A-Scan: Efficient Scale-up Analytics via Throughput-Guided Data Movement

Hamish Nicholson* Aunn Raza† Viktor Sanca† Anastasia Ailamaki*
*EPFL, Lausanne, Switzerland †Oracle, Switzerland / United States
{hamish.nicholson, anastasia.ailamaki}@epfl.ch {aunn.raza, viktor.sanca}@oracle.com
†Work done while the authors were at EPFL.

***Abstract*—Modern scale-up analytical systems, from single servers to rack-scale deployments, face increasing complexity in data movement across their storage hierarchies, NUMA nodes, and high-speed interconnects like CXL and PCIe. The performance gap between optimal and suboptimal data-movement strategies can be substantial, as the best approach depends on complex interactions between hardware capabilities (storage bandwidth, interconnect speeds, near-storage compute) and workload characteristics such as query selectivity, access patterns, and data volume. While existing systems employ data-movement strategies fixed at query planning time, the growing diversity of storage technologies, memory topologies, and analytical workloads makes such static approaches brittle, demanding adaptive runtime selection.**

We propose Throughput-Guided Data Movement that uses end-to-end operator-pipeline throughput as the adaptation signal to select how and where bytes are realized at query runtime. Throughput implicitly captures the interplay between hardware and query properties without an explicit performance model for data movement. We integrate throughput-guided data movement into a leaf operator, Adaptive Scan (A-Scan), that encapsulates data-movement within the operator and selects among them at runtime to maximize observed end-to-end throughput. A-Scan keeps the query plan fixed and presents the same rowset interface to its parent operator while internally controlling how and where data are realized across nodes and memories. We validate A-Scan by implementing it in a push-based, code-generating analytical engine, showing that it approaches the per-query best static choice across tested configurations and interference conditions, without hardware-specific tuning.

I. Introduction

The traditional view of storage as a hierarchy, from CPU registers through memory to persistent storage, no longer captures the reality of modern scale-up systems. Single-server and rack-scale topologies are increasingly complex with compute, memory, and storage devices separated by interconnects. Data movement across a networked topology is well studied in distributed systems, but as scale-up architectures adopt technologies like Compute Express Link (CXL) [1], [2] and increase internal heterogeneity, the same trade-offs now determine performance within individual servers and racks. Scale-up analytical systems, often designed in the CPU-memory-(hard disk) era, lack the flexibility to adapt their data-movement strategies to these diverse internal topologies, resulting in significant performance penalties when static approaches misalign with hardware capabilities and workload characteristics.

Query performance in scale-up systems hinges on where bytes are realized relative to bandwidth boundaries. We call a scale-up node a set of hardware resources that have high internal bandwidth relative to external bandwidth; boundaries occur at bandwidth bottlenecks. In a two-socket server, each socket is a node: local DRAM and often local storage is faster than the inter-socket link. In heterogeneous CPUs/GPUs (e.g., Grace/Hopper), CPU and GPU form distinct nodes with coherent memory; in rack-scale systems, many nodes may share an address space. Query engines therefore face a choice when accessing data: (i) direct transfer of blocks to the compute node, (ii) staging data into storage-local memory and letting the compute node access it at fine granularity, or (iii) pushdown of selective work to storage-local compute before moving data. Which choice wins depends jointly on selectivity and per-node resources (interconnect bandwidth, memory hierarchy, available compute). Figure 1 contrasts these three choices on two Taxi queries; Section II analyzes the underlying bottlenecks.

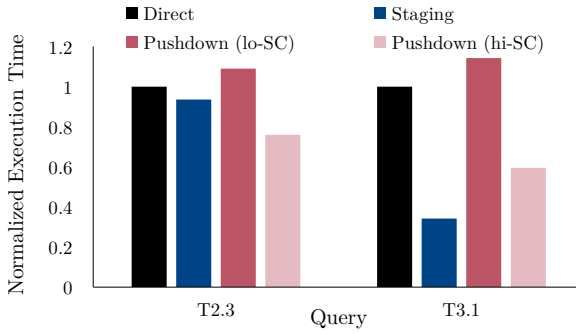


Fig. 1: The fastest data-movement strategy flips by query and hardware. On two NYC Taxi queries, a single selective filter (T2.3) favors pushdown when near-storage compute does not bottleneck filter evaluation, whereas a filter+selective-join (T3.1) favors staging by reducing interconnect traffic to a sparse subset of columns. No single static strategy dominates; the winner depends on selectivity, column access patterns, and near-storage compute.

Ideally, each query would use the data-movement strategy that minimizes execution time on the available hardware. Any policy that always loads full blocks, always uses fine-grained access, or always pushes down operators will be suboptimal

for some hardware–workload combinations. Predicting the best strategy at compile time is especially hard in modern engines because performance emerges from shared bottlenecks contended by parallel workers, such as interconnect links, memory controllers, caches, and storage bandwidth. The effective throughput of a strategy depends not only on per-operator/per-thread costs but also on how these resources are multiplexed, which can change non-linearly with degree of parallelism and over time. Modeling these interactions exhaustively is brittle and requires continual retuning. Instead, we make the choice at *query runtime*, where the outcome of workload behavior and resource contention can be *observed rather than predicted*, allowing the system to select the most effective strategy on the fly.

This paper introduces Throughput-Guided Data Movement: at query runtime select among candidate data-movement strategies by maximizing end-to-end operator-pipeline throughput. End-to-end throughput captures both bottlenecks at the pipeline's leaf operator (e.g., near-storage compute and memory bandwidth), bottlenecks in downstream operators (e.g., accessing staged data over an interconnect), and contention on shared resources across parallel workers. Using a single throughput objective makes data-movement strategies directly comparable and remains portable across topologies and implementations without per-platform weights or re-tuning. We focus on the critical path of transferring data from storage to compute during query execution, particularly for scan-heavy analytical workloads where efficient data movement is crucial for query performance.

The **contributions** of this paper are as follows:

- We introduce Throughput-Guided Data Movement, which uses *end-to-end operator-pipeline throughput* as the signal, capturing shared-resource bottlenecks local counters miss.
- We realize A-Scan in a parallel code-generating push-based engine: by aligning the control and back-pressure boundary at the scan's output queue, A-Scan obtains per-mode end-to-end throughput without instrumenting consumers.
- Our experiments using the Star Schema Benchmark [3] and the NYC Taxi dataset [4] show that A-Scan improves query time by up to $2.78\times$ over direct transfers, approaches the per-query best static mode while showing robustness to interference, hardware setups, and query characteristics.

II. The Art of Data Movement

In this section, we describe three strategies for data movement and show that query performance depends on the strategy chosen, motivating runtime adaptation.

A. Data-Movement Strategies

We focus on three fundamental approaches, each making different assumptions about available hardware capabilities. Firstly, at one end of the spectrum are direct transfers, which require only basic storage connectivity, making no assumptions about near-storage memory or compute resources. Then, data staging requires near-storage memory, which is used to enable more flexible data access patterns. Finally, operator pushdown represents the fullest utilization of near-storage capabilities, requiring both memory and compute resources near storage. These strategies span the spectrum of hardware capabilities in modern storage-compute architectures and demonstrate the sensitivity of data movement performance to both query and hardware characteristics.

Direct transfer moves data directly from storage to the compute node's memory over the interconnect. While simplest to implement, it cannot leverage such capabilities when available, missing opportunities to reduce data movement through selective access or intermediate processing.

Data staging loads data to an intermediate location (e.g., near-storage memory) before the compute device either accesses it there or copies it to compute-local memory. The intermediate medium's properties, such as support for cache-line-granular access, can better align with workload access patterns than direct block-based transfers. Despite the additional movement step, staging can reduce total data transfer across a bottleneck when workloads benefit from selective access patterns. For example, in GPU-accelerated analytical query processing, the input data must be transferred to the GPU, and the GPU's interconnect becomes the main bottleneck [5]. Staged lazy transfers for GPUs consuming NVMe resident data are one example of data staging that alleviates this bottleneck [6].

Staging is sensitive to hardware characteristics; it depends on the achievable bandwidth of fine-grained memory accesses over the interconnect and the minimum access granularity. For example, across a CPU-CPU interconnect, most CPUs can load individual cache lines (64B on x86), while in the previous example of GPUs accessing system memory, the granularity was observed to be 128B for an NVIDIA V100 [7]. The minimum access granularity determines the access selectivity required to move less data over an interconnect; in the CPU example, if four bytes are required in each cache line, the same amount of data will be moved over the interconnect as accessing every byte in each cache line.

Operator pushdown moves the execution of initial operators in a processing pipeline, particularly those that reduce the volume of data to upstream operators such as filters or aggregates, closer to the data. This relies on having compute available near the data, which is beneficial when both the near-data compute has higher bandwidth to the data and the bottleneck in query processing is the movement of data to the primary compute over the network or interconnect. Operator pushdown is a well-studied topic, and has been used in the context of Smart Disks [8]–[11], processing-in-memory (PIM) [12], [13], cloud databases [14]–[17], and DB appliances [18]–[20].

