Rubberband: Memory-Elastic, Skew-Tolerant GPU Hash Joins

Hangdong Zhao
Gray Systems Lab, Microsoft
USA
hangdongzhao@microsoft.com

Rathijit Sen
Gray Systems Lab, Microsoft
USA
rathijit.sen@microsoft.com

Craig Peeper
Microsoft Azure
USA
craigp@microsoft.com

Matteo Interlandi
Gray Systems Lab, Microsoft
USA
matteo.interlandi@microsoft.com

ABSTRACT

GPU-accelerated data analytics is maturing from research into production, as witnessed by commercial database vendors offloading complex SQL fragments onto GPUs [25, 26]. This makes it timely to revisit the hash join—an analytical workhorse—under two compounding realities of GPUs: on-device high-bandwidth memory (HBM) is scarce, and its memory hierarchy sharply penalizes the atomic contention and scattered random accesses that conventional hash joins incur.

Rubberband addresses both via two novel, co-designed contributions. The first is a logical rewrite—equality hoisting—that lifts the equality test out of the probe loop and applies it as a post-join filter. The hash table is then free to return a superset of matches, turning its false-positive rate—which a conventional design must pin near zero—into a tunable knob. The result is **memory elasticity** as HBM pressure tightens. The second is a new hash table layout—a compact CSR-style bucket directory of row indices, built by additive atomics only (no compare-and-swap, retries) and probed by a single contiguous bucket scan (no chained random walks). Under data skew, a hot key becomes a longer contiguous scan rather than a pile-up of CAS retries on scattered chain walks, which gives Rubberband **skew tolerance** without a separate skew-handling path.

Rubberband is a new hash join in CoddSpeed [25], Microsoft Fabric's production GPU-accelerated analytical engine. On an NVIDIA H100 NVL, against CoddSpeed's already optimized open-addressing baseline, Rubberband is up to 3.3× faster on TPC-H SF100 PK–FK joins, ~722× on an extreme-skew production case, and 1.8× on average across 11 production-customer datasets, while using 43–53% less hash-table memory in the default configuration and 71–76% less in a memory-constrained one.

KEYWORDS

GPU hash join, equality hoisting, memory-elastic hashing, skew tolerance, analytical query processing on GPUs

VLDB Workshop Reference Format:
Hangdong Zhao, Rathijit Sen, Craig Peeper, and Matteo Interlandi. Rubberband: Memory-Elastic, Skew-Tolerant GPU Hash Joins. VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'26).

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

As CPU performance saturates, while AI-motivated datacenter investment scales GPU compute and memory bandwidth far faster, commercial database vendors are moving intensive SQL fragments onto the accelerators already provisioned at scale [25]. A decade of GPU-accelerated database designs and engineering has produced increasingly capable on-GPU hash-join implementations [7, 9, 16, 27, 41]. Recent customer queries on Microsoft Fabric, however, expose two compounding realities that uniform benchmarks rarely surface: (1) on-device high-bandwidth memory (HBM) is scarce and contested—the build-side hash table competes for HBM with every other database component and routinely strains it; and (2) the GPU memory hierarchy sharply penalizes the atomic contention and scattered random accesses that conventional open-addressing hash joins—compare-and-swap inserts and chained slot walks—incur whenever key distributions skew. This paper rethinks the GPU hash joins under both.

On GPUs, hash join implementations have largely converged on open addressing: a flat array of $H \geq N$ slots, sized as a small constant multiple of the build cardinality N to keep collision chains short. Each distinct key claims its own slot; build-side rows are inserted via atomic compare-and-swap (CAS); probe-side rows walk the resulting chain until they find a match or an empty slot. Most existing GPU hash-join designs inline the full (key, payload) pair in each slot [6]; CoddSpeed—our production GPU analytical engine [25]—instead stores only a build-row index per slot and fetches the key and payload on demand. Even on this already-lean baseline, two shortcomings of open addressing remain in production.

Memory overhead. The $H \geq N$ constraint binds even with row-id-only slots: the probe must resolve each slot to a single, unambiguous build row, so distinct keys cannot share a slot. Using a companion H-entry step-counter array for early build/probe termination, the hash-join footprint can claim a non-trivial fraction of HBM at production cardinalities (Figure 3). Once the table outgrows HBM, the query *spills* to a CPU path that largely erases the GPU advantage.

Skew sensitivity. When many keys hash to the same slot, build threads pile onto the same atomic CAS while probe threads walk long chains across scattered HBM memory. In a captured production query (§6.4) where a handful of hot keys absorb most of the build,

Rubberband is ~722× faster than the open-addressing baseline on this join—exactly the pathology this work targets.

Rubberband addresses each by a novel ingredient. The first is a join rewrite—equality hoisting—that pulls the equality test into a post-join filter. The hash-table lookup may now return a *superset*: distinct keys can share a table slot (i.e. a *bucket*), and the join emits candidates that the filter validates. Conventional open-addressing tables already admit some false positives (two keys hashing to the same slot), but size $H \gg N$ to keep that rate near 0; Rubberband deliberately relaxes that constraint. The bucket count K therefore decouples from the build cardinality N and, under HBM pressure, can drop below N, trading a higher false-positive rate for a smaller directory. The table remains $O(N)$—an N-entry row-index array is intrinsic to indexing build rows—but its constant factor falls well under the baseline. This yields **memory elasticity**: the hash table shrinks as HBM pressure tightens, keeping queries on the GPU that would otherwise spill.

The second ingredient is a compact hash-table design: a Compressed Sparse Row (CSR)-style bucket directory [3] that departs from the open-addressing (OA) slot-array family. Rubberband keeps the baseline CoddSpeed's row-id-only storage, but lays it out as an N-entry array of build-row indices grouped by a K+1-entry offset array (that encodes K bucket boundaries). Build runs a three-phase count, prefix-sum, and scatter pipeline using only additive atomics and reusing one K+1 buffer across all phases; probe is a single offset load followed by a contiguous scan inside a bucket. Under heavy skew, a hot key becomes concurrent atomic-adds on one counter at build time and a long but contiguous bandwidth-bound scan at probe time—both absorbed gracefully by the GPU memory hierarchy. This makes the design **skew tolerant**. The same scaffolding generalizes to semi-, anti-, and outer joins as small variations on a single kernel (§5).

