



One Ring to Shuffle Them All: Scalable Intra-Process Data Redistribution with Ring-Buffer Shuffle in Redpanda Osla

Adam Szymański

Redpanda Data

Warsaw, Poland

adam.szymanski@redpanda.com

Tyler Akidau

Redpanda Data

Seattle, USA

takidau@redpanda.com

ABSTRACT

As server CPUs scale to dozens and now hundreds of cores per socket, parallel query engines must rethink how they redistribute data between threads. Partitioned operators such as hash joins and aggregations require frequent data redistribution across threads, yet existing intra-process shuffle designs fundamentally fail to scale with core count: *batch partitioning* avoids cross-thread synchronization in the hot path but materializes all intermediate data, introduces a global producer/consumer barrier, and requires a consumption approach with low cache locality, while *channel-based streaming* avoids materialization but incurs per-channel synchronization that scales poorly with core count. As core counts rise, these architectural tradeoffs increasingly prevent engines from fully utilizing modern hardware.

We present a *ring-buffer streaming* shuffle design that addresses these shortcomings through lock-free atomic slot acquisition into fixed-size batch groups, achieving amortized $O(1)$ synchronization cost per batch and $O(M)$ memory independent of input size. Ring-buffer shuffle has been implemented in Redpanda’s Osla query engine for two years, where it currently powers production queries for Redpanda SQL users.

We evaluate all three approaches on a 72-core NVIDIA Grace-Hopper, a 192-core dual-socket AWS Graviton4, and a 96-core (192-thread) AMD EPYC. On a 72-core single-socket system the ring buffer outperforms channel streaming by up to 44% and batch partitioning by up to 79%; at 192 cores the advantage over channel grows to over 100% and over 300% versus batch partitioning. Even so, on chiplet architectures with many partitioned L3 caches, the shared atomic counter becomes a cross-die bottleneck and channel-based streaming remains competitive.

End-to-end Graviton4 evaluation on TPC-H (21 queries) and ClickBench (43 queries) shows the advantage is workload-shape-dependent: the ring buffer wins TPC-H in aggregate (1.07×) across diverse query shapes, but a handful of consumer-heavy patterns—COUNT(DISTINCT), wide aggregates—favor channels and swing ClickBench’s aggregate the other way.

PVLDB Reference Format:

Adam Szymański and Tyler Akidau. One Ring to Shuffle Them All: Scalable Intra-Process Data Redistribution with Ring-Buffer Shuffle in Redpanda Osla. PVLDB, 19(12): 4372 – 4384, 2026.

doi:10.14778/3827998.3828039

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828039

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/redpanda-data/ring_shuffle.

1 INTRODUCTION

Modern analytical database systems exploit multi-core hardware by executing query operators in parallel across threads. A wide range of relational operators—hash joins, hash aggregations, window functions—require that at some point during processing, all data related to a given key converges on a single thread. One way to achieve this is through shared synchronized state: multiple threads can concurrently read and update a shared data structure such as a partitioned hash map, avoiding explicit data movement altogether. Today this approach is used for specific operators—notably parallel hash join builds [10, 16] and, more recently, group-by aggregation [18]—and ongoing research suggests it may become competitive in a broader range of scenarios. Nevertheless, many production systems still rely on explicit *data redistribution*: given a stream of data produced in parallel by M threads, route items to N consumer threads based on a partitioning function. Importantly, what gets redistributed need not be the original input rows—systems may shuffle partial aggregates, hash map partitions, or other intermediate representations. This redistribution operation, commonly referred to as *shuffle* or *exchange*, is on the critical path of every partitioned operator and is a fundamental primitive in parallel query execution [6, 7].

The database community has extensively studied shuffle in the distributed setting—network protocols, serialization formats, compression, fault tolerance, and spilling strategies have all received significant attention [13, 14]. By contrast, the *intra-process* variant of the problem—exchanging rows between threads within a single process on a single machine—has received far less dedicated treatment. It is often regarded as a solved problem, either subsumed into operator-specific implementations or inherited wholesale from the Volcano exchange model [6]. Yet on modern many-core servers with 64 to 256 or more cores, this local redistribution is itself a significant performance bottleneck. As core counts grow, per-core memory bandwidth has not scaled proportionally—making cache misses increasingly costly. With the rapid growth in core counts, the efficiency of data movement between threads has become more important than ever.

Existing approaches to intra-process shuffle face a fundamental tension between *memory overhead* and *contention*. Batch-oriented designs, exemplified by morsel-driven systems such as HyPer and DuckDB [8, 10], have each thread accumulate input data locally,

partition it, and then merge matching partitions from all threads after a barrier. This avoids cross-thread synchronization during the accumulation phase but requires full materialization of intermediate results, consuming memory proportional to the input size and causing cache misses when consumers access data written by distant threads. Streaming designs, rooted in the Volcano exchange operator [6] and employed by systems such as StarRocks [15] and DataFusion [9], route rows to consumers as they are produced via per-partition channels. This avoids materialization but introduces synchronization on every channel operation; the resulting contention grows with the number of threads and becomes a scalability bottleneck on many-core hardware.

In this paper, we present a novel ring-buffer-based shuffle design that navigates between both extremes. Our approach streams data between producers and consumers without full materialization, yet achieves $O(1)$ memory per core and minimal cross-thread contention through a lock-free protocol based on atomic counters and fixed-size batch groups. We have implemented and deployed this design in Osla, a distributed analytical SQL engine developed at Redpanda, where it serves as the shuffle primitive for all partitioned operators in production workloads.

This paper makes the following contributions:

- We frame intra-process shuffle as a first-class problem, distinct from distributed shuffle, and identify the memory-versus-contention tradeoff that governs existing designs.
- We present a lock-free ring-buffer shuffle design with $O(1)$ memory per core, providing temporal data locality while avoiding the contention-scaling problems of channel-based approaches.
- We evaluate all three approaches—batch partitioning, channel-based streaming, and ring-buffer streaming—in controlled benchmarks across varying core counts and data distributions.

The remainder of this paper is organized as follows: Section 2 discusses related work on parallel query execution and exchange operators. Section 3 describes the three shuffle approaches in detail. Section 4 presents our experimental evaluation. Section 5 discusses production deployment experience in Osla. Section 6 concludes.

2 RELATED WORK

We organize related work along the conceptual progression from the foundational exchange model to the two dominant families of intra-process redistribution (batch partitioning and channel-based streaming), and finally to shared-state alternatives that sidestep explicit data movement.

2.1 The Exchange Operator

Graefe introduced the *exchange* operator in the Volcano system as a meta-operator that encapsulates parallelism without modifying the logic of data-processing operators [6, 7]. Exchange interposes between a producer and a consumer, using bounded packet queues in shared memory to transfer data between forked processes. A support function—hash, range, or round-robin—routes each packet to the appropriate consumer port. Within each process, execution remains demand-driven (iterators pull tuples); between processes, communication is data-driven (producers push packets into queues).

This clean separation of parallelism from operator logic made the exchange approach the template for plan-driven parallel query execution, and most modern systems—including Presto [14], StarRocks, and DataFusion—still employ exchange-derived operators for data redistribution.

However, the original exchange model was designed for relatively low process counts and does not address the memory hierarchy effects that dominate on modern many-core hardware. Rödiger et al. explicitly criticize exchange-based architectures for “unnecessary materialization” and inflexibility when scaling to high core counts [13]. The two families of approaches described below can be understood as different responses to these limitations.

2.2 Batch Partitioning

Morsel-driven parallelism, introduced in HyPer [10], eliminates the exchange operator entirely for intra-process parallelism. Instead of routing tuples through channels, threads pull fixed-size *morsels* from a shared input, process them independently into thread-local storage, and synchronize only at pipeline boundaries through barrier-like phase transitions. A dispatcher assigns morsels with NUMA-aware scheduling, ensuring threads primarily access local memory. For hash joins, the build phase inserts into a shared hash table via compare-and-swap (CAS); redistribution is implicit in the transition between pipeline phases rather than explicit in a dedicated operator.

Bandle et al. integrate radix-partitioned hash joins into the morsel-driven framework in Umbra [3]. Their approach uses two-pass radix partitioning with software write-combining buffers (SWWCBs) [12] and non-temporal stores to avoid polluting the cache hierarchy. Each thread partitions into thread-local pages; after a barrier, partitions from all threads are concatenated by exchanging metadata pointers rather than copying data. The key contribution is a cost model that predicts when radix partitioning pays off relative to a non-partitioned hash join, based on table sizes and available cache. In a companion paper, Bandle and Giceva decompose the exchange operator into composable *sub-operators*—scan, map, scatter, gather, fold—providing a vocabulary for reasoning about the individual steps of radix partitioning [2].

