increase by over an order of magnitude, from 1Gbps to 12.5Gbps and from 37.5K to 500K, respectively. Similar trends are observed in other enterprise clusters [7, 8, 30]. With emerging EDSFF form factors [10], future storage servers will be even denser. Such growth makes CPU resources increasingly scarce on storage nodes, motivating a re-examination of pushdown database design under modern hardware constraints.

3 Characterizing Database Pushdown

3.1 Prior Pushdown Strategy

Existing databases push down operators mainly in two ways:

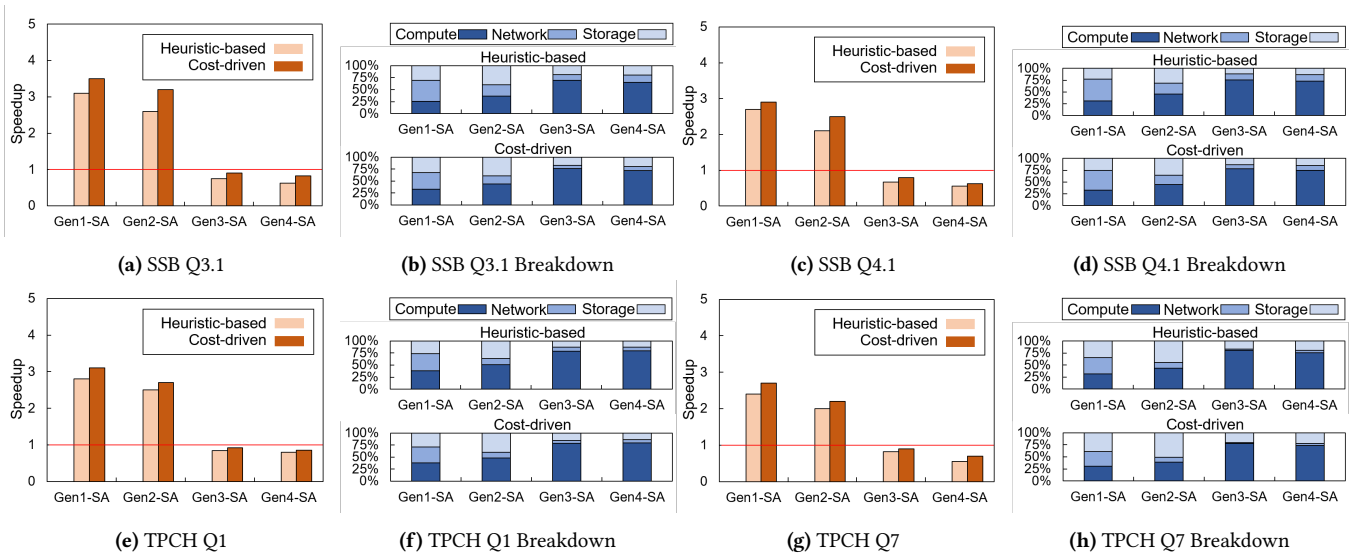
- **#1: Heuristic-based.** Some databases [39, 116, 117, 120] employ static heuristic rules to determine whether an operator can be pushed down based on offline analyses. For example, Taurus [39, 74] runs *selection*, *projection*, and *aggregation* operations close to the storage since empirical characterizations show they are faster than the pullup mode (fetching data to the compute layer and performing execution locally). Amazon S3 Select [4] supports *filter*, *project*, *join*, *group-by*, and *top-K* operator pushdown. This approach captures the execution characteristics of individual database operators without considering the computing resource dynamics at the remote storage layer;
- **#2: Cost-driven.** Others [15, 89, 106, 110] use a cost model to evaluate the computation-communication trade-off when making the pushdown decision. A query optimizer first uses a sample-based cardinality estimation scheme [71, 126] to gauge the data input size of each operator. Next, it calculates the operator's execution time when pushed down as $\frac{size(op)}{Proc\_Rate(op,\ Storage\_Node)}$, where $Proc\_Rate$, referring to the data processing rate, is platform dependent and obtained offline. Finally, the engine uses a watermark, i.e., $\frac{size(op)}{Proc\_Rate(op,\ Compute\_Node)} + \frac{size(op)}{BW_{Net}}$, where BW_{Net} is measured at runtime, to decide if the operator running remotely is beneficial. This mechanism accounts for network conditions but still relies on offline profiling to determine how fast an operator runs on the compute and storage nodes. **Adaptive Pushdown** [118] follows the cost-driven design and introduces a theoretical model to balance concurrent pushdown and pushback execution. Our TapDB implementation builds on it and retains its core components, including the pushback mechanism. § 5.7 provides a dedicated comparison of FlexpushdownDB and TapDB.

3.2 The Problem: Policy is NOT Enough!

We examine how pushdown databases perform under four generations of disaggregated storage architectures. To do this, we ported a recently open-sourced pushdown database (i.e., FPDB [117]) onto our storage backend, replaced its AWS S3 service with a similar self-developed object store, implemented all pushdown operators, and integrated both pushdown logics into the DBMS query execution engine.

Gen1-SA and Gen2-SA. Both pushdown strategies accelerate query execution on old-gen storage nodes. On average across all queries in SSB and TPCH benchmarks under different scaling factors, the heuristic-based approach achieves 2.8× and 2.3× speedups on Gen1-SA and Gen2-SA, while the cost-driven one yields 3.1× and 2.7× speedups. This is expected since pushdown operators reduce data transfers and bring near-data computing benefits. The cost-driven approach outperforms the heuristic one because it considers the network dynamism and only outsources a database operator when query acceleration between a compute and storage node outweighs the data movement across the network.

Gen3-SA and Gen4-SA. However, the two pushdown designs become ineffective on recent storage nodes. The heuristic-based method entails a 40% and 55% degradation on average across all queries in SSB and TPCH benchmarks on Gen3-SA and Gen4-SA nodes, while the cost-driven solution slows down queries by 36% and 45%. This indicates that reduced data transfers due to pushdown don't yield performance improvements. Instead, operators are running more slowly on storage nodes than on compute nodes. To further explore why, we break down the query execution time into three components: operator running (*Compute*), data movement (*Network*), and table access (*Storage*). Figure 2 shows the results of four queries. *Compute* dominates the query execution, accounting for 76.8% and 74.1% on average on Gen3-SA and Gen4-SA nodes. When the CPU of storage nodes becomes over-subscribed, the running speed of its housed operators is jeopardized, causing query performance drops. This happens exactly in Gen3-SA and Gen4-SA under fast network and



(a) SSB Q3.1 **(b)** SSB Q3.1 Breakdown **(c)** SSB Q4.1 **(d)** SSB Q4.1 Breakdown

(e) TPCH Q1 **(f)** TPCH Q1 Breakdown **(g)** TPCH Q7 **(h)** TPCH Q7 Breakdown

Figure 2. We report the query speedup (compared with the non-pushdown case) and query runtime breakdown of four queries from the SSB and TPCH benchmarks when enabling heuristic-based and cost-driven pushdown under four storage generations. We break down the query execution into three components: Compute (operator execution), Network (data transfer across the wire), and Storage (database table read/write). §5.1 describes our experimental setup. The benchmark scaling factor is 10.

I/O (§2.2). However, heuristic-based and cost-driven pushdown policies hardly capture this. In Gen1-SA and Gen2-SA scenarios, the storage-side computing resources are usually over-provisioned, whereas slow network and I/O mainly contribute to the query execution (which is more than 65–75% mostly, such as the four queries in Figure 2). This matches the hypothesis of existing pushdown design strategies, where trading compute for I/O helps accelerate the query.

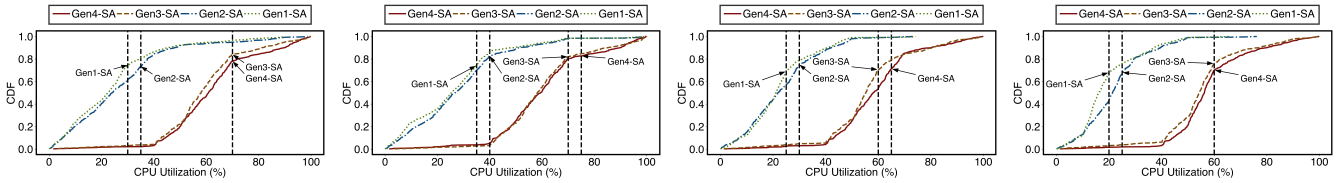
Takeaways. Databases formulate operator pushdown as a policy question, i.e., whether running an operator on storage nodes is beneficial. Existing pushdown approaches (heuristic-based and cost-driven) then evaluate the trade-off between computing and communication in an online/offline manner and outsource an operator when reduced data transfers yield performance improvement. However, the hardware trends of disaggregated storage architecture challenge such designs in two ways: (a) the performance bottleneck of pushdown query execution shifts from network and I/O to compute; (b) the computing resources on storage nodes become scarce. As a result, these render existing policy-based pushdown database designs ineffective because the operator's processing speed can be easily affected, given the fluctuating computing capacity at the storage node. For example, a pushdown operator at the storage layer would run much slower than at the compute layer, outweighing the savings of reduced data transfers and hurting the query performance. Therefore, we need a set of new mechanisms to (1) assist DBMS in making the right decision in a changing environment and (2) ensure the pushdown computations deliver the target promises. Next, we explore and identify the root causes that make existing policy-driven pushdown designs inefficient.