The performance impact of pushdown depends on the compute available on storage nodes, the bandwidth differential between near-storage data access and data transfer over the interconnect, and the computational intensity and data reduction factor of pushed-down operators. With limited compute on the storage node or more complicated operators, evaluating the pushed-down operators can become a larger bottleneck than moving the data directly over the interconnect. In contrast, when near-storage compute is not a bottleneck and the

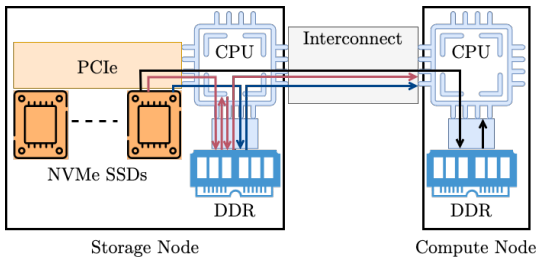


Fig. 2: Example topology with both near-storage compute and memory showing three possible data-movement strategies: direct (black), staging (blue) and pushdown (red)

near-storage bandwidth exceeds the interconnect bandwidth, pushdown will improve performance over direct transfers.

B. Impact on Query Performance

We demonstrate the impact of data-movement strategy on query execution times using two queries from the NYC yellow taxi dataset [4], stored using a columnar layout. Query T2.3 filters taxi trips by fare amount and computes average speed grouped by day of week. Query T3.1 filters trips for credit card payments, then joins with the zone lookup table filtered for specific zones to calculate four aggregates for trips with drop-off locations in those zones grouped by trip distance ranges.

We execute these queries using three data-movement strategies on a system with storage and compute nodes connected via a processor interconnect (Figure 2). The platform is a dual-socket server with one socket serving as a storage node (with NVMe drives) and the other as a compute node, simulating rack-scale storage-compute disaggregation. This setup also models CXL memory sharing between nodes with inter-socket memory access performance characteristics, an approach adopted in several recent studies of disaggregated memory, as commercial CXL hardware that supports memory sharing is not widely available [21]. Section V-A explains the complete experimental setup. By default, query operators run on the compute node. With direct transfers, the data is directly loaded from the drives on the storage node to compute node memory. For pushdown, we evaluate two configurations showing the impact of the amount of near-storage compute: hi-SC and lo-SC using 16 and 4 threads, respectively, for filter/projection evaluation on the storage node.

Figure 1 shows the execution time for both queries using all three data-movement strategies normalized to the direct transfer execution time. Direct transfers bottleneck on interconnect bandwidth for both queries. For both T2.3 and T3.1, pushdown with lo-SC is slower than direct because the processing throughput of the filter and project operations become the bottleneck and the interconnect becomes underutilized. Pushdown with hi-SC, using more compute on the storage node, improves performance and shifts the bottlenecks. For T2.3, materializing intermediate results from pushed-down operations to memory causes high CPU cache pressure. This cache pressure frequently evicts kernel data structures from cache, making the kernel IO path (read using io_uring with `O_DIRECT`) the limiting factor in performance. Using non-temporal stores to materialize the data achieves a 1.25× speedup for pushdown with hi-SC on T2.3 compared to standard stores used in Figure 1. In contrast, T3.1 using pushdown with hi-SC saturates the memory bandwidth on the storage node. T3.1 scans 9 columns compared to T2.3's 4 columns, and also has a less selective filter than T2.3, requiring more data materialization after pushed-down operations. Staging is slightly faster than direct on T2.3, but offers limited benefit. While staging accesses only required values in the fare amount column for filtering and subsequent columns for passing rows, each access loads an entire cache line over the interconnect. Thus, most cache lines still transfer despite the moderately selective access pattern. Staging outperforms direct and pushdown for T3.1 because after filtering payment type, the join with zone lookup requires only `DOLocationID` values. Since the zone filter is highly selective, few rows match, and other columns load only for these matching rows, substantially reducing data movement. Unlike T2.3 where most cache lines transfer anyway, T3.1's selective join avoids loading most cache lines from non-join columns, substantially reducing data movement.

These results demonstrate that no single static strategy dominates. The effectiveness of any static data-movement strategy is highly dependent on both query characteristics and hardware conditions. The direct approach, while simple, fails to leverage available near-storage resources and becomes bottlenecked by interconnect bandwidth. Data staging can reduce data movement through selective access patterns, but its effectiveness depends critically on query selectivity and join patterns; as shown by the difference between T2.3 (where most cache lines transfer regardless) and T3.1 (where selective joins avoid most data transfers). Similarly, operator pushdown can leverage near-storage compute to reduce data volume over the interconnect, but can introduce new bottlenecks: compute throughput with limited threads (lo-SC), CPU cache pressure from materialization (T2.3 with hi-SC), or memory bandwidth (T3.1 with hi-SC). The best-performing data-movement strategy depends on complex interactions between query selectivity, access patterns, and hardware bottlenecks.

C. Selecting a Strategy

Ideally, we would use a performance model to decide which data-movement strategy to use at query planning time. However, as our experiments demonstrate, predicting the optimal strategy requires modeling complex interactions between query characteristics and hardware bottlenecks. For example, a model would need to predict not only data reduction factors but also subtle effects like cache-line granularity impacts (T2.3 staging), join selectivity patterns (T3.1 staging), CPU cache pressure from materialization (T2.3 pushdown with hi-SC), and shifting bottlenecks as compute resources vary (lo-SC vs hi-SC). Further, seemingly minor implementation details, such as using non-temporal stores can yield 1.25× performance differences. Such a model would need updates as operator

implementations evolve and hardware configurations change, making static optimization brittle and costly to maintain.

We therefore select at runtime within the leaf operator rather than at plan time. Section III presents *Adaptive Scan (A-Scan)*, which preserves a fixed plan and uniform rowset interface while deciding during execution how and where to realize bytes. Decisions are guided by an end-to-end throughput signal: the measured rate from assigning a unit of scan work to its consumption by the parent operator. This reflects both the data-movement strategy throughput and its impact on downstream operator throughput. The end-to-end rate subsumes effects of selectivity, interconnect and memory bandwidth, and implementation details, without requiring an explicit model.

III. Throughput-Guided Data Movement

A. Runtime Strategy Selection via End-to-End Throughput

Section II showed that the fastest data-movement strategy depends on query selectivity, column access patterns, near-data compute, and shared bottlenecks; no single static choice dominates. We therefore adopt *throughput-guided data movement*: defer selection to runtime and choose the strategy that maximizes measured end-to-end operator-pipeline throughput, i.e., the rate at which work units flow through the pipeline.

Signal. We measure throughput as the rate of pipeline progress: work units completed per second, where a work unit represents a fixed-size chunk of the input data (e.g., a range of table rows to process). The measurement spans the full operator pipeline: from initiating work-unit processing (data loading, any required data transfers, operator evaluation) through to the pipeline breaker. This single end-to-end rate makes strategies directly comparable and aggregates the net effect of:

- **Data movement costs**: Storage bandwidth, interconnect latency and bandwidth, memory placement decisions.
- **Operator placement effects**: Whether operations execute near-storage or on compute nodes, and the relative performance and resource availability at each location.
- **Downstream bottlenecks**: If operator execution cannot keep pace (e.g., complex joins, remote memory access stalls), back-pressure propagates upstream, inflating measured work-unit completion times.
- **Resource contention**: Shared bottlenecks (interconnect links, memory controllers, storage bandwidth) surface as throughput degradation across parallel workers.

When a strategy reduces bytes crossing the interconnect (e.g., pushdown filtering), measured throughput rises; when a strategy suffers from access granularity inefficiencies (e.g., staging with cache-line-granular access where most cache lines transfer despite sparse column reads, as in T2.3 from Section II), throughput falls; when bottlenecks shift from interconnect to near-storage memory bandwidth (e.g., T3.1's pushdown materializing many columns after filtering), the new constraint surfaces as a throughput change. These performance effects manifest in measured throughput without requiring us to understand their underlying causes; a strategy that suffers from cache pressure will simply complete work units more slowly than alternatives under current conditions.

The challenge in realizing runtime strategy selection lies in obtaining reliable throughput measurements and translating them into selection decisions. The following subsections address where to implement this methodology (Section III-B), how strategies interact with the rest of the query plan (Section III-C), and the policy for measuring and selecting strategies during execution (Section III-D).

B. Operator-Level Realization

Runtime strategy selection requires infrastructure to instantiate multiple physical strategies, measure their throughput during execution, and route work units to the selected strategy. This could be realized at multiple system levels: query planner logic, middleware interceptors, or storage-layer shims. We encapsulate the methodology within a leaf table-scan operator, which confines adaptation complexity to a single component while preserving standard query plan structure.

Isolation of complexity. All adaptation logic (strategy management, throughput measurement, work routing, selection policy) resides within a single operator. Parent operators consume a uniform stream of tuple batches without awareness of active strategies or switches.

Preserves optimizer reasoning. Query optimizers make plan decisions based on cardinality estimates and cost models. Confining adaptation to the scan operator's internals keeps the logical plan fixed while only physical data movement varies at runtime, avoiding plan instability.