Kuiper et al. extend morsel-driven radix partitioning to external (larger-than-memory) hash joins in DuckDB [8]. Threads partition into spillable pages using a two-phase radix scheme; thread-local partitions are merged after a barrier. The design maintains the morsel-driven property that threads operate independently during partitioning, synchronizing only to merge partition metadata.

The key tradeoff of batch partitioning is clear: no cross-thread synchronization during the accumulation phase, but a barrier is required before consumers can proceed, and the accumulated data must be materialized in its entirety. When thread-local hash tables provide significant reduction—as in low-cardinality aggregation—the materialized state is compact and the merge phase is inexpensive. However, when key cardinality is high or the operation does not aggregate (e.g., a partitioned join build), memory consumption is proportional to the input size, and consumers accessing data written by distant cores incur cache misses during the merge phase. Balkesen et al. provide a comprehensive experimental study of these cache effects for multi-core hash joins [1].

2.3 Channel-Based Streaming

The alternative family preserves the streaming nature of the original exchange model, routing data to consumers as it is produced rather than accumulating it. Modern implementations replace Volcano’s process-level packet queues with thread-level channels using contemporary concurrency primitives. StarRocks [15] implements a LocalExchanger with multiple strategies—hash partitioning, round-robin, broadcast, and adaptive passthrough—where sink operators push chunks into per-partition buffers and source operators pull from them. DataFusion’s [9] RepartitionExec creates $N \times M$ asynchronous channels (for N input and M output partitions) with gate-based backpressure, and supports sort-preserving repartition and spill-to-disk under memory pressure.

At the lowest level, the choice of channel implementation matters significantly. Baumstark and Pohl benchmark lock-free single-producer single-consumer (SPSC) ring buffers against mutex-based queues for inter-thread tuple transfer, demonstrating 4–5× throughput improvement with lock-free designs [4]. Their work highlights that the synchronization primitive—not just the architectural pattern—is a critical performance factor.

Velox, Meta’s unified execution engine, also employs exchange operators for local data redistribution [11]. While the published description focuses on the broader execution framework rather than exchange internals, the architecture appears to follow the channel-based pattern, and the authors identify shuffle as a performance-sensitive component for partitioned operators.

The fundamental tradeoff of channel-based streaming is the inverse of batch partitioning: it avoids full materialization (achieving low memory overhead and low latency to first results), but requires synchronization on every channel operation. As the number of threads grows, contention on channel data structures becomes a scalability bottleneck.

2.4 Shared Synchronized State

A third approach avoids explicit redistribution entirely by having multiple threads concurrently access a shared data structure. Leis et al. use CAS-based insertion into a shared hash table for the join build phase [10]. PostgreSQL’s parallel hash join has multiple workers cooperatively build a shared hash table in dynamic shared memory [16]. More recently, Xue and Marcus show that purpose-built concurrent hash tables can match partitioning approaches for GROUP BY aggregation, arguing that shared state is viable and underexplored as a general strategy [18].

These techniques are currently operator-specific—tied to hash joins or aggregations—rather than general-purpose exchange mechanisms. They represent an important complementary direction but are outside the scope of this paper, which focuses on general data redistribution between parallel pipeline stages.

3 DESIGN

We now describe three approaches to intra-process shuffle in detail: *batch partitioning*, *channel-based streaming*, and *ring-buffer streaming*. The former two see broad adoption across the industry, as detailed in the previous section, while the latter is our novel approach from Redpanda Oxa.

Table 1: Comparison of shuffle design properties. M : producers, N : consumers.

Property	Batch	Channel	Ring-Buffer
Memory	$O(input)$	$O(N)$	$O(M)$
Sync rate [†]	none*	$O(M)$	$O(1)$
Latency	barrier	streaming	streaming
Consumer temporal locality	stale	uncorrelated	correlated

*No sync during accumulation; barrier at end.

[†]Total lock/mutex acquisitions per time unit across all producers.

All three solve the same problem: given M producer threads that generate columnar batches and N consumer threads that must each receive all rows assigned to them by a partitioning function h , route every input row to its designated consumer. A *batch* is a fixed-capacity column-oriented container holding up to B rows. Correctness requires that every input row is delivered to exactly one consumer, determined by h , with no duplication or loss.

The three designs differ in when and how they synchronize producers and consumers. We evaluate them along four axes: (1) total memory footprint, (2) synchronization rate as a function of thread count, (3) latency to first output, and (4) consumer cache locality. Table 1 summarizes the key properties of the three approaches.

All three approaches share a common preprocessing step: *batch indexing*. When a producer receives an input batch of up to B rows, it evaluates h for every row to determine each row’s target partition. It then constructs an index structure that allows any consumer to efficiently extract the rows belonging to its partition. This indexing requires a single pass over the batch ($O(B)$) and is entirely thread-local. What differs between the three designs is what happens after indexing: batch partitioning copies rows into thread-local buffers; channel-based and ring-buffer streaming both push indexed batches onward without copying, but differ in how they coordinate between producers and consumers.

3.1 Batch Partitioning

In batch partitioning, each producer accumulates its entire input locally before any consumer begins processing. The approach proceeds in three phases, illustrated in Figure 1.

In the *accumulation phase*, each of the M producer threads pulls input morsels and partitions rows by h into N thread-local buffers. Because each thread writes only to its own memory, no cross-thread synchronization is required during this phase.

A *barrier* follows: all producers must complete before any consumer can start. This is a global synchronization point that separates the write phase from the read phase.

In the *merge phase*, each consumer j reads partition j from all M producers’ thread-local buffers. Since the data was written by remote threads—potentially on distant NUMA nodes—these reads typically miss the local caches and incur main memory latency.

The total memory consumed is $O(|input|)$: the entire input is materialized in the thread-local buffers before consumers begin. When thread-local aggregation provides significant reduction (e.g., low-cardinality GROUP BY), the materialized state is only $O(M \times K)$

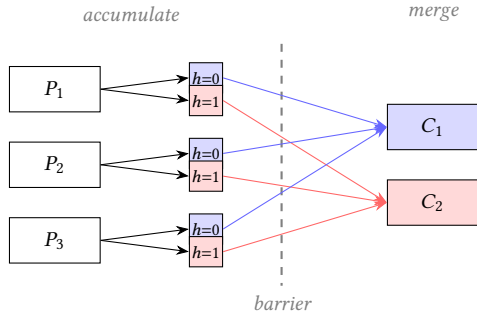


Figure 1: Batch partitioning with $M=3$ producers and $N=2$ consumers. Producers write to thread-local partition buffers without synchronization. After a barrier, each consumer reads its partition from all producers.

where K is the number of distinct keys, and the merge phase is correspondingly cheaper. However, for operations that do not aggregate—such as a partitioned join build with high key cardinality—the full input must be materialized, and the barrier prevents any pipelining between production and consumption.

Beyond the latency cost of the barrier itself, the two-phase structure introduces a *tail-latency* problem: within each phase, the available cores are fully utilized only until the fastest thread finishes. Partition skew or uneven morsel sizes cause some threads to complete earlier than others, leaving their cores idle until the slowest thread reaches the barrier. This underutilization occurs twice—once at the end of the accumulation phase and once at the end of the merge phase. In contrast, streaming approaches naturally self-balance: when a producer stalls (e.g., on a full channel), the OS schedules the co-resident consumer, keeping the core occupied.

3.2 Channel-Based Streaming

Channel-based streaming preserves the data-driven nature of the Volcano exchange model: producers route batches to consumers as they are produced, with no barrier between production and consumption. The architecture is shown in Figure 2.

Each output partition is associated with a *channel*—a bounded buffer with synchronization primitives that supports concurrent push and pull operations. After batch indexing, the producer pushes the indexed batch to each of the N output channels; the consumer on the other end extracts its partition’s rows using the precomputed index. Consumers pull from their channels, blocking when no data is available.

Backpressure is inherent: when a channel is full, the producer blocks until the consumer drains space. This bounds total memory at $O(N)$ —one channel per output partition, independent of the input size. Some systems use $M \times N$ dedicated single-producer single-consumer (SPSC) channels instead of N multi-producer channels, trading higher memory for lower per-channel contention.