Contributions. Specifically, Rubberband contributes:

(1) **A memory-elastic CSR-style hash table for GPU hash joins (§4, §6.2).** Rubberband replaces the open-addressing convention by a CSR-style layout—to our knowledge a first for GPU join indexes. This cuts hash-table sizes by 43–53% at the production default K=1.05N and exposes a smooth space–time knob below it, governed by an elegant false-positive bound (§3.2).
(2) **Structural skew tolerance with minimal regression.** Rubberband is ~722× faster than our baseline on an extreme-skew production case (§6.4), rising smoothly with skew on a parametric Zipf sweep from 1.23× at s=0 to 1.80× at s=0.6 (Figure 4), and up to 3.3× on TPC-H SF100 [36]—no special-case handling or regression on uniform data.
(3) **Production deployment in a commercial GPU-accelerated SQL engine.** Rubberband embeds inside CoddSpeed [25], Microsoft Fabric's production GPU data analytical engine. On 11 datasets captured from production queries (§6.4), it is 1.8× faster on average than the OA baseline at the default K=1.05N, and 1.6× on average even when the directory is shrunk to K=0.15N (roughly half the K=1.05N hash-table footprint).

Organization. §2 isolates the structural constraint in the standard formulation. §3 develops equality hoisting—the rewrite—and derives its closed-form false-positive bound (§3.1–§3.2). §4 explains the CSR-style hash table: its additive-atomic build and contiguous-scan probe (§4.1–§4.3). §5 extends the same scaffolding to other join types. §6 reports memory elasticity, and experimental results on TPC-H and production workloads at both default and memory-constrained settings. §7–8 position related work and conclude.

2 BACKGROUND

Open-addressing GPU hash joins. Open addressing is the dominant on-GPU hash-join layout, shipped as the off-the-shelf primitive in NVIDIA's `cuCollections` [6] behind RAPIDS `cuDF` and engines such as BlazingSQL [42] and Sirius [31]; §7 discusses the broader design space. The build phase inserts via compare-and-swap (CAS) and retry-on-collision; the probe phase re-walks the resulting collision chain, checking each candidate's key for equality.

Production OA hash join baseline. The concrete OA baseline against which every speedup and memory delta throughout this paper is reported is Microsoft Fabric's CoddSpeed [25], the production descendant of the Tensor Query Processor [26], which targets the same two costs as Rubberband—memory footprint and skew—by two pragmatic optimizations on top of the standard OA template:

- **For memory: row-index-only slots.** While `cuCollections` inlines a full (key, payload) pair per slot, CoddSpeed stores only an integer build-row index and fetches the actual key and payload from memory on demand—the most HBM-frugal OA layout we know of in production. The extra indirection per probe can dent raw throughput, but leaves the baseline already memory-tuned.
- **For skew: $H \approx 2N$ slots + a per-slot step counter.** The slot array is sized at $\approx 2N$ (load factor 0.5) to keep the probe-sequence-length tail short, and a companion H-entry step-counter array records the longest probe length seen at each slot. Each insert bumps the counter at its hashed slot; subsequent inserts skip past the existing chain rather than CAS-walking it, and probes terminate at the recorded length instead of walking to the next empty slot. The counter accelerates both phases under skew over basic OA at the memory cost of one extra integer per slot. Collapsing it to a single global max-chain scalar slows both phases. Entries below are the resulting slowdowns using a global max only (N=2M, D=1M):

	Uniform	Zipf s=0.3	Zipf s=0.5	Zipf s=0.8
Build	3.0×	3.3×	3.6×	6.5×
Probe	1.1×	1.0×	1.2×	2.5×

Both optimizations are always on: Rubberband is benchmarked against this tuned production OA join throughout.

3 EQUALITY HOISTING

§3.1 states the rewrite identity and the two design knobs it exposes; §3.2 bounds its only new cost—false positives—in closed form. The memory–runtime envelope this opens up is characterized empirically in §6.2.

3.1 The Rewrite Identity

For any equi-join and any function ϕ satisfying $x{=}y \Rightarrow \phi(x){=}\phi(y)$,

$$A \bowtie_{x=y} B \quad \equiv \quad \sigma_{x=y}\big(A \bowtie_{\phi(x)=\phi(y)} B\big).$$

Table 1: Notations used throughout the paper, for a single equi-join $A \bowtie_{x=y} B$ with build side A and probe side B.

Parameter	Meaning
N	Build-side input row count
D	Distinct build-side keys; $D \le N$
H	Open-addressing slot count (baseline); $H \ge N$, conventionally $\sim 2N$
K	Rubberband bucket count; the memory knob (§6.2). Production default $K \approx 1.05\,N$; the rewrite admits any $K{>}0$, with the characterized envelope reaching $K \approx 0.05N$
ϕ	Many-to-one bucket map: $x{=}y \Rightarrow \phi(x){=}\phi(y)$
FP	Per-probe false-positive rate: $\max(0,\ 1 - K/D)$ (§3.2)

The equality test $x{=}y$ moves out of the hash-table probe and into a post-filter on the output. The probe now operates on ϕ-values, so the hash table only has to group rows that share a ϕ-value—it no longer needs a slot per distinct key. We call this move **equality hoisting**; the rest of Rubberband realizes it on the GPU.

What hoisting buys us. Keys that share a ϕ-value form a *bucket*—the group the table stores in place of a single-key slot. Two parameters are now tunable design choices under the rewrite:

- **The choice of ϕ.** Any ϕ obeying the equivalence is admissible. Setting ϕ to identity recovers the inner join; $\phi(v) = \texttt{hash}(v) \bmod K$ yields the many-to-one hash mapping Rubberband uses; range bucketization can be an effective optimization for band joins (§8). The right ϕ is a query-optimization decision.
- **The bucket count K.** Rows ϕ-mapping to the same bucket are grouped together; the post-join filter $\sigma_{x=y}$ validates which actually match. The index now needs only K entries instead of N; K becomes free to decouple from N (analyzed in §3.2). This is Rubberband's memory elasticity: shrinking K trims the only term that depends on it, leaving the $\Theta(N)$ row-id payload—the floor for any structure returning matching rows—untouched.

