Scout-and-Reduce: Budget-Aware Storage-Side Reduction for Memory-Safe GPU Analytics

Soon Hwang[1], Jungki Noh[2], Woosuk Chung[2], Youngjae Kim[1,*]
[1]*Dept. of Computer Science and Engineering, Sogang University*, Seoul, Republic of Korea
[2]*Memory System Research, SK hynix*, Seongnam, Republic of Korea
{soonhw, youkim}@sogang.ac.kr, {jungki.noh, woosuk.chung}@sk.com

***Abstract*—GPU-based analytics over object-stored Parquet data exposes a sharp memory cliff: a workload that completes at one scale can fail with out-of-memory at a modestly larger one. We identify two budgets that jointly govern this cliff: B_{transfer}, the decoded size of data admitted to the GPU, and B_{merge}, the intermediate hash-table state sustained during group-by aggregation. We show that existing storage-side offloading addresses neither. We present Scout-and-Reduce (S&R), a storage-side framework that enables memory-safe GPU analytics by shaping data before it reaches the GPU. S&R uses a two-level planner to estimate GPU memory demand from Parquet footer metadata and selectively applies exact local reduction on storage servers. For high-cardinality group-by, S&R partitions results into key-disjoint buckets, allowing the GPU to merge one bounded bucket at a time and preventing intermediate state from exceeding device capacity. On two datasets (NYC Yellow Taxi and OTTO recommendation events) with a 4 GB GPU, S&R sustains completion at all tested scales where the direct path and filter offloading fail with OOM. Our footer-based estimator predicts decoded footprint within 1.8% of the actual size. S&R keeps peak GPU memory well below the device limit regardless of data scale, while running entirely on existing storage-server CPU and memory.**

***Index Terms*—GPU-based analytics, storage-side computation, object storage, memory admission safety, cuDF**

I. Introduction

Single-node GPU dataframe workflows are an important target for interactive tabular analytics, where users prepare and query data on local machines rather than deploying a full distributed stack. Recent work treats the single-machine scenario as a primary setting for dataframe-based data preparation and shows that most real-world analytical workloads fit within a single node [1], [2]. At the same time, GPU analytics systems already operate over cloud- and object-backed columnar data. BlazingSQL supports tables on local or cloud bucket storage and returns query results as cuDF DataFrames [3], while recent GPU query engines execute directly over remote Parquet files on AWS S3 [4]. These systems converge on open columnar formats such as Parquet, which have become standard substrates for analytical DBMSs more broadly [5]. Together these trends make direct GPU analytics over object-stored Parquet data a natural target for lightweight notebook-oriented workflows.

*Youngjae Kim is the corresponding author.

In practice, however, this direct path exposes a sharp GPU memory cliff. A workload that completes at one input scale can fail abruptly at a modestly larger one, either with an out-of-memory (OOM) error or with memory-management overheads severe enough to erode the benefit of GPU acceleration. The problem is compounded by the fact that Parquet combines columnar encodings and page compression, so the in-memory footprint after decode can be substantially larger than the on-disk size, making OOM failures difficult to predict from file size alone. Recent GPU analytics work consistently identifies limited GPU memory capacity, together with bandwidth-limited CPU-GPU communication, as a first-order systems constraint [2], [6]–[9]. Unlike CPU execution, which often degrades gradually through paging or other out-of-core mechanisms, GPU execution presents a much narrower boundary between success and failure [10], [11]. For object-stored analytics, this makes it critical to control not just what computation eventually runs on the GPU, but what data is admitted to GPU memory in the first place. [2], [7]

A natural response is to reduce data before GPU admission. Scale-out execution can mitigate the problem but introduces distributed complexity that interactive single-node workflows seek to avoid [1]. Existing storage-side computation systems are designed around bandwidth reduction: they minimize the bytes transferred to the client but remain oblivious to the memory constraints of the downstream execution engine. For GPU workloads, where the gap between transferred bytes and decoded in-memory footprint can exceed an order of magnitude (Section III), bandwidth reduction alone does not guarantee safe execution. S3 Select [12] and Skyhook [13], for example, target CPU-based SQL analytics and do not support GPU dataframe workloads or address GPU admission safety. A filter query whose final result is small may still require admitting a large intermediate unit into GPU memory. We therefore argue that admission safety must be defined in terms of the transfer unit admitted to the GPU, rather than the final result size alone.

cuDF [14] also provides a spill-to-host mechanism that evicts GPU buffers not currently referenced by an active operation to host memory on demand [15]. This is useful when pressure arises after substantial work has already materialized reusable state. However, spill is reactive, not preventive: it responds after pressure occurs and does not explicitly bound

the admission-time working set or the in-flight state maintained by an operator [16]. For example, a single cuDF filter or group-by call may allocate an intermediate buffer that exceeds available GPU memory in one request, leaving no opportunity for the spill mechanism to intervene. When the next allocation or the effective unspillable working set is itself too large relative to available headroom, spill may still fail to prevent the cliff. In our evaluation, enabling spill does not move the failure boundary for either filter or high-cardinality group-by workloads.

Group-by aggregation introduces a second failure mode beyond admission-unit sizing. Even if each object is delivered in smaller pieces, the GPU can still exhaust memory during merge. With high-cardinality keys, partial aggregates from many objects must be combined into an in-flight hash table whose size grows with the accumulated distinct-key state [17], [18]. Chunked execution, introduced earlier as a way to control transfer granularity, does not help here because it is key-unaware: the same key can appear in many chunks, so the GPU may still need to sustain a large shared merge state across them. Controlling this failure mode requires partitioning that is aware of key distribution, not just input size [17].

These observations point to two budgets that jointly govern GPU admission safety. We call the first B_{transfer}, the maximum size of a transfer unit admitted to the GPU in one step. We call the second B_{merge}, the amount of in-flight merge state the GPU must sustain during aggregation. Prior storage-side computation works do not reliably control either quantity. Filter offloading may reduce rows but does not explicitly bound admission units. Spill reacts only after pressure appears. Client-side reduction cannot reduce network bytes before transfer. Chunked-only delivery does not bound high-cardinality merge state. Object storage should therefore not merely deliver bytes to the GPU analytics engine: it should export semantically exact units already shaped to the client GPU's admission and merge budgets.

We present Scout-and-Reduce (S&R), a storage-side framework for GPU-backed object analytics. Rather than exporting raw object bytes and leaving memory safety entirely to the client, S&R performs exact local reduction on storage servers and exports semantically exact outputs already shaped to budgets derived from the client GPU's reported memory capacity.

S&R consists of a GPU client and a storage-side sidecar co-located with the object store. The client submits a query along with GPU memory information. The sidecar's two-level planner first reads the Parquet footer of each object to estimate the decoded GPU footprint and, for group-by queries, the merge-state upper bound. It then checks whether the combined memory demand of the decoded data and the merge state fits within the GPU's effective capacity. If so, the planner selects the direct path and no storage-side processing occurs. Otherwise, the planner activates the S&R execution engine, which performs exact local reduction on each object using the storage server's CPU and memory. For filter queries, the engine emits exact filtered results. For aggregation queries, it computes local partial aggregates that can be merged on the GPU. For high-cardinality group-by, the engine further partitions results into key-disjoint buckets so that the GPU merges one bounded bucket at a time, preventing the intermediate merge state from exceeding GPU capacity. S&R runs on existing storage-server CPU and memory resources and requires no specialized hardware.

We evaluate S&R on two real-world datasets, NYC Yellow Taxi [19] and OTTO recommendation events [20]. S&R completes at every tested scale, whereas the direct GPU path and filter-only offloading both fail with OOM. The footer-based estimator predicts decoded GPU footprint within 1.8% of the actual size without reading any data pages. S&R keeps peak GPU memory well below the device limit by merging one bounded bucket at a time through partitioned delivery.

We make four contributions.

1) We identify B_{transfer} and B_{merge} as the two budgets that jointly govern GPU admission safety for object analytics, and show that existing filter offloading and spill mechanisms address neither.
2) We design Scout-and-Reduce, a storage-side framework that receives the client GPU's memory capacity as a query-time parameter and emits semantically exact outputs shaped to the reported budgets. This redefines the storage-client interface: rather than optimizing for bandwidth reduction, the storage layer takes responsibility for producing units that the GPU can safely admit.
3) We show that high-cardinality group-by requires partitioned merge with key-disjoint cross-object bucketing, and derive a ceiling formula that selects the minimum partition count P sufficient to keep per-bucket merge state within budget.
4) We demonstrate that S&R sustains completion where direct, filter-offloading paths fail, while keeping peak GPU memory bounded and running entirely on existing storage-server resources.

II. Background

A. Parquet Format and Decode

Apache Parquet has become the de facto columnar storage format for analytical workloads across data lake, warehouse, and scientific computing environments [21], [22]. Its widespread adoption stems from several properties: columnar layout enables efficient column pruning and vectorized reads, the open specification allows interoperability across engines (Spark, DuckDB, Polars, cuDF), and built-in encoding and compression achieve high compression ratios that reduce storage cost and I/O volume. As a result, Parquet is the default substrate for analytical tables in systems ranging from cloud data lakes to single-node dataframe workflows [5].

A Parquet file is organized as a sequence of row groups, each containing one column chunk per column. Each column chunk is further divided into pages, which are the smallest units of encoding and compression. Parquet applies columnar encodings such as dictionary encoding, run-length encoding (RLE), and delta encoding, and then applies page-level compression (for example, Snappy or ZSTD) [21], [22].

When a reader decodes a Parquet file into an in-memory columnar representation such as Apache Arrow [23], it must decompress pages, expand encoded values, and materialize null bitmaps and other validity metadata. As a result, the in-memory footprint after decode can be substantially larger than the on-disk file size, which makes memory pressure difficult to predict from file size alone.