The key advantage over batch partitioning is latency: the first output batch is available as soon as the first input batch is routed,

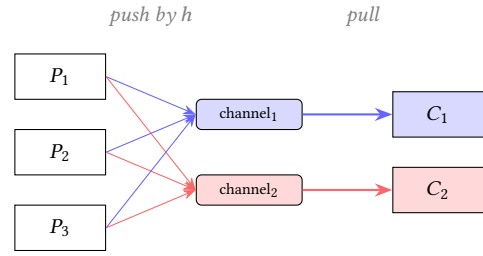


Figure 2: Channel-based streaming with $M=3$ producers and $N=2$ consumers. Each producer routes rows by h and pushes to the corresponding channel. Consumers pull from their dedicated channel. Synchronization occurs on every push and pull.

enabling pipelining with downstream operators. The disadvantage is synchronization cost. Each push and pull operation requires coordination—typically via *futex*, *mutex*, or atomic compare-and-swap—and with M producers competing on each of N multi-producer single-consumer (MPSC) channels, contention on individual channels grows with thread count. The total synchronization cost per input batch is $O(N)$: one channel operation per output partition.

3.2.1 Producer-Buffer Variant. A natural alternative to shared MPSC channels is a *producer-buffer* model, in which each producer maintains N private single-consumer output buffers—one per partition. Consumers poll all M producers’ buffers for their partition, yielding $M \times N$ independent SPSC channels. Because each channel has exactly one writer and one reader, synchronization can be entirely lock-free (e.g., a single-producer single-consumer ring buffer [4]), eliminating the multi-producer contention that limits the MPSC channel design. The tradeoff is threefold. First, memory grows to $O(M \times N)$ channel instances, which at 192 producers and 192 consumers means 36,864 buffers—each requiring its own allocation and bookkeeping. Second, consumers must visit M separate buffers per batch-group cycle, and the data in those buffers was written by M different producer cores at uncorrelated times, so consumers lose the temporal L3 sharing that the ring-buffer design provides (subsubsection 3.3.9). Third, consumers must implement a polling or notification strategy across M sources, adding per-consumer coordination complexity. The producer-buffer model thus trades lower write-path contention for higher memory, weaker cache locality on the read path, and more complex consumer scheduling. We did not benchmark this variant; a quantitative comparison is an interesting direction for future work.

3.3 Ring-Buffer Streaming

Oxla’s ring-buffer design retains the streaming property of channel-based exchange while achieving amortized $O(1)$ synchronization cost per batch and temporal cache locality across consumers. The key idea is to replace per-partition channels with a single shared ring buffer of fixed-size *batch groups*, where producers write concurrently using lock-free atomic counters and consumers independently advance through completed groups—drawing on the

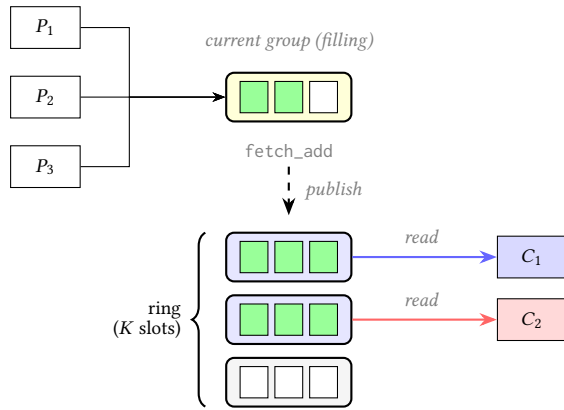


Figure 3: Ring-buffer streaming. Producers acquire slots in the current batch group via atomic `fetch_add`. Full groups are published to the ring buffer (K slots). Each consumer independently tracks its read position and advances through the ring. When consumers are faster than producers, they converge on the same group, enabling temporal cache sharing. Filled slots shown in green; empty in white.

sequenced ring-buffer pattern introduced by the LMAX Disruptor [17]. Figure 3 illustrates the architecture. Figure 4 summarizes both producer and consumer algorithms in pseudocode.

3.3.1 Indexed Batches. As in channel-based streaming, each producer constructs an indexed batch after the common batch indexing step. The critical difference is how this indexed batch reaches consumers. In channel-based streaming, the producer pushes the indexed batch to N separate channels, incurring a synchronization operation per channel. In the ring-buffer design, the producer places the indexed batch into a single shared structure—a *batch group*—and all consumers read from it there.

3.3.2 Batch Groups. A *batch group* is a fixed-capacity array of G slots, each holding one indexed batch. The capacity G is set proportional to M (the number of producers), so that each producer can contribute approximately one batch before the group fills. Producers acquire a slot by performing a single `fetch_add` on an atomic counter—this is the core lock-free operation of the design. If the returned value exceeds G , the group is already full and the producer retries with the next group. Otherwise, the producer writes its indexed batch to the acquired slot without any further synchronization. When a producer fills the final slot in a group, it becomes the *publisher* for that group: it pushes the completed group into the ring buffer and installs a fresh group for subsequent writes.

3.3.3 Data Structure. The ring-buffer shuffle consists of a shared queue, per-producer state, and per-consumer state.

Batch group. A batch group is a fixed-capacity array of G slots, each holding one indexed batch, plus three atomic counters: `writes_started`, incremented via `fetch_add` to claim a slot; `writes_completed`, incremented after writing, which triggers the full flag when it reaches G ; and `consumers_left`, initialized to N and decremented by each consumer when it finishes reading the

group. An atomic boolean full flag signals producers that the group is exhausted.

Shared state. The shuffle queue holds a circular ring buffer of K batch group slots (where K is a small constant, typically 1–3), an atomic published counter tracking how many groups have been published, the current *insertion buffer* (the batch group being filled by producers), and a mutex with condition variables for publish, consumer blocking, and producer backpressure. Ring position and occupancy are protected by the mutex.

Per-producer state. Each producer holds a private reference to the current insertion buffer, protected by a per-producer mutex. This avoids a shared-pointer bottleneck: when the publisher installs a new insertion buffer, it updates each producer’s reference individually, so producers wake and acquire only their own lock—eliminating cross-producer contention. Each producer also holds a pre-allocated replacement batch group, ready to donate when it triggers a publish.

Per-consumer state. Each consumer tracks its read position (an absolute index into the sequence of published groups) and a cached copy of the published counter. The cached copy avoids atomic loads on the hot path: the consumer first compares its position against the local cache, performs an atomic load only when caught up, and blocks on a condition variable only when the atomic load also shows no new groups.

3.3.4 Producer Algorithm. When a producer has an indexed batch to submit, it proceeds as follows:

- (1) Check the full flag on the batch group referenced by its private pointer. If the group is full, wait on the per-producer condition variable until the publisher installs a new group and updates this producer’s reference.
- (2) Claim a slot via atomic `fetch_add` on `writes_started`. If the returned index exceeds G , the group filled concurrently—return to step 1.
- (3) Write the indexed batch to the claimed slot. No synchronization is needed for the write itself.
- (4) Increment `writes_completed`. If this was the G -th completion:
 - Set the full flag.
 - Acquire the queue mutex. If all K ring slots are occupied, block until a consumer frees a slot (*backpressure*).
 - Push the full group into the ring, install the pre-allocated replacement as the new insertion buffer, and increment the published counter.
 - Update all producers’ private references to point to the new buffer, and notify waiting consumers and producers.
 - After releasing the mutex, allocate a fresh replacement group—off the critical path.

The total number of batches in flight is bounded by $K \times G$, independent of the input size.

3.3.5 Consumer Algorithm. Each consumer independently advances through published batch groups:

- (1) Wait for a new batch group. The consumer checks a three-tier progression of increasing cost: first a cached local copy

Producer(batch) <pre> 1: loop 2: if group.full then wait(cv_{prod}) 3: s ← fetch_add(group.started, 1) 4: if s < G then break 5: group.slots[s] ← batch 6: c ← fetch_add(group.completed, 1) 7: if c + 1 = G then // publish 8: group.full ← true 9: lock(μ) 10: while ring is full: wait(cv_{bp}) 11: ring.push(group) 12: group ← replacement 13: notify(cv_{cons}, cv_{prod}) 14: unlock(μ) </pre>	Consumer(id) <pre> 1: loop 2: if pos < cached_pub then break 3: cached_pub ← load(published) 4: if pos < cached_pub then break 5: wait(cv_{cons}) 6: g ← ring[pos mod K] 7: for each slot b in g 8: extract(b, id) 9: r ← fetch_sub(g.consumers_left, 1) 10: if r = 1 then // last reader 11: ring.free(pos mod K) 12: if occ. ≤ K/2: notify(cv_{bp}) 13: pos ← pos + 1 </pre>
---	--

Figure 4: Ring-buffer pseudocode. Left: per-batch producer path (lines 1–6) requires one lock-free fetch_add; the publish path (lines 7–14) acquires a mutex but runs only once per G batches. Right: the consumer is lock-free or local except for the condition variable wait on line 5, which is reached only when caught up to producers.

of the published counter (no shared access), then a single atomic load to refresh it, and finally a condition variable wait. This eliminates shared-state access entirely when groups are buffered ahead.