3.2 False Positives and Bucket-factor Sweep

Shrinking K incurs a probe overhead: when distinct keys hash to the same bucket, the probe scans all of them and the equality post-filter discards the non-matches. Each discard is a false positive—the only cost the rewrite can amplify. Under bag semantics, a bucket holds N/K rows on average; under uniform multiplicity, a probe whose key exists in the build finds N/D true matches among those N/K candidates, and the remaining $N/K - N/D$ are false positives. Dividing gives the false-positive rate FP and the work amplification—the number of candidates a probe scans per true match:

$$FP = \frac{N/K - N/D}{N/K} = 1 - \frac{K}{D}, \qquad \text{work amp.} = \frac{N/K}{N/D} = \frac{D}{K},$$

both valid for $K \le D$; for $K > D$, $FP{=}0$ and work amp. $= 1$. The per-key multiplicity N/D cancels in both expressions: cost is governed by K/D, not K/N. The implication is surprising: a heavily duplicated build ($D \ll N$) can shrink its directory all the way to $K \approx D$ at **zero** false-positive cost—any $K \ge D$ suffices—and pushing K lower raises work only linearly, as $O(D/K)$. Open addressing, however, has no such smooth trade-off: it falls off a collision cliff as its slot over-provisioning H/N approaches 1.

4 CSR BUCKET DIRECTORY

Rubberband realizes its hash table as a compact CSR-style bucket directory: an N-entry array of build-row indices grouped by a $K{+}1$-entry offset array. The next three subsections outline this layout (§4.1), the additive-atomic build that retires CAS contention (§4.2), and the contiguous-scan probe that retires scattered chain walks (§4.3); the resulting per-phase costs and structural skew tolerance are characterized empirically across the Zipf sweep of Figure 4 and the production captures of §6.4.

4.1 Directory Layout

Equality hoisting (§3.1) instantiates ϕ as

$$\phi(v) \;=\; \texttt{hash}(v) \bmod K,$$

and a natural data structure is a contiguous bucket grouping (Figure 2b). Rubberband uses two arrays: ❶ an N-entry `rowIds` array of build-side row indices, and ❷ a $K{+}1$-entry `offsets` array holding an exclusive prefix sum of bucket sizes.

Consecutive offsets frame each bucket, so all rows mapping to a given bucket sit in one contiguous slice of `rowIds`—no per-slot key, payload, and no over-provisioning. Every entry in `rowIds` is a build-side row-id, never an empty slot, and the total footprint is $K{+}1{+}N$ integers: the N-entry payload is the information-theoretic floor for any structure that returns matching rows, and the $K{+}1$-entry directory is the design's only knob. Structurally, the directory mirrors the CSR layout used widely for sparse matrices and adjacency lists on GPUs [3]: §7 discusses this resemblance.

Since K is elastic, the same layout spans both regimes: at $K \approx N$ (our default) most buckets hold 0–1 rows and it acts as an open-addressing hash table, while under memory pressure (as K shrinks) buckets coarsen into radix-style partitions the probe scans in one single pass (§7). Under data skew, hot keys simply fatten one bucket, and the probe stays bandwidth-bound on contiguous memory—the structural source of Rubberband's skew tolerance. §6.4 stress-tests this empirically.

4.2 Building the Directory

The directory is built in three phases that all read and write the same $K{+}1$-entry buffer—no separate count array or write cursors. The trick is a simple one-slot shift: every atomic targets `offsets[b+1]` (never `offsets[b]`), so `offsets[0]` stays at its zero-initialized value throughout. This address policy lets each slot `offsets[b+1]` carry three successive meanings, in turn:

(1) **Tally.** Each thread hashes its key to a bucket b and increments via `atomicAdd(&offsets[b+1], 1)`. Unlike CAS, the increment always succeeds on the first try—skew just becomes a bigger number in one counter. After this pass, `offsets[b+1]` holds the count of bucket b.

(2) **In-place exclusive scan.** A single CUB `ExclusiveSum` over the same $K{+}1$ slots rewrites every entry from count to running start. `offsets[0]` stays at 0, and `offsets[b+1]` now holds the start of bucket b—precisely the write cursor that scatter will advance through next.

(3) **Scatter.** Each thread issues the same atomic-add as in the tally pass—only the meaning of the address has changed. The atomic returns the next free slot inside bucket b and bumps the cursor

for the next writer. When the last writer finishes, `offsets[b+1]` has advanced exactly count[b] times and now equals the start of bucket b+1.

The same buffer thus reads as counts, then offsets, then write cursors in turn. After phase 3 (scatter), bucket b's slice in `rowIds` runs from `offsets[b]` to `offsets[b+1]`: its right edge is finalized by bucket b's own scatter, and its left edge by bucket $b-1$'s scatter—except for b=0, whose left edge is simply the zero-init in `offsets[0]` that no atomic ever touches. Figure 1 traces the buffer through all three phases for the running example of Figure 2.

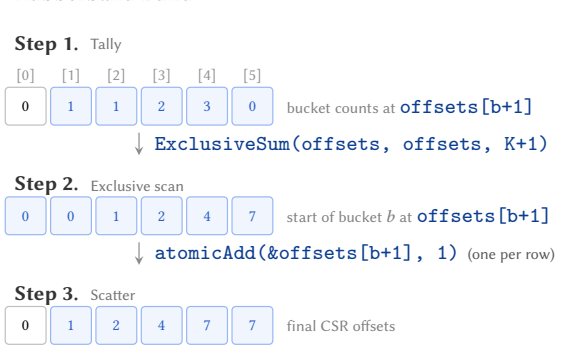

Figure 1: One K+1 buffer carries three meanings in turn (N=7, K=5; bucket sizes $(1, 1, 2, 3, 0)$, matching Fig. 2b). `offsets[0]` is never touched—its zero-init is already the start of bucket 0.