Parquet also embeds metadata that readers can use to skip irrelevant data without reading the full file. File-level metadata records row-group boundaries. Footer statistics expose per-column min, max, and null-count summaries at row-group granularity. Page indexes expose finer-grained page-level statistics that help range scans and point lookups avoid unnecessary page reads. Bloom filters provide probabilistic membership tests on individual columns and can be used for predicate pushdown even when dictionaries are absent or ineffective. These mechanisms reduce I/O for selective queries, but they operate at fixed Parquet granularity and do not reshape the decoded result into units tailored to a particular downstream memory budget.

B. cuDF Execution and Spill

RAPIDS cuDF [14] is a GPU dataframe library that provides a pandas-like API backed by GPU-accelerated columnar operators. It targets interactive single-node analytics where users load, transform, and query tabular data in notebook-style workflows. cuDF achieves significant speedups over CPU dataframe libraries by exploiting GPU parallelism for operations such as hash joins, group-by aggregation, and string processing [14]. This makes it attractive for workloads where data fits in GPU memory, but GPU memory is typically an order of magnitude smaller than host DRAM, creating a narrow operating envelope [6].

In a typical direct path, Parquet data is read, decoded, and materialized into GPU-resident columnar structures, after which operators such as filter, projection, and group-by run in device memory. GPU memory allocation differs from CPU allocation in a way that matters here. Operators such as filter and hash-based group-by allocate intermediate buffers whose size depends on the input and operator state. If a required device allocation exceeds available memory, the operation fails immediately with an OOM error rather than degrading gradually. This failure mode is not unique to cuDF: other GPU and even CPU query engines that manage their own memory pools (e.g., DuckDB's buffer manager) similarly raise OOM when the working set exceeds the configured memory limit [24]. The difference is that GPU memory limits are much tighter, making the cliff more immediate.

cuDF provides a spill-to-host mechanism that can evict spillable GPU buffers to host memory on demand [15]. Spilling is useful when memory pressure arises from accumulated reusable state, but it does not change the size of the next allocation request. If a single operator requires a contiguous allocation larger than available GPU memory, spill cannot intervene because the allocation fails before any eviction opportunity arises. This distinction is important for understanding when spill can and cannot relieve GPU memory pressure.

C. Storage-Side Filter Offloading

Filter offloading reduces data movement by evaluating filter conditions closer to where data resides instead of transferring all records to the query engine first. In object-storage settings, several systems have explored this idea.

Amazon S3 Select exposes a restricted SQL interface that returns only matching rows from an object [12]. However, it operates on a per-object basis with no cross-object coordination. It supports scalar aggregates (`SUM`, `COUNT`) but not `GROUP BY`, so it cannot produce per-key partial aggregates. Its output is a single filtered or scalar-aggregated result per object, with no mechanism to bound the size of the unit that the client must subsequently admit to memory.

Skyhook embeds data access libraries such as Arrow and Parquet readers into Ceph OSD (Object Storage Daemon) processes, enabling filter and projection pushdown within the storage cluster [13]. While Skyhook demonstrates the feasibility of embedding analytical operators in storage nodes, it is tightly coupled to the Ceph storage stack [25] and requires deploying custom OSD plugins. Additionally, the Skyhook Arrow integration was removed from the Apache Arrow mainline [26], making it incompatible with current Arrow releases. Extending Skyhook to new operators requires modifying and redeploying OSD code, which limits its practical extensibility.

These systems show that storage-side computation can substantially reduce network transfer. However, their primary objective is bandwidth reduction, not explicit control over the decoded transfer unit or the merge state admitted to GPU memory. None of them consider downstream GPU memory constraints or provide grouped partial aggregation needed for group-by workloads.

D. Hash Group-By and Merge-State Growth

Hash-based group-by aggregation maintains aggregate state keyed by the distinct grouping keys. The operator builds a hash table in memory, where each entry stores the grouping key and the running aggregate values (e.g., a running sum and count for each group). On GPUs, the hash table resides in device memory, and its size is determined primarily by the number of distinct groups rather than the number of input rows [27]. A query over millions of rows with a few hundred groups produces a small hash table, whereas a query with millions of distinct key combinations can consume gigabytes of device memory for the hash table alone.

Some aggregation functions are exactly mergeable across partitions. Previous work classify `SUM`, `COUNT`, `MIN`, and `MAX` as distributive aggregates: computing them independently on each partition and merging with the same operator yields the globally correct answer [28]. `AVG` is algebraic: it can be decomposed into `SUM` and `COUNT`, which are each distributive. This property is what makes exact local partial aggregation possible. Each partition can compute a local partial result, and the final answer can be reconstructed by merging those

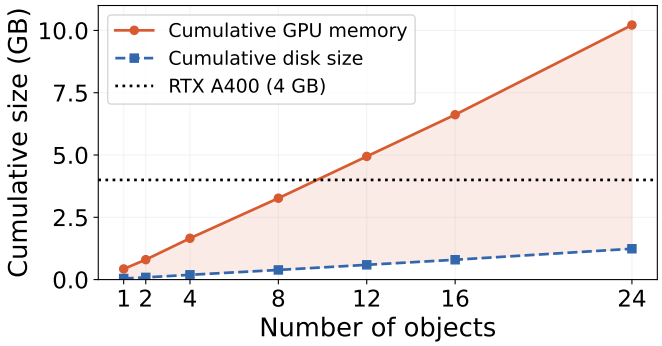


Fig. 1: Cumulative on-disk vs. decoded GPU memory when all columns are decoded (NYC Yellow Taxi 2023 and 2024).

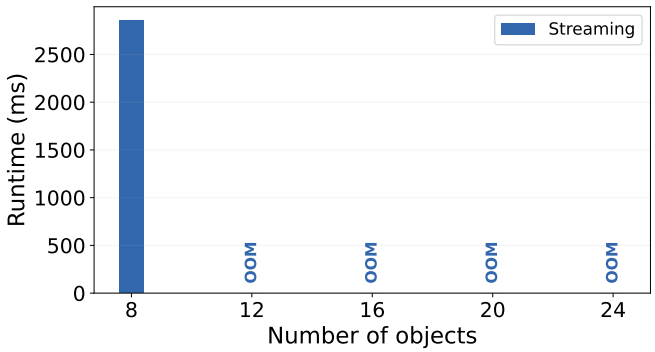


Fig. 2: Merge cliff under streaming delivery (NYC Yellow Taxi, group-by).

partial states without approximation. Holistic aggregates such as `MEDIAN` and `PERCENTILE` require access to all values for a given key and cannot be decomposed into independently mergeable partial states. Unlike distributive aggregates, they do not permit per-object row reduction through partial aggregation.

When data arrives from multiple objects, however, local partial aggregates still have to be combined into a global result. If the per-object partials overlap in key space, the global merge must keep state for all groups seen so far until all contributing partials have been incorporated. On GPUs, this merged state resides in device memory and grows with the accumulated number of distinct keys across all objects. Critically, this merge-state growth is independent of the transfer unit size: even if each arriving partial is small, the shared hash table that accumulates their state can still exceed GPU capacity. This creates a failure mode that cannot be addressed by controlling input granularity alone and motivates key-aware partitioning as a separate mechanism.

III. Motivation

This section uses measured behavior on an NVIDIA RTX A400 GPU (4 GB VRAM, Ampere, FHHL form factor) to demonstrate the two failure modes introduced in Section I. The object store is MinIO [29], an S3-compatible object storage server, accessed over the network. All experiments use the NYC Yellow Taxi dataset (2023 and 2024, one Parquet object per month) and enable cuDF spill-to-host, so the failures shown persist even with the spill mechanism active.

A. Admission Cliff

Parquet files use dictionary encoding, run-length encoding, and general-purpose compression (Snappy, Zstandard) to minimize on-disk footprint. When cuDF decodes a Parquet file into a GPU DataFrame, every encoded column is materialized into its full-width representation and every compressed page is decompressed. The resulting in-memory size can be substantially larger than the on-disk size.

Figure 1 plots the cumulative on-disk Parquet size and the cumulative cuDF in-memory footprint when all columns are decoded. The two curves diverge rapidly as objects accumulate. At 8 objects the on-disk total is approximately 0.4 GB while the decoded GPU footprint already reaches 3.3 GB. By 24 objects the gap widens to roughly 8× (10.2 GB decoded vs. 1.3 GB on disk). Even when a query reads only a subset of columns, the per-column expansion ratio remains, and the gap grows with the number of objects. A planner that estimates GPU working set from compressed Parquet byte counts will systematically underestimate memory demand. Accurately predicting the decoded footprint requires inspecting Parquet-internal metadata to estimate per-column decoded sizes before data is read.

B. Merge Cliff

The admission cliff can be addressed by reducing data on the storage side before transfer. However, existing offloading systems do not address group-by aggregation. S3 Select supports scalar aggregates such as `SUM` and `COUNT` but does not support `GROUP BY`, so it cannot produce per-key partial aggregates. Skyhook focuses on filter and projection within storage nodes. None of these systems provide the grouped partial aggregation needed to reduce data volume for group-by workloads.

Even if storage-side aggregation were available, the per-object computation model that existing systems such as S3 Select employ would not guarantee completion. Figure 2 shows the runtime of a group-by query without filters on the NYC Taxi dataset, grouping by `PULocationID`, `DOLocationID`, and `tpep_pickup_datetime`. In this experiment, we use a streaming approach where the storage side computes a partial aggregate for each object independently and delivers only the reduced results to the GPU. Unlike Figure 1, which shows raw decode of all columns, each streamed piece here contains only the queried columns after aggregation, so the per-unit size is far smaller than the full-decode footprint. The workload succeeds at 8 objects but fails with OOM from 12 objects onward. The failure occurs not during admission but during merge: the GPU must maintain a single hash table to combine partial aggregates across all arriving pieces, and this table grows with the accumulated

number of distinct keys regardless of how small each piece is. This confirms that merge-state growth is a distinct failure mode, independent of transfer-unit sizing.

C. Two Budgets

The observations above identify two distinct memory constraints that govern GPU admission safety.