3.3 Root Cause #1: Table Structure Matters

Description. Fast I/Os exacerbate the impact of table structures on operator processing speed. We find that *Proc_Rate* is highly affected by row/column organization, data type, and data distribution on Gen3-SA and Gen4-SA nodes, making pre-execute profiling-based estimations ineffective. For instance, numeric types (like integer and boolean) are generally faster to read, compare, and jointly process than strings. `Sort` and `Join` operators favor a uniform distribution rather than a skewed one due to balanced partitions and reduced hash collisions. All these factors are unnoticeable in old-gen storage architecture since I/O access dominates.

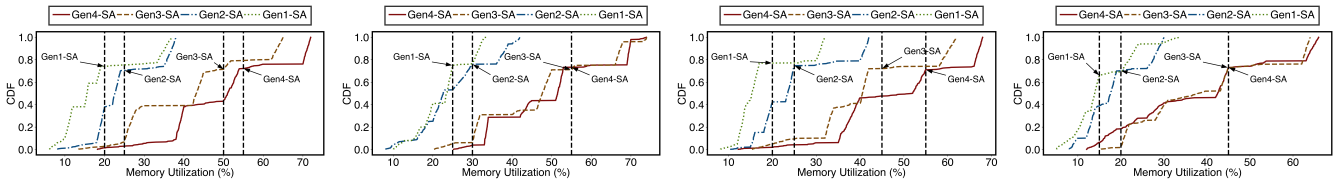
Characterization. We choose four database operators usually considered for pushdown, vary table sizes and structures, and explore how existing table-agnostic *Proc_Rate* estimations impact the query performance. On average across all scenarios, we observe that a 1.9(1.7)×, 2.6(1.9)×, 6.3(3.2)×, and 7.3(3.1)× difference between the estimated and actual processing speed for `Filter`, `JoinProbe`, `Aggregate`, and `Group` operators on Gen3-SA (Gen4-SA) nodes, causing inaccurate pushdown decisions, thereby yielding 45.0%(37.8%), 85.4%(54.1%), 62.5%(57.8%), and 54.0%(52.2%) performance degradation. However, this doesn't happen to old-gen storage architectures. Even though the estimated *Proc_Rate* sometimes differs significantly from the actual one on Gen1-SA and Gen2-SA nodes, it doesn't alter the pushdown decision because the threshold that accounts for both data transfer and operator execution is large enough.

- **Data Type.** For the `Filter` operator, the queries apply `(ORDERPRIORITY = '1-URGENT' AND ORDERDATE BETWEEN`



(a) SSB Q2.1 **(b)** SSB Q3.1 **(c)** TPCH Q1 **(d)** TPCH Q7

Figure 3. The CDF of CPU utilization of four queries on different storage nodes and their corresponding tolerant regions.



(a) SSB Q2.1 **(b)** SSB Q3.1 **(c)** TPCH Q1 **(d)** TPCH Q7

Figure 4. The CDF of MEM utilization of four queries on different storage nodes and their corresponding tolerant regions.

`'2023-01-01' AND '2023-12-31'`) and (`TAX > 50`) conditions to the `LINEORDER` table which processes complex data types (like `String`, `Date`, and `UINT8`-based subfields). The actual processing speed differs by factors of 1.7× and 1.3× from the estimated one, causing 36.3%(37.0%) and 53.6%(38.5%) degradation on Gen3-SA(Gen4-SA) nodes;

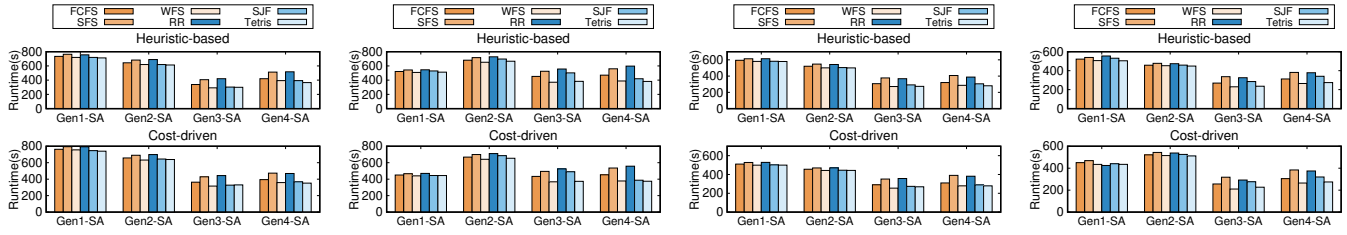
- **Table Organization.** We use the `JoinProbe` operator to illustrate this. Applying the (`JOIN CUSTOMER C ON L.LO_CUSTKEY = C.C_CUSTKEY`) and (`JOIN CUSTOMER C ON L.LO_CUSTKEY = C.C_REGION`) conditions over `LINEORDER` table and `CUSTOMER` tables generate either over-estimated or under-estimated amount of rows than expected, resulting in 1.4×(1.4×) and 1.7×(2.0×) processing cost difference on a Gen3-SA (Gen4-SA) node. This then yields false negative pushdowns and hurts query performance by 57.0%(34.0%) and 113.7%(74.2%);
- **Meta-operator Execution.** Operators (like `Aggregate`) incur non-trivial online running because they perform multiple arithmetic operations (e.g., SUM and AVG) over numeric columns while processing wide tables with irrelevant attributes, leading to significant computation and memory overhead during query execution. For example, queries perform (`SUM(L.REVENUE) AS TOTAL_REVENUE, AVG(L.TAX) AS AVG_TAX`) and (`SUM(L.REVENUE) AS TOTAL_REVENUE`), triggering arithmetic over complex data. We observe 2.0×(2.5×) and 3.3×(3.3×) estimation differences, causing operators to be wrongly pushed down, resulting in 47.8%(36.5%) and 77.2%(79.1%) query slowdown on Gen3-SA(Gen4-SA) nodes, respectively;
- **Data Distribution.** The computation cost of the `Group` operator is considerably affected by how data is distributed across a table. For example, the query (`GROUP BY CUSTKEY`) processes an evenly distributed table and incurs a light computing load. The actual cost is 3.3×(3.3×) less than the estimated on a Gen3-SA(Gen4-SA) node. Thus, the operator is not pushed down, hurting 44.2%(40.8%) performance. The 3rd query instead targets a skewed table and results in many hash collisions. We thereby observe a 2×(3.3×) estimation error and a 63.7%(63.6%) slowdown.

Takeaways. The rising I/O speed amplifies the impact of table structures on making pushdown decisions. Existing approaches relying on pre-execute profiling are not effective because they cannot capture data type, table organization, data distribution, and meta-operator execution, yielding many false negative operator pushdowns and jeopardizing query performance. Instead, we need a table-aware strategy to estimate the operator's table processing rate in situ.

3.4 Root Cause #2: Less Interference-Tolerance

Description. On recent-gen storage appliances, even if an operator is correctly pushed down, it can be affected by co-located operators or other I/O activities. Fast network and I/O expedite operator execution and cause more concurrent operator consolidation, thereby aggravating computational resource variability and interfering with pushdown operators. This is not an issue on old-gen storage nodes since their computing capacity is generally over-provisioned.

Characterization. To study the sensitivity to computing resource (CPU and memory) variability, we select four SSB and TPCH queries and measure their CPU and memory usage under interference-free execution with sufficient resources. We then co-locate a synthetic background operator on the storage node, which gradually increases CPU and memory consumption, to identify the threshold at which query performance degrades. We define the *interference-free window* as the utilization range [0, 100%−threshold] where query performance remains unaffected. Figures 3/4 depict the CDF of CPU/Memory usage of four queries and highlight their interference-tolerant regions on different storage



(a) SSB Q2.1 **(b)** SSB Q3.1 **(c)** TPCH Q1 **(d)** TPCH Q7

Figure 5. Query runtime comparison among 6 different operator schedulers of four queries from the SSB and TPCH benchmarks under heuristic-based and cost-driven approaches over four generations of storage nodes. The scaling factor is 10.

nodes. In terms of CPU, Gen1-SA and Gen2-SA expose a large interference-free window, i.e., [0, 68%] and [0, 65%], which is shrunk to [0, 33%], and [0, 28%] on Gen3-SA and Gen4-SA, respectively. Regarding memory, similarly, Gen1-SA and Gen2-SA can uphold other tasks that use up to 53% and 49% resources without slowing down contemporary operators. However, this upper bound is reduced to 25% and 20% on Gen3-SA and Gen4-SA. Such a narrow interference-free region indicates that pushdown operators are very sensitive to fluctuating computing resources. For example, when co-locating a background job whose CPU/memory consumption is slightly above the threshold, we observe 2.9× and 4.3× performance degradation on average across all evaluated SSB and TPCH queries in the Gen3-SA and Gen4-SA cases.