Build cost. ❶ The build runs in two passes over the N input keys—tally and scatter—costing $2N$ atomic-adds, plus a single $O(K)$ prefix sum on the K+1 buffer. That is one extra input pass compared to the OA baseline, traded for retry-free atomics: every `atomicAdd` commits on the first try, so build time scales by N alone and is independent of the key distribution, where the baseline's CAS retries spiral under skew. ❷ Memory is the K+1+N integers of the directory—if set to a default $K = 1.05N$, about half of the $2H \approx 4N$ integers the OA baseline carries across its $H \approx 2N$ slot array and companion step-counter (§2).

4.3 Probing the Directory

A probe key v maps to bucket $b = \texttt{hash}(v) \bmod K$. The probe loads the two adjacent offsets `offsets[b]` and `offsets[b+1]`, which delimit a contiguous slice of `rowIds`, scans that slice, and applies the hoisted equality post-filter $x = y$ to each candidate (Figure 2b). Each probe touches just two memory regions—one offset pair and one contiguous slice—both sequential and cache-friendly.

Rubberband picks one of two scan strategies based on whether the build is unique on the join key (informed by the database schema and the optimizer):

- **Unique-build fast path (PK–FK case).** One thread per probe walks the bucket and stops on the first equality hit; uniqueness guarantees there is no second match, so the probe collapses to a short contiguous read.
- **Cooperative scan (otherwise).** Lacking uniqueness guarantee, every probe must scan its entire bucket, and a fat bucket would stall a single-threaded walker. Rubberband therefore splits each probe across $\max(1, \lfloor B/64 \rfloor)$ cooperating threads, where B is the build's recorded maximum bucket size; each thread's portion stays near 64 consecutive rows even on the fattest bucket.

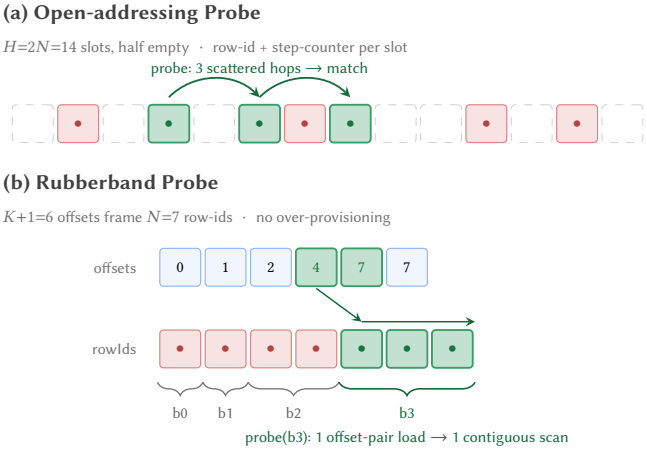

Figure 2: Probing N=7 build rows over K=5 buckets of sizes $(1, 1, 2, 3, 0)$. Cells containing dots store row-ids; green cells are what one probe touches. (a) OA scatters rows across H=2N=14 slots (half empty), so the probe chases a 3-hop chain. (b) Rubberband packs rows into K contiguous slices; offsets $(4, 7)$ frame bucket b_3, read as one contiguous scan.

Both code paths scan the directory contiguously and apply the same equality post-filter, so a hot key becomes a longer bandwidth-bound scan rather than a chain walk. Both mitigate the slowdown as K shrinks and buckets fatten: the early break has more rows to skip past, and the cooperative split has more probes for parallelism. Coupled by the additive-atomic build of §4.2, this makes Rubberband skew-tolerant; §6.3 and §6.4 present the resulting per-phase costs.

5 SEMI-, ANTI-, AND OUTER JOINS

Rubberband extends to semi-, anti-, and outer joins, all implemented in CoddSpeed [25]. We support semi- and anti-joins by instantiating Rubberband for the mark join [24]—a physical primitive that writes a Boolean "matched?" bit per row instead of emitting (probe, build) pairs, exposed in CoddSpeed as the `isIn` kernel. Only the per-thread action at the end of the bucket scan differs; build, directory, and equality post-filter are unchanged. Semi- and anti-joins are then one `isIn` call plus a mask filter (invert for anti). Outer joins reuse the inner-join probe and add a small tensor-level post-pass that recovers the preserved side's unmatched rows. No new GPU kernels are introduced.

6 EVALUATION

We validate Rubberband's two contributions in concert—memory elasticity from equality hoisting (§6.2) and structural skew tolerance from the CSR-style directory—on the same production code path, across three workloads of increasing realism:

- a controlled sweep of K against skew (§6.2);
- the TPC-H SF100 PK–FK joins [36] (§6.3); and
- 11 datasets captured from production customer queries (§6.4).

Two questions organize the reading of the figures that follow: ❶ does shrinking the directory buy back HBM along a graceful curve rather than a cliff, and ❷ does the distribution-oblivious Rubberband implementation absorb the skew that stalls the baseline? We run each end-to-end benchmark at two operating points—the default K=1.05N and a memory-constrained K=0.15N at roughly half that footprint—the latter reaching into the K<D regime where the false-positive bound of §3.2 turns active. The pair brackets Rubberband's full space–time envelope.

6.1 Setup

All experiments run on an NVIDIA H100 NVL (94 GB HBM3) with CUDA 12.8; we report 5-run medians after one warmup. The baseline is CoddSpeed's in-engine OA hash join (§2); Rubberband is built into the same engine and shares CoddSpeed's hash function, allocator, layout, and output format, so each delta isolates hash join algorithm from everything else.

Unless noted, Rubberband runs at the default K=1.05N, the factor that keeps the FP bound of §3.2 near zero for every tested key distribution. The 5% headroom is a production conservative: Rubberband sets K from N alone, so we err on the side of over-allocating HBM rather than risk regressing where D approaches N. The elastic regime K<N is characterized in §6.2. Auto-picking K from the optimizer, histograms on the join keys, or runtime memory pressure is left to future work.

6.2 Memory-performance tradeoff across K

The hash table dominates the join's peak HBM footprint: it stays resident throughout the probe pipeline—and across any downstream pipelines that pin it for further probes. We first sweep K alone to isolate the memory savings (Figure 3), then sweep K against skew to map the runtime envelope (Figure 4).

Hash-table memory across K. Figure 3 sweeps K/N through [0.05, 1.05] across three TPC-H SF 100 builds that recur across the query set: `customer` (15 M PK), `orders` (150 M PK), and a duplicate-heavy build on `lineitem.L_ORDERKEY` (600 M rows, 150 M distinct), reporting Rubberband's directory as a percentage of the baseline.