B_{transfer} bounds the decoded in-memory size of the largest transfer unit admitted to the GPU in one step. The admission cliff (Section III-A) shows that the decoded GPU footprint can be many times larger than the on-disk size, making admission failures unpredictable without metadata-level estimation. Filter offloading partially addresses B_{transfer} only when selectivity is sufficiently low. Under high-pass-rate filters the filtered output can still exceed the admission budget.

B_{merge} bounds the in-flight merge state the GPU must sustain while combining partial aggregates across objects. The merge cliff (Section III-B) shows that even when each transfer unit fits in memory, the accumulated merge state can independently exceed GPU capacity.

Neither budget alone is sufficient: B_{transfer} without B_{merge} leaves the merge cliff open, and B_{merge} without B_{transfer} leaves the admission cliff open. A storage layer that remains oblivious to these downstream GPU constraints can trigger either cliff even when query semantics are preserved. This motivates a fundamental shift in the storage-side computation model. Existing systems decide what to compute based on operator support alone and optimize for bandwidth reduction. The two-budget analysis shows that the storage layer must instead receive the client GPU's capacity and shape its output to respect both admission and merge constraints before data crosses the storage-to-GPU boundary.

IV. Design

To address the two memory cliffs identified in Section III, we propose Scout-and-Reduce (S&R), a storage-side reduction framework for GPU-backed object analytics. S&R reshapes object storage from a passive byte source into a GPU-aware producer that exports semantically exact outputs already sized to the client GPU's admission and merge budgets.

This design delivers three benefits under a fixed GPU capacity. First, it extends the feasible workload envelope: workloads that fail with OOM on the direct path can complete without requiring a larger GPU. Second, it makes admission behavior predictable: instead of discovering whether a workload fits by trial and error, the planner estimates the decoded footprint from Parquet footer metadata and plans delivery against explicit budgets. Third, it redefines storage-side computation for GPU workloads: existing offloading systems provide functional support such as filtering and aggregation closer to storage, but do not consider GPU admission safety or merge-state budgets. S&R makes storage computation aware of the client GPU's memory constraints, shaping its output into units that respect both admission and merge budgets.

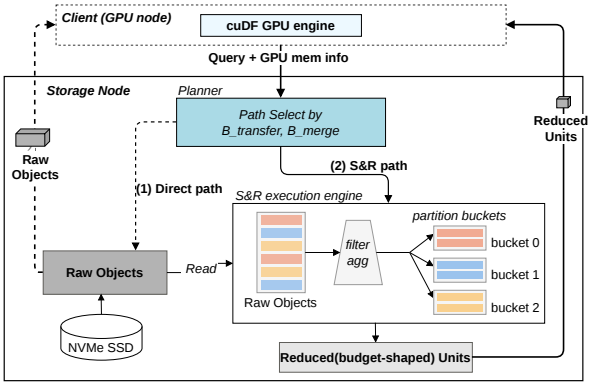


Fig. 3: S&R architecture overview.

A. Overview

S&R consists of two components: a GPU client and a storage-side sidecar co-located with the object store (Figure 3). The client submits a query along with GPU memory information. The sidecar's planner derives two budgets from the reported GPU capacity: B_{transfer}, which bounds the largest unit admitted to the GPU in one step, and B_{merge}, which bounds the in-flight merge state during aggregation. The planner reads the Parquet footer of each object to estimate the decoded GPU footprint, then decides whether the query can proceed on the direct path or should be routed through the S&R execution engine. On the direct path (1), raw Parquet objects are fetched to the GPU without modification. On the S&R path (2), the execution engine performs exact local reduction on each Parquet object using the storage server's CPU and memory, then emits results in units that respect the derived budgets. For high-cardinality group-by, the engine partitions results into key-disjoint buckets, each bounded by B_{merge}. When filters accompany aggregation, the engine applies the filter first and then computes partial aggregates over the surviving rows, so the merge-state estimate is based on the actual post-filter cardinality rather than a pre-execution guess. The client fetches these units and performs final processing on the GPU. The sidecar runs on the storage server using only existing CPU and memory resources and requires no specialized hardware.

B. Budget Derivation

The client reports the target GPU's total and available memory along with each query. The sidecar's planner derives two budgets from this information.

B_{transfer} bounds the decoded in-memory size of the largest unit admitted to the GPU in one step. B_{merge} bounds the in-flight merge state the GPU must sustain while combining results across objects. Both budgets are drawn from the same effective headroom $T = M_{\text{gpu}} - \text{overhead}$. The overhead term captures GPU memory occupied independently of the query, primarily the CUDA context and the cuDF runtime's internal pools. Because the decoded data and the merge state co-exist in GPU memory, the admission check is $F_{\text{total}} + H \leq T$ (Section IV-D). In Level 2, the reduced data size R is known

after execution, so the remaining headroom available for merge state is $T - R$. This bound is conservative: under partitioned delivery the GPU holds only one bucket at a time, so the actual available headroom per bucket exceeds $T - R$.

Because the budgets are derived from the GPU memory capacity received with each query, the same sidecar serves GPUs of different capacities without reconfiguration. This makes GPU memory capacity a first-class parameter of the storage-client interface. Unlike existing storage-side systems, which produce output independent of the client's hardware constraints, S&R couples its delivery decisions to the reported capacity, so the same object may be delivered raw to a 40 GB GPU but reduced and partitioned for a 4 GB GPU. When the planner's footer-based estimate indicates that all objects already fit within the budgets, it selects the direct path with only lightweight metadata inspection overhead.

C. Exact Local Reduction

The sidecar performs exact local reduction on each Parquet object. Exact means that processing each object separately on the storage side and then merging the per-object results on the GPU produces the same answer as loading all objects into the GPU and processing them in a single pass.

S&R targets operators that can reduce data volume on the storage side, bringing the transferred output within the GPU's admission budget. It handles three configurations. When only filter predicates are present, the sidecar evaluates them and emits only matching rows with optional column projection, reducing the transfer unit proportionally to the filter selectivity. When only group-by aggregation is present, the sidecar computes local partial aggregates over each object, collapsing potentially millions of rows into a much smaller set of per-key aggregate state. When both are present, the sidecar first applies filters to reduce the input and then computes partial aggregates over the surviving rows, achieving the largest reduction.

The degree of row reduction depends on the aggregate type. Distributive aggregates (`SUM`, `COUNT`, `MIN`, `MAX`) and algebraic aggregates (`AVG`, decomposed into `SUM` and `COUNT`) can be computed independently on each object and merged exactly [28]. For these functions, partial aggregation collapses many input rows into one output row per group, achieving significant transfer-volume reduction. Holistic aggregates such as `MEDIAN` require access to all values for a given key and cannot be partially aggregated, so the sidecar cannot reduce row count for these functions. The partitioned delivery mechanism (Section IV-E) bounds per-bucket scope for all aggregate types through key-disjoint bucketing, though the current merge-state bound H (Eq. 4) and evaluation target distributive and algebraic functions.

D. Two-Level Planner

The planner operates in two levels. Level 1 runs on the sidecar before any data pages are read and decides whether to send raw objects directly to the GPU or to activate storage-side reduction. Level 2 runs after reduction has been performed and selects the delivery mode.

Level 1: Footer-based route selection. The planner reads the Parquet footer of each object via a range read of the final few kilobytes of the file. The footer contains the file schema, per-row-group row counts, and per-column-chunk compressed and uncompressed byte sizes. Because cuDF materializes decoded Parquet data as Apache Arrow columnar arrays in GPU memory, the estimate must reflect the Arrow in-memory layout rather than the compressed Parquet representation.

For each column c in the set of columns $\mathcal{C}$ referenced by the query, the estimated decoded size in row group g is:

$$D(c,g) = \begin{cases} N_g \times w(c) & \text{if } c \text{ is fixed-width} \\ U(c,g) & \text{if } c \text{ is variable-length} \end{cases} \tag{1}$$

where N_g is the number of rows in row group g, $w(c)$ is the physical type width in bytes (e.g. 4 for `INT32`, 8 for `INT64` and `DOUBLE`), and $U(c,g)$ is the uncompressed size recorded in the column chunk metadata for variable-length types such as `BYTE_ARRAY`.

The per-object footprint estimate sums over all row groups and queried columns, representing the maximum data volume the object can contribute before any filter or aggregation reduction:

$$F_o = \sum_{g \in \mathcal{G}_o} \left(\sum_{c \in \mathcal{C}} D(c,g) \; + \; |\mathcal{C}| \times \left\lceil \frac{N_g}{8} \right\rceil \right) \tag{2}$$

where the second term accounts for the per-column validity bitmap that each nullable Arrow array may maintain to track null entries. Although Eq. 2 accounts for value buffers and validity bitmaps, it estimates the logical decoded footprint rather than the exact physical allocation made by the GPU dataframe runtime. Small physical-layout effects such as Arrow buffer alignment, allocator granularity, and padding are not represented in Parquet footer metadata. We therefore apply a fixed layout-correction factor γ to convert the logical footer estimate into a conservative physical planning estimate. The value of γ adds a small fixed margin for these layout effects and is held constant across all queries, object counts, and planner decisions. It is not tuned from query success/failure outcomes or end-to-end runtime. We use $\gamma = 1.03$ in all experiments and evaluate the resulting estimation accuracy in Section V-I. The cumulative footprint estimate across all queried objects is:

$$F_{\text{total}} = \gamma \cdot \sum_{o \in \mathcal{O}} F_o \tag{3}$$

For group-by queries, the planner additionally estimates a merge-state upper bound. During GPU hash group-by execution, three objects co-exist in device memory at peak: the input DataFrame, the hash table being built, and the output result materialized from the hash table. In the worst case every row carries a unique key, so the total memory occupied by the hash table and the output, which co-exist at peak, is bounded by:

$$H = \left(\sum_{o \in \mathcal{O}} N_o \right) \times (w_{\text{key}} + w_{\text{agg}}) \times \lambda \tag{4}$$

where N_o is the total row count of object o, w_{key} is the sum of type widths of the group-by key columns, w_{agg} is the sum of per-group state widths across all aggregate functions in the query (e.g. 8 bytes for `SUM`, 8 bytes for `COUNT`, $w(c)$ for `MIN`/`MAX`), and λ is a memory multiplier. GPU hash tables typically maintain around 50% occupancy, requiring approximately twice the entry count in allocated slots. The output result co-exists with the hash table at peak and, for all supported aggregate types (distributive and algebraic), the per-group output is at most as large as the per-group aggregate state. Together these yield $\lambda \approx 3.0$ as a conservative default. For queries without group-by, $H = 0$.