Takeaways. Emerging storage nodes are less tolerant of computing resource variability under operator pushdown due to computing scarcity and increased consolidation. This calls for an admission control-like mechanism to ensure co-located operators stay within their interference-free regions.

3.5 Root Cause #3: Operator Scheduler Matters

Description. Since database pushdown queries shift from I/O-bound to compute-bound on new-gen storage architecture, operator scheduling becomes another performance impact factor. Given the over-subscribed computing resources in the storage layer, the problem is how to allocate CPU cycles among competing database operators. We first characterize the issue below and will then formalize the problem in §4.5. Again, this is not an issue in Gen1-SA and Gen2-SA nodes, as their computing capability is adequate.

Characterization. We examine six widely used schedulers and see how they impact the performance of pushdown queries on four generations of storage nodes. These schedulers are: (1). First-Come-First-Serve (FCFS): operators are scheduled based on the enqueue order to the runnable queue; (2). Simple Fair Scheduler (SFS)[83]: operators are scheduled in a fair scheduling fashion (equal timeshare); (3). Weighted Fair Scheduling(WFS)[43, 91]: operators receive a timeshare that is proportional to their cost; (4). Round-Robin Scheduling (RR): competing operators equally receive a fixed time slice in each round; (5). Shortest Job First (SJF): operators are prioritized based on their estimated running cost; (6). Tetris[46]: operators are scheduled via a multi-resource DAG-aware packing algorithm that greedily selects operators to maximize the dot product of resource vectors.

Figure 5 compares the runtime differences among six scheduling algorithms across different SA generations. While the differences are limited on Gen1-SA and Gen2-SA nodes (up to 3.1% (3.0%) and 4.0% (3.4%)), they increase substantially to 45% (44%) and 47% (42%) on Gen3-SA and Gen4-SA nodes under the two pushdown approaches. **WFS** performs best on Gen3-SA nodes by allocating CPU cycles according to operator resource consumption, while **Tetris** outperforms others on Gen4-SA by jointly considering DAG dependencies and resource demands. **FCFS** is effective only for simple query structures where the enqueue order reflects computing demand. **SFS** and **RR** treat all operators equally, leading to execution stalls under heterogeneous demands (e.g., `JoinProbe` should be prioritized over `JoinBuild`). **SJF** prioritizes lightweight operators (e.g., `Scan` and `Filter`), breaking DAG dependencies and resulting in suboptimal execution.

Takeaways. The scheduler at the storage node determines how computing resources are allocated among competing pushdown operators, becoming pivotal under the hardware trends of disaggregated storage. This is generally overlooked in old-gen storage architectures since the query performance is dominated by network and I/O. An optimal scheduler should consider the per-operator execution cost, DAG structure, as well as runtime computing availability.

4 TapDB: Design and Implementation

4.1 Key Idea

TapDB is a pushdown database designed for the recent disaggregated storage. Our system design goal is to maximize database pushdown benefits without hurting query performance. Based on the gathered insights (§3), we build TapDB following two principles. One is *lazy evaluation (P1)*, inspired by the programming language theory [54], where we should make the informed pushdown decision just before the operator is scheduled. The other is *trading excessive network and I/O bandwidth for scarce computing resources (P2)*. Based on these, we develop four techniques: (1) table-aware operator cost estimation based on in-situ meta-learning and cardinality estimation (§4.2), which samples operator execution

and fits a linear model to predict execution cost on storage and compute nodes; (2) an admission control scheme by dynamically adjusting the learned cost to ensure pushdown operators stay within the inference-free window (§4.3); (3) a DRAM-SSD hybrid table, borrowing the memory-ballooning idea to spill intermediate results to SSD when memory is scarce, thereby implicitly extends the memory interference-tolerance region and storage-layer memory capacity (§4.4); (4) a critical path-driven operator scheduler, which identifies the critical path and data dependencies in the query DAG, and preferentially allocates the limited computing resources based on the operator's running urgency (§4.5).

4.2 Table-aware Learning-based Cost Estimator

The first issue (§3.3) happens since the input data size and table structure are unknown. Existing databases make *early* pushdown decisions during the query optimization phase via a cost model, which is developed using offline profiling and cardinality estimation [64, 69, 107, 113]. Yet effective on old-gen storage nodes, such a data-driven approach only considers static computation and communication costs, completely ignoring database runtime factors (§3.3) that contribute to incorrect operator pushdown on newly storage architecture.

We apply the *lazy evaluation (P1)* principle and determine whether an operator is pushed down until its upstream producer emits partial results, from which the schema of the data table operated on can be derived and the actual computation-communication trade-off is clear. We enhance the de facto cardinality estimation with meta-learning [108, 123] that gauges table processing cost in situ via enhanced sampling. For an operator op, our approach works in three steps: (1) extracting the cardinality of its input table $input_{op}$, (2) measuring the execution time of representative samples $sample(input_{op})$, and (3) predicting the full-table computational cost via an online-trained regression model based on dual-dimensional features.

First, we use *Equidistant Sampling* to generate curated table samples with low overheads. Data rows are sampled at equal intervals through two sequential steps: *(1) Interval Calculation*, which calculates the sampling interval k as $\left\lfloor \frac{size(input_{op})}{size(sample_{op})} \right\rfloor$; *(2) Sample Row Selection*, which selects the first row and every k-th rows. Sampling at equal intervals allows us to choose rows that capture the overall table characteristics. We evaluate different sampling ratios (taking 10% in our TapDB), where the marginal accuracy gain becomes negligible relative to the added runtime overhead. We then use the execution time of these samples as a predictive feature for estimating the operator's total computational cost. We further validate if this approach is effective by examining the Pearson coefficient [38].

Second, we develop an eager input estimation scheme that approximates the actual input size of an operator based on the first received data message size, scaled by a constant factor. The scaling ratio depends on the processing rate of the upstream-producing and downstream-consuming operators. When an operator has multiple upstream producers of different types, it estimates the total input after receiving one message per type by scaling the observed message sizes by the number of producers in each category, i.e., $\langle m \cdot |A_i|,\ n \cdot |H_i| \rangle$, assuming uniform data volumes within each type. This method is essential because many database operators run in a non-blocking fashion for performance optimization, i.e., beginning execution without waiting for complete input tables (Precedence Constraints, §4.5). The naive approach, assuming an equal-sized partition, works well for tables with uniform data production volumes across all producer operators. But for skewed tables, such approximations are inadequate because processing entails significant imbalances across different table partitions. We thus proactively introduce a special *Pre-Scan Balancing Operator* to replace the conventional scan logic based on the uncertainty of data skewness patterns. It reshapes skewed data into a uniform distribution by redefining how tables are loaded and partitioned. Specifically, we divide each partition into $m = \frac{partitionSize}{segmentSize}$ uniform segments, where `segmentSize` denotes the granularity of subdivision. A segment is uniquely identified by a tuple $(partitionId, offset) = \left(\left\lfloor \frac{i}{mn} \right\rfloor, (i \bmod mn) \cdot segmentSize\right)$. The pre-scan balancing operator assigns $segment_i$ to a scanner $scan_{i \bmod n}$ which reads the corresponding $segmentSize$-byte data block from the specified partition and offset via $seekg(offset)$ and $read(segmentSize)$ operations. Fine-grained segmentation thus enables more uniform data distribution.

Finally, we construct the cost estimation model of an operator by considering table size, table structure, and sampled running cost. We investigated several ML models and found that a linear regression is effective enough. The execution cost regressor is modeled as a linear regression formula: $T_{op} = k_1 * T_{sample_{op}} + k_2 * size(input_{op})$. The parameter vector (k_1, k_2) captures the table processing characteristics, obtained offline initially. We develop a random query generator and a runtime telemetry system that collects the online training datasets, serving as the basis for fitting and training the linear model. (k_1, k_2) is updated continuously via incremental learning[52, 90], capturing the operator running statistics. The training is performed on synthetic queries produced by our random query generator. Runtime telemetry collects execution metrics from these queries, which are then used to train a more robust and accurate basic model, which will be tuned online.

Our proposed table-aware learning-based cost estimator performs the majority of its estimation computations at the compute node and incurs little overhead (§5.9) to the storage layer. Upstream operators on the compute/storage layer only sample a small amount of data. This mechanism operates inside our query execution engine (§4.5).

Algorithm 1 Admission Control Algorithm

```
Input: Initial parameter A = 1.0, learning rate η > 0
opLog: Log operators' estimated and measured costs
procedure onlineUpdateN
    while opLog.size() > 0 do
        T_est, T_target ← opLog.pop()
        Compute the prediction error: e = T_target − T_est
        Compute the gradient: gradient = −2 · e · T_p
        Update A: A ← A − η · gradient
procedure scheduleOP(op)
    Estimate the base cost using the estimator (§4.2): T_p
    Estimate the pushdown cost: T_est = A · T_p
    Estimate the pushdown threshold: Thr
    if T_est ≤ Thr then
        Pushdown operator op
    else
        Pullup operator op
```

4.3 Admission Control for Operator Pushdown

The second issue (§3.4) highlights that storage nodes are less tolerant of computing resource variability under operator pushdown, especially when the storage layer is saturated. Existing solutions eagerly push operators down and can inadvertently slow down query execution. Our first technique (§4.2) can accurately estimate the operator execution time, but fails to consider the storage target contention. The key question is to figure out the computing availability of the storage node when making the pushdown decision.