At the default K=1.05N, all three builds cut hash-table memory by 43–53% versus the baseline. This is the regime that where $D \le N \le K$ makes the FP bound of §3.2 to near zero. As K shrinks, the savings grow—memory falls monotonically and then flattens around K=0.05N, bottoming out at the N-entry row-id array that every build must keep.

Runtime across K and skew. The equality hoisting rewrite admits any K>0, so we sweep well below the default. Figure 4 reports an inner join (N=2 M, probe=2N, D=N/2, both sides sampled from Zipf(s)) over $K \in$ [0.05, 1.05] N at varying s values for skew. Two observations stand out under skew.

- **Higher skew, larger speedup over OA.** The right-half plateau rises from 1.23× at uniform to 1.80× at s=0.6: the baseline's CAS retries and probe chain-walks lengthen while Rubberband's CSR-style build and contiguous-scan probe stay distribution-oblivious.
- **High skew, small K hurts less.** All curves flatten at $K \approx D$, where $FP = \max(0,\ 1 - K/D)$ (§3.2) drops to zero. For K<D the regression of Rubberband stems from larger FP, but skew mitigates the slowdown: at K=0.05N, uniform drops to 0.41×, s=0.4 to 0.60× (against the fixed OA baseline), while s=0.6 stays above parity throughout the sweep at 1.64–1.80×.

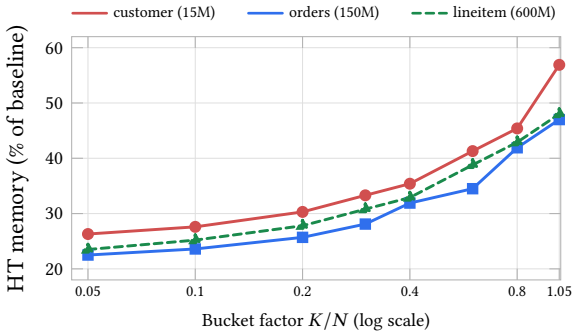

Figure 3: Hash-table memory as a percentage of the OA baseline, swept over $K/N \in$ [0.05, 1.05] on 3 TPC-H SF100 builds: customer (15 M PK), orders (150 M PK), and a duplicate-heavy self-join on `lineitem.L_ORDERKEY` (600 M rows, 150 M distinct, as in Q21). Lower is better; runtime companion: Figure 4.

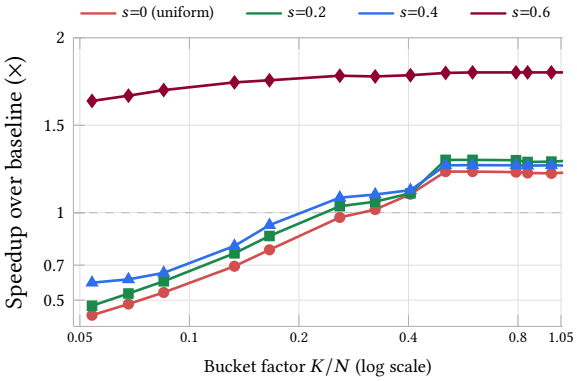

Figure 4: Speedup over baseline on an inner join (N=2 M, probe=2N, D=N/2, build and probe sampled from the same Zipf(s)), swept over $K/N \in$ [0.05, 1.05] at 4 Zipf levels: s=0 (uniform), s=0.2, 0.4, 0.6 (heavy-skew stress). Higher is better; dashed line marks parity. Memory companion: Figure 3.

The contrast to the OA baseline is structural. When an OA table outgrows HBM, the query falls back to CPU—a sharp performance decline (often 5–10× slower). Rubberband's K knob replaces that cliff with a continuous curve: any HBM headroom maps to a corresponding K, the join can stay on GPU, and even in the uniform worst case it degrades gracefully—while under high skew it stays at or above parity throughout.

6.3 TPC-H SF100

PK–FK joins on TPC-H SF100. We benchmark Rubberband on TPC-H SF100 PK–FK joins (Figure 5): (1) the open-addressing baseline once, and Rubberband—(2) at the default K=1.05N (Figure 5a) and (3) at the memory-constrained K=0.15N (Figure 5b). Figure 5b

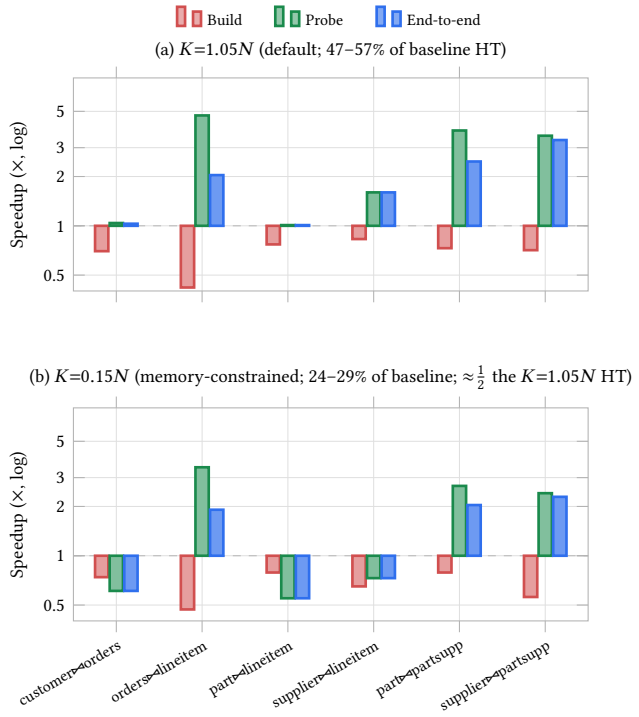

Figure 5: TPC-H SF100 PK–FK joins: build, probe, and end-to-end speedup over the OA baseline. Panel (a) is the default K=1.05N (47–57% of baseline HT memory; 43–53% saved); panel (b) is the memory-constrained K=0.15N (24–29% of baseline; 71–76% saved—roughly half the (a) footprint). Log y-axis; dashed line marks parity. Each bar is the median of 5 warm runs.

enters the $K<D$ regime where FP (§3.2) turns non-zero on PK builds, trading a probe slowdown for further memory savings.