The planner checks whether the total GPU memory demand fits within the available capacity. Let $T = M_{\text{gpu}} - \text{overhead}$ denote the effective GPU headroom, as defined in Section IV-B. The route decision depends on the query type.

When both filter and group-by are present, the planner activates S&R unconditionally. The reason is that Level 1 has no access to data pages and therefore cannot determine filter selectivity. Without selectivity, the post-filter cardinality is unknown, making the merge-state bound H impossible to estimate. Level 2 resolves this by applying the filter via eager execution and using the realized cardinality to compute H exactly.

For filter-only queries, cuDF materializes both the input DataFrame and the filtered output simultaneously, so the peak GPU footprint is bounded by input plus output. Under worst-case selectivity, the planner selects the direct path if:

$$2 \cdot F_{\text{total}} \;\leq\; T \tag{5}$$

For group-by-only queries, the input, hash table, and output co-exist at peak. The planner selects the direct path if:

$$F_{\text{total}} \;+\; H \;\leq\; T \tag{6}$$

If the applicable condition is not met, the planner activates the S&R execution engine to reduce data on the storage side before transfer. Because overestimation only adds sidecar latency while underestimation can cause GPU OOM, the planner defaults to a conservative bias. We evaluate the accuracy of this estimator in Section V-I.

Level 2: Post-reduction delivery selection. Once the engine has performed reduction on an object, it knows the exact size of the reduced output. This is the key property that makes the reduction double as a scouting signal: no separate probe step is needed, and the reduced result is unconditionally reused regardless of the delivery decision. Based on the realized output size, the engine selects one of the following delivery modes.

For filter-only queries, the engine applies the filter on each object independently and streams the per-object filtered results to the GPU, so the transfer unit is bounded by the single-object filtered output rather than the cumulative result across all objects. When both filter and group-by are present, the engine applies the filter first via eager execution and observes the post-filter cardinality. It then computes partial aggregates over the surviving rows and uses the actual result size to decide whether partitioned delivery is needed. For group-by queries, the engine estimates the post-reduction merge state M_{est} from the realized partial-aggregate sizes: it sums the actual number of distinct groups observed across all reduced objects and multiplies by the per-group entry size and memory multiplier λ. If M_{est} exceeds $B_{\text{merge}} = T - R$, where R is the total reduced data size, the engine switches to partitioned

Algorithm 1 Two-level planner

Require: Query Q, object set $\mathcal{O}$, GPU memory M_{gpu}
Ensure: Delivery plan for each object

$T \leftarrow M_{\text{gpu}} - \text{overhead}$
{**Level 1: footer-based route selection**}
for each $o \in \mathcal{O}$ **do**
  Read Parquet footer, compute F_o via Eq. (2)
end for
$F_{\text{total}} \leftarrow \gamma \cdot \sum_o F_o$
if Q has filter and group-by **then**
  goto Level 2 {selectivity unknown}
else if Q has group-by **then**
  Compute H via Eq. (4)
  if $F_{\text{total}} + H \leq T$ **then**
    return DIRECT
  end if
else
  if $2\,F_{\text{total}} \leq T$ **then**
    return DIRECT
  end if
end if
{**Level 2: reduce, then decide delivery**}
for each $o \in \mathcal{O}$ **do**
  if Q has filter **then**
    Apply filter (eager)
  end if
  if Q has group-by **then**
    Compute partial aggregate for o, measure R_o
  end if
end for
if Q has group-by **then**
  $R \leftarrow \sum_o R_o$ {total reduced data size}
  $M_{\text{est}} \leftarrow \sum_o n_{\text{groups},o} \times (w_{\text{key}} + w_{\text{agg}}) \times \lambda$
  $B_{\text{merge}} \leftarrow T - R$ {remaining headroom after reduced data}
  if $M_{\text{est}} \leq B_{\text{merge}}$ **then**
    Emit all partials as SINGLE
  else
    $P \leftarrow \lceil M_{\text{est}} \,/\, B_{\text{merge}} \rceil$
    Re-partition all partials into P buckets, emit PARTITIONED
  end if
else
  Emit per-object filtered results
end if

delivery (Section IV-E) to bound per-bucket merge state. If the query lacks reducible operators and the engine cannot shrink the data, it falls back to returning the original object URL so the client can attempt the direct path or report the workload as infeasible for the current budget. Algorithm 1 summarizes the complete two-level decision procedure.

Even when each transfer unit fits within B_{transfer}, the GPU can still exhaust memory during group-by merge. The problem is not the final aggregation result, which may be compact, but the intermediate hash table the GPU must build and maintain during the merge process. When partial aggregates from multiple objects are merged without regard to key distribution, the GPU sustains a single hash table whose size grows with the accumulated number of distinct keys across all arriving units. This intermediate state can far exceed the final output size, especially with high-cardinality keys.

E. Partitioned Merge

Partitioned merge solves this problem by making delivery key-aware. The engine hashes each object's local partial aggregates on the group-by keys into P key-disjoint buckets: $\text{bucket\_id} = \text{hash}(\text{group\_by\_keys}) \bmod P$. Because the same key combination always maps to the same bucket, the GPU can merge one bucket at a time across all objects, finalize its result, free the associated memory, and move to the next bucket. This bounds the intermediate merge state per bucket to B_{merge}.

Choosing P. The planner computes the minimum partition count that, under near-uniform hash distribution, keeps the expected per-bucket merge state within B_{merge}, the remaining GPU headroom after the reduced data:

$$P = \left\lceil \frac{M_{\text{est}}}{B_{\text{merge}}} \right\rceil, \quad B_{\text{merge}} = T - R \tag{7}$$

where M_{est} is the estimated merge state from realized partial-aggregate sizes and R is the total reduced data size. A smaller P incurs less partitioning overhead but produces larger per-bucket state. A larger P reduces per-bucket state but increases the number of transfer rounds and the associated I/O. P is therefore a runtime-memory tradeoff knob rather than a value that can be fixed at design time. The evaluation confirms this with a sweep over P values (Section V-F). Because M_{est} sums local distinct-group counts across objects without deducting cross-object key overlap, P may exceed the minimum required value. This is conservative: it increases the number of I/O rounds but does not produce incorrect results or allow an OOM that a tighter estimate would have prevented.

Correctness. Partitioned merge preserves exactness for distributive and algebraic aggregates. Each local partial aggregate is exactly mergeable (Section II-D), and key-disjoint bucketing guarantees that all partial states for a given key combination reside in the same bucket, so per-bucket merge produces the same result as a global single-pass aggregation. This paper focuses on distributive and algebraic functions, for which the per-bucket merge state remains bounded by B_{merge}.

V. Evaluation

We evaluate S&R on two datasets to answer four questions: (1) Does the memory cliff arise across different datasets and query types? (2) Does S&R sustain completion where existing paths fail? (3) Does the footer-based planner accurately estimate GPU memory demand? (4) How does S&R compare against client-side reduction? We begin with the OTTO recommendation-event dataset [20] to establish that the cliff and S&R's benefits generalize beyond a single schema, then conduct detailed experiments on the NYC Yellow Taxi dataset [19].

A. Experimental Setup

Hardware. The GPU node is equipped with a 24-core CPU (2.3 GHz), 96 GB DRAM, and an NVIDIA RTX A400 GPU (4 GB VRAM, Ampere, FHHL form factor). The MinIO storage server runs on a separate node with a 16-core CPU (2.1 GHz), 32 GB DRAM, and two 1 TB NVMe SSDs. The two nodes are connected via Mellanox ConnectX-5 100 GbE.

Software. The S&R sidecar is implemented using Polars 1.39 [30], co-located with MinIO on the storage server. The GPU client uses cuDF 26.02 with spill-to-host enabled in all experiments.

Datasets. We use two datasets: (1) OTTO recommendation-event dataset, with per-object on-disk size of approximately 120 MB, (2) NYC Yellow Taxi dataset (2023, 2024), consisting of 24 monthly Parquet objects with a cumulative on-disk size of approximately 1.3 GB.

Comparison. We compare three execution paths:

- **Direct**: the client fetches raw Parquet objects and loads them into cuDF on the GPU without any storage-side processing.
- **Filter Offload**: the storage side applies filter offloading before transfer, analogous to existing systems such as S3 Select. Group-by aggregation is not supported.
- **S&R (proposed)**: the full S&R pipeline with footer-based planning, exact local reduction, and key-disjoint partitioned delivery.