- (2) Access the batch group at ring position $read \bmod K$ and iterate over its indexed batches, extracting the rows designated for this consumer’s partition.
- (3) Decrement `consumers_left`. If this consumer is the last reader (count reaches zero): free the ring slot and notify producers if ring occupancy drops below a threshold (e.g., half capacity), allowing multiple slots to accumulate before producers wake. This avoids excessive context switching when consumers are slower than producers.

Different consumers may be reading different batch groups at any given time. When consumers are faster than producers and no new group is available, all waiting consumers block on the same condition variable and are woken simultaneously when the next group is published.

3.3.6 Synchronization Analysis. The producer *hot path*—slot acquisition within a batch group—requires a single atomic `fetch_add`, which is $O(1)$ and lock-free. The *cold path*—publishing a full batch group to the ring—requires one mutex acquisition per G batches. Since $G \propto M$, if each of the M producers generates one batch per time unit, the group fills once per time unit regardless of M , yielding a constant mutex acquisition rate. By contrast, in channel-based streaming each batch requires a lock per channel, so the total lock acquisition rate is $O(M)$ per time unit—contention grows linearly with producer count. The ring-buffer design eliminates this scaling.

On the consumer side, the fast path is a single atomic read of the published group counter—if a new group is available, the consumer proceeds with no lock. When consumers are faster than producers and no new group is available, the contention on the condition variable is benign: the consumers are idle anyway, and once woken they operate independently on disjoint slices of the same data without further synchronization.

3.3.7 Practical Considerations. The asymptotic properties above—amortized $O(1)$ synchronization, $O(M)$ memory—describe the design but do not guarantee good performance on many-core hardware. Two implementation techniques are necessary to realize these properties in practice.

Pre-allocated replacement groups. Publishing a full batch group requires installing a fresh empty group for producers to write into. Allocating this group under the queue mutex—initializing G slots—extends the critical section unnecessarily, increasing the window during which other threads are blocked. Instead, each producer maintains a pre-allocated replacement group. When a producer happens to fill the final slot and triggers a publish, it donates its replacement via a pointer swap (zero-cost under the mutex). After releasing the mutex, it allocates a fresh replacement for next time—off the critical path.

Selective producer notification. When consumers are slower than producers, producers fill the ring and block waiting for space. If a producer is woken as soon as a single slot frees, it fills that slot and immediately blocks again—each group published triggers a round of context switches between producers and consumers for minimal useful work. Instead, producers are notified only when ring occupancy drops to a threshold (e.g., half capacity), allowing multiple slots to accumulate before producers wake. This gives producers enough work to justify the context-switch cost and lets consumers run uninterrupted for longer stretches.

3.3.8 Memory Analysis. The ring buffer holds at most $K \times G = O(M)$ indexed batches in flight. Per-producer private state is $O(1)$: one batch under construction. The total memory footprint is bounded by the ring capacity and proportional to the number of producers, independent of input size. This contrasts with batch partitioning, where memory is $O(|input|)$.

3.3.9 Cache Locality. The ring-buffer design provides a form of temporal cache locality that neither alternative achieves. Each consumer independently tracks its position in the ring, so different consumers may be processing different batch groups at any given time. However, when consumers are faster than producers—a common scenario when shuffle is the performance bottleneck—consumers that finish a batch group quickly have no new group to advance to and block until the next group is published. At that point, all waiting consumers are notified simultaneously and converge on the same batch group. They each read the same underlying batch data, extracting only their partition’s rows via the index. The first consumer to touch a batch pulls its columns into a shared cache level (typically L3); subsequent consumers find the data already warm, paying only L3 hit latency rather than main memory latency. In channel-based streaming, consumers also hold references to the same indexed batches—the data is not copied. However, each consumer pulls from its own channel independently, so consumers process the same batch at different, uncorrelated times. Without the synchronized wake-up that the ring-buffer provides, temporal clustering is weaker and cache sharing is less reliable. In batch partitioning, consumers read data long after producers wrote it (post-barrier), by which time the data has typically been evicted, especially on many-core systems where the aggregate working set exceeds L3 capacity.

3.3.10 Partition Key Skew. When the partitioning function h distributes rows unevenly (e.g., a small number of hot keys dominate the input), some consumers receive substantially more data than others. In the ring-buffer design, all N consumers must finish reading a batch group before its ring slot can be reclaimed (consumers_left must reach zero). A slow consumer handling a hot partition therefore gates slot reclamation for the entire group, reducing the effective ring capacity and increasing the likelihood that producers stall on backpressure. This effect is not unique to the ring buffer: in channel-based streaming, backpressure on a hot channel similarly stalls producers that hash to the same partition, and in batch partitioning, the barrier-to-merge latency is determined by the slowest consumer. The ring-buffer design is arguably better positioned than batch partitioning for skewed workloads, because streaming allows fast consumers to proceed immediately rather than waiting at a barrier, and backpressure from a slow consumer affects only the publication rate—not the ability of other consumers to process already-published groups. Nevertheless, extreme skew can degrade ring-buffer throughput, and workloads with known hot keys may benefit from pre-splitting hot partitions or using skew-aware partitioning functions upstream of the shuffle.

4 EVALUATION

The ring-buffer design relies on atomic counters shared across all producer threads. The efficiency of these atomic operations depends critically on whether the underlying cache line can be transferred between cores through a shared last-level cache or must traverse a slower interconnect between separate cache domains. We therefore select as our primary evaluation platform a dual-socket AWS Graviton4 system (c8g.metal-48xl) with 2×96 Arm Neoverse V2 cores (192 total), where each socket provides a 36 MB L3 shared across its 96 cores via the Arm CMN-700 mesh. In subsection 4.3, we contrast these results with a 72-core single-socket Neoverse V2 server (NVIDIA GraceHopper, 117 MB unified LLC) and a chiplet-based AMD EPYC processor to quantify the impact of cache topology on each design.

We evaluate the three shuffle designs using a standalone C++ benchmark¹. Each experiment uses $M=N$ producer-consumer threads, 8192 rows per chunk, and 1000 chunks per producer. All results report the median of 5 runs; variance across runs was low (coefficient of variation below 5% for all configurations). Producers and consumers are pinned to the same physical cores to simulate a fixed core budget, as is typical in database engines. Note that this approach simulates limited compute but not limited memory bandwidth—lower-core-count systems typically also have proportionally lower memory bandwidth, which may shift the crossover point between synchronization-bound and bandwidth-bound regimes. All three implementations were developed by the authors with equal optimization effort; the benchmark code is available in the artifact repository. The Channel baseline uses one bounded MPSC queue per output partition (N total), each backed by a `std::vector` under a `std::mutex` with separate condition variables for not-full and not-empty; capacity is fixed at M batches per partition. The Batch baseline gives each producer N thread-local `std::vector` buckets that hold indexed-batch pointers; producers

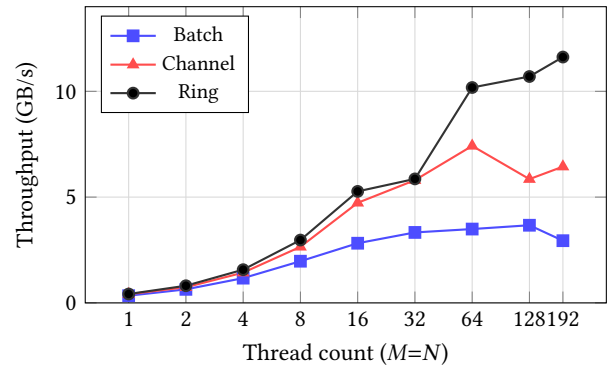


Figure 5: Throughput scaling with thread count on a 192-core dual-socket Graviton4 (Neoverse V2). Row size is 8 bytes (uniform). Beyond 96 cores (socket boundary), channel throughput drops while ring-buffer streaming continues to scale; batch saturates early.

append without synchronization, and after all producers join, consumers iterate across all M producer buckets to extract their partition. All three designs shuffle indexed-batch pointers rather than copying row payloads, so throughput differences reflect coordination cost rather than data movement.