- K=1.05N **(default).** Rubberband's count-then-scatter doubles the build scan, so build runs modestly slower—the OA baseline is already near-optimal on unique PK keys, where each insert is almost always a single CAS—no chain walk. Probe inverts the picture: a single offset read plus a contiguous in-bucket scan exploits cache locality, reaching up to 4.72× on the FK-duplicated joins. Because the smaller PK sits on the build side, probe dominates total time, so these wins carry end-to-end—up to 3.34× on the duplicated joins, and parity on the rest.
- K=0.15N **(memory-constrained).** Compacting the hash table leaves build largely unchanged but pushes every PK build into $K<D$, where the false-positive bound bites (e.g. $FP\approx0.85$): each probe scans many more candidates before the equality post-filter discards them, so probe—and end-to-end—slow. That is the price of the extra memory being saved.

Full-query audit. Beyond the standalone PK–FK tests, we run all 22 TPC-H SF100 queries end-to-end at the default K=1.05N. Figure 6 reports, per query, peak intermediate GPU memory saved (top) and end-to-end runtime change (bottom). Join-heavy queries

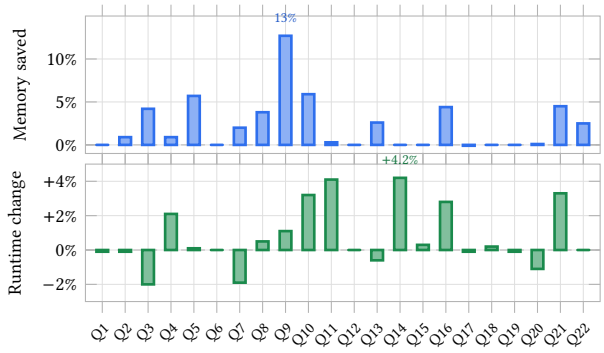

Figure 6: Per-query peak intermediate GPU memory saved (top) and end-to-end runtime change (bottom) under Rubberband vs. baseline, on all TPC-H SF100 queries; K=1.05N. Higher is better in both panels.

save memory—up to **13**% on Q9, a 6-table join—while the scan-, filter-, and aggregate-bound queries sit at parity.

Runtime changes span [−2.0%, +4.2%] at mean +0.7%. Wins concentrate on join-heavy queries (e.g. Q14 +4.2%, Q11 +4.1%, Q21 +3.3%, Q10 +3.2%); and the 4 measurable regressions (Q3 −2.0%, Q7 −1.9%, Q20 −1.1%, Q13 −0.6%) all trace back to joins where Figure 5a already shows a sub-baseline build performance (0.42–0.83×). Crucially, the engine aggressively pushes selective predicates (e.g. bitmap filters [32] created from the build side) down the join probe side, so at SF100 the hash join (e.g. the join probe time) is a fraction of total query time and the structural wins of Figures 5 and 7 compress to single-digit-percent end-to-end deltas here.

6.4 Production Datasets and Queries

The 12 production captures (#0–#11) are anonymized snapshots of real customer joins, deliberately collected from traces flagged as slow under the open-addressing baseline. Each is run at the default K=1.05N and the memory-constrained K=0.15N. They fall into two regimes: #0–#2 pair a mostly-distinct build against a large probe, while #3–#10 feed heavily duplicated builds holding only a few hundred distinct keys. The most adversarial capture (#11)—a single key taking ~55% of the build—is reported separately in Figure 7.

- K=1.05N **(default).** Rubberband outperforms the OA baseline on every capture (geomean **1.84**× end-to-end), and the dominant source is build-side key skew. On the 8 duplicate-build captures #3–#10—where 353 K rows hash to only 15–300 distinct keys—the baseline piles thousands of CAS retries onto each hot slot while Rubberband's additive count step touches every row once, so even the build side runs up to an order of magnitude faster on datasets #4–#7. The mostly-distinct cases #0–#2 hold near parity on build, and the probe wins on top lift every dataset above parity end-to-end under Rubberband.
- K=0.15N **(memory-constrained).** The FP formula of §3.2 predicts that exactly these build-skewed captures should tolerate a smaller K: with $D \le 300$ on #3–#10, even the shrink to K=0.15N leaves $K \gg D$, so FP stays small and probe walks the same buckets it did at K=1.05N. Indeed, across #3–#11 the bars are

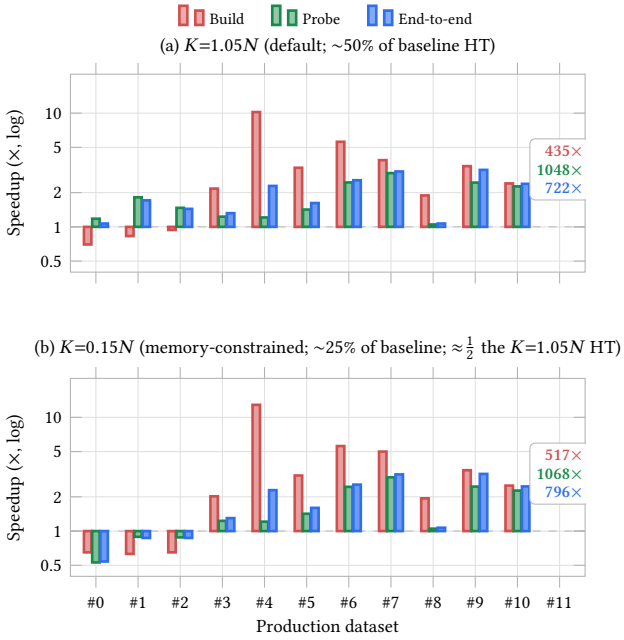

Figure 7: 12 production datasets: build, probe, and end-to-end speedup over the OA baseline. Panel (a) uses K=1.05N (~50% of baseline); panel (b) uses K=0.15N (~25% of baseline—~ 50% the (a) footprint). Log y-axis. Extreme-skew #11 values listed in-place (no bars, color matches the legend).

essentially indistinguishable from panel (a). Only #0–#2 push $D > K$, and slow down end-to-end. The OA baseline uses ~4× more hash-table memory at this memory-constrained regime.