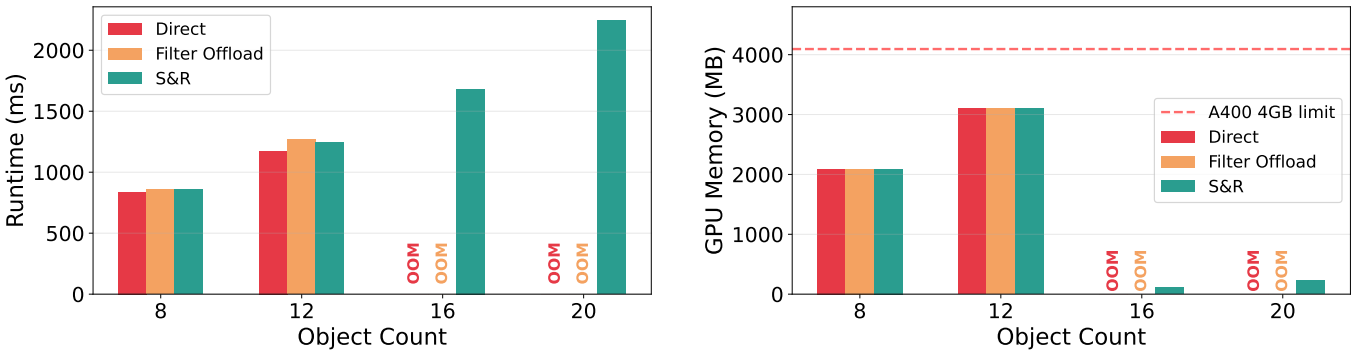
All configurations run with cuDF spill-to-host enabled. For queries without filter predicates, Filter Offload is functionally equivalent to Direct because there is no filter to offload.

Methodology. Each experiment is repeated 3 times. We report the median, and error bars in all figures show the min–max range across the three runs. We measure end-to-end runtime, peak GPU memory, and completion status (success or OOM).

B. Memory Cliff on OTTO Dataset

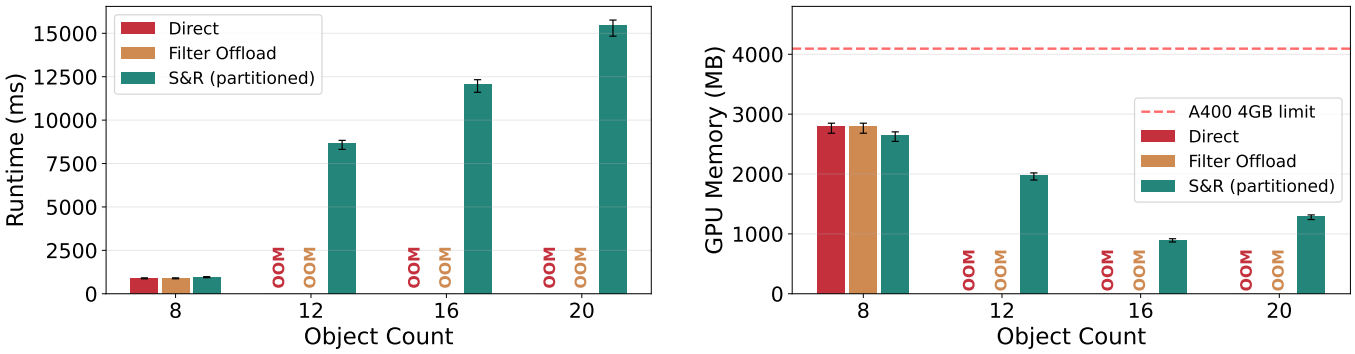
We first verify that the memory cliff arises on a dataset with different schema and per-object characteristics than NYC Taxi.

Filter-only (Figures 4a and 4b). Direct and Filter Offload succeed at 8 and 12 objects but fail with OOM from 16 objects onward. S&R succeeds at all tested scales by applying storage-side filtering before transfer, keeping peak GPU memory well below the 4 GB limit. This demonstrates the admission cliff (B_{transfer}) on a dataset with larger per-object decode expansion,



(a) Runtime. (b) GPU memory.

Fig. 4: OTTO filter-only query. Dashed line marks the A400 4 GB device limit.



(a) Runtime. (b) GPU memory.

Fig. 5: OTTO group-by query. Dashed line marks the A400 4 GB device limit.

confirming that admission control is necessary even for filter-only workloads when per-object size is large.

Group-by-only (Figures 5a and 5b). Direct and Filter Offload fail from 12 objects onward. Filter Offload provides no benefit here because the query contains no filter predicate, leaving the group-by merge state as the sole source of OOM. S&R with partitioned delivery succeeds at all scales, bounding the intermediate merge state per bucket.

Having established that both cliffs arise on OTTO, the remaining experiments use the NYC Yellow Taxi dataset for detailed analysis of S&R's mechanisms.

C. High-Cardinality Group-By

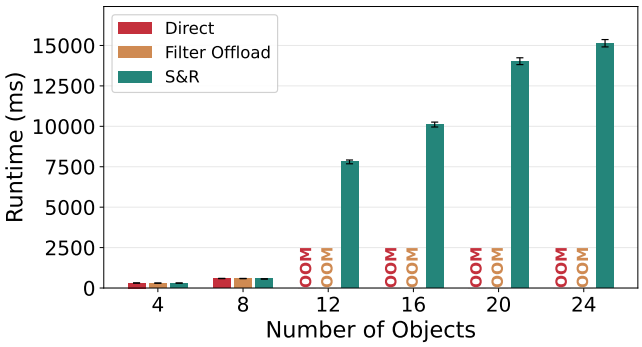
We first evaluate the merge cliff identified in Section III-B under a high-cardinality group-by without filters. The query groups by `PULocationID`, `DOLocationID`, and `tpep_pickup_datetime`, and computes `SUM(fare_amount)` and `COUNT(*)`. Figures 6a and 6b show the runtime and peak GPU memory, respectively.

`Direct` and `Filter Offload` both succeed at 4 and 8 objects but fail with OOM from 12 objects onward. As shown in Figure 6b, their peak GPU memory approaches the 4 GB device limit at 8 objects and exceeds it at 12, confirming that the failure is caused by the intermediate merge hash table growing with the accumulated distinct-key count. S&R succeeds at all tested scales up to 24 objects by applying partitioned merge with key-disjoint bucketing, keeping peak GPU memory well below the device limit. The runtime overhead of S&R relative to the direct path at 4–8 objects reflects the cost of storage-side reduction and partitioned delivery. However, this overhead is the cost of completion: at 12 objects and beyond, the direct path produces no result at all.

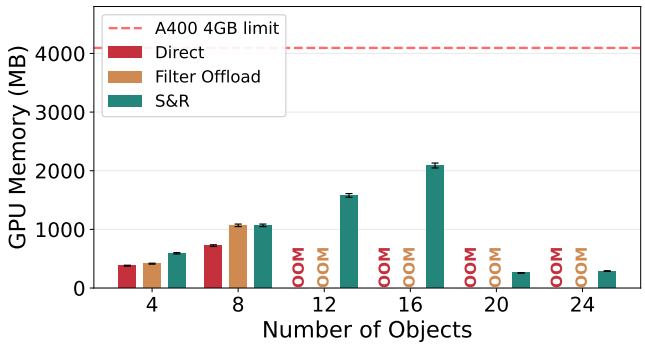
D. Low-Cardinality Group-By

We next test a lower-cardinality group-by to examine whether S&R adds unnecessary overhead when the direct path can already succeed. The query groups by `PULocationID` × `DOLocationID` (approximately 65 000 potential groups), which produces a much smaller merge state than the high-cardinality case. Figure 7 shows the results.

`Direct` and `Filter Offload` succeed up to 20 objects but fail with OOM at 24. The merge cliff arrives later because the per-key state is smaller, but it still arrives. S&R succeeds at all scales, including 24 objects where the direct path fails. At smaller scales (4–12 objects) where the direct path succeeds, S&R incurs higher runtime due to storage-side reduction overhead. This confirms that S&R is not intended to replace



(a) Runtime.



(b) GPU memory.

Fig. 6: High-cardinality no-filter group-by (`PU × DO × tpep_pickup_datetime`). Dashed line marks the A400 4 GB device limit.

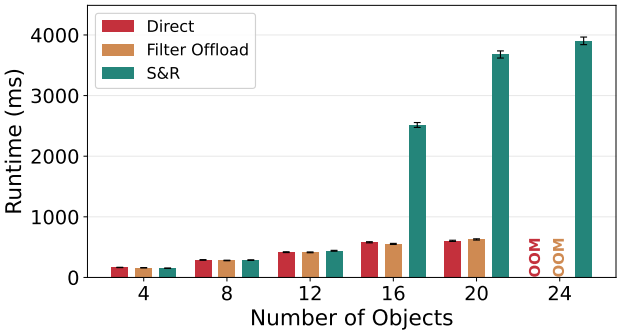


Fig. 7: Low-cardinality no-filter group-by (`PU × DO`).

the direct path when the workload already fits, but to extend the completion envelope when it does not.

E. Filter + Group-By with Varying Selectivity

We evaluate the Level 2 eager-filter path by combining a filter predicate with the high-cardinality group-by. The filter selects trips originating from a single `PULocationID` value at three selectivity levels: ∼24% (`PU = 64`), ∼6% (`PU = 16`), and ∼1.5% (`PU = 4`). Figure 8 shows the runtime and Figure 9 shows the peak GPU memory for the highest-selectivity case.

`Direct`, `Filter Offload`, and S&R succeed throughout because the filter reduces the post-filter cardinality enough to keep the merge state within GPU capacity at all three selectivities. At low selectivity (∼1.5%), all three paths perform similarly because the filter eliminates most rows before aggregation. At moderate and high selectivity (∼6% and ∼24%), the gap between the direct path and both S&R and `Filter Offload` widens as the object count grows, because the post-filter data fits entirely in GPU memory and executing filter and group-by on the GPU in a single pass is faster than performing the filter on the storage side first.

In all three cases, the Level 2 planner observes the realized post-filter cardinality, correctly determines that the merge state fits within B_{merge}, and selects single-unit delivery without activating partitioned merge, confirming that the eager-filter path avoids unnecessary partitioning.

TABLE I: Sidecar overhead breakdown at 24 objects.

	High-card	Low-card
End-to-end runtime (ms)	15 138	3 903
Sidecar total (ms)	7 160	1 097
Download (ms)	1 226	907
Compute (ms)	2 292	688
Upload (ms)	2 529	174
Client GPU merge (ms)	7 978	2 806
Partitions (P)	8	8
Transfer units	192	192
Peak GPU memory (MB)	1 107	73

F. Partition Count Sweep

We sweep the partition count $P \in \{2, 4, 8, 16\}$ alongside the automatic selection ($P = \lceil M_{\text{est}}/B_{\text{merge}} \rceil$) to validate that P is a meaningful runtime-memory tradeoff knob. All runs use forced S&R to isolate the effect of P from the planner's route decision. Figure 10 shows the results.