Independent per-pipeline adaptation. Each scan operator adapts independently based on its pipeline's observed throughput. In multi-table queries, different scans can simultaneously use different strategies (e.g., pushdown for selective fact table scans, direct for small dimensions) without coordination.

We call this operator *Adaptive Scan (A-Scan)*. A-Scan targets scan-heavy analytical workloads in pipelined query execution engines [22], [23].

C. Interface & Semantics

A-Scan encapsulates multiple data-movement modes behind a table-scan interface. It targets pipelined (streaming) execution [22]–[27]. Internally, A-Scan assigns scan slices to modes and delivers rowsets (materialized tuple batches) to its parent via a bounded output queue; a dispatch shim consumes this queue and drives parent execution (Section IV-A). All adaptation signals are computed over slice consumption, while the parent sees only rowsets. Adaptation is strictly intra-operator: operators above A-Scan receive the same stream of rowsets regardless of the active mode. The plan remains fixed; only where and how bytes are realized changes.

Terminology. We distinguish two notions: *scan slice*: the unit of work A-Scan assigns to a mode (a bounded portion of the table identified by storage/range metadata); a slice yields zero or more passing tuples. *rowset*: the unit of delivery to the parent operator (a materialized batch of tuples with column-major memory layout produced by one or more completed slices).

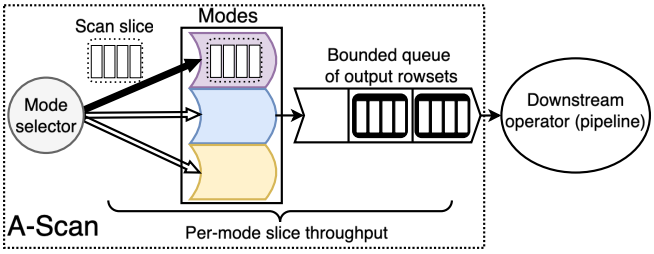


Fig. 3: A-Scan encapsulates multiple data-movement modes (Direct, Staging, Pushdown) and emits a uniform stream of rowsets. The mode selector assigns slices (dashed) to one mode at a time during sampling, and each mode emits rowsets (solid) into a bounded output queue. A-Scan measures slice throughput from assignment to downstream consumption (brace) to obtain an end-to-end signal that reflects shared-resource bottlenecks; the queue couples downstream back-pressure to slice admission.

Inputs. A-Scan is created with (i) a table handle, (ii) a projection (set of columns), (iii) optional pushed predicates P (owned by A-Scan), and (iv) optional conservative runtime filters R (e.g., Bloom filters).

Outputs. A-Scan emits a stream of *rowsets* with the declared projection. Each rowset represents a batch of tuples in column-major layout. The parent accesses column values through the rowset's interface, which abstracts the physical data location and realization strategy.

Predicate evaluation contract. A-Scan guarantees that emitted rowsets contain only tuples satisfying the pushed predicate set P, independent of the active mode. Parent operators need not re-apply P for correctness (they may apply additional predicates not in P). How this guarantee is realized differs by mode:

- **Direct and staging modes:** Predicates in P are evaluated on the compute node when the parent accesses column values through the rowset interface. This deferred evaluation allows selective access patterns in the parent to reduce data movement in staging mode.
- **Pushdown mode:** Predicates in P are evaluated on the storage node before data transfer. The parent accesses pre-filtered data.

The parent's code is unchanged across modes; it accesses column values through the same rowset interface, but the physical behavior differs.

Runtime filters. In addition to exact pushed predicates P, A-Scan can optionally consume conservative *runtime filters* R (e.g., Bloom filters) produced by other pipelines; these may admit false positives but not false negatives. Using R does not change the correctness contract for P: every emitted rowset still satisfies P in all modes. Because our goal is to reduce data movement, A-Scan exploits R only in modes that can apply it near data (e.g., pushdown), where it can reduce materialized bytes before crossing the interconnect; in direct and staging, applying R after transfer does not save interconnect traffic.

Progress and downstream back-pressure. A-Scan exposes a bounded rowset output queue to its parent. A rowset occupies one slot until the parent consumes it. Internally, modes process scan slices; once enough passing tuples accumulate to form a rowset, the mode enqueues it (blocking until a slot becomes available if the queue is full). If the parent falls behind, the rowset queue fills, propagating back-pressure to the active mode. Slices are never duplicated or speculatively processed: each slice is assigned once.

D. Mode Selection Policy

The mode selector implements a sample-exploit-resample strategy (Algorithm 1). Routing slices by concurrent back-pressure across modes is unbiased only under performance isolation between modes (sending work to one mode does not alter the throughput of the others). With data movement, that assumption often breaks: modes contend for shared resources such as interconnect bandwidth and memory controllers. In such cases, back pressure across modes can converge to suboptimal allocations: a lower-efficiency mode (e.g., direct transfers saturating the interconnect with unfiltered bytes) can depress the achievable throughput of a more efficient mode (e.g., staging that benefits from sparse access) and attract less work than it should. Therefore A-Scan uses mode-at-a-time sampling, i.e., it routes a short burst of normal scan slices (in scan order, without duplicating work) exclusively to one mode to estimate per-mode steady-state scan-slice rates, exploits the arg-max, and resamples on drift (Algorithm 1). Section V-E quantifies both destructive and constructive interference patterns and shows why mode-at-a-time sampling yields more predictable choices.

After sampling all modes, A-Scan selects $m^\star = \arg\max_{m \in \mathcal{M}} \widehat{\tau}_m$ and enters the *exploitation phase*, routing all subsequent slices to $m^\star$. During exploitation, A-Scan continues measuring $m^\star$'s throughput. When a drift detector signals sustained performance degradation (e.g., a windowed throughput average drops more than 15% below the smoothed estimate), A-Scan re-enters sampling to re-evaluate all modes. This handles query phase changes (e.g., transitioning from selective to non-selective predicates over sorted data) or external resource contention.

End-to-end signal: Because progress is gated by admission to the bounded rowset output queue, the measured slice rate reflects end-to-end pipeline throughput, capturing bottlenecks at the data-movement boundary (storage bandwidth, interconnect, near-storage compute) as well as in downstream operators (joins, aggregations, remote-access latency).

E. Scope and Discussion

Scope: A-Scan currently adapts only scan-local predicates and projections; other data-reducing operators (e.g., partial aggregation, top-k) remain statically placed.

Optimizer interaction: For join ordering, the optimizer's join order choice is independent of mode: all modes produce identical output cardinality, so join cost estimates are mode-independent; staging benefits from downstream selectivity,

Algorithm 1: A-Scan mode selection (concept sketch)

Input: Enabled modes $\mathcal{M}$
State: Per-mode throughput estimate $\widehat{\tau}_m$ for $m \in \mathcal{M}$

Sample:
for $m \in \mathcal{M}$ **do**
  Route a short burst of *scan slices* exclusively to m
  Measure slices/second over *assignment* $\rightarrow$ *downstream consumption*
  Update $\widehat{\tau}_m$ with the observed rate
$m^\star \leftarrow \arg\max_{m \in \mathcal{M}} \widehat{\tau}_m$ `// best estimated mode`

Exploit:
while *scan not finished* **do**
  Route the next scan slice to $m^\star$; update $\widehat{\tau}_{m^\star}$
  if DRIFTDETECTED$(\widehat{\tau}, m^\star)$ **then**
    goto Sampling `// re-evaluate modes`

which typically align with optimizer preferences for selective joins that reduce intermediate sizes. Reciprocally, optimizer estimates could warm-start mode selection by biasing toward likely winners. Operator placement has more potential for mode-aware optimization, but exposing predicted modes to the optimizer requires modeling, reintroducing the brittleness that motivates runtime adaptation; extending A-Scan to own additional data-reducing operators would preserve encapsulation while letting throughput-guided selection apply unchanged.

Scaling the strategy space: As the strategy space grows beyond a small discrete set to include tunable parameters and combinatorial choices, exhaustive sampling becomes impractical; more scalable selection (e.g., hierarchical coarse-to-fine search or bandit-based optimization) remains future work.

Concurrent adaptation: Our design assumes at most one A-Scan adapts at a time; in our implementation, this holds because the host engine executes pipelines sequentially. In engines with concurrent pipelines or queries, independent A-Scan instances could interact and oscillate. Two mitigations are: (1) *staggered sampling*, so only one A-Scan samples at a time and others remain in steady state; and (2) *switching hysteresis*, so after resampling a mode change occurs only if the best alternative exceeds the current mode by a margin, otherwise reset the drift baseline. Coordinating adaptation across concurrent scans is future work.

IV. System Architecture and Implementation

We now describe how A-Scan is realized in a parallel push-based query engine with just-in-time compilation [23]. Section IV-A explains how A-Scan integrates with code generation to produce mode-specific pipeline variants while keeping parent operators mode-agnostic. Section IV-B details how each mode's materializer realizes different data-movement strategies. Sections IV-C and IV-D describe the queue topology, scheduling, and adaptive policy.