Extreme-skew case #11. Dataset #11 is the most adversarial production join we have collected: a single key absorbs ~55% of the 12.5 M-row build column. The baseline degenerates as millions of build threads contend a few hot slots and probes then re-walk the resulting long chain. Rubberband maps both phases into one additive atomic per row (§4.2) and a contiguous, albeit long, parallel bucket scan (recall cooperative scan at §4.3): 435×/1048× build/probe at K=1.05N (end-to-end ~722×), and essentially identical at K=0.15N (517×/1068×, end-to-end ~796×) since the small number of distinct keys keeps $K \gg D$ (see the FP analysis at §3.2).

7 RELATED WORK

Hash joins on GPUs. On-GPU hash joins fall into two families. Open addressing has been the dominant single-GPU layout since the non-partitioned hash join of Kaldewey et al. [9], and is today the off-the-shelf default in NVIDIA's `cuCollections` [6]—the primitive behind `cuDF` and the engines built on it (BlazingSQL [42], Sirius [31], and NVIDIA's recent GQE [40]): it keeps a flat slot array sized as $H = \alpha N$ (over-provisioning $\alpha \geqslant 1$) and resolves collisions by walking the slot array—fast on uniform data, fragile under skew [15, 22]. Radix-partitioned joins [7, 19, 20], in contrast, first hash *both* inputs into $K \ll N$ buckets—count, prefix-sum, scatter, often multi-pass—and then join the co-located partition pairs in fast on-chip memory. They improve locality, but at the cost of writing both inputs out to partition buffers prior to any matching. Rubberband can be seen as a single structure that interpolates between the two families. Its directory mirrors CSR [3], the standard GPU sparse-matrix format, and is built by the same count, prefix-sum, scatter pass as a radix join [1, 10, 23], but the bucket count K is tunable rather than fixed. At $K \approx N$ almost every bucket holds a single row, so a probe is a near-direct lookup as in open addressing; as K shrinks—e.g. under memory pressure—buckets coarsen into short partitions the probe scans, mimicking radix partitioning. Unlike a radix join, though, Rubberband partitions only the build side and keeps that directory as its index, so the probe stays single-pass and **pipeline-friendly**, i.e. no probe-side materialization. To our knowledge Rubberband is the first deployed GPU hash join that is both memory-elastic and structurally skew-tolerant.

Approximate join indices. Lossless indices—open addressing, chained tables [17], the Concise Hash Table [2], and perfect hashing [5, 18]—largely pin every distinct key to its own resolvable address. Approximate indices—most commonly, Bloom filters [4]—are far smaller but answer set membership, so in joins they sit as a *probe-side* pre-filter in front of a full-sized lossless build-side index. Rubberband fills the missing mirroring cell: an approximate *build-side* index that returns rows. To our knowledge this "approximate but returns rows" point is otherwise empty in the join literature.

Hash join under data skew. Prior GPU work targets skew along three lines: sync-free OA builds [21], dual-path designs that route hot keys to a dedicated kernel [22], and radix-partitioned joins [19, 20]. All rely on either a skew-detection pre-pass or a second skew-aware code path. Rubberband instead absorbs hot keys structurally—additive atomics for build, contiguous scans for probe—on a single code path having no skew detection. As a complementary payload-side trick, GFTR's payload co-permutation [20] fits naturally into Rubberband's row-ids-in-contiguous-slice output: pre-permuting payload columns to follow each bucket's row-id ordering turns the downstream payload fetch into a sequential stream (rather than a gather), compounding the spatial locality for the bucket scans.

Spilling hash joins. A deep CPU lineage [33–35] and GPU analogues that stage join states over fast interconnects [8, 11, 12] take over once memory pressure exceeds Rubberband's elasticity range. Rubberband is a cushion that yields smooth performance degradation until spilling: the K knob (§6.2) exposes a space–time curve rather than a cliff, so the engine can shed memory in fine-grained increments and only spill at the edge of the HBM budget.

Multi-device scaling. Multi-GPU and distributed designs [13, 14, 39] are orthogonal to the per-device structure Rubberband sits. Rubberband slots most cleanly into partition-based multi-GPU engines that hash-shuffle keys into per-device partitions—such as Wu et al. [37] and PystachIO [38]: Rubberband drops in as each device's hash-join kernel and every shard inherits its robust properties.

8 CONCLUSION

Rubberband brings both memory elasticity and skew tolerance to on-GPU hash joins via two ideas: equality hoisting is a flexible join rewrite and a CSR-style bucket layout is used as a memory-elastic hash table. Rubberband integrates into Microsoft's CoddSpeed [25], covering inner, semi, anti, and outer joins.

Generalization to theta joins. The two ingredients suggest an extension for theta joins: under memory pressure, a join's index can over-approximate, provided correctness is restored by a cheap post-filter. Range and band joins [28–30] are the near-term case—bucketization gives the same coarse-key-plus-post-filter shape as equality hoisting, only now ϕ is set-valued: a point maps to a singleton bin id and a range to a list of consecutive bins (i.e. bucketized). Rubberband's additive-atomic build and cooperative bucket scan already handle the resulting data skews, e.g. caused by long-interval replication—and other theta predicates (similarity, spatial, etc.) fit once a bucketization scheme is properly supplied.

Robust on-GPU query execution. Two desirable properties recur for GPU-resident SQL execution: memory elasticity and skew tolerance. HBM is often a hard ceiling, and production data are rarely uniform—and both apply beyond joins. Aggregation, group-by, sort, distinct, and window functions all hash, partition, or materialize state and share the same fragility under tight memory and skewed inputs. Rubberband addresses these properties for hash join; extending such robustness across the rest of the GPU operator set is left to future work.

ACKNOWLEDGMENTS

We used AI-based language tools to assist with drafting and editing this paper. All technical content, experiments, and claims are the authors' own.

REFERENCES

[1] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. ICDE*, pages 362–373, 2013.

[2] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, 2014.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. SC '09*, pages 18:1–18:11, 2009.

[4] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes Cuckoo at high throughput. *Proc. VLDB Endow.*, 12(5):502–515, 2019.