However, accurately collecting statistics before query execution is challenging because resource utilization in the storage layer may change at runtime. We instead propose to use the historical prediction error between the estimated and actual operator execution time. Intuitively, a large estimation gap indicates that the target is overloaded, and pushdown operators experience great interference. Based on this, we develop an admission control mechanism following the *first principle (P1)*. It readjusts the estimated cost to dynamically control the number of concurrent pushdown operators staying within the interference-tolerant window (Figure 3). Specifically, an adaptive parameter (A) is introduced, $T_{\text{est}} = A \cdot T_p$, for readjusting the estimated cost. We use the squared error, defined as: $\mathcal{L} = \left(T_{\text{target}} - A \cdot T_p\right)^2$. To minimize this loss, we update A via gradient descent, where the gradient of $\mathcal{L}$ with respect to A is: $\frac{\partial \mathcal{L}}{\partial A} = -2 \cdot (T_{\text{target}} - A \cdot T_p) \cdot T_p$.

Algorithm 1 updates A using runtime telemetry of estimated and actual operator costs (§4.2). Based on these estimates, `scheduleOP` decides whether to push down or pull up operators according to cost thresholds. The learning rate η balances responsiveness and stability (§5.4).

4.4 DRAM-SSD Hybrid Ballooned Table

The second issue (§3.4) raises the DRAM limitation on Gen3-SA/GEN4-SA, caused by concurrent pushdown operators. Simply monitoring and throttling the runtime memory usage at the system level would hurt performance (§5.5).

We apply the *second principle (P2)* and trade excessive I/O bandwidth for scarce computing resources. TapDB introduces a new table structure (dubbed **HBTable**) to expand the memory interference-tolerant region. It stores temporary table tuples (e.g., streamed messages) in both DRAM and SSD and implicitly expands the memory capacity in the storage layer, leading to more consolidation. The idea behind HBTable is memory ballooning [29], it rebalances DRAM usage of all HBTables based on their operators' needs and only uses SSDs when necessary. HBTable is ephemeral, whose lifetime closely aligns with the operator.

HBTable Data Structure. HBTable comprises three components (Figure 6-a): *manifest*, *DRAM region*, and *SSD region*. Both DRAM and SSD parts are organized as a resizable circular buffer, in which the number of entries hinges on the operator's demands. *Manifest* handles table indexing and maintaining the metadata. It contains the following fields: (a) *Operator ID*, the ID of the operator using the table; (b) *Priority*, table processing order, discussed in §4.5; (c) *Tuple Metadata*, including column name, data type, and other subfields; (d) *SSD Head/Tail*, pointing to the start and end LBA (logical block address) of the SSD; (e) *DRAM Usage*, indicating the current size of the DRAM region. Each tuple consists of several attributes (*Tuple Metadata*), each of which corresponds to a column. Every attribute has its own name and data type, and combined with the column data (per-entry in DRAM/SSD Region) to form a complete tuple.

HBTable Intra-Table Primitives. HBTable provides two basic operations: *append* and *pop*. Upon an append, it first checks the system memory (organized as a memory pool) availability to see if there is enough space to hold the tuple in memory (Figure 6-b). When there is no more space, HBTable triggers a memory lending operation (discussed later) from other tables via ballooning. If the memory pool can accommodate the tuple, it is appended to the tail of the DRAM. Otherwise, the tuple is appended to the tail of the SSD. For a pop, HBTable first checks the `DRAM Usage` field and returns the head if there are pending in-memory entries. Otherwise, HBTable fetches tuples from the SSD region, which then releases the space with the head pointer updated.

HBTable Inter-Table Ballooning. We realize ballooning via two operations *lend* and *redistribute*. A high-priority table can proactively borrow memory space from low-priority ones when its DRAM usage runs below a predefined threshold (Figure 6-c). A HBTable that lends memory to others then spills out its tuples to the SSD. When the borrowed space is returned, we would redistribute the memory to all active HBTables in priority to their priorities. As discussed below, this approach expedites high-priority operator execution and avoids stalls on the critical path due to memory limitations.

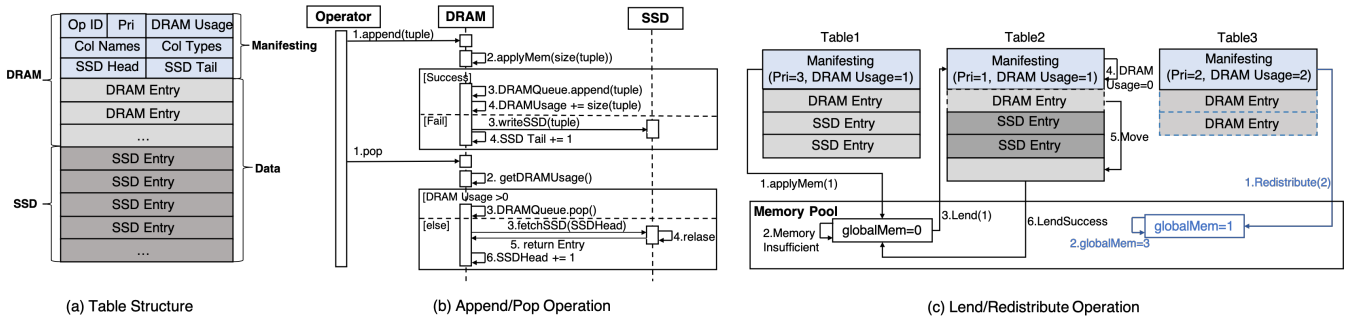


Figure 6. (a) depicts the HBTable data structure. (b)/(c) illustrate the execution flow of intra-table primitives and inter-table operations.

Algorithm 2 Critical Path-Driven Scheduler

```
1: readyQue: A queue for execution-ready operators;
2: activeQue: A queue for executing operators;
3: R_total: The dynamic available resource at current time;
4: procedure CPSched(op, pri)
5:     while ResourceDemand(op) > R_total do
6:         op_low, pri_low ← activeQue.head()
7:         if pri_low > pri then
8:             return
9:         else
10:            readyQue.append((op_low, pri_low))
11:            activeQue.pop()
12:            R_total ← R_total + ResourceDemand(op_low)
13:    activeQue.append(op)
14:    readyQue.pop(op)
15: procedure Schedule
16:    while True do
17:        if readyQue.len > 0 then
18:            op, pri ← readyQue.head()
19:            CPSched(op, pri)
```

4.5 Critical Path-driven Operator Scheduler

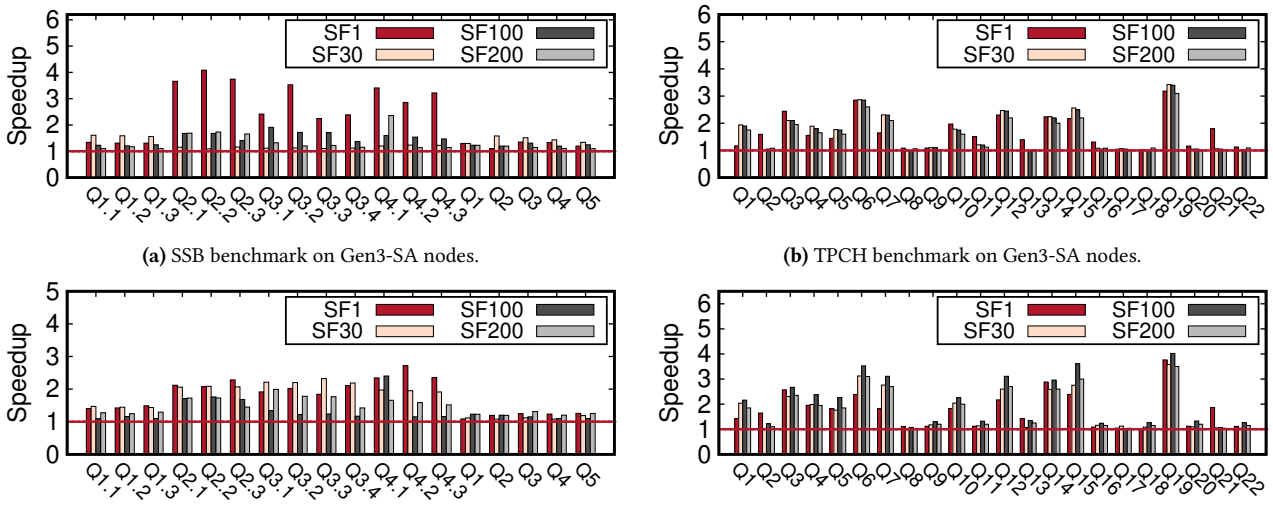
Section 3.5 shows that six common schedulers perform inconsistently. We formalize query scheduling as an NP-hard problem and propose a new critical-path–based scheduler that follows the *lazy evaluation (P1)* principle, dynamically allocating CPU cycles to read-to-execute operators based on resource availability.