A. Code Generation and the Dispatch Shim

A-Scan integrates with parent operators through a *dispatch shim*, generated code that addresses a key integration challenge: **how to generate pipeline variants with different predicate evaluation timing and data placement assumptions while keeping parent operator code mode-agnostic**.

In a code-generating DBMS [22], [23], parent operators generate their tuple-processing logic, without knowledge of how data arrives. A-Scan requires three execution strategies: direct and staging modes must check predicates P during tuple iteration, while pushdown mode skips predicate checks (already evaluated storage-side). A naive solution would require parent operators to contain mode-specific logic for handling predicates differently by mode. This couples parent operators to A-Scan's internal implementation: every operator that might process scanned data needs awareness of direct/staging/pushdown semantics, and adding new modes requires modifying the entire operator stack.

The dispatch shim solves this by generating three for-loops, one per mode, that form the outer loop iterating over the tuples in a rowset. The direct and staging loops evaluate pushed predicates P inline: for each tuple in the rowset, the loop accesses predicate column values, evaluates P, and skips to the next tuple immediately when P fails. The pushdown loop iterates without evaluating P (predicates already evaluated near-storage). These loop structures serve as the integration boundary: parent operators' code generation is invoked three times, once per mode, to produce the loop body (the tuple-processing logic). The shim fuses each body with its corresponding outer loop structure, producing three complete compiled pipeline variants. Following standard operator fusion [22], this compilation inlines predicate checks, column value accesses, and parent logic into tight loops, eliminating function call boundaries. This allows the staging loop to benefit from parent operators' conditional access patterns: when parent code branches to skip column accesses (e.g., after failed join probes), those remote cache-line reads never occur, reducing interconnect traffic.

At runtime, materializers asynchronously load data, construct rowsets, and enqueue them to A-Scan's output queue. The shim's dispatcher dequeues rowsets and routes them to the appropriate compiled variant based on internal mode tags. From the execution engine's perspective, the dispatch shim is a standard operator consuming A-Scan's output queue and driving parent pipeline execution.

B. Data Placement and Rowset Construction

Each mode implements a materializer, a component that converts scan slices (storage metadata identifying which data to read) into rowsets. Materializers are responsible for: 1) loading data from storage to memory, 2) constructing rowsets with base pointers to the loaded data, and 3) enqueuing rowsets tagged with the mode to A-Scan's output queue. Materializers use asynchronous I/O (io_uring with O_DIRECT) to overlap data loading with pipeline execution, maintaining a bounded in-flight queue of slices being loaded. When this queue fills, back-pressure prevents new slice assignments from the scheduler. Regardless of mode, rowsets have uniform structure: an array of column base pointers (one per column in the projection), a tuple count, and an internal mode tag for dispatch

routing. Modes differ in where base pointers reference (local vs. remote memory) and whether data is filtered (pushdown) or not (direct/staging). In our implementation, each scan slice produces a rowset whose per-column segment is capped at 2 MiB (hugepage-sized); for 8-byte fixed-width columns this corresponds to 262k rows per slice.

Direct and staging materialization. The direct materializer issues asynchronous reads into compute-local memory buffers. For a scan slice spanning columns $C = \{c_1, c_2, \ldots, c_n\}$, it loads the column segments to local memory. Once all segments load, it constructs a rowset with base pointers to the buffers and enqueues it with mode tag DIRECT. The direct pipeline variant will later access column values from local memory when evaluating predicates P inline.

The staging materializer reads column segments into near-storage memory and enqueues a rowset whose pointers reference these remote buffers. The staging pipeline variant then evaluates predicates by accessing remote memory on demand; if parent code conditionally skips column accesses (Section IV-A), staging avoids those remote cache-line reads and reduces interconnect traffic.

Pushdown materialization. The pushdown materializer evaluates predicates P on the storage node before transferring data. It loads column segments to storage-local memory, spawns worker threads affinitized to the storage node's NUMA domain to evaluate P and compact data to include only P-satisfying tuples, gathering column values into an output buffer that is then transferred to compute-local memory. The materializer constructs a rowset with base pointers to these compacted local buffers and enqueues it. The pushdown pipeline variant iterates over a rowset's tuples without evaluating pushdown predicates P again; every tuple satisfies P by construction. Pushdown reduces interconnect traffic proportional to $(1 - \text{selectivity}(P))$ but introduces potential bottlenecks: (a) near-storage compute may be slower or more contended, (b) compaction adds a storage-side materialization step before the final transfer to the compute node, (c) cannot exploit downstream selectivity (all column segments of the output rowset are materialized for P-passing tuples before the rowset is enqueued in the output queue).

C. Queue Architecture and Control Loop

A-Scan's internal architecture uses a two-stage queue topology that decouples slice assignment from rowset consumption while enabling isolated throughput measurement. The system maintains: 1) per-mode input queues holding scan slices awaiting materialization, 2) a bounded unified output queue that collects rowsets from all modes and feeds the dispatch shim. This topology provides the scheduler with assignment flexibility, slices can be routed to any mode via input queues, while presenting parent operators with a mode-agnostic interface: a single output queue producing rowsets.

Execution flow. The complete dataflow proceeds as follows: 1) the scheduler assigns a scan slice to mode M by enqueuing it to M's input queue, 2) a worker thread affinitized to mode M dequeues the slice and invokes the corresponding materializer, 3) the materializer loads data and constructs rowsets tagged with mode M, enqueuing them to the unified output queue, 4) the dispatch shim dequeues rowsets and routes each to its corresponding pipeline variant, 5) parent operators consume tuples from the rowset, after which the rowset is freed and the output queue slot becomes available. This flow proceeds asynchronously across parallel workers, with the bounded output queue providing back-pressure that couples materialization rate to consumption rate.

Throughput measurement. Workers measure slice processing time from input-queue dequeue to successful output-queue enqueue of all rowsets produced from that slice. This interval spans the full cost: data loading, any storage-side computation (pushdown), rowset construction, and, critically, output queue admission. When the output queue fills due to slow downstream consumption, materializers block on enqueue, directly inflating measured slice times. Each worker maintains a local throughput estimate (slices per second) and periodically reports it to the scheduler.

The scheduler aggregates worker reports when constructing routing contexts for slice assignment. For mode M with recent reports $\{\tau_{M,i}\}$ from n active workers, the scheduler computes the current aggregate throughput $\tau_M = \sum_{i=1}^{n} \tau_{M,i}$ and updates an exponentially weighted moving average:

$$\widehat{\tau}_M \leftarrow \alpha \cdot \tau_M + (1 - \alpha) \cdot \widehat{\tau}_M$$

The EWMA provides robustness to transient noise while remaining responsive to sustained performance shifts. Unless stated otherwise, we use α=0.2 for EWMA smoothing. The aggregated estimate $\widehat{\tau}_M$ reflects mode M's throughput under current conditions: because the bounded output queue constrains slice completion whenever downstream operators fall behind, the observed rate captures end-to-end pipeline throughput rather than only data-loading speed. During mode transitions, the scheduler excludes throughput measurements from the first k slices of the newly activated mode, where k equals the combined capacity of materializer in-flight queues and the output queue. This warm-up period allows the output queue to reach a fill level representative of steady-state back-pressure before measurements contribute to $\widehat{\tau}_M$.

Control loop and policy interface. For each scan slice, the scheduler: (1) updates a context with per-mode throughput estimates $\{\widehat{\tau}_M\}$ and query state (slices processed, active mode), (2) invokes the pluggable policy: `mode = policy.selectMode(context)`, (3) enqueues the slice to that mode's input queue. This cleanly separates policy (mode choice and when to sample) from mechanism (queues, throughput tracking, and worker coordination). The interface is intentionally small (`selectMode(context)`), enabling future extensions that explore different sampling and switching strategies without changing the execution mechanism.

Although the architecture can execute modes concurrently, A-Scan's default policy enforces mode-at-a-time execution during throughput measurement: It routes new slices only to the active mode, so prior in-flight work drains, and subsequent throughput measurements reflect that mode alone.

D. Adaptive Policy Implementation

Our policy implements the sample-exploit-resample strategy introduced in Section III. The policy maintains a state machine with three phases: SAMPLING (measure each mode's throughput in isolation), EXPLOITATION (assign all slices to the highest-throughput mode), and RE-SAMPLING (triggered when throughput degrades). The policy adapts purely on observed throughput from the routing context.

Sampling phase. During SAMPLING, the policy takes a per-mode sample by executing a burst of scan slices, one mode at a time to avoid interference (Section III). For each mode, the policy assigns N_{sample} slices but discards throughput measurements from the first k slices. This warm-up period flushes transient effects that would otherwise contaminate steady-state measurements. After the warm-up, the policy collects throughput estimates $\widehat{\tau}_M$ from the routing context over the remaining slices, providing stable measurement of mode M's end-to-end performance. Once all modes are sampled, the policy selects $M^* = argmax_M \widehat{\tau}_M$ and transitions to EXPLOITATION.