[5] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[6] NVIDIA. cuCollections: a header-only library of GPU-accelerated, concurrent data structures. https://github.com/NVIDIA/cuCollections, accessed 2026.

[7] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *Proc. SIGMOD*, pages 511–524, 2008.

[8] M. Kabić, B. Wu, J. Dann, and G. Alonso. Powerful GPUs or fast interconnects: Analyzing relational workloads on modern GPUs. *Proc. VLDB Endow.*, 18(11):4350–4363, 2025.

[9] T. Kaldewey, G. M. Lohman, R. Müller, and P. B. Volk. GPU join processing revisited. In *Proc. DaMoN*, pages 55–62, 2012.

[10] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. Di Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[11] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proc. SIGMOD*, pages 1633–1649, 2020.

[12] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Triton join: Efficiently scaling to a large join state on GPUs with fast interconnects. In *Proc. SIGMOD*, pages 1017–1032, 2022.

[13] T. Maltenberger, I. Tolovski, and T. Rabl. Efficiently joining large relations on multi-GPU systems. *Proc. VLDB Endow.*, 18(11):4653–4667, 2025.

[14] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. Distributed GPU joins on fast RDMA-capable networks. *Proc. ACM Manag. Data*, 1(1):29:1–29:26, 2023.

[15] J. Paul, B. He, S. Lu, and C. T. Lau. Revisiting hash join on graphics processors: A decade later. *Distributed and Parallel Databases*, 38(4):771–793, 2020.

[16] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt. WarpCore: A library for fast hash tables on GPUs. In *Proc. IEEE HiPC*, pages 11–20, 2020.

[17] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A dynamic hash table for the GPU. In *Proc. IPDPS*, pages 419–429, 2018.

[18] D. Bez, F. Kurpicz, H.-P. Lehmann, and P. Sanders. High performance construction of RecSplit based minimal perfect hash functions. In *Proc. ESA*, LIPIcs 274, pages 19:1–19:16, 2023.

[19] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on GPUs. In *Proc. ICDE*, pages 698–709, 2019.

[20] B. Wu, D. Koutsoukos, and G. Alonso. Efficiently processing joins and grouped aggregations on GPUs. *Proc. ACM Manag. Data*, 3(1):39:1–39:27, 2025.

[21] J. Yuan, C. Gong, Y. Deng, S. Chen, and B. Tang. Free your hash join from slow synchronizations: A fast hash join implementation on GPU. In *Proc. ACM TUR-C*, 2024.

[22] Y. Cai and S. Chen. CPU and GPU hash joins on skewed data. In *Proc. ICDE Workshops (HardBD/Active)*, pages 402–408, 2024.

[23] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated group-by and aggregation. In *Proc. ADMS@VLDB*, pages 13–24, 2015.

[24] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 383–402, 2015.

[25] M. Interlandi, N. Bruno, B. Haynes, C. Curino, R. Sen, Y. Li, K. Rajan, B. Ding, L. M. Maas, W. Cui, et al. CoddSpeed: Hardware accelerated query processing in Microsoft Fabric. In *Companion of the 2026 International Conference on Management of Data (SIGMOD Companion '26)*, pages 359–372, 2026. https://doi.org/10.1145/3788853.3803077.

[26] D. He, S. Nakandala, D. Banda, R. Sen, K. Saur, K. Park, C. Curino, J. Camacho-Rodríguez, K. Karanasos, and M. Interlandi. Query processing on tensor computation runtimes. *Proc. VLDB Endow.*, 15(11):2811–2825, 2022. https://doi.org/10.14778/3551793.3551833.

[27] A. Shanbhag, S. Madden, and X. Yu. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proc. SIGMOD*, pages 1617–1632, 2020.

[28] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. VLDB*, pages 443–452, 1991.

[29] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *The VLDB Journal*, 26(1):125–150, 2017.

[30] Databricks. Range join optimization. https://docs.databricks.com/aws/en/optimizations/range-join.html, accessed 2026.

[31] B. Yogatama, Y. Yang, K. Kristensen, D. Sarda, A. Kim, A. Cockcroft, Y. Teng, J. Patterson, G. Kimball, W. McKinney, W. Gong, and X. Yu. Rethinking analytical processing in the GPU era. In *Proc. CIDR*, 2026.

[32] H. Zhao, Y. Tian, R. Alotaibi, B. Ding, N. Bruno, J. Camacho-Rodríguez, V. Papadimos, E. Cervantes Juárez, C. Galindo-Legaria, and C. Curino. I can't believe it's not Yannakakis: Pragmatic bitmap filters in Microsoft SQL Server. In *Proc. CIDR*, 2026.

[33] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[34] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, 1986.

[35] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[36] Transaction Processing Performance Council. TPC Benchmark™ H (Decision Support) standard specification, revision 3.0.1. https://www.tpc.org/tpch/, 2022.

[37] B. Wu, W. Cui, C. Curino, M. Interlandi, and R. Sen. Terabyte-scale analytics in the blink of an eye. *Proc. VLDB Endow.*, 19(2):141–155, 2025. https://doi.org/10.14778/3773749.3773754.

[38] J. Luo, N. Boeschen, M. El-Hindi, and C. Binnig. PystachIO: Efficient distributed GPU query processing with PyTorch over fast networks & fast storage. arXiv:2512.02862, 2025.

[39] F. Aramburú, W. Malpica, K. Abrougui, et al. Theseus: A distributed and scalable GPU-accelerated query processing platform optimized for efficient data movement. arXiv:2508.05029, 2025.

[40] NVIDIA. Designing GPU-accelerated query engines with NVIDIA GQE. NVIDIA Developer Blog, https://developer.nvidia.com/blog/designing-gpu-accelerated-query-engines-with-nvidia-gqe/, accessed 2026.

[41] HEAVY.AI. HeavyDB: GPU-accelerated SQL analytics database (formerly MapD/OmniSci). https://github.com/heavyai/heavydb, accessed 2026.

[42] BlazingSQL. BlazingSQL: A GPU-accelerated SQL engine on RAPIDS/cuDF. https://github.com/BlazingDB/blazingsql, accessed 2026.