Problem Formalization. We have a set of operators $O = \{o_1, o_2, \ldots, o_n\}$ representing the operators in the query DAG $G = (O, E)$. Each operator $o_i \in O$ has: (1) a resource requirement $R(o_i)$, the amount of CPU cycles needed by the operator to execute; (2) an execution time function $T(o_i, R(o_i))$, which describes the time taken to execute operator o_i given a resource allocation $R(o_i)$. The system has a limited amount of available resources R_{total}, and at any given time, the sum of resources allocated to all active operators cannot exceed this value: $\sum_{i \in \text{Active}(t)} R(o_i) \le R_{\text{total}}, \quad \forall t$.

Precedence Constraints. The operators are subject to precedence constraints based on the DAG structure. During execution, there is no strict dependency between operators, meaning that operator o_j is not required to start only after o_i completes. If an operator o_i (such as JoinProbe) depends on data from its producer o_j (like JoinBuild) before it can begin execution, the operator must wait for the completion of o_j, and temporarily buffer the data received from other producers $o_k \in Producer(o_i) \setminus \{o_j\}$. An operator may also face potential blockages due to dependencies on specialized data sources. An operator o_j must be completed before operator o_i can start: $S(o_i) = \max(C(o_j) + T(o_j, o_i), C(o_{k^*}) + T(o_{k^*}, o_i))$, where $o_{k^*} = \underset{o_k \in Producer(o_i) \setminus \{o_j\}}{\arg\min} C(o_k)$, $C(o_i)$ is the completion time of operator o_i, and $S(o_j)$ is the start time of operator o_j. Conversely, an operator o_i, such as NestedLoopJoin that compares each row from one table with every row from another table to find matching pairs, treats different producers equally, either storing or processing data from different upstream producers. It starts after any of its producers $o_j \in Producer(o_i)$ have completed or after all producers complete. This operator must satisfy: $S(o_i) = C(o_{j^*}) + T(o_{j^*}, o_i)$ where $o_{j^*} = \underset{o_j \in Producer(o_i)}{\arg\min} C(o_j)$ or $o_{j^*} = \underset{o_j \in Producer(o_i)}{\arg\max} C(o_j)$.

Objective. We aim to minimize makespan, the time required to complete all operators, *Minimize* $\max_i C(o_i)$, where $C(o_i)$ is the completion time of operator o_i. The problem can be formulated as a Job Shop Scheduling Problem, which is NP-hard[84, 102]. It can also be formulated to minimize the execution time of the critical path L as *Minimize* $\max_{o_i \in L} C(o_i)$. This is a classic critical path scheduling problem, which is inherently difficult to solve optimally [31, 81].

The Scheduling Algorithm. We divide all execution paths into the longest execution (critical) one (L) and the bypass ones (B). Based on the precedence constraints analysis from the problem formalization above, we further categorize the bypass paths into two types: urgent bypass paths (U) and common bypass paths (C). An urgent bypass path determines the readiness of the longest execution path. For example, if an operator $o_i \in L$ (e.g., JoinProbe) depends on data from its producer o_j (e.g., JoinBuild) before it can begin execution, then $o_j \in U$, while other producers(eg. NestedLoopJoin) belong to C. Note that we complete this classification by checking the execution logic of all involved operators, using it as prior knowledge. Thus, we schedule operators based



(a) SSB benchmark on Gen3-SA nodes.

(b) TPCH benchmark on Gen3-SA nodes.

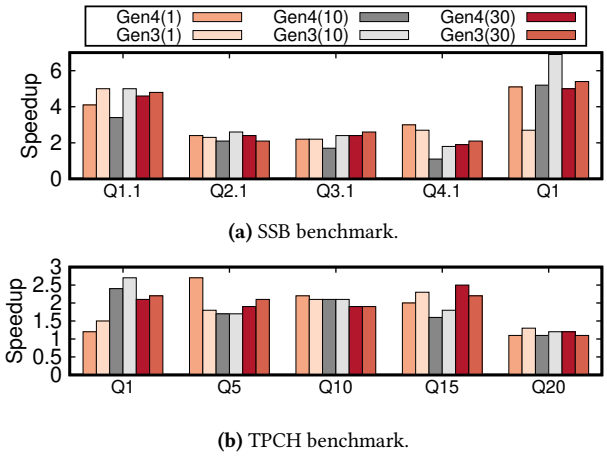
(c) SSB benchmark on Gen4-SA nodes.

(d) TPCH benchmark on Gen4-SA nodes.

Figure 7. Speedup achieved by TapDB over the cost-driven approach under SSB and TPCH benchmarks on Gen3-SA and Gen4-SA.

on the current execution progress and allocate resources between critical and bypass operators to minimize total execution time. Resources should be allocated in a way to make sure that operators on L are executed as quickly as possible, while bypass operators that affect operators in U are also given sufficient resources to maintain P's fast execution.

As shown in Algorithm 2, we assign different priorities to the operators on $L/U/C$, with L having the highest priority, while the priority between operators of the same category follows topological order. `CPSCHED` employs the strict priority scheduling policy and runs an operator when its resource requirements are met. Otherwise, we yield lower-priority operators in *activeQue* for higher-priority ones. The blocked operators release the computing and memory resources and are re-appended to the `readyQue`. The scheduler continuously attempts to run as long as the `readyQue` is not empty.



(a) SSB benchmark.

(b) TPCH benchmark.

Figure 8. Achieved speedup (compared with the non-pushdown case) by TapDB when running SSB and TPCH benchmarks.

5 Evaluation

5.1 Experimental Methodology

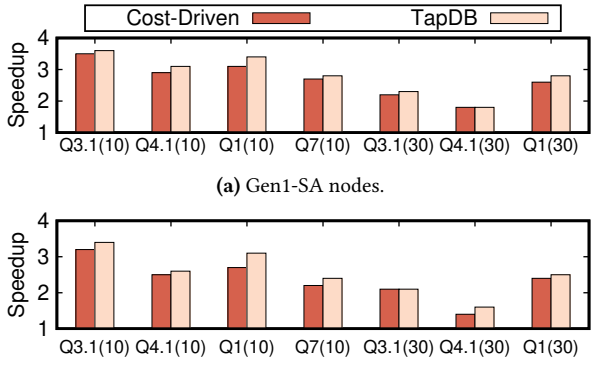
Implementation Details. TapDB extends FPDB [1] where we disable the caching, retain the pushdown logic, and integrate it with our approach. It uses a learning-based cost estimator (River [22]) and runs on CAF [21] with operator parallelization and CPU allocation.

Testbed Setup. Our testbed comprises an RDMA-capable rack with X86-based compute nodes and four generations of storage nodes. The compute server has two Intel Xeon CPUs, 96GB memory, and a dual-port Nvidia/Mellanox CX-5 NIC.

Application and Metric. We use the Star Schema Benchmark (SSB) [19] and TPC-H benchmark [20] under four scaling factors (1, 30, 100, 200). The workloads use a wide range of data types, including DECIMAL, INTEGER, DATE, TIMESTAMP, VARCHAR, and CHAR. Each table is sized into 50MB partitions in the Parquet [6] format. We mainly report the query execution time, and measure the speedup relative to the cost-driven (§3.1), FPDB adaptive [118], and non-pushdown approaches to quantify performance gains.

5.2 Overall Performance

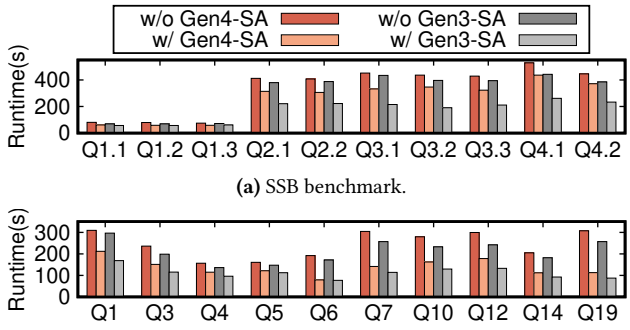
TapDB vs. Cost-Driven Approach. As shown in Figure 7, TapDB achieves average speedups of 2.3×/1.3×/1.4×/1.3× and 1.9×/1.7×/1.3×/1.5× for SSB under SF1/SF30/SF100/SF200 on GEN3-SA and GEN4-SA nodes, respectively. For TPCH, the speedups are 1.7×/1.7×/1.7×/1.6× and 1.7×/1.8×/2.1×/1.8× under the same scaling factors. This is because our proposed techniques dynamically adapt to resource variations via online pushdown optimization, unlike static cost-driven models that ignore table structures and runtime conditions. When network or storage dominates, TapDB makes the same pushdown decisions as the cost-driven approach, resulting in only marginal speedups (SSB Q1–5, TPCH Q17,18,20–22).



(a) Gen1-SA nodes.

(b) Gen2-SA nodes.

Figure 9. Achieved speedup by TapDB and the cost-driven approach when running SSB and TPCH benchmarks.



(a) SSB benchmark.

(b) TPCH benchmark.