4.1 Scalability

Figure 5 shows throughput as thread count increases from 1 to 192 on the dual-socket Graviton4, with 8-byte rows and uniform row sizes. At low core counts (≤ 8), all three approaches perform similarly—contention on shared synchronization state is minimal, so the ring-buffer’s mechanisms for avoiding it provide no benefit. Starting at 16 cores, contention becomes the dominant cost and the ring-buffer design ($K=1$) begins to pull ahead. At 64 cores (within a single socket), ring-buffer streaming achieves 10.18 GB/s, compared to 7.42 GB/s for channel-based streaming (+37%) and 3.49 GB/s for batch partitioning (+192%). Beyond 96 cores, threads span both sockets; channel throughput drops from 7.42 GB/s at 64 cores to 5.85 GB/s at 128 as cross-socket synchronization costs increase, while ring-buffer throughput continues to rise. At 192 cores, ring-buffer streaming achieves 11.62 GB/s, compared to 6.44 GB/s for channel-based streaming (+80%) and 2.94 GB/s for batch partitioning (+295%).

4.2 Row Size and Distribution Sensitivity

Table 2 compares the three approaches at 192 cores across batch sizes from 64 KB to 2048 KB (8192 rows with row sizes from 8 to 256 bytes), under both uniform and normal ($\mu=\text{row_size}$, $\sigma=\mu/4$) row-size distributions. For each configuration, we report ring-buffer throughput ($K=1$) alongside the percentage improvement over batch and channel. The highest throughput per configuration is shown in bold. At 1024 KB and 2048 KB, batch partitioning exceeds the 128 GB memory limit and cannot run—a direct consequence of its $O(|input|)$ memory footprint at 192 producers.

The ring-buffer advantage is largest at small batch sizes: at 64 KB with uniform rows, ring-buffer streaming achieves +253% over

¹https://github.com/redpanda-data/ring_shuffle

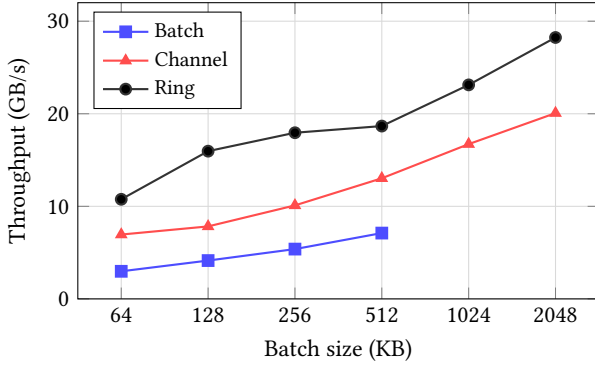


Figure 6: Throughput vs. batch size at 192 cores (Graviton4) with ring-buffer $K=1$ and normally distributed row sizes. Batch partitioning exceeds memory at ≥ 1024 KB. The ring-buffer advantage is largest at small batch sizes where synchronization dominates.

batch and +106% over channel. Small batches are processed quickly, causing producers and consumers to hit synchronization points more frequently per unit time—maximizing contention on locks and condition variables. As batch size increases, each chunk takes longer to process, reducing the synchronization rate, and the workload shifts from synchronization-bound to memory-bandwidth-bound. At 2048 KB, ring-buffer leads channel by +67%—the gap narrows but remains substantial at 192 cores.

Figure 6 visualizes this trend under normal row-size distribution, which better reflects real-world workloads where row sizes vary. The ring-buffer curve remains above the other two across all batch sizes.

The normal distribution shows slightly lower absolute throughput. Variable row sizes cause uneven batch sizes across partitions: some consumers receive more data than others, leading to stragglers. In streaming approaches, faster consumers back-pressure their producers while waiting for slower consumers to catch up. In batch partitioning, the barrier amplifies the imbalance—the slowest consumer determines the completion time of the merge phase. Despite this, the relative advantage of ring-buffer streaming is preserved across both distributions.

4.3 Impact of Cache Topology

The results in subsection 4.1 and subsection 4.2 were measured on the dual-socket Graviton4 system described in section 4. To evaluate the impact of cache topology, we compare with two additional platforms. First, a 72-core single-socket Arm Neoverse V2 server (NVIDIA GraceHopper) with a unified 117 MB LLC shared by all cores—a single coherence domain. Second, an AMD EPYC 7R13 server with 96 physical cores (192 vCPUs via SMT), which uses a chiplet architecture: 12 Core Complex Dies (CCDs), each containing 8 cores with a private 32 MB L3 cache. There is no shared cache across CCDs—cross-CCD coherence is maintained via a directory-based protocol over the Infinity Fabric interconnect, with measured latency increasing significantly across CCD boundaries [5].

Table 2: Throughput (GB/s) at 192 cores (Graviton4). Ring-buffer uses $K=1$. Batch size is rows $\times$ row size (8192 rows). Percentage gain of ring-buffer over each baseline shown in parentheses. Best throughput per configuration bolded. “—” indicates out-of-memory (> 128 GB).

Batch (KB)	Dist.	Batch	Channel	Ring $K=1$
64	uniform	3.29 (+253%)	5.64 (+106%)	11.61
128	uniform	3.97 (+335%)	9.17 (+88%)	17.26
256	uniform	5.26 (+300%)	11.41 (+84%)	21.04
512	uniform	7.30 (+301%)	15.26 (+92%)	29.31
1024	uniform	—	19.05 (+85%)	35.17
2048	uniform	—	23.85 (+67%)	39.82
64	normal	2.98 (+261%)	6.95 (+55%)	10.76
128	normal	4.14 (+286%)	7.83 (+104%)	15.96
256	normal	5.38 (+234%)	10.10 (+78%)	17.95
512	normal	7.11 (+163%)	13.03 (+43%)	18.67
1024	normal	—	16.72 (+38%)	23.12
2048	normal	—	20.07 (+41%)	28.24

Table 3 compares ring-buffer and channel-based streaming at full core count on all three architectures. On the 72-core Grace system, ring-buffer streaming outperforms channel streaming at every batch size, with gains up to +44%. On the 192-core Graviton4, the ring-buffer advantage is substantially larger—up to +106% at 64 KB—despite the dual-socket NUMA topology, because the atomic counter bounces between only 2 sockets rather than 12 CCDs. The higher core count amplifies channel contention, making the ring buffer’s amortized $O(1)$ synchronization rate more valuable. Additionally, the Graviton4’s higher aggregate memory bandwidth (dual-socket DDR5) raises the bandwidth ceiling, keeping the workload in the synchronization-bound regime across all tested batch sizes. On Grace and EPYC, memory bandwidth saturates at smaller batch sizes, compressing the gap between approaches—on EPYC this effect is particularly pronounced, as 192 vCPUs sharing the available bandwidth achieve lower peak throughput (18.42 GB/s at 2048 KB) than Grace with only 72 cores (21.79 GB/s). Notably, peak shuffle throughput at 2048 KB is roughly proportional to each platform’s theoretical memory bandwidth—Grace (LPDDR5X, 546 GB/s), Graviton4 (dual-socket DDR5, $\sim 1,075$ GB/s), and EPYC (dual-socket DDR4, ~ 410 GB/s)—and Grace achieves the highest per-core shuffle throughput (0.30 GB/s versus 0.21 for Graviton4 and 0.10 for EPYC), consistent with its 7.6 GB/s per-core theoretical bandwidth. On the EPYC, the picture is mixed: channel streaming matches or exceeds the ring buffer at 256 KB, 512 KB, and 2048 KB batches, while the ring buffer regains the lead at 1024 KB—likely due to interaction between batch size and CCD-local cache capacity.

We hypothesize that the performance difference stems from how atomic operations are resolved on each architecture. On the Grace system, atomic operations on shared counters (slot acquisition, completion tracking) resolve within the unified LLC at low latency. On the EPYC, no shared cache exists across CCDs. Ownership of the cache line holding the ring buffer’s atomic slot counter must transfer between CCDs via the Infinity Fabric, with each transfer incurring significantly higher latency than an intra-CCD operation [5]. Since the ring buffer funnels all slot acquisition through a single atomic

Table 3: Ring vs. channel throughput (GB/s) across cache topologies. Grace: 72 Neoverse V2 cores, unified 117 MB LLC. Graviton4: 2×96 Neoverse V2 cores, 2×36 MB L3 (dual socket). EPYC Milan: 192 vCPUs (96 cores + SMT), 12 CCDs with 32 MB L3 each. All use ring $K=1$. Best per configuration bolded.