Exploitation and drift detection. During EXPLOITATION, the policy assigns all slices to M^*, monitoring its throughput via the routing context. The policy maintains a sliding window of throughput observations. When the window average drops more than $\delta = 15\%$ below the mode's exponentially weighted moving average, the policy interprets this as sustained performance degradation (indicating query phase change, different data characteristics, or external resource contention) and re-enters SAMPLING to re-evaluate all modes. This threshold responds to performance shifts while tolerating transient fluctuations in parallel execution (OS scheduling, cache effects). By default, the system uses $N_{\text{sample}} = 350$ and $\delta = 15\%$, Section V-D evaluates sensitivity to δ and N_{sample}.

V. EXPERIMENTAL EVALUATION

We describe our experimental setup and evaluate A-Scan on the Star Schema Benchmark and NYC Taxi workloads. We then study intra-query adaptation and parameter sensitivity, and evaluate robustness against back-pressure baselines, across hardware configurations, and under interference.

A. Hardware & Software Setup

Hardware. All the experiments were conducted on a two-socket server running Ubuntu 22.04 with kernel version 6.5.0. Each socket has a 24-core AMD EPYC Milan 7413 processor with simultaneous multithreading enabled and 256 GB of DRAM. The server has three x8 xGMI inter-processor links. One socket has 12 Corsair MP600 Pro NVMe drives. We observe a maximum memory read bandwidth of 114 GB/s per socket and 24 GB/s remote socket read memory bandwidth using STREAM triad [28].

We measure 86 GB/s `O_DIRECT` sequential read bandwidth from all 12 NVMe drives when using fio [29] with socket-local memory. Each NVMe drive is mounted separately; data is evenly striped across drives, and all I/O uses `O_DIRECT`. We treat the NVMe-attached socket as the storage node and the other socket as the compute node.

Software. A-Scan is implemented as a fork of Proteus [23], [30]. The implementation of relational operators is directly inherited from Proteus; relational operators are fused and just-in-time compiled using LLVM code generation. As is typical for columnar engines, physical query plans specify which columns of a table are required.

Benchmark. We evaluate A-Scan with the Star Schema Benchmark (SSB) [3] at scale factor (SF) 1000. SSB has 4 query groups, and within each group, query selectivity decreases with rank; for example, Q1.3 is more selective than Q1.2, which is more selective than Q1.1. In A-Scan's binary format, groups 1–3 scans 96GB and group 4 scans 144GB per query from storage; in both cases, scan volume is dominated by the `lineorder` fact table. We also evaluate on the NYC yellow taxi dataset [4] for years 2011–2024. The dataset is 207GB in binary format, and 11.5GB per yellow trips column. We use three query sets. The first two, T1 and T2, are inspired by the Google BigQuery public workload, similar to Boeschen et al. [31]. T1 scans the yellow trip table, filtering on trip distance, and then aggregates trips per month. T2 scans the yellow trip table filtering on fare amount and valid trip distances, computes average speed by day of the week. For both T1 and T2, there are three variants (T1.1-T1.3 / T2.1-T2.3) with increasing selectivity of the trip distance and fare amount filters, respectively. T3 scans the yellow trip table, filtering for credit card payments and valid trip distances, then joins with the zone lookup table filtered for specific zones to compute aggregates (avg fare, tip, MTA tax, and speed) grouped by trip distance ranges. T3 also has three variants with increasing selectivity of the filter on the zone lookup table (T3.1–T3.3).

B. Star Schema Benchmark (SSB)

Figure 4 plots the execution time normalized to the naive strategy, that is, direct transfers, for all SSB queries using the static pushdown and staging strategies in comparison to A-Scan using the throughput-guided policy (A-Scan (TP)). In this experiment, A-Scan is configured to use 16 threads on the storage node for pushdown and 48 parallel pipeline instances (one per logical core) on the compute socket for all modes.

The performance characteristics of the direct, staging, and pushdown strategies exhibit notable variations across SSB query groups. With sufficient storage compute resources (16 threads), pushdown delivers substantial performance gains for most queries, achieving speedups of up to 2.93× over direct transfers (Q1.3). However, staging remains the optimal strategy for specific queries: Q3.1, Q4.2, and Q4.3. For Q3.1, staging achieves 0.95× the execution time of direct, marginally outperforming pushdown at 1.02×. The difference is more pronounced for Q4.2 and Q4.3, where staging is 1.29× and 1.24× faster than pushdown, respectively.

These exceptions highlight important workload characteristics. With SSB queries, early operators (filters/joins) reduce cardinality. Staging results in loading all values for columns needed by that operator; once tuple cardinality is reduced, other columns are accessed sparsely by later operators, and this benefit compounds across multiple pipelined data-

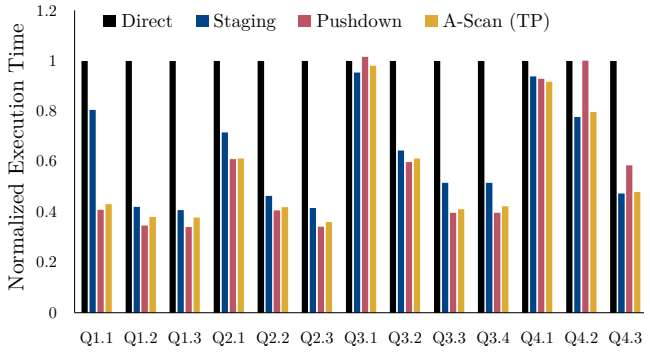


Fig. 4: Execution time of SSB queries normalized to direct transfer. Pushdown uses 16 threads on the storage node, while all strategies use 48 threads on the compute node.

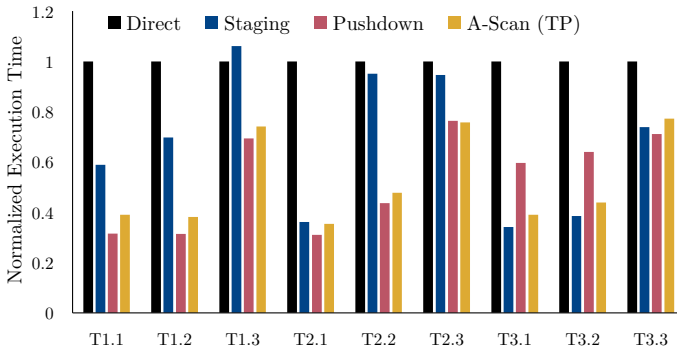


Fig. 5: Execution time of Taxi queries normalized to direct transfer. Pushdown uses 16 threads on the storage socket, all strategies use 48 threads on the compute socket.

reducing operators. Pushdown reduces total data movement by early filtering but incurs the cost of materializing filtered tuples on the storage node. For Q4.2 and Q4.3, this trade-off favors staging due to two factors: (1) the Bloom filter is less selective, reducing pushdown's data reduction advantage, and (2) the pushed-down operator materializes more columns, amplifying the cost of materialization. When pushdown's data reduction is modest and materialization overhead is high, staging's selective column access becomes more efficient despite transferring data at cache-line granularity. This shows that optimal mode depends not just on filter selectivity, but on the interplay between columnar access patterns, data reduction effectiveness, and materialization costs.

A-Scan navigates these trade-offs: across all SSB queries, A-Scan achieves up to a 2.78× speedup and a geometric mean speedup of 1.92× compared to direct transfers, demonstrating its ability to avoid poor strategy selections. When compared against an oracle that selects the best static strategy for each individual query, A-Scan achieves 96% of the oracle's performance (geometric mean), representing a 4% overhead for runtime adaptability. This oracle selects the single mode that yields the lowest total query time when applied uniformly to all scans within a query. Across queries, A-Scan spends on average 9.1% of execution time in the initial sampling phase; no re-sampling was triggered. Since sampling processes actual query work, overhead relative to the oracle depends on the performance gap between sampled modes: for Q3.1, where all modes perform within 7% of each other, 7.3% sampling time yields only 3% overhead; for Q2.3, where the slowest mode takes 2.9× as long as the fastest, 14.1% sampling time results in 5% overhead. For the challenging cases discussed above, A-Scan correctly identifies staging as optimal for Q4.2 and Q4.3, while selecting pushdown where it dominates. This shows that throughput-guided adaptation successfully captures the interplay between workload and hardware characteristics without requiring explicit modeling.

C. NYC Taxi Workload

Figure 5 presents the execution time normalized to direct transfers for the NYC Taxi workload using the static pushdown and staging strategies compared to throughput-guided A-Scan. The experimental configuration matches that of SSB: 16 threads for pushdown on the storage node and 48 threads on the compute node. In contrast to the relatively uniform SSB results, the Taxi workload exhibits significant diversity in optimal data-movement strategies across query types. For the aggregation queries (T1 and T2), pushdown demonstrates strong performance, achieving speedups of up to 3.2× over direct transfers (T1.1, T1.2, T2.1). However, the join queries (T3) reveal a strikingly different pattern: staging outperforms pushdown by 1.74× for T3.1 and 1.66× for T3.2.