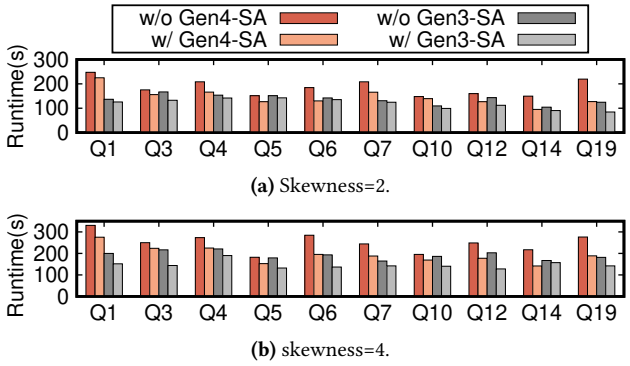
Figure 10. Query performance comparison with TLCE enabled (w/) and disabled (w/o). The scaling factor is 10.

Pushdown v.s. Non-Pushdown. As expected, TapDB outperforms the non-pushdown cases by 1.1–6.9× on Gen3-SA/Gen4-SA nodes (Figure 8), and only achieves 6.5%–8.1% improvements on Gen1-SA/Gen2-SA nodes (Figure 9).

5.3 Table-aware Learning-based Cost Estimator

Efficiency. Our table-aware learning-based cost estimator (TLCE) allows TapDB to learn the table's meta-features online by choosing representative samples. It then constructs the learning regression model to estimate the computing cost and enable better pushdown decisions. As shown in Figure 10, TLCE reduces the query runtime by an average of 22.7% and 36.3% across all evaluated SSB queries on Gen4-SA and Gen3-SA nodes, respectively. Similarly, it reduces the query runtime by an average of 42.1% and 45.4% across all evaluated TPCH queries on Gen4-SA and Gen3-SA nodes.

Cost Estimation Accuracy. We employ an incremental learning technique for fine-tuning by tracking operator runtime statistics. We collect both true and estimated execution times across all SSB queries with scale factors of 1, 10, and 30 on Gen3-SA/Gen4-SA nodes. The evaluation metric is Relative Mean Absolute Error (RMAE), which normalizes the Mean Absolute Error (MAE). The maximum, minimum, and average RMAE are 0.27, 0.07, and 0.16. This indicates that



(a) Skewness=2.

(b) skewness=4.

Figure 11. Query performance comparison with Pre-Scan Balancing Operator enabled (w/) and disabled (w/o).

the regression model built via input table size and execution time of table samples as features can accurately capture the table structure, leading to smaller estimation bias.

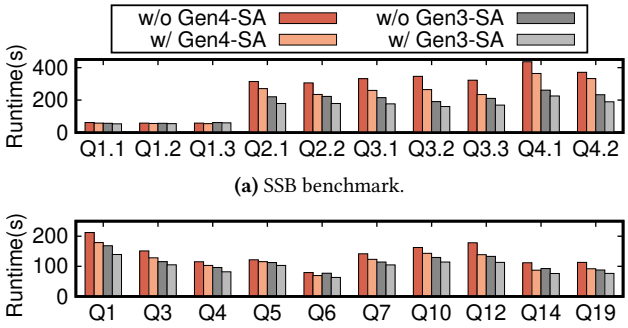
Input Estimation under Skewness. TapDB uses an eager input estimation scheme that approximates the actual input size of an operator. It also introduces a Pre-Scan Balancing Operator to shuffle skewed data, further improving estimation accuracy. To evaluate this, we generate TPC-H skewed data tables [23] with scale factors of 1, 10, and 30. Each column in these tables follows a Zipfian distribution. Our results show that the estimation accuracy using segment sizes of 9000 bytes is 98.6%, 98.1%, and 94.9% when skewness is 0, 2, and 4, respectively, showing an accuracy improvement of 19.6%, 62.7%, and 73.4% compared with the conventional scan. Under a scaling factor of 10, we examine the effectiveness of the Pre-Scan Balancing Operator. As shown in Figure 11, it reduces the query runtime by an average of 21.1% (12.9%) and 22.4% (23.0%) across all TPCH queries on Gen4-SA (Gen3-SA) when the skewness is set to 2 and 4, respectively.

5.4 Admission Control for Operator Pushdown

Efficiency. TapDB uses an admission control scheme to limit concurrent pushdown operations such that they can stay within the inference-free window. When running SSB and TPCH benchmarks, we find that the query performance is improved since less inference happens. As shown in Figure 12, our proposed scheme saves performance by 15.1% and 13.6% on average on Gen4-SA and Gen3-SA nodes for SSB queries, and 14.6% and 13.3% when running TPCH.

Sensitivity to Learning Rate (η). A large η leads to overshooting, preventing convergence and increasing query runtime by 35.19% and 40.7% on Gen3-SA and Gen4-SA nodes. Conversely, a small η results in slow adjustments, degrading query performance by 19.9% and 20.9% on Gen3-SA and Gen4-SA nodes. when η = 0.1 compared to the optimal η.

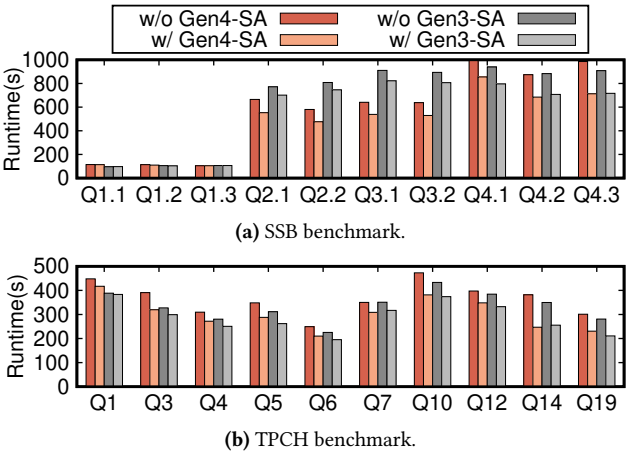
Cost Estimation Accuracy. The key reason why admission control works is that it uses an adaptive parameter A to dynamically adjust the estimated operator costs, which



(a) SSB benchmark.

(b) TPCH benchmark.

Figure 12. Query performance comparison between AC enabled (w/) and disabled (w/o). The scaling factor is 10.



(a) SSB benchmark.

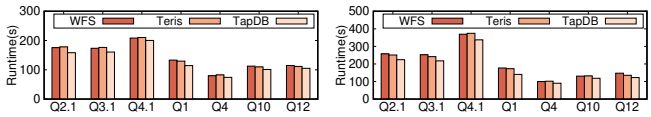
(b) TPCH benchmark.

Figure 13. Query performance comparison with HBTable enabled (w/) and disabled (w/o). The scaling factor is 30.

reflects the current system load, with higher values indicating a heavier load. To assess the accuracy of the adjusted estimated execution times, we evaluate them in the same way as §5.3 and show that the max, min, and avg. RMAE are 0.21, 0.04, and 0.19, respectively, compared to the values of 0.38, 0.06, and 0.27 when admission control is not applied.

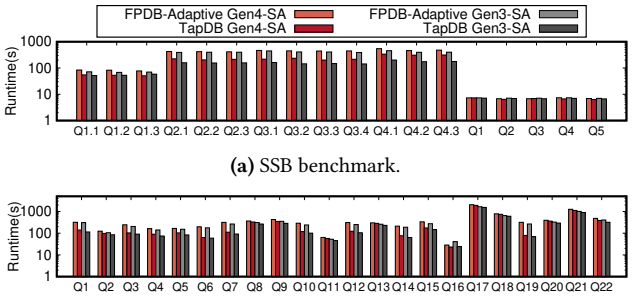
5.5 DRAM-SSD Hybrid Ballooned Table

TapDB uses HBTable to expand the memory interference-tolerant region, enabling pushdown operators with large memory footprints and allowing more operator consolidation. We disable the HBTable, control the number of concurrent operators, and induce query execution stalls when memory overflows, thus assessing the impact of HBTable. As shown in Figures 13-a/b, HBTable improves the query performance by an average of 14.9%/9.4% and 17.2%/13.8% across all evaluated SSB and TPCH queries on Gen4-SA/Gen3-SA nodes, respectively. The ballooned table uses extra I/O bandwidth to consolidate more operators and associated tables when exceeding the memory capacity, which leads to more pushdowns with little performance drops, making the CPU cycles of the storage nodes fully used.



(a) Gen3-SA nodes **(b)** Gen4-SA nodes

Figure 14. Runtime comparison among TapDB scheduler, WFS and Teris: The first 4 evaluated queries from SSB benchmark and the remaining ones from TPCH. The scaling factor is 10.



(a) SSB benchmark.

(b) TPCH benchmark.

Figure 15. The query performance comparison between TapDB and Adaptive Pushdown. The scaling factor is 10. y-axis is log scale.