Batch	Grace (72c)		Graviton4 (192c)		EPYC (192v)	
	Chan.	Ring	Chan.	Ring	Chan.	Ring
64 KB	7.87	11.30	5.64	11.61	3.45	3.94
128 KB	12.33	15.37	9.17	17.26	5.35	5.65
256 KB	15.29	18.35	11.41	21.04	8.43	8.26
512 KB	17.24	18.86	15.26	29.31	12.54	11.85
1024 KB	17.73	19.98	19.05	35.17	15.87	18.01
2048 KB	19.45	21.79	23.85	39.82	18.42	17.77

counter, this cache line bounces across 12 CCDs on every group fill, serializing producers at interconnect speed—a pattern analogous to the timestamp counter scalability collapse observed by Yu et al. [19]. The EPYC’s mixed results thus stem from two compounding factors: cross-CCD atomic bouncing penalizes the ring buffer’s single-counter design, while early bandwidth saturation limits the throughput ceiling for all approaches, leaving little room for the ring buffer’s lower synchronization overhead to translate into gains.

These results suggest that the ring-buffer design is best suited for architectures with *few coherence domains*—whether a single unified LLC (Grace) or a small number of sockets with per-socket shared caches (Graviton4). The critical factor is not a single unified cache per se, but the number of domains across which the atomic counter must bounce: 1 domain (Grace) and 2 domains (Graviton4) both strongly favor the ring buffer, while 12 domains (EPYC CCDs) erode the advantage. On multi-CCD architectures such as AMD EPYC and multi-tile Intel Xeon, channel-based streaming remains competitive, particularly at larger batch sizes where the workload is memory-bandwidth-bound. A NUMA-aware variant of the ring-buffer design—using per-CCD batch groups with local atomic counters—could potentially recover the advantage on chiplet architectures but is beyond the scope of this work.

4.4 Effect of Ring Capacity K

The experiments above all use $K=1$, the smallest possible ring buffer. subsection 3.3 noted that K is typically 1–3; we now measure how K affects throughput and show that the optimal value depends on cache topology.

Figure 7 sweeps $K \in \{1, 2, 3, 4\}$ across batch sizes at full core count on each platform (Graviton4 was measured for $K \in \{1, 2\}$ only). K controls the slack between producers and consumers: with $K=1$ the ring holds at most one already-published group, so producers stall when the ring is full and they need to publish a fresh group, and consumers stall when no new group has yet been published. The frequency of both stalls scales inversely with per-group time—when each group fills or drains in microseconds, even brief rate variance causes frequent stalls; when each group takes milliseconds, stalls are rare regardless of K . Larger K provides additional in-flight groups that absorb this variance, but at the cost of a larger working set across in-flight groups and weaker consumer convergence on shared groups (subsection 3.3.9).

Grace. At small batch sizes (≤ 128 KB), $K=2$ and $K=3$ outperform $K=1$ by 9–17%. Two factors align in $K>1$ ’s favor: the aggregate working set across multiple in-flight groups still fits in the 117 MB L3 (no cache penalty), and per-group times are short enough that stall frequency is high, so the additional buffering pays for itself. Beyond 256 KB the picture flips: the working set under $K>1$ begins to exceed L3, consumers drift onto different groups (breaking the convergence described in subsection 3.3.9), and stall frequency drops below the threshold where buffering matters. $K=1$ retakes the lead by 2–6%. $K=4$ falls off most steeply at medium batch sizes (256–512 KB) where the four-group working set is first to exceed L3.

Graviton4. $K=1$ dominates $K=2$ at every batch size, with the gap peaking at 256–512 KB (32%) and remaining substantial (18%) at 2048 KB. The dual-socket NUMA topology turns every additional in-flight group into cross-socket cache-line traffic on the shared atomic counters and ring bookkeeping; this fixed coordination cost outweighs the stall-reduction benefit observed on Grace, even at small batch sizes.

EPYC. $K=1$ wins at most batch sizes; the only exception is 512 KB, where $K=2$ leads by 14% before $K=1$ resumes its lead at larger sizes. The chiplet topology’s high cross-CCD coherence latency penalizes additional in-flight state, mirroring the Graviton4 pattern.

The rule that emerges: default to $K=1$, and tune toward $K=2$ –3 only on architectures with a single coherence domain at small batch sizes, where the high stall frequency makes the additional buffering pay for itself. Production Oxla deployments on Grace-class hardware use $K=2$. On chiplet or multi-socket hardware where atomic-counter traffic crosses cache domains, $K=1$ is consistently the safe default. Across all three platforms $K=4$ is dominated by smaller K values, validating the “typically 1–3” design guideline of subsection 3.3.

4.5 End-to-End Query Impact

The microbenchmarks above isolate the shuffle primitive. To measure how that translates to real queries, we compared the ring-buffer build (currently in production at Oxla) against a channel-based build that wires Oxla’s hash-aggregation pipeline through one MPSC channel per output partition, with rows hash-routed and row-copied into per-bucket batches before being pushed onto each channel. The channel-based shuffle is not a hypothetical baseline: it was Oxla’s production shuffle before the ring-buffer design replaced it. Both builds run on the c8g.metal-48xl Graviton4 from section 4; ring capacity is $K=2$ unless noted. We report median wall-clock from warmed-up runs; coefficient of variation is below 3% across all measurements. We evaluate two standard analytic suites at the scales they are typically run: TPC-H Q1–Q22 at SF=100 (Q21 omitted—it requires EXISTS subqueries that Oxla does not support) and the full 43-query ClickBench suite at the standard 99.9 M-row scale.

Table 4 reports the suite-level results alongside the microbenchmark from subsection 4.2. On TPC-H—21 queries spanning hash joins with multi-table aggregations, sub-queries, complex predicates, and string aggregations—the ring-buffer wins in aggregate (86.6 s vs 92.9 s, 1.07×). On ClickBench—43 variations on *filter* + *GROUP BY* + *aggregate*—the channel build wins the suite total (9.0 s

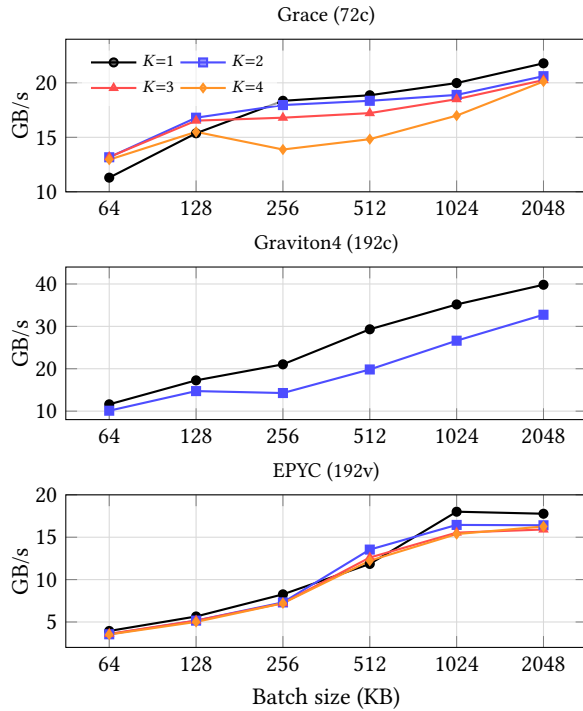


Figure 7: Ring-buffer throughput vs. batch size at full core count, sweeping $K \in \{1, 2, 3, 4\}$ (Graviton4 measured for $K \in \{1, 2\}$ only; uniform row sizes, flat distribution). On Grace, $K=2$ and $K=3$ win at small batch sizes where stall frequency is high; on Graviton4 and EPYC, additional in-flight groups generate cross-domain cache-line traffic that outweighs the stall-reduction benefit, so $K=1$ dominates almost everywhere.

Table 4: Ring-buffer vs channel-based shuffle on Graviton4 (192 cores). Ring uses $K=2$. Microbenchmark row from subsection 4.2; suite rows are warmed-up sums-of-medians (TPC-H 21 queries, ClickBench 43 queries).

Workload	Ring	Channel	Ring vs Ch.
Microbench (8 B rows, 64 KB)	11.6 GB/s	5.6 GB/s	Ring 1.6×
TPC-H 21 queries (sum)	86.6 s	92.9 s	Ring 1.07×
ClickBench 43 queries (sum)	13.5 s	9.0 s	Ch. 1.49×

vs 13.5 s, 1.49×), driven by four queries on the COUNT(DISTINCT) and wide-aggregate end.

The two suite-level numbers measure different things. TPC-H spans diverse query shapes—hash joins, sub-queries, multi-table aggregations, string aggregations, predicates of widely varying selectivity—across 21 queries; ClickBench densely samples a narrow band of *filter + GROUP BY + aggregate* variants across 43 queries. The aggregate ratio for TPC-H therefore measures how the two designs compare across diverse query shapes; the aggregate ratio for ClickBench measures how they compare under one dominant shape repeated many ways.