At small scale (4–8 objects), all values of P produce similar runtime because the merge state is small regardless of partitioning. As the object count grows, the gap widens. $P = 2$ has the lowest per-round overhead but produces the largest per-bucket merge state, which can cause OOM at larger scales. $P = 16$ keeps per-bucket state minimal but incurs the highest runtime due to more transfer rounds. The automatic selection tracks $P = 4$ at moderate scales and increases to higher values only when needed, confirming that the ceiling formula (Eq. 7) selects the minimum P that keeps the merge state within budget.

This validates the design choice of computing P dynamically rather than fixing it at design time. A fixed P would either waste I/O rounds at small scales or risk OOM at large scales.

G. Sidecar Overhead Breakdown

To understand where S&R spends its time, we instrument the sidecar pipeline into three phases: object download from

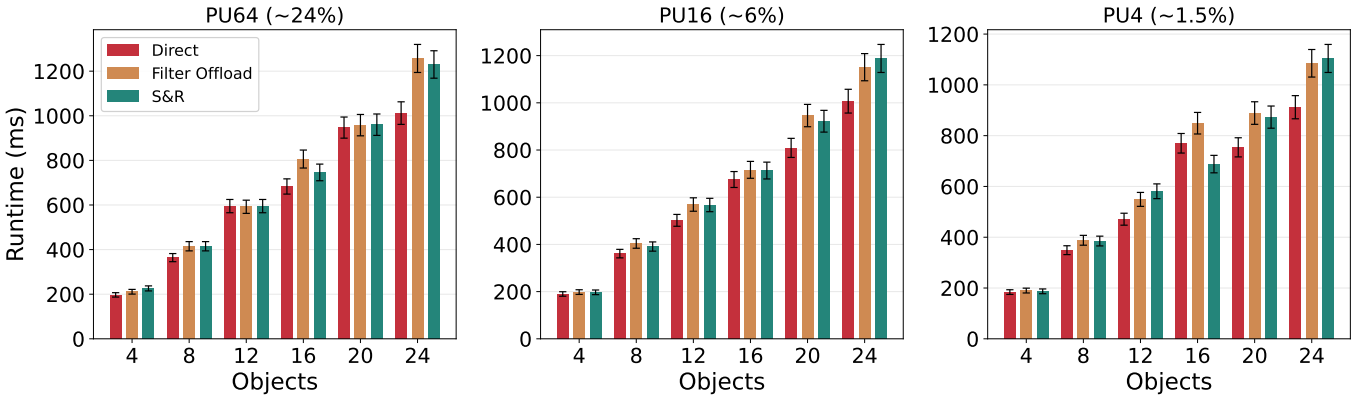


Fig. 8: Filter + group-by with varying selectivity. Left: `PU = 64` (∼24%). Center: `PU = 16` (∼6%). Right: `PU = 4` (∼1.5%).

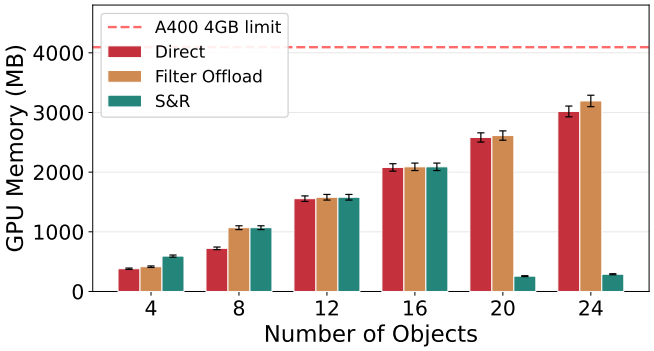


Fig. 9: Peak GPU memory for the filter + group-by case (`PULocationID = 64`, ∼24% selectivity).

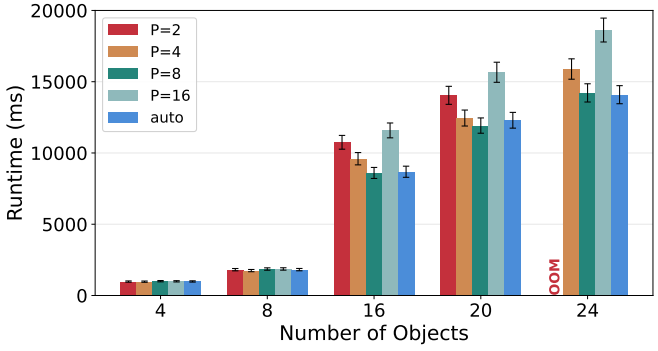


Fig. 10: Runtime vs. partition count P for the high-cardinality group-by.

MinIO, compute (reduction and partitioning on Polars), and result upload back to the object store. The three phases are pipelined across objects, so their per-phase totals do not sum to the sidecar total. Table I reports the breakdown at 24 objects for the high-cardinality and low-cardinality group-by workloads.

Both workloads use the same partition count ($P = 8$) and the same number of transfer units (192), so the structural overhead of partitioned delivery is identical. The runtime difference is driven by two factors. First, the high-cardinality query produces larger partial aggregates per object, increasing both upload volume and GPU merge time. Second, the client-side GPU merge is proportional to the number of distinct keys that must be combined across buckets, which is substantially larger in the high-cardinality case. In the low-cardinality case, the sidecar accounts for roughly 28% of end-to-end runtime, while in the high-cardinality case it accounts for 47%. In both cases, the compute phase (Polars reduction and partitioning) is not the dominant cost.

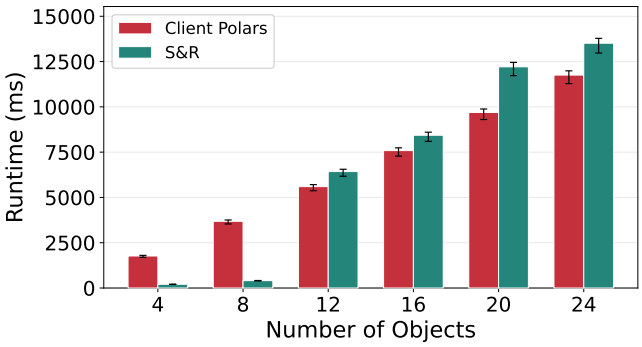
H. Comparison with Client-Side Reduction

An alternative to storage-side reduction is to download the query-referenced columns to the client and reduce on the client CPU before GPU admission. We implement this as a `Client Polars` pipeline: the client downloads only the needed columns from MinIO, performs group-by aggregation and cross-object merge entirely on the client CPU using Polars, and streams the final merged result to the GPU. Unlike the streaming approach in Section III-B, where the GPU receives per-object partial aggregates and must build a merge hash table, Client Polars completes all merging on the CPU and transfers only the final merged result to the GPU.

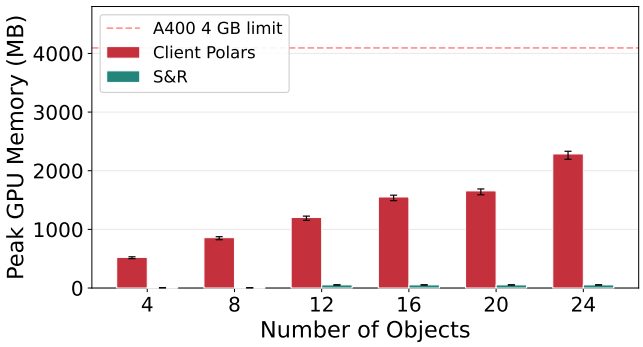
Figures 11a and 11b compare the two approaches at 100 GbE for the high-cardinality group-by.

Figure 11a shows Client Polars achieves lower end-to-end runtime than S&R across larger scales, because it avoids the sidecar upload phase and benefits from the high-bandwidth link for raw column download. However, this runtime advantage depends on the high-bandwidth local link used in our testbed (100 GbE). In typical cloud object-storage deployments, client-to-storage bandwidth is often limited to 1–10 Gbps, which would increase the raw column download cost for Client Polars proportionally while S&R's reduced transfer volume remains unaffected. Under such bandwidth constraints, the runtime gap narrows or reverses in favor of S&R.

Figure 11b reveals a critical difference in GPU memory consumption. Client Polars must materialize the entire merged result on the GPU in a single handoff: for the high-cardinality



(a) Runtime.



(b) Peak GPU memory.

Fig. 11: S&R vs. Client Polars (high-cardinality group-by, 100 GbE).

group-by, this produces over 7.6 million output rows, and converting this result from Arrow to cuDF consumes 2.3 GB of GPU memory at 24 objects. In contrast, S&R uses partitioned delivery: the GPU receives one key-disjoint bucket at a time and merges it. Because cuDF spill-to-host is enabled, previously merged bucket results are evicted to host memory as new buckets arrive, so only one bucket occupies GPU memory at any given time. The peak GPU memory is therefore bounded by the largest single bucket rather than the total result, keeping it at well below the device limit, regardless of the overall data scale.

Client Polars shifts the entire reduction and merge burden to the client. Its CPU merge phase dominates runtime, accounting for over 75% of end-to-end time at 24 objects. This cost scales with the total row count and is independent of network bandwidth. Furthermore, Client Polars requires the client to have sufficient CPU and memory resources for the full cross-object merge, which may not be available on lightweight GPU workstations. More fundamentally, Client Polars peak GPU memory grows linearly with the number of objects because the Arrow-to-cuDF conversion materializes the entire merged result on the GPU in a single allocation, leaving no opportunity for spill to intervene. At 24 objects the peak reaches 2.3 GB. S&R sustains completion where the direct path and filter offloading fail, while keeping peak GPU memory far lower than client-side reduction and preserving headroom for larger scales.