5.6 Critical Path-driven Operator Scheduler

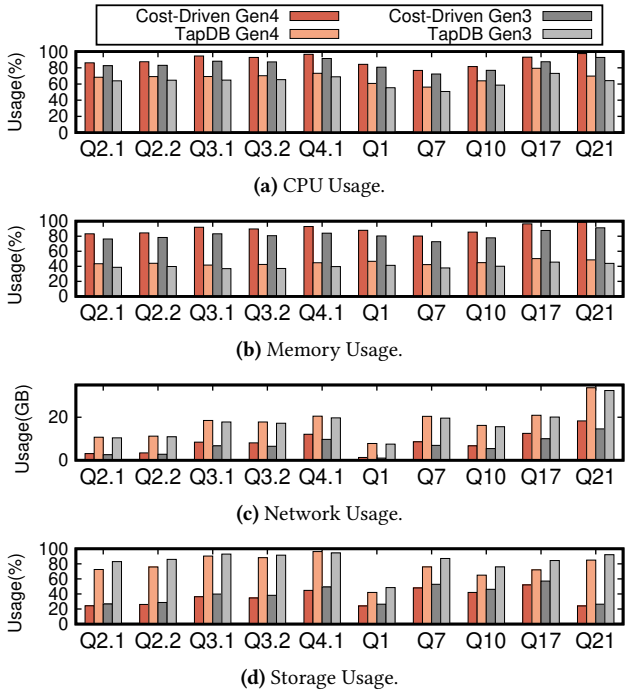
TapDB designs a critical path-driven operator scheduler that allocates available CPU cycles based on the longest execution path within the DAG. We compare with two schedulers (**WFS** and **Teris**) that perform the best in our characterization study. Figure 14 shows that our scheduler outperforms WFS/Teris by 13.1% and 11.5% on Gen4-SA nodes, and by 8.7% and 8.7% on Gen3-SA nodes, on average across 8 queries. By identifying the critical execution path and data dependencies, the scheduler can allocate CPU cycles more efficiently, ensuring that resources are utilized where they will have the greatest impact. This results in improved performance, particularly for complex, compute-intensive queries.

5.7 Compared with Adaptive Pushdown

The performance of FlexPushdownDB's adaptive pushdown scheme [118] is with execution time differences mostly within 1.4–4.2%. However, as shown in Figure 15, a significant performance gap remains when compared to TapDB: on the SSB benchmark, average speedups of 1.6× and 1.9× are achieved on GEN4-SA and GEN3-SA, respectively, while on TPC-H, speedups of 1.8× and 1.9× are attained.

5.8 Resource Usage

We compare resource utilization between the cost-driven approach and TapDB across CPU, memory, network, and storage. TapDB reduces CPU and memory usage by 21.2% and 42.6%, while increasing network traffic by 2× and storage I/O by 1.3× (Figure 16). This trade-off is enabled by pushdown admission control and the DRAM–SSD Hybrid Ballooned Table, which prevent storage overload and reduce memory



(a) CPU Usage.

(b) Memory Usage.

(c) Network Usage.

(d) Storage Usage.

Figure 16. The query average resource usage between Cost-Driven approach and TapDB evaluated using SSB (first five queries) and TPC-H (remaining queries). The scaling factor is 30.

pressure. As a result, TapDB shifts bottlenecks from compute to network and I/O on Gen3-SA and Gen4-SA nodes, trading abundant bandwidth for scarce compute resources.

5.9 System Overhead

TapDB incurs two major overheads. First, the table-aware learning-based cost estimator introduces sampling overhead, increasing total execution time by 0.7%/6.9%/0.8% at sampling ratios of 1%/10%/20%. The query runtime increases by 0.6%/0.4%/0.8% on Gen3-SA nodes and by 0.8%/0.7%/0.9% on GEN4-SA nodes. Second, the Pre-Scan Balancing Operator for skew mitigation increases Scan operator time by 3.8×/1.5×/1.4×, and raises query runtime by 1.6×/1.1×/1.1× on GEN3-SA nodes and by 1.6%/1.3%/1.1% on GEN4-SA nodes for segment sizes of 90/900/9000 bytes.

6 Related Work

Storage disaggregation. Disaggregated storage has been extensively explored and widely adopted [42, 78, 105, 122] with prior work focusing on remote storage protocol optimization [57, 62, 66], storage stack enhancements [25, 50, 58, 86, 124], programmable networks-based acceleration [36, 49, 53, 72, 75, 77, 79, 80, 82, 92, 94, 95, 98, 100, 101, 121, 125], and hardware offloading [55, 60, 73, 76, 103, 104, 115]. We analyze hardware trends in disaggregated storage and revisit database operator pushdown under modern architectures.

Pushdown database. Prior work explores computation pushdown to save network cost, such as data reduction [74, 117, 120] and early filtering [61]. FlexPushdownDB [118] proposes an adaptive pushdown model with storage-side request rejection, based on a static theoretical model that assumes data transfer dominates latency. Our characterization shows these policy-driven approaches work on Gen1-SA/Gen2-SA but fail on Gen3-SA/Gen4-SA, motivating four new mechanisms for next-generation storage hardware.

In-storage-device pushdown. In-storage-device pushdown executes computation inside storage devices to reduce host–storage data movement [40, 67]. Prior work [34, 45, 48, 68, 96, 99, 112] explores embedded CPUs, programmable controllers, and FPGA acceleration. TapDB relies on general-purpose CPUs and standard NVMe SSDs, complementing these approaches without assuming in-storage accelerators.

NVMe SSDs. People have exploited NVMe characteristics for system optimizations. including optimizing LSM-based KV stores [119] and maximizing bandwidth [70]. Besides, data placement techniques such as FDP [24] and ZNS SSDs improve performance and reliability by reducing write amplification and enhancing parallelism, with extensive studies on zone management and extensions [32, 37, 51, 59, 87, 88]. TapDB explores computation pushdown and query-level scheduling without modifying the storage stack.

Cost and cardinality estimation. Query cost and cardinality estimation are fundamental to query optimization. Prior work explores tree-based feature encoding [107], index-based sampling [69], hybrid histogram–learning models [113], and set-based neural architectures [64]. Our learning-based cost estimator follows a similar feature-encoding approach.

Query scheduling. Prior work improves scheduling efficiency using heuristic or learning-based methods. Graphene [47] and Decima [83] target DAG- and job-level scheduling, while LSched [97] applies reinforcement learning to database queries via query encoding. In contrast, our design explicitly allocates CPU cycles along the query critical path, making it better suited for compute-constrained storage nodes.

7 Conclusion

This paper rethinks pushdown database design under emerging storage disaggregation. Existing policy-driven solutions designs become ineffective due to the performance bottleneck shift and the increasing network and I/O density at the storage layer. We perform root cause exploration, identify three limitations, and then develop a pushdown database with four new mechanisms. Evaluations using SSB and TPCH benchmarks over our prototype demonstrate 1.3–2.3× performance improvement compared with prior solutions.

Acknowledgement

We would like to thank the anonymous reviewers for their comments and feedback. Ming Liu is supported in part by NSF grants CNS-2106199 and CNS-2212192.

References

[1] 2023. FlexPushdownDB. https://github.com/cloud-olap/FlexPushdownDB.

[2] 2024. AlloyDB for PostgreSQL. https://cloud.google.com/alloydb.

[3] 2024. Amazon Graviton Processors. https://aws.amazon.com/ec2/graviton/.

[4] 2024. Amazong S3 SELECT. https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-select-sql-reference-select.html.

[5] 2024. Apache Calcite. https://calcite.apache.org.

[6] 2024. Apache Parquet. https://parquet.apache.org.

[7] 2024. Chameleon Hardware. https://www.chameleoncloud.org/hardware/.

[8] 2024. CloudLab Hardware. https://cloudlab.us/hardware.php.

[9] 2024. Drill Query Plan. https://sparkbyexamples.com/spark/spark-execution-plan/.

[10] 2024. Enterprise and Datacenter Standard Form Factor (EDSFF). https://www.snia.org/forums/cmsi/knowledge/formfactors.

[11] 2024. Google Axion Processors. https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu.

[12] 2024. IBM DB2 Optimizer. https://www.ibm.com/docs/en/db2/11.5?topic=performance-query-optimization.

[13] 2024. Microsoft announces acquisition of Fungible to accelerate datacenter innovation. https://blogs.microsoft.com/blog/2023/01/09/microsoft-announces-acquisition-of-fungible-to-accelerate-datacenter-innovation/.

[14] 2024. NVMe over Fabrics (oF) Specification. https://nvmexpress.org/specification/nvme-of-specification/.

[15] 2024. Oracle Exadata Database Machine. https://www.oracle.com/engineered-systems/exadata/database-machine/.

[16] 2024. Oracle Query Optimizer. https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm.

[17] 2024. PostgreSQL Query Optimizer. https://www.postgresql.org/docs/current/planner-optimizer.html.

[18] 2024. Spark Execution Plan. https://sparkbyexamples.com/spark/spark-execution-plan/.

[19] 2024. Star Schema Benchmark(SSB. https://doris.apache.org/docs/dev/benchmark/ssb/.

[20] 2024. TPC-H Benchmark. https://www.tpc.org/tpch/.

[21] 2025. CAF: the C++ Actor Framework. https://github.com/actor-framework/actor-framework.

[22] 2025. River. https://github.com/online-ml/river.

[23] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 841–855. doi: 10.1145/3183713.3183761

[24] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan R. Nair, and Vivek Shah. 2025. Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1142–1160. doi: 10.1145/3689031.3696091

[25] Seunghyun An, Joontaek Oh, and Ming Liu. 2025. Server Chiplet Networking. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks (HotNets'25)*. 289–299.