Figure 8 shows the per-query distribution behind those numbers. Each panel plots $\log_2(t_{\text{channel}}/t_{\text{ring}})$ for every query in the suite, sorted: positive bars are queries where the ring-buffer is faster, negative bars where the channel build is faster. The two distributions look very different. TPC-H clusters within roughly ± 0.55 on the log axis (i.e. no query is more than $1.47\times$ apart between the two builds), and the bulk of the queries are positive. ClickBench is bimodal: a long tail of small ring-buffer wins and ties on the right, plus a handful of queries on the left where the ring-buffer loses by a much larger margin than anywhere on the TPC-H panel.

Where the ring-buffer wins, and where it loses. The ring-buffer’s wins concentrate where a query exposes the shuffle as a throughput-bound stage with simple per-group state: hash joins on integer keys with a small set of post-shuffle aggregates (TPC-H Q22, Q8, Q17, Q5, Q9, Q12, Q19), or GROUP BY on a moderate-cardinality string column with a counter aggregate (ClickBench Q15, Q2, Q31, Q13, Q32). The wins are bounded—typically $1.1\text{--}1.5\times$, and only Q22 on TPC-H reaches $1.47\times$. The ring-buffer’s losses concentrate in two narrow query shapes that put heavy per-row work on the consumer side: COUNT(DISTINCT) per group (ClickBench Q14: $4.2\times$ slower; TPC-H Q13: $1.17\times$ slower) and wide per-row arithmetic over many simultaneous aggregates (ClickBench Q33: $4.9\times$ slower for its 90 SUM expressions). When the shuffle’s consumer maintains a hash set per group, or evaluates dozens of arithmetic expressions per row before aggregating, the ring-buffer’s shared-batch + indexed-read access pattern is unfavorable for the consumer’s working set, while the channel build delivers each consumer’s rows already grouped contiguously.

The ring-buffer’s advantage is therefore distributed as many small wins concentrated in queries that expose its design regime; its disadvantage is concentrated in a few specific query shapes that defeat that regime. Whether a workload nets out positive or negative for the ring-buffer depends on how many queries of each shape it contains. TPC-H, dominated by hash joins on integer keys with simple aggregates, contains many ring-favorable queries and few unfavorable ones; ClickBench contains both, plus a handful of COUNT(DISTINCT) and wide-aggregate queries large enough in absolute time to swing the suite total.

K does not flip the ranking. We sweep ring capacity $K \in \{1, 2\}$ on the ring-buffer build. Across both suites the two settings are within 3% of each other (TPC-H sums: $K=1$ 83.9 s, $K=2$ 86.6 s; ClickBench sums: $K=1$ 13.8 s, $K=2$ 13.5 s). The per-query pattern in Figure 8 is structural—a property of the design comparison, not of the parameter setting. Detailed K-vs-batch-size data is in subsection 4.4.

Synthesis. The TPC-H result is the broad-applicability evidence: across 21 queries that exercise hash joins, sub-queries, multi-table aggregations, string aggregations, and filters of widely varying selectivity, the ring-buffer wins in aggregate, and the wins are distributed across diverse query shapes rather than concentrated in any one. The microbenchmark in subsection 4.2—CRC-only consumers, continuous full-rate production, row size swept from 8 to 256 bytes—describes a regime (light per-row consumer work, no per-group state, throughput-bound shuffle) that recurs inside real queries as the inner loop of a hash join with simple aggregates, or a

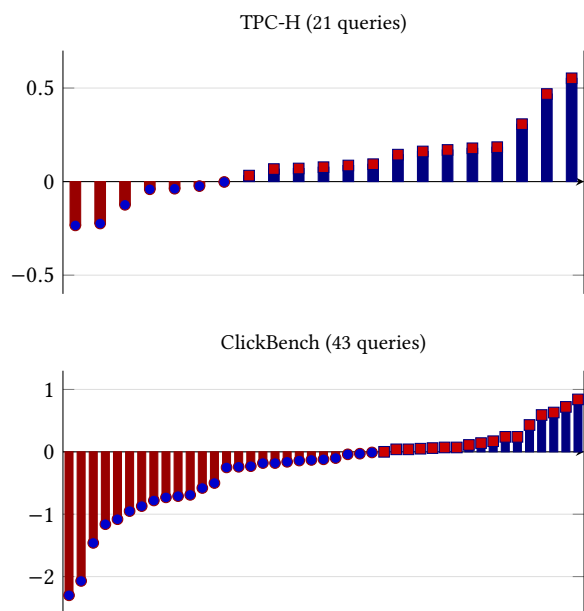


Figure 8: Per-query $\log_2(t_{\text{channel}}/t_{\text{ring}})$ on Graviton4 (192 cores), sorted. Positive bars: ring-buffer faster. TPC-H bars stay within ± 0.55 on the log axis (max $1.47\times$ either way) and the bulk are positive. ClickBench is bimodal: a long tail of small ring-buffer wins and ties on the right, plus four queries on the left where the ring-buffer loses by $2\text{--}5\times$.

GROUP BY on a moderate-width key with a counter aggregate. Whenever a query plan exposes that regime to the shuffle, the microbenchmark’s prediction lands directly: $1.1\text{--}1.5\times$. The ring-buffer’s small but consistent wins across the bulk of TPC-H, and its larger wins on the right tail of ClickBench, both come from this mechanism. The ClickBench suite total swings the other way because ClickBench’s narrower shape distribution oversamples a handful of queries that put heavy per-row work on the consumer—COUNT (DISTINCT) per group, wide-row arithmetic over many simultaneous aggregates—where the channel design’s per-bucket contiguous delivery beats the ring-buffer’s shared-batch indexed-read access pattern. The design’s win is workload-shape-dependent: TPC-H’s broad variety is the case for the design choice; ClickBench’s narrow stress on adversarial shapes is the case for tuning K and consumer access patterns when those shapes dominate.

5 PRODUCTION CONSIDERATIONS

Oxla is Redpanda’s distributed analytical SQL engine. Ring-buffer shuffle has been the intra-process redistribution primitive in production for two years. We summarize where shuffle is invoked, how the ring-buffer parameters and thread counts are configured in production, how the design behaves under failure, and the implementation lessons that emerged at production scale.

5.1 Where Shuffle Is Used

Three operator families invoke ring-buffer shuffle in Oxla query plans. *Hash joins* shuffle both build and probe sides on the join

key, so matching rows converge on the same thread. *Hash aggregation* pre-shuffles rows by the grouping key for high-cardinality GROUP BY, where local-then-merge would materialize too much intermediate state. *Window functions* with PARTITION BY route rows so all rows sharing a partition key are evaluated on a single thread, allowing per-thread ordered window computation without cross-thread coordination during evaluation.

5.2 Tuning K and Batch Size

Ring capacity K is chosen based on cache topology: $K=1$ on chiplet or multi-socket hardware, $K=2$ on single-coherence-domain architectures such as the Grace-class servers, consistent with the sweep in subsection 4.4. The group capacity is set to $G=M$ (one slot per producer thread), so each producer contributes approximately one batch before the group fills, keeping the slot counter contended for at most G fetch_add operations per group.

Batch size is configured per query as a function of input cardinality and column types: smaller cardinalities use 8192 rows; large analytic columns (strings, JSON) use smaller row counts to keep total per-batch bytes within the working-set budget identified in subsection 4.4.

5.3 Thread Counts

Oxla uses $M=N$ in all shuffle instances: every core runs one producer and one consumer, co-scheduled on the same physical core. This maximizes core utilization by ensuring that when a producer stalls (e.g., on a full ring), the consumer on the same core can run, and vice versa. The ring-buffer design does not require $M=N$, but the group capacity G is set proportional to M , so asymmetric configurations change the dynamics. When $M > N$, groups fill faster (more producers contributing) while fewer consumers drain them, increasing backpressure frequency. When $M < N$, groups fill slowly and consumers frequently block waiting for new groups; the temporal convergence described in subsubsection 3.3.9 becomes stronger (more consumers converge on each group), but the reduced production rate may underutilize consumer cores. All experiments in section 4 use $M=N$; characterizing asymmetric configurations is left to future work.

5.4 Failure and Cancellation

The ring-buffer shuffle is single-use and scoped to one query plan node; there is no in-shuffle recovery. All error and cancellation paths converge on a single stop() primitive, which the query coordinator invokes whenever any thread reports an error or the query is cancelled.

Producer fault during slot write. A producer first atomically increments writes_started to claim a slot, writes its batch, then increments writes_completed. If it throws between these two atomics (e.g., from a downstream evaluation error), writes_completed never reaches the group capacity G , the full flag is never set, and the group is never published. Producers waiting to publish into the now-stuck insertion buffer also hang. The faulted thread reports the error and the coordinator calls stop(); all blocked threads then exit through the cancellation path.