This divergence stems from differences in query structure and data access patterns. T1 and T2 consist of a single selective filter before group by aggregations, a pattern that strongly favors pushdown. While pushdown directly reduces data movement by filtering at the storage node, staging must load every value of the filter column over the interconnect from the storage node's memory in order to apply the predicate. Moreover, since the aggregation consumes every tuple passing the filter, staging cannot access patterns more sparse than the initial filter for subsequent columns. The filter's selectivity directly determines the data volume for both strategies, but only pushdown fully capitalizes on this reduction before crossing the interconnect. This pattern, where all filtered data is consumed by an aggregation without further reduction, eliminates staging's primary advantage of increasingly selective data accesses by operators later in a query plan.

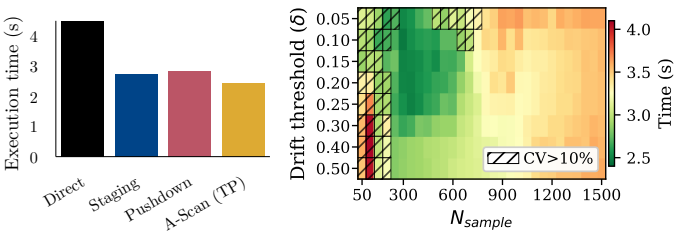
For T3 queries, multiple data-reducing operators shift the trade-off. These queries first filter on payment type (60% selectivity) before joining with a filtered zone lookup table. With pushdown, the payment type filter reduces the data volume moved over the interconnect by 40% before transfer, but it must materialize all columns needed for the following join and aggregations for every tuple passing the initial filter on the trips table. In contrast, staging exploits the cumulative selectivity of both operators: while it must transfer all payment type values to evaluate the filter, it then accesses join key columns only for the tuples passing the filter, and finally accesses the aggregation columns only for tuples surviving both the filter and join. This cascading reduction in data

access allows staging to amortize its initial overhead across increasingly sparse column accesses. When the initial filter has only moderate selectivity and multiple columns must be materialized, staging's ability to defer column materialization until after all data-reducing operators outweighs pushdown's advantage of early but limited data reduction.

A-Scan successfully navigates this diverse landscape of query characteristics, achieving a 2.01× geometric mean speedup over using direct transfers. When compared to the same oracle as in Section V-B, A-Scan achieves 89% of oracle performance. This 11% overhead is higher than the 4% observed for SSB. Concretely, A-Scan spends 17.1% of execution time in initial sampling, with occasional re-sampling bringing total sampling time to 19.3%; these re-evaluations did not result in mode switches. Despite these challenges, A-Scan correctly identifies pushdown for the single-operator queries (T1, T2) and staging for the multi-operator join queries (T3.1, T3.2). Future work could reduce overhead by skipping predictable modes (e.g., direct, given storage/interconnect specifications) and focusing sampling on less certain modes.

D. Intra-Query Adaptation and Parameter Sensitivity

We evaluate whether the policy (Section IV-D) reacts within a single query when the optimal mode changes over time. Using Taxi query T3.2, we keep the query plan, hardware and degree of parallelism fixed and vary the data distribution of the filter (payment type) and join probe key (drop-off zone) columns of the yellow trip table. In the first phase of the scan, staging is optimal (filter selectivity 80% & join selectivity 1%) and in the second phase of the scan pushdown is optimal (filter selectivity 15% and join selectivity 50%). We use the same configuration as the previous SSB and Taxi experiments, reporting the mean of 20 independent iterations, and additionally vary A-Scan's parameters to assess sensitivity.



(a) Execution time (b) Parameter sensitivity

Fig. 6: T3.2 with mid-query phase change (staging optimal in phase 1, pushdown in phase 2). (a) A-Scan outperforms static strategies by adapting mid-execution. (b) A-Scan is robust across parameter choices; hatched regions indicate high variance (coefficient of variation $>10\%$).

Figure 6a plots the results of this experiment. A-Scan achieves the lowest end-to-end time (2.4 s), outperforming both static choices (staging (2.7 s) and pushdown (2.8 s)) by 1.12× and 1.16×, respectively, and beating Direct (4.4 s) by 1.84×. The gain comes from switching: in phase 1 (high filter selectivity, sparse probe), A-Scan exploits staging's advantage; after the distribution flips in phase 2 (lower filter selectivity, dense probe), it transitions to pushdown. This shows that throughput-guided re-evaluation reacts within a query when the optimal mode changes, avoiding the worst-case behavior of locking into the wrong strategy.

Sensitivity to N_{sample} and δ. Figure 6b shows execution time across the parameter space of sampling size N_{sample} and drift threshold δ, on the same workload. A-Scan is robust across a range of parameter choices: the green region ($N_{sample}\in[300, 500]$, $\delta\in[0.10, 0.25]$) achieves execution times within 7% of the best configuration. Two failure modes bound this region. *Under-sampling* ($N_{sample}<200$) yields unreliable mode selection; throughput estimates have high variance, leading A-Scan to occasionally pick suboptimal modes (hatched cells indicate coefficient of variation $>10\%$). *Over-sampling* ($N_{sample}>500$) increases execution time as more slices are processed by suboptimal modes during sampling; approaching $N_{sample}=900$, re-sampling after drift detection consumes most remaining slices, and beyond 900, sampling itself crosses the phase boundary where optimal mode switches from staging to pushdown. The drift threshold has a smaller effect but matters at extremes: $\delta\geq0.40$ never triggered re-evaluation for $N_{sample}>150$, leaving the initial mode selection unchanged despite the mid-query shift; $\delta<0.10$ triggers spurious re-sampling from transient fluctuations, and when combined with borderline N_{sample} (500–700), can cause resampling to straddle the phase boundary, producing high variance. Our default ($N_{sample}=350$, $\delta=0.15$) balances sufficient samples for reliable estimates against bounded overhead, and was used for all other experiments without per-query tuning.

Sensitivity to α (EWMA). In a sweep over $\alpha \in [0.05, 0.5]$ with default $N_{sample}=350$ and $\delta=0.15$, we observed no consistent effect on re-evaluation frequency or mode switches, so we fix $\alpha=0.2$ for the other experiments.

E. Interference Effects in Back-pressure Routing

Back-pressure offers an appealingly simple approach to adaptive data movement: faster strategies drain their queues more quickly and thus receive more work, yielding a self-balancing system. To evaluate this approach, we implement back-pressure routing through randomly selecting among non-full mode-input queues. Figure 7 compares this back-pressure approach (A-Scan (BP)) against our throughput-guided policy on two representative Taxi queries. In practice, back-pressure's effectiveness depends on an implicit assumption that strategies operate independently, which breaks down when strategies compete for shared hardware resources like interconnects.

For T3.1 at SC-16, the optimal static strategy (staging) is 1.6× faster than back-pressure routing and 1.14× faster than throughput-guided A-Scan. The slowdown of back-pressure relative to throughput-guided stems from resource contention: staging saturates the interconnect bandwidth when running in isolation while transferring the least data across the interconnect. In back-pressure, both direct and pushdown also compete for the same interconnect bandwidth. Since these

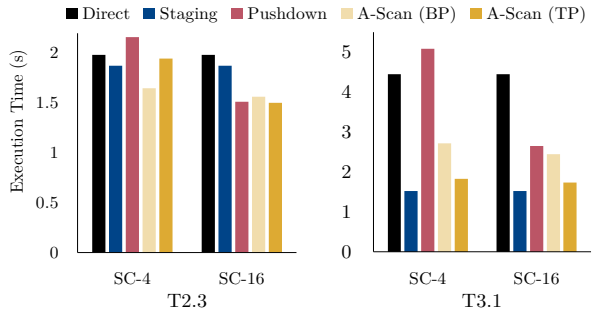


Fig. 7: Execution time for Taxi queries T2.3 and T3.1 showing contrasting interference patterns with back-pressure (BP) routing. All strategies use 48 threads on the compute node with 4 (SC-4) or 16 (SC-16) storage threads for pushdown.

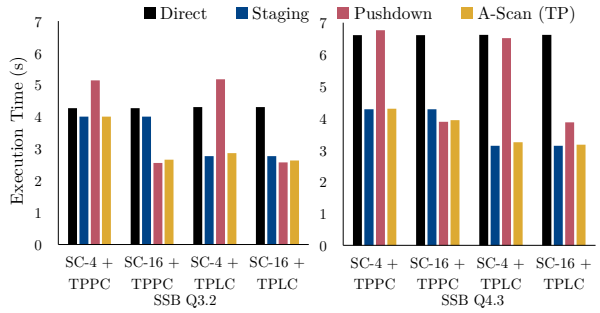


Fig. 8: Execution time for SSB Q3.2 and Q4.3 with TPPC vs. TPLC on the compute socket and SC-4 vs. SC-16 on the storage socket

two strategies transfer data less efficiently than staging, the contention on the interconnect results in performance loss.

T2.3 shows the opposite behavior: back-pressure outperforms all static strategies, including staging. Here, pushdown with SC-4 is compute-bottlenecked on the storage socket and does not saturate shared resources (interconnect, storage-socket memory, or storage bandwidth). When running alone, staging saturates the interconnect but utilizes bandwidth less efficiently than pushdown. Under back-pressure, these strategies compete for interconnect bandwidth, creating an unexpected synergy: pushdown contributes efficient filtered transfers while staging fills the remaining bandwidth capacity, together achieving higher effective throughput.