I. Estimation Accuracy

We evaluate the two estimation stages of the Level 1 planner: per-object footprint estimation (Eq. 2) and peak GPU memory prediction (Eq. 5, 6).

Per-object footprint. We compute the footer-based footprint estimate F_o for all 24 NYC Taxi Parquet objects (2023 and 2024 combined, 24 files) and compare against the actual cuDF in-memory size when all columns are decoded. Table II summarizes the results. Across all files, the raw estimate (before applying γ) underestimates the actual decoded size by 1.8% on average, with a maximum deviation of 1.9%. The gap is due to Arrow buffer alignment padding, which the footer metadata does not capture. Applying the safety factor $\gamma = 1.03$ brings the estimate to approximately 1.2% above the actual size, making the planning estimate consistently conservative. This confirms that the Parquet footer provides sufficient information for accurate GPU footprint estimation without reading any data pages.

TABLE II: Footer-based footprint estimation accuracy (24 NYC Taxi files, all columns decoded).

	Estimated	Actual	Ratio
Per-file range (MB)	382–494	389–503	0.981–0.983
Total (MB)	10 269	10 462	0.982
Mean underestimation			1.8%
After $\gamma = 1.03$			+1.2% (conservative)

TABLE III: Peak GPU memory prediction vs. actual outcome. ✓: prediction matches actual. **C**: conservative (predicts OOM but actual succeeds).

	High-card		Low-card	
Obj	Est. peak	Actual	Est. peak	Actual
4	1 607 MB	OK ✓	1 221 MB	OK ✓
8	3 199 MB	OK ✓	2 430 MB	OK ✓
12	4 859 MB	OOM ✓	3 690 MB	OK ✓
16	6 517 MB	OOM ✓	4 949 MB	OK **C**
20	8 206 MB	OOM ✓	6 232 MB	OK **C**
24	10 081 MB	OOM ✓	7 656 MB	OOM ✓

Peak GPU memory prediction. The planner uses the footprint estimate together with the merge-state bound H (Eq. 4) to predict whether a workload fits in GPU memory. Since H assumes the worst case where every row carries a unique key ($n_{\text{groups}} = n_{\text{rows}}$), the prediction is inherently conservative. Table III shows the estimated peak versus the actual outcome for two group-by cardinalities.

For the high-cardinality group-by (`PU×DO×tpep_pickup_datetime`), the prediction matches the actual outcome at every scale. The estimated peak crosses the 4 GB device limit between 8 and 12 objects, exactly where the direct path begins to fail.

For the low-cardinality group-by (`PU×DO`), the prediction is conservative at 16 and 20 objects: the planner predicts OOM, but the direct path actually succeeds. This is because the worst-case assumption $n_{\text{groups}} = n_{\text{rows}}$ greatly overestimates

the actual number of distinct groups. The low-cardinality query produces approximately 53 000 distinct groups, far fewer than the 51–65 million rows used as the upper bound. In these cases the planner routes the query through S&R unnecessarily, adding sidecar overhead but preserving correctness. This is the expected behavior of a conservative planner: it may sacrifice runtime for safety, but it never allows an OOM that could have been prevented.

VI. Related Work

Distributed GPU dataframes. Dask-cuDF and the RAPIDS Accelerator for Apache Spark extend cuDF to multi-GPU and multi-node settings [31], [32]. Dask-cuDF partitions DataFrames across workers and coordinates execution through Dask's task graph, while the RAPIDS Accelerator maps Spark SQL operators to GPU-accelerated kernels across a Spark cluster. Both address GPU memory limits through scale-out, distributing data across multiple GPUs rather than fitting it into one. However, distributed execution introduces cluster management overhead, inter-node communication, and task scheduling complexity that interactive single-node notebook workflows seek to avoid [1]. S&R targets the single-node setting where scale-out is not available or desirable, extending the feasible workload envelope of a single GPU through storage-side reduction rather than horizontal scaling.

GPU analytics over object storage. BlazingSQL executes SQL queries on cuDF DataFrames over local or cloud bucket storage [3]. Theseus runs GPU queries directly over remote Parquet files on S3 [4]. These systems assume that queried data fits in GPU memory or rely on chunked execution without explicit budget control. S&R addresses the gap between object-stored data and GPU memory capacity by interposing a budget-aware reduction layer between the object store and the GPU client.

GPU query engines and out-of-core execution. Several systems address GPU memory limitations through engine-internal mechanisms. Vortex introduces a GPU-aware query optimizer that partitions operators across GPU and CPU based on memory availability [6]. Mordred uses heterogeneous memory management to spill intermediate state between GPU and host memory during query execution [7]. These approaches manage memory pressure from within the query engine, assuming the engine controls the full data pipeline and has sufficient CPU and memory resources for out-of-core management. In object-storage settings where the client is a lightweight notebook workflow, the engine may lack the memory or CPU budget for out-of-core management, making storage-side preshaping a more natural intervention point. S&R operates at a different layer: it shapes data before it reaches the GPU engine, so the two approaches are complementary. Budget-shaped output from S&R could serve as input to an out-of-core GPU engine, combining storage-side reduction with engine-internal memory management.

Storage-side filter offloading. S3 Select [12] and Skyhook [13] demonstrate that pushing filter evaluation closer to storage reduces data transfer. However, S3 Select operates per-object with no cross-object coordination and does not support `GROUP BY`, so it cannot produce per-key partial aggregates. Skyhook is tightly coupled to Ceph and requires custom OSD plugins. Its Arrow integration was removed from the Apache Arrow mainline, making it incompatible with current releases. Since neither system is directly deployable for comparison, we implement a `Filter Offload` baseline that reproduces the same filter-offloading strategy and compare against it experimentally (Section V-C). S&R differs from these systems in two respects. First, it supports partitioned partial aggregation, not just filter offloading. Second, it shapes its output to explicit GPU memory budgets rather than optimizing solely for bandwidth reduction.

TABLE IV: Merge cliff across GPU capacities. Query: `GROUP BY (PULocationID, DOLocationID, tpep_pickup_datetime)` over NYC Yellow Taxi monthly files (2022–2025). GPU memory varies from 4 GB (A400) to 10–80 GB (A100 MIG).

GPU Mem.	Cliff (file)	Groups	Peak	Merge-state growth
3.7 GB	12 (OOM)	34.6 M	2.41 GB	98 MB/file
9.5 GB	30 (OOM)	93.9 M	6.46 GB	100 MB/file
19.5 GB	∼62 (proj.)	161.6 M*	11.10 GB*	106 MB/file
39.3 GB	∼125 (proj.)	161.6 M*	11.10 GB*	106 MB/file
79.3 GB	∼253 (proj.)	161.6 M*	11.10 GB*	106 MB/file

*Values at end of dataset (47 files), no OOM observed.
Projected cliffs extrapolated from the observed rate of 0.31 GB per file.

Computational storage and near-data processing. Computational storage devices (CSDs) and smart SSDs embed processing elements (FPGAs, ARM cores) inside storage hardware to reduce data movement [33]. Recent work explores SQL offloading on CSDs for scan and filter operations [34], [35]. These systems target fixed-function acceleration at the device level and typically do not address cross-object aggregation or GPU-specific memory constraints. S&R achieves storage-side reduction using commodity server CPUs without requiring specialized storage hardware, making it deployable on any object-storage node.

In summary, existing storage-side systems are designed around a bandwidth-reduction objective: they minimize transferred bytes but produce output without regard for the client's memory constraints. Engine-internal systems manage GPU memory reactively but assume the engine controls the full pipeline. S&R bridges this gap by making the client GPU's capacity a query-time input to the storage layer, so the storage-side output is shaped to budgets derived from the client rather than optimized for bandwidth alone. This positions storage-side computation as a proactive safety layer for GPU analytics, complementary to engine-internal memory management.

VII. Discussion

Generality beyond 4 GB GPUs. To confirm that the memory cliff is not an artifact of the 4 GB A400, we repeat the merge experiment on an Elice Cloud [36] G-NAHP-80 instance equipped with an NVIDIA A100 80 GB PCIe GPU. We extend

the dataset to 2022–2025 (up to 47 monthly files) and use MIG partitioning to expose 10, 20, 40, and 80 GB of device memory while keeping all other variables constant (Table IV). Note that the usable capacity after MIG and runtime overhead is slightly smaller than the nominal value (e.g., 9.5 GB for the 10 GB partition). Across both the A400 and the MIG configurations, the total peak GPU memory grows at a consistent rate of approximately 0.31 GB per file, comprising both the per-file merge-state growth (98–106 MB, shown in Table IV) and the decoded-data contribution of the queried columns. The 3.7 GB A400 reaches the cliff at file 12 (34.6 M groups), and the 10 GB MIG partition at file 30 (93.9 M groups). Extrapolating this rate, the 20 GB partition is projected to reach the cliff at approximately 62 files; it completes all 47 files without OOM, consistent with this projection. The 40 and 80 GB partitions likewise complete all 47 files, as the dataset is too small to trigger their projected cliffs. The larger GPU defers the cliff but does not eliminate it. S&R's partitioned merge bounds the working state to B_{merge}, preventing unbounded accumulation regardless of GPU capacity.

Generalization beyond Parquet. S&R's core mechanisms are not tied to the Parquet format. The two-budget abstraction ($B_{transfer}$ and B_{merge}), exact local reduction, and key-disjoint partitioned merge depend only on the decoded in-memory footprint and the merge-state growth of group-by aggregation. These quantities are independent of how data is encoded on storage. The single format-specific component is the Level 1 estimator (Section IV-D), which reads Parquet footer metadata to predict the decoded footprint without touching data pages. This strategy transfers directly to other self-describing columnar formats. Apache ORC, for example, records per-stripe and file-level column statistics and uncompressed sizes in its footer, enabling the same metadata-only estimation. For formats that lack rich footer statistics such as CSV or JSON, the footer read can be replaced by a lightweight sampling probe that estimates per-column decoded width from a small prefix, while the reduction engine and partitioned delivery remain unchanged. Adapting S&R to a new format therefore mainly requires replacing the metadata estimator and the sidecar reader adapter, while the budget abstraction, exact reduction logic, and partitioned delivery protocol remain unchanged.