Consumer cancellation. A cancelled consumer skips the per-batch decrement on consumers_left, so the group’s ring slot is never

returned to the free pool; producers eventually fill the ring and block on backpressure. There is no in-flight detection of consumer cancellation—it is always handled by query-level teardown via `stop()`.

Stop and error propagation. `stop()` sets a finish flag under the queue mutex and broadcasts on every condition variable. All blocked threads observe the flag and exit cleanly; partial in-flight data is dropped rather than flushed. Errors captured by any producer or consumer are stored as a status code on the queue and surfaced when other threads next call into the queue, avoiding cross-thread exception propagation. Consolidating recovery into a single `stop()` primitive rather than building per-mode handling keeps the hot path free of failure-detection logic, which would add cost for a behavior invoked at most once per query.

5.5 Lessons Learned at Production Scale

The ring-buffer design described in this paper is itself a response to production scaling failures. Oxla’s prior shuffle implementation was the channel-based design we use as the baseline in subsection 4.5 (one MPSC channel per output partition); it scaled acceptably up to roughly 32 cores but collapsed at 64 cores and beyond as per-channel mutex contention dominated on the customer workloads of the time. The ring-buffer redesign was driven specifically by this 64-core wall, and the implementation techniques below were necessary to make the new design hold its $O(1)$ synchronization rate at production core counts. None of these techniques were part of the initial design—each emerged from observed production behavior that smaller-scale microbenchmarks did not surface. We note that the reconstructed channel-based build in subsection 4.5, running on modern Neoverse V2 hardware, does not reproduce that historical collapse uniformly: the ring-buffer advantage at the query level is workload-shape-dependent, and the original production motivation reflects a particular point in the engine’s history rather than a permanent ranking between the two designs.

Selective producer notification. An earlier implementation woke a producer as soon as a single ring slot was freed. At high core count this caused context-switch storms when consumers ran slightly faster than producers, with throughput collapsing under scheduler overhead. Notifying only when ring occupancy dropped below half capacity restored throughput.

Pre-allocated replacement groups. The first implementation allocated each new group under the queue mutex. At 64+ producers this extended the critical section and showed up as a flat mutex-contention plateau in profiles. Donating a producer-owned replacement via pointer swap moved the allocation off the critical path.

Per-producer buffer references. An early design used a shared atomic pointer to the current insertion buffer. The atomic refcount became a cross-core cache-line hotspot; the per-producer container with private mutexes eliminated this contention.

6 CONCLUSION

Intra-process shuffle—how a query engine repartitions data across parallel execution threads within a single server—is a critical primitive whose design space has received surprisingly little dedicated study. In this paper, we framed the problem in terms of a fundamental tension between memory overhead and synchronization

contention, and systematically compared three designs that span the resulting tradeoff space: batch partitioning, channel-based streaming, and ring-buffer streaming. Our ring-buffer approach achieves amortized $O(1)$ synchronization cost per batch and $O(M)$ memory through atomic slot acquisition and fixed-size batch groups. Several implementation techniques—per-producer buffer references, pre-allocated replacement groups, and selective notification—proved essential to realizing these properties at high core counts.

Our evaluation reveals that the relative advantage of each design depends critically on cache topology and core count. On the 72-core unified LLC GraceHopper system (Arm Neoverse V2), ring-buffer streaming outperforms channel streaming by up to 44% and batch partitioning by up to 79%. At 192 cores on a dual-socket Graviton4, the advantage over channel grows to over 100% and over 300% versus batch partitioning—both because higher core counts amplify channel contention, and because the Graviton4’s higher aggregate memory bandwidth keeps the workload synchronization-bound across all tested batch sizes, whereas on Grace and EPYC memory bandwidth saturates earlier and compresses the gap. However, on chiplet architectures with many partitioned L3 caches (AMD EPYC, 12 CCDs), the ring buffer’s shared atomic counter becomes a cross-CCD bottleneck, and channel-based streaming remains competitive at medium and large batch sizes. The gap narrows as batch sizes grow and the workload shifts from synchronization-bound to memory-bandwidth-bound, though at 192 cores the ring buffer retains a substantial lead even at the largest batch sizes due to growing congestion of the channel approach.

Our channel-based baseline uses multi-producer single-consumer (MPSC) channels; an alternative using $M \times N$ dedicated single-producer single-consumer (SPSC) channels could reduce per-channel contention at the cost of higher memory, and is a natural baseline for future comparison. The end-to-end results in subsection 4.5 show that the ring-buffer’s wins and losses are predictable from query plan shape (specifically, the downstream operator’s access patterns), suggesting that a query optimizer could select the shuffle implementation per pipeline stage at plan time. Another promising direction for future work is a NUMA-aware ring-buffer variant that uses per-CCD batch groups with local atomic counters, avoiding cross-die traffic on the critical acquisition path while preserving the ring buffer’s streaming and memory-efficiency advantages on chiplet architectures.

ACKNOWLEDGMENTS

The authors would like to thank the members of the Oxla team who have helped build and maintain our query engine over the years: Wojciech Chłapek, Grzegorz Dudek, Mikołaj Gagatęk, Jacek Gałazka, Krzysztof Grabski, Mateusz Grzonka, Marcin Grzebieluch, Michał Korbel, Witold Kozłowski, Franciszek Kucza, Kajetan Litwinowicz, Michał Maślanka, Jan Mikuła, Tomasz Nowak, Wojciech Oziębły, Wojciech Padło, Konrad Piotrowski, Mariusz Rokicki, Jacek Seliga, Krzysztof Smogów, Paweł Sobótka, Vagelis Sofikitis, Paweł Szczur, Szczepan Szpilczyński, Przemek Zglinicki.

REFERENCES

- [1] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware.

- In *Proceedings of the 29th IEEE International Conference on Data Engineering*. 362–373. <https://doi.org/10.1109/ICDE.2013.6544839>
- [2] Maximilian Bandle and Jana Giceva. 2021. Database Technology for the Masses: Sub-Operators as First-Class Entities. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2483–2490. <https://doi.org/10.14778/3476249.3476296>
 - [3] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data*. 168–180. <https://doi.org/10.1145/3448016.3452831>
 - [4] Alexander Baumstark and Constantin Pohl. 2019. Lock-free Data Structures for Data Stream Processing—A Closer Look. *Datenbank-Spektrum* 19, 3 (2019), 209–218. <https://doi.org/10.1007/s13222-019-00329-4>
 - [5] Alessandro Fogli, Bo Zhao, Peter Pietzuch, and Jana Giceva. 2025. ARCAS: Adaptive Runtime System for Chiplet-Aware Scheduling. (2025). <https://doi.org/10.48550/arXiv.2503.11460> arXiv:2503.11460 [cs.DC]
 - [6] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. 102–111. <https://doi.org/10.1145/93597.98720>
 - [7] Goetz Graefe. 1994. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135. <https://doi.org/10.1109/69.273032>
 - [8] Laurens Kuiper, Mark Raasveldt, Hannes Mühleisen, and Peter Boncz. 2025. Saving Private Hash Join. *Proceedings of the VLDB Endowment* 18, 10 (2025), 2748–2761. <https://doi.org/10.14778/3742728.3742762>
 - [9] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*. 5–17. <https://doi.org/10.1145/3626246.3653368>
 - [10] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 743–754. <https://doi.org/10.1145/2588555.2610507>
 - [11] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta’s Unified Execution Engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384. <https://doi.org/10.14778/3554821.3554829>
 - [12] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 755–766. <https://doi.org/10.1145/2588555.2610522>
 - [13] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2016. High-Speed Query Processing over High-Speed Networks. *Proceedings of the VLDB Endowment* 9, 4 (2016), 228–239. <https://doi.org/10.14778/2856318.2856319>
 - [14] Raghav Sethi, Masha Basmanova, Andrii Rosa, et al. 2023. Presto: A Decade of SQL Analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25. <https://doi.org/10.1145/3589769>
 - [15] StarRocks Contributors. 2025. StarRocks: A High-Performance Analytical Database. Retrieved July 10, 2026 from <https://github.com/StarRocks/starrocks> A Linux Foundation project.
 - [16] The PostgreSQL Global Development Group. 2018. *Parallel Hash*. Retrieved July 10, 2026 from https://wiki.postgresql.org/wiki/Parallel_Hash Introduced in PostgreSQL 11.
 - [17] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. 2011. Disruptor: High Performance Alternative to Bounded Queues for Exchanging Data Between Concurrent Threads. Retrieved July 10, 2026 from <https://lmax-exchange.github.io/disruptor/disruptor.html>
 - [18] Daniel Xue