Overall, back-pressure is sensitive to whether shared resources are already bottlenecked (risking destructive interference, as in T3.1) or underutilized (allowing beneficial interactions, as in T2.3), making performance difficult to predict. A-Scan mitigates this by sampling each strategy in isolation and selecting based on end-to-end throughput; it stays close to the best available strategy, with overhead primarily from sampling. Thus, we trade occasional super-optimal outcomes (e.g., T2.3) for more predictable near-optimal performance.

F. Adaptation Across Hardware Configurations

Figure 8 compares SSB Q3.2 and Q4.3 across two hardware resource allocation dimensions: storage compute threads (SC-4 vs. SC-16) and compute-node parallelism, where TPPC denotes thread-per-physical-core (24 threads total) and TPLC

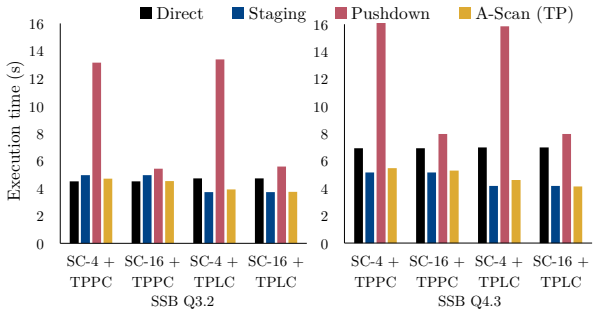


Fig. 9: Execution time for SSB Q3.2 & Q4.3 with the same configurations as Figure 8 while an interfering workload contends for hardware resources on the storage socket.

denotes thread-per-logical-core (simultaneous multithreading (SMT), 48 threads total).

Hardware setup alone can reverse the best strategy, even for the same query. For Q3.2, increasing storage compute from 4 to 16 threads improves pushdown performance by 2.02×, making it 1.57× faster than staging with TPPC. However, for Q4.3, the same 4× increase in storage compute yields only a 1.74× improvement, and staging remains 1.1× faster than direct at SC-16 + TPLC. This highlights that a fixed strategy does not generalize across hardware configurations.

The underlying bottlenecks differ by strategy. Pushdown exhibits sublinear scaling with storage compute resources: quadrupling threads yields only a 1.74× to 2× improvement on these queries because the bottleneck shifts from compute to memory bandwidth. With SC-16, Q3.2 sustains 134 GB/s of memory bandwidth measured with AMD uProf PCM. This performance counter-based measurement exceeds the software-based STREAM measurement for single socket bandwidth, indicating that pushdown is saturating, or close to saturating, the available memory bandwidth.

In contrast, staging is insensitive to near-storage compute since it performs no computation on the storage node. However, using TPLC on the compute node improves staging by 1.45× for Q3.2 and 1.37× for Q4.3, because SMT partially masks stalls from staging's sparse, cache-line–granular remote accesses to remote memory. SMT increases staging's sustained interconnect bandwidth by 1.41× (15.8 GB/s vs. 11.2 GB/s for Q3.2), translating directly to improved query performance.

These results show that the optimal strategy depends on subtle interactions between query properties and hardware bottlenecks. Measuring end-to-end throughput at runtime lets A-Scan adapt across hardware configurations.

G. Robustness to Runtime Interference

We evaluate A-Scan under resource contention by co-running STREAM [28] on the storage socket with SSB Q3.2 and Q4.3.

In Figure 9, interference affects strategies differently. Direct transfer slows by 5–10% since it depends little on storage-socket memory bandwidth, whereas pushdown slows by up to 2.59× (Q3.2, SC-4+TPLC) because filter evaluation and materialization contend with STREAM for storage-node memory

bandwidth. With SC-4, pushdown increases by >2.4× on both queries; SC-16 reduces this to ≈2.1×. Staging degrades only 1.21–1.34× since it performs no storage-side computation; slowdowns are larger under TPLC, where baseline storage-socket bandwidth utilization is higher.

Contention can also change the best strategy. For Q3.2 with SC-16 + TPPC, pushdown's 1.57× advantage over staging (no interference) reverses to a 10% disadvantage under contention. For Q4.3, pushdown becomes 1.5–3× slower than staging in configurations where it was previously competitive.

A-Scan tracks throughput and switches modes when contention degrades a strategy. For example, in Q3.2 with SC-16+TPPC, under interference A-Scan switches from the pushdown to staging (Figure 9); without interference, pushdown is best in the corresponding configuration (Figure 8). Across configurations, A-Scan matches the best strategy under contention without explicit interference detection.

VI. Related Work

Adaptive query processing. Traditional AQP adapts to runtime conditions and cardinality estimation errors by changing *which* operators run and *in what order* or *which algorithm* an operator uses, while treating data movement as fixed.

Inter-Operator (Plan-Level) Adaptation: These techniques adapt the operator evaluation order within a query plan. Eddies [32] routes individual tuples through different operator orderings; its lottery-based scheme tracks tuple consumption and production rates per operator. POLAR [33] routes chunks of tuples through alternative join paths, adapting thread-locally based on intermediate result sizes, an effective heuristic for join ordering cost, which is invariant to other operators or parallelism. SkinnerDB [34] employs reinforcement learning to discover optimal join orders by measuring execution progress within time slices. These systems assume a fixed data-movement strategy; they adapt which operators process data, not how data reaches those operators. Re-optimization techniques [35], [36] take a different approach: they monitor execution and, at pipeline breakers or between sub-queries, may re-invoke the optimizer using observed materialized intermediate statistics, potentially producing a different physical plan for the remaining operators. Re-optimization primarily corrects for cardinality estimation errors; A-Scan addresses data-movement trade-offs across hardware bandwidth boundaries, which cost-based optimization does not model.

Intra-Operator (Physical-Operator-level) Adaptation: These techniques adapt physical operator implementations at runtime. Micro-Adaptivity [37] selects between multiple function implementations, for example, between branching and non-branching filters, based on per-invocation runtime. Rosenfeld et al. extend the Micro Adaptivity approach to select aggregation and selection implementation variants on heterogeneous hardware [38]. Smooth Scan morphs its *access path* between non-clustered index probes and full scans based on observed selectivity and distribution [39].

Positioning of A-Scan: A-Scan performs intra-operator adaptation along the *data-movement* axis. It differs from prior AQP in two respects. **(1) target:** It selects among *data-movement modes* (direct, staging, pushdown) that determine where/when bytes cross bandwidth boundaries, while keeping the logical plan and scan interface fixed. In contrast, most prior AQP work focuses on adapting operator ordering or per-operator algorithms, while treating data movement/placement as an external concern. **(2) signal:** It selects modes using *end-to-end pipeline throughput* aggregated across worker threads. Because modes contend for shared bandwidth resources, their performance depends on parallelism and concurrent activity, which local metrics can misrepresent. In contrast, systems like POLAR adapt per-worker using *workload-intrinsic* signals (e.g., intermediate result size) that are determined by the data and predicates and that are robust to runtime contention. A-Scan is complementary to conventional AQP: a system may adapt join order or operator algorithms while A-Scan independently chooses the movement mode best matched to current hardware and workload conditions.

Modern and heterogeneous hardware: Heterogeneous architectures have prompted rethinking database design principles. HetExchange [23] extends exchange operators to encapsulate CPU-GPU parallelism with JIT-compiled engines. GOLAP [31] accelerates table scans through GPU-accelerated pruning and decompression. Modern storage technologies are transforming query processing economics and performance. Umami abstracts materialization so hash-based operators can switch between in-memory execution and partition/spill seamlessly [40] to high performance NVMe drives. Chronis et al. [41] argue that memory-centric computing is increasingly necessary with disaggregated resources, while Lerner and Alonso [1], [42] explore how CXL enables coherent access to disaggregated memory and its impact on database design. With evolving storage, compute, and interconnect characteristics, our work introduces an adaptive and extensible framework that minimizes the optimization complexity arising from evolving workloads and system characteristics.

VII. Conclusion

In this paper, we propose throughput-guided data movement, which uses end-to-end operator-pipeline throughput to select how and where bytes are realized at query runtime. We integrate this idea into a leaf operator, A-Scan, which encapsulates direct, staging, and pushdown as runtime-selectable modes. Looking ahead, our approach provides a foundation for query processing in an era of increasingly heterogeneous hardware. As emerging interconnect technologies like CXL result in complex topologies of storage, memory, and compute devices and more flexible approaches to data movement between them, adaptive data movement will become increasingly important for achieving efficient utilization and consequently, performance across diverse hardware topologies.

Acknowledgements

This work was partially funded by the Swiss State Secretariat for Education, Research and Innovation (SERI) for the Prodasys project, a European Research Council (ERC) grant.

AI-Generated Content Acknowledgement

OpenAI ChatGPT was used for proofreading. We prompted ChatGPT to identify typos, confusingly worded sentences, and to check for consistent use of terminology across the full paper.