[26] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. 1743–1756.

[27] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. doi: 10.1145/2723372.2742797

[28] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*. 49–67.

[29] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[30] Luis Andre Barroso and Jimmy Clidaras. 2022. *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Springer Nature.

[31] Harsh Bhasin and Nandeesh Gupta. 2018. Critical path problem for scheduling using genetic algorithm. In *Soft Computing: Theories and Applications: Proceedings of SoCTA 2016, Volume 1*. Springer, 15–24.

[32] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[33] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. 2020. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *18th USENIX conference on file and storage technologies (FAST'20)*. 29–41.

[34] 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 *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 29–41. https://www.usenix.org/conference/fast20/presentation/cao-wei

[35] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1849–1862.

[36] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*. 1–12.

[37] Jungyun Choi, Yuhun Jun, Jongseok Kim, Jinkyu Jeong, and Euiseong Seo. 2024. An Adaptive Zone-Grouping Scheme Enabling General-Purpose File Systems on ZNS SSDs. In *Proceedings of the 17th ACM International Systems and Storage Conference* (Virtual, Israel) *(SYSTOR '24)*. Association for Computing Machinery, New York, NY, USA, 132–145. doi: 10.1145/3688351.3689151

[38] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. *Noise reduction in speech processing* (2009), 1–4.

[39] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to Be Fast, Available, and Frugal in the Cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. 1463–1478.

[40] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230. doi: 10.1145/2463676.2465295

[41] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, Vol. 86. 209–219.

[42] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[43] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types

[44] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. 2022. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*. 1249–1266.

[45] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, and Myoungsoo Jung. 2024. DockerSSD: Containerized In-Storage Processing and Hardware Acceleration for Computational SSDs. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 379–394. doi: 10.1109/HPCA57654.2024.00036

[46] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466. doi: 10.1145/2740070.2626334

[47] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 81–97. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_graphene

[48] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: a framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 153–165. doi: 10.1109/ISCA.2016.23

[49] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 916–929.

[50] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. 2023. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*. 1012–1027.

[51] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 147–162. https://www.usenix.org/conference/osdi21/presentation/han

[52] Haibo He, Sheng Chen, Kang Li, and Xin Xu. 2011. Incremental Learning From Stream Data. *IEEE Transactions on Neural Networks* 22, 12 (2011), 1901–1914. doi: 10.1109/TNN.2011.2171713

[53] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*. 33–47.

[54] Peter Henderson and James H Morris Jr. 1976. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. 95–103.

[55] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. 2024. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. 297–312.

[56] John L Hufferd. 2003. *iSCSI: The universal storage connection*. Addison-Wesley Professional.

[57] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*.

[58] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[59] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. 2024. ZMS: Zone Abstraction for Mobile Flash Storage. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 173–189. https://www.usenix.org/conference/atc24/presentation/hwang

[60] Sheng Jiang and Ming Liu. 2025. Building an Elastic Block Storage over EBOFs Using Shadow Views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1137–1153.

[61] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.* 9, 12 (aug 2016), 924–935. doi: 10.14778/2994509.2994512

[62] Yuyuan Kang and Ming Liu. 2025. Understanding and Profiling NVMe-over-TCP Using ntprof. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1117–1136.

[63] Robert W Kembel. 2009. *Fibre Channel A Comprehensive Introduction*. Northwest Learning Assoc.

[64] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[65] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*. 1–15.

[66] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[67] Gunjae Koo, Kiran Kumar Matam, Te I., H.V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 219–231.

[68] Miryeong Kwon, Donghyun Gouk, Eunjee Na, Jiseon Kim, Junhee Kim, Hyein Woo, Eojin Ryu, Hyunkyu Choi, Jinwoo Baek, Hanyeoreum Bae, et al. 2025. Containerized In-Storage Processing and Computing-Enabled SSD Disaggregation. *IEEE Micro* (2025).

[69] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[70] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. doi: 10.1145/3341301.3359628

[71] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2017. Wander Join and XDB: Online Aggregation via Random Walks. *SIGMOD Rec.* 46, 1 (may 2017), 33–40.

[72] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[73] Xiao Li, Zerui Guo, Yuebin Bai, Mehash Ketkar, Hugh Willkinson, and Ming Liu. 2025. Understanding and Profiling CXL.mem Using PathFinder. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM'25)*.

[74] Shu Lin, Arunprasad P. Marathe, Per-Åke Larson, Chong Chen, Calvin Sun, Paul Lee, Weidong Yu, Jianwei Li, Juncai Meng, Roulin Lin, Xiaoyang Chenxi, and Qingping Zhuxii. 2022. Near Data Processing in Taurus Database. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1662–1674. doi: 10.1109/ICDE53745.2022.00170

[75] Ming Liu. 2020. *Building Distributed Systems Using Programmable Networks*. University of Washington.

[76] Ming Liu. 2023. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*. 118–126.

[77] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. 318–333.

[78] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. 2020. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 305–323.

[79] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 795–809.

[80] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 363–378.

[81] Errol L. Lloyd. 1980. Critical path scheduling of task systems with resource and processor constraints (Extended Abstract) *(STOC '80)*. Association for Computing Machinery, New York, NY, USA, 436–446. https://doi.org/10.1145/800141.804692

[82] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2017. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack*.

[83] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. doi: 10.1145/3341302.3342080

[84] Monaldo Mastrolilli and Ola Svensson. 2011. Hardness of Approximating Flow and Job Shop Scheduling Problems. *J. ACM* 58, 5, Article 20 (Oct. 2011), 32 pages. doi: 10.1145/2027216.2027218

[85] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*. 753–766.

[86] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*. 106–122.

[87] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. 461–477.

[88] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2024. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage* 20, 3, Article 16 (June 2024), 41 pages.

[89] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New query optimization techniques in the Spark engine of Azure synapse. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 936–948. doi: 10.14778/3503585.3503601

[90] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, et al. 2021. River: machine learning for streaming data in python. *The Journal of Machine Learning Research* 22, 1 (2021), 4945–4952.

[91] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 663–676. doi: 10.14778/3184470.3184471

[92] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 663–679.

[93] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*. 66–85.

[94] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th*

ACM Workshop on Hot Topics in Networks (HotNets'20). 16–22.

[95] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 772–787.

[96] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 379–394. https://www.usenix.org/conference/atc19/presentation/ruan

[97] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1228–1242. doi: 10.1145/3514221.3526158

[98] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 740–755.

[99] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 67–80. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri

[100] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 1–16.

[101] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 685–699.

[102] Pankaj Sharma and Ajai Jain. 2016. A review on job shop scheduling with setup times. *Proceedings of the Institution of Mechanical Engineers, Part B* 230, 3 (2016), 517–533. arXiv:https://doi.org/10.1177/0954405414560617 doi: 10.1177/0954405414560617

[103] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*.

[104] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 783–799. https://www.usenix.org/conference/osdi24/presentation/shu

[105] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. 2023. Disaggregated RAID Storage in Modern Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 147–163. doi: 10.1145/3582016.3582027

[106] Malcolm Singh and Ben Leonhardi. 2011. Introduction to the IBM Netezza Warehouse Appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. 385–386.

[107] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (nov 2019), 307–319. doi: 10.14778/3368289.3368296

[108] Dennis Treder-Tschechlov, Manuel Fritz, Holger Schwarz, and Bernhard Mitschang. 2024. Ensemble Clustering Based on Meta-Learning and Hyperparameter Optimization. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 2880–2892. doi: 10.14778/3681954.3681970

[109] Michael Ubell. 1985. The intelligent database machine (idm). In *Query processing in database systems*. 237–247.

[110] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*. 1041–1052.

[111] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 449–462.

[112] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: an intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. doi: 10.14778/2732967.2732972

[113] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *PACMMOD*.

[114] Jianguo Wang Xi Pang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. In *Proceedings of the 2024 ACM International Conference on Management of Data (SIGMOD'24)*.

[115] Xincheng Xie, Wentao Hou, Zerui Guo, and Ming Liu. 2025. Building Massive MIMO Baseband Processing on a Single-Node Supercomputer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1221–1242.

[116] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2, Article 136 (jun 2023), 28 pages.

[117] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2101–2113.

[118] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (July 2024), 1643–1670. doi: 10.1007/s00778-024-00867-8

[119] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[120] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1802–1805.

[121] Jie Zhang, Hongjing Huang, Xuzheng Chen, Xiang Li, Jieru Zhao, Ming Liu, and Zeke Wang. 2025. RpcNIC: Enabling Efficient Datacenter RPC Offloading on PCIe-attached SmartNICs. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA'25)*. 1379–1394.

[122] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. 2024. What's the story in EBS glory: evolutions and lessons in building cloud block store. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies* (Santa Clara, CA,

USA) *(FAST '24)*. USENIX Association, USA, Article 17, 16 pages.

[123] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2102–2114. doi: 10.1145/3448016.3457291

[124] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 37 (June 2022), 30 pages.

[125] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 427–449.

[126] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. 1525–1539.