Aggregation coverage and trade-offs. S&R's reduction benefit depends on the aggregate type. Distributive aggregates (SUM, COUNT, MIN, MAX) and algebraic aggregates (AVG, decomposed into SUM and COUNT) are exactly mergeable. Each object collapses into one partial state per group, so storage-side reduction shrinks both the transfer unit and the merge state. Holistic aggregates such as MEDIAN and PERCENTILE cannot be decomposed into independently mergeable partial states, so the sidecar cannot reduce row count for them. For these functions S&R still provides a benefit through partitioned delivery. Key-disjoint bucketing bounds the per-bucket scope so the GPU finalizes one bucket at a time, but each bucket must carry all values for its keys rather than a compact partial aggregate. The trade-off is thus explicit. Distributive and algebraic queries gain both transfer-volume and merge-state reduction, whereas holistic queries gain only merge-state bounding at unchanged transfer volume. Our current merge-state bound H (Eq. 4) and evaluation target distributive and algebraic functions. Extending the bound to holistic aggregates, where per-bucket state scales with value count rather than group count, is left to future work.

Conservative cardinality estimation. The worst-case assumption $n_{\text{groups}} = n_{\text{rows}}$ in the merge-state bound (Eq. 4) leads to significant overestimation for low-cardinality queries (Section V-I). This causes the planner to activate S&R unnecessarily at scales where the direct path would succeed, adding sidecar overhead without a completion benefit. A tighter bound could be obtained by maintaining lightweight cardinality sketches (e.g., HyperLogLog) in the Parquet footer metadata or as sidecar-maintained auxiliary statistics, reducing unnecessary S&R activations for low-cardinality workloads. The worst-case bound is conservative by design, ensuring the planner never under-provisions, and the sketch-based estimate would refine accuracy without changing this guarantee.

Runtime-completion tradeoff. S&R is not designed to accelerate queries that already complete on the direct path. When the direct path succeeds, S&R adds storage-side reduction overhead that increases runtime. The value of S&R lies in extending the completion envelope: workloads that fail with OOM on the direct path can complete through S&R at the cost of additional latency. This is analogous to how spill-to-host trades performance for capacity, but S&R operates proactively at the storage layer rather than reactively at the GPU.

VIII. Conclusion

GPU-based object analytics faces a sharp memory cliff: workloads that succeed at one scale fail abruptly at a modestly larger one. We presented Scout-and-Reduce (S&R), a storage-side framework that closes this gap by introducing two budgets, B_{transfer} and B_{merge}, that jointly govern GPU admission safety, and by reshaping storage-side output to respect them. A footer-metadata planner selects the execution path without reading data pages, and partitioned merge with key-disjoint bucketing bounds intermediate state for high-cardinality group-by. Experiments on the NYC Yellow Taxi and OTTO recommendation-event datasets show that S&R sustains completion at all tested scales where the direct path and filter offloading fail due to OOM, while keeping peak GPU memory well below client-side reduction and imposing minimal overhead when the planner determines the direct path is safe.

Acknowledgments

This work was supported by SK hynix. This work was also partially supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (RS-2024-00453929 and RS-2025-00564249).

REFERENCES

[1] A. Mozzillo, L. Zecchini, L. Gagliardelli, A. Aslam, S. Bergamaschi, and G. Simonini, "Evaluation of dataframe libraries for data preparation on a single machine," in *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*, pp. 337–349, 2025.

[2] Y. Li, B. Ding, Z. Wei, L. M. Maas, M. Al-Ghosien, S. Blanas, N. Bruno, C. Curino, M. Interlandi, C. Peeper, K. Rajan, S. Chaudhuri, and J. Gehrke, "Scaling gpu-accelerated databases beyond gpu memory size," *Proc. VLDB Endow.*, vol. 18, p. 4518–4531, July 2025.

[3] J. Glaser, F. Aramburú, W. Malpica, B. Hernández, M. B. Baker, and R. Aramburú, "Scaling sql to the supercomputer for interactive analysis of simulation data," in *Driving Scientific and Engineering Discoveries Through the Integration of Experiment, Big Data, and Modeling and Simulation*, vol. 1512 of *Communications in Computer and Information Science*, pp. 327–339, Springer, 2022.

[4] F. Aramburú, W. Malpica, K. Abrougui, A. Aramoon, R. Auccapuclla, C. Brisson, M. Brobbel, C. Farrell, P. Garigipati, J. Hoozemans, S. Kamburugamuve, A. Nair, A. Ocsa, R. Quesada López, D. Sihag, A. Uyar, D. Vats, M. Wendt, J. M. Patel, and R. Aramburú, "Theseus: A distributed and scalable gpu-accelerated query processing platform optimized for efficient data movement," *arXiv preprint arXiv:2508.05029*, 2025.

[5] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes, "A deep dive into common open formats for analytical dbmss," *Proc. VLDB Endow.*, vol. 16, p. 3044–3056, July 2023.

[6] Y. Yuan, A. Iyer, L. Ma, and N. Talati, "Vortex: Overcoming memory capacity limitations in GPU-accelerated large-scale data analytics," *Proceedings of the VLDB Endowment*, vol. 18, no. 4, pp. 1250–1263, 2025.

[7] B. W. Yogatama, W. Gong, and X. Yu, "Orchestrating data placement and query execution in heterogeneous cpu-gpu dbms," *Proc. VLDB Endow.*, vol. 15, p. 2491–2503, July 2022.

[8] J. Choi, H. Y. Yeom, and Y. Kim, "Implementing cuda unified memory in the pytorch framework," in *Proceedings of the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pp. 20–25, 2021.

[9] Y. Sun, A. R. Shovon, T. Gilray, S. Kumar, and K. Micinski, "Optimizing datalog for the gpu," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 762–776, 2025.

[10] W. Shen, Y. Chen, R. Chen, and H. Chen, "Towards fully-fledged gpu multitasking via proactive memory scheduling," *arXiv preprint arXiv:2512.24637*, 2026.

[11] J. Ravi, T. Nguyen, H. Zhou, and M. Becchi, "Pilot: a runtime system to manage multi-tenant gpu unified memory footprint," in *Proceedings of the 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 442–447, 2021.

[12] Amazon Web Services, "Querying data in place with Amazon S3 Select." Amazon S3 User Guide, 2018. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html.

[13] J. Chakraborty, I. Jimenez, S. A. Rodriguez, A. Uta, J. LeFevre, and C. Maltzahn, "Skyhook: Towards an arrow-native storage system," in *Proceedings of the 2022 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 81–88, 2022.

[14] RAPIDS Development Team, "Welcome to the cuDF documentation!." https://docs.rapids.ai/api/cudf/stable/, 2026. Accessed: 2026.

[15] RAPIDS Development Team, "Library design — cudf documentation." https://docs.rapids.ai/api/cudf/stable/developer_guide/library_design/, 2026. Accessed: 2026-04-02.

[16] W. Lu, K. He, X. Qin, C. Li, Z. Wang, T. Yuan, X. Liao, F. Zhang, Y. Chen, and X. Du, "Xorbits: Automating operator tiling for distributed data science," in *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 5211–5223, 2024.

[17] B. Wu, D. Koutsoukos, and G. Alonso, "Efficiently processing joins and grouped aggregations on gpus," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–27, 2025.

[18] D. G. Tomé, T. Gubner, M. Raasveldt, E. Rozenberg, and P. A. Boncz, "Optimizing group-by and aggregation using gpu-cpu co-processing.," in *ADMS@ VLDB*, pp. 1–10, 2018.

[19] New York City Taxi and Limousine Commission, "NYC taxi and limousine commission trip record data," 2026.

[20] P. Normann, S. Baumeister, and T. Wilm, "OTTO recommender systems dataset: A real-world e-commerce dataset for session-based recommender systems research," 2023.

[21] Apache Parquet, "Encodings." https://parquet.apache.org/docs/file-format/data-pages/encodings/, 2026. Accessed: 2026-04-02.

[22] Apache Parquet, "Compression." https://parquet.apache.org/docs/file-format/data-pages/compression/, 2026. Accessed: 2026-04-02.

[23] Apache Arrow, "Arrow columnar format." https://arrow.apache.org/docs/format/Columnar.html, 2026. Accessed: 2026-04-02.

[24] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 international conference on management of data*, pp. 1981–1984, 2019.

[25] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pp. 307–320, 2006.

[26] Apache Arrow Contributors, "[C++] remove Skyhook," 2025. GitHub Issue #47225.

[27] T. Karnagel, R. Müller, and G. M. Lohman, "Optimizing gpu-accelerated group-by and aggregation," in *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pp. 13–24, 2015.

[28] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[29] MinIO, Inc., "MinIO: High performance object storage," 2026.

[30] Polars Contributors, "Polars: Dataframes for the new era," 2026.

[31] NVIDIA Corporation, "Dask cudf: Gpu-accelerated backend for dask dataframe." https://docs.rapids.ai/api/dask-cudf/stable/, 2024. Part of RAPIDS, version 26.04.

[32] NVIDIA Corporation, "RAPIDS accelerator for Apache Spark." https://docs.nvidia.com/spark-rapids/, 2024. Accessed: 2026.

[33] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[34] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," *Proc. VLDB Endow.*, vol. 9, p. 924–935, Aug. 2016.

[35] W. U. Zaman, C. S. Mishra, S. AlSaleh, A. Aghayev, and M. T. Kandemir, "Cord: Parallelizing query processing across multiple computational storage devices," in *Proceedings of the 2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1141–1153, 2025.

[36] Elice, Inc., "Elice: AI education and cloud platform," 2026.