References

[1] A. Lerner and G. Alonso, "CXL and the return of scale-up database engines," *Proc. VLDB Endow.*, vol. 17, no. 10, pp. 2568–2575, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p2568-lerner.pdf

[2] D. D. Sharma, R. Blankenship, and D. S. Berger, "An introduction to the compute express link (CXL) interconnect," *ACM Computing Surveys*, vol. 56, no. 11, pp. 290:1–290:37, 2024. [Online]. Available: https://doi.org/10.1145/3669900

[3] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The Star Schema Benchmark and Augmented Fact Table Indexing," in *Performance Evaluation and Benchmarking*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer, 2009, pp. 237–252.

[4] NYC Taxi & Limousine Commission, "TLC Trip Record Data - TLC," 2025. [Online]. Available: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[5] A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. G. Anadiotis, and A. Ailamaki, "Gpu-accelerated data management under the test of time," in *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[6] H. Nicholson, A. Raza, P. Chrysogelos, and A. Ailamaki, "HetCache: Synergising NVMe storage and GPU acceleration for memory-efficient analytics," in *13th Conference on Innovative Data Systems Research*. Amsterdam, The Netherlands: www.cidrdb.org, Jan. 2023. [Online]. Available: https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf

[7] P. Chrysogelos, "Efficient Analytical Query Processing on CPU-GPU Hardware Platforms," Ph.D. dissertation, EPFL, 2022. [Online]. Available: https://infoscience.epfl.ch/handle/20.500.14299/190430

[8] L. Woods, Z. István, and G. Alonso, "Ibex - an intelligent storage engine with support for advanced SQL off-loading," *Proc. VLDB Endow.*, vol. 7, no. 11, pp. 963–974, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p963-woods.pdf

[9] S. Lee, A. Lerner, P. Bonnet, and P. Cudré-Mauroux, "Database kernels: Seamless integration of database systems and fast storage via CXL," in *14th conference on innovative data systems research, CIDR 2024, chaminade, HI, USA, january 14-17, 2024*. www.cidrdb.org, 2024. [Online]. Available: https://www.cidrdb.org/cidr2024/papers/p43-lee.pdf

[10] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: opportunities and challenges," in *Proceedings of the ACM SIGMOD International Conference on Management of Datayork*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. New York, USA: ACM, 2013, pp. 1221–1230.

[11] S. Kim, H. Oh, C. Park, S. Cho, S.-W. Lee, and B. Moon, "In-storage processing of database scans and joins," vol. 327, pp. 183–200, 2016. [Online]. Available: https://doi.org/10.1016/j.ins.2015.07.056

[12] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the wall: Near-data processing for databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, I. Pandis and M. L. Kersten, Eds. Melbourne, Australia: ACM, May 2015, pp. 2:1–2:10.

[13] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves, "Database processing-in-memory: An experimental study," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 334–347, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p334-kepe.pdf

[14] X. Yu, M. Youill, M. E. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, "PushdownDB: Accelerating a DBMS using S3 computation," in *36th IEEE international conference on data engineering*. Dallas, USA: IEEE, Apr. 2020, pp. 1802–1805.

[15] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry, "Amazon redshift re-invented," in *SIGMOD '22: International Conference on Management of Data*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. Philadelphia, USA: ACM, Jun. 2022, pp. 2205–2217.

[16] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford, "Spanner: Becoming a SQL system," in *Proceedings of the 2017 ACM International Conference on Management of Data,*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. Chicago, IL, USA: ACM, May 2017, pp. 331–343. [Online]. Available: https://doi.org/10.1145/3035918.3056103

[17] Y. Yang, M. Youill, M. E. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS," *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2101–2113, 2021.

[18] "A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server," Oracle Corporation, Oracle White Paper, Dec. 2012. [Online]. Available: https://www.oracle.com/technetwork/server-storage/engineered-systems/exadata/dbmachine-x3-twp-1867467.pdf

[19] P. Francisco, "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics," IBM International Technical Support Organization, IBM Redguide, 2011. [Online]. Available: https://public.dhe.ibm.com/software/ch/de/pdf/Netezza_Appliance_Architecture_WP.pdf

[20] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of A parallel relational database machine GRACE," in *VLDB'86 twelfth International Conference on Very Large Data Bases*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Kyoto, Japan: Morgan Kaufmann, Aug. 1986, pp. 209–219. [Online]. Available: http://www.vldb.org/conf/1986/P209.PDF

[21] A. Baumstark, M. Paradies, K.-U. Sattler, S. Kläbe, and S. Baumann, "So far and yet so near - accelerating distributed joins with CXL," in *Proceedings of the 20th International Workshop on Data Management on New Hardware*, C. Binnig and N. Tatbul, Eds. Santiago, Chile: ACM, Jun. 2024, pp. 7:1–7:9. [Online]. Available: https://doi.org/10.1145/3662010.3663449

[22] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf

[23] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," *Proceedings of the VLDB Endowment*, vol. 12, no. 5, pp. 544–556, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.14778/3303753.3303760

[24] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-pipelining query execution," in *Second Biennial Conference on Innovative Data Systems Research*. Asilomar, CA, USA,: www.cidrdb.org, Jan. 2005, pp. 225–237. [Online]. Available: http://cidrdb.org/cidr2005/papers/P19.pdf

[25] G. Graefe, "Volcano - an extensible and parallel query evaluation system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994. [Online]. Available: https://doi.org/10.1109/69.273032

[26] A. Shaikhha, M. Dashti, and C. Koch, "Push versus pull-based loop fusion in query engines," *Journal of Functional Programming*, vol. 28, p. e10, 2018. [Online]. Available: https://doi.org/10.1017/S0956796818000102

[27] R. A. Lorie, "XRM - an extended (n-ary) relational memory," *Research Report / G / IBM / Cambridge Scientific Center*, vol. G320-2096, 1974.

[28] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[29] J. Axboe, "Fio," 2022. [Online]. Available: https://github.com/axboe/fio

[30] "Proteus," Jun. 2025. [Online]. Available: https://github.com/epfl-dias/proteus

[31] N. Boeschen, T. Ziegler, and C. Binnig, "GOLAP: A GPU-in-data-path architecture for high-speed OLAP," *Proc. ACM Manag. Data*, vol. 2, no. 6, pp. 237:1–237:26, 2024. [Online]. Available: https://doi.org/10.1145/3698812

[32] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data,*, W. Chen, J. F. Naughton, and P. A. Bernstein, Eds. Dallas, USA: ACM, May 2000, pp. 261–272. [Online]. Available: https://doi.org/10.1145/342009.335420

[33] D. Justen, D. Ritter, C. Fraser, A. Lamb, N. Tran, A. Lee, T. Bodner, M. Y. Haddad, S. Zeuch, V. Markl, and M. Boehm, "POLAR: adaptive and non-invasive join order selection via plans of least resistance," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1350–1363, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1350-justen.pdf

[34] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari, "SkinnerDB: Regret-bounded query evaluation via reinforcement learning," *ACM Transactions on Database Systems*, vol. 46, no. 3, pp. 9:1–9:45, 2021. [Online]. Available: https://doi.org/10.1145/3464389

[35] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," *SIGMOD Rec.*, vol. 27, no. 2, pp. 106–117, 1998. [Online]. Available: https://doi.org/10.1145/276305.276315

[36] J. Zhao, H. Zhang, and Y. Gao, "Efficient Query Re-optimization with Judicious Subquery Selections," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3589330

[37] B. Raducanu, P. A. Boncz, and M. Zukowski, "Micro adaptivity in vectorwise," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. New York, USA: ACM, Jun. 2013, pp. 1231–1242. [Online]. Available: https://doi.org/10.1145/2463676.2465292

[38] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl, "The operator variant selection problem on heterogeneous hardware," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, R. Bordawekar, T. Lahiri, B. Gedik, and C. A. Lang, Eds., Hawaii, USA, Aug. 2015, pp. 1–12. [Online]. Available: http://www.adms-conf.org/2015/ADMS_Viktor_Rosenfeld_CR.pdf

[39] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser, "Smooth scan: Statistics-oblivious access paths," in *31st IEEE International Conference on Data Engineering*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. Seoul, South Korea: IEEE Computer Society, Apr. 2015, pp. 315–326.

[40] M. Kuschewski, J. Giceva, T. Neumann, and V. Leis, "High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance," *Proceedings of the ACM on Management of Data*, vol. 2, no. 6, pp. 1–27, Dec. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3698813

[41] Y. Chronis, A. Ailamaki, L. Benson, H. Caminal, J. Gičeva, D. Patterson, E. Sedlar, and L. W. Wills, "Databases in the era of memory-centric computing," in *15th Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, Jan. 2025. [Online]. Available: https://vldb.org/cidrdb/papers/2025/p6-chronis.pdf

[42] A. Lerner and G. Alonso, "Data flow architectures for data processing on modern hardware," in *40th IEEE international Conference on Data Engineering*. Utrecht, The Netherlands: IEEE, May 2024, pp. 5511–5522. [Online]. Available: https://doi.org/10.1109/ICDE60146.2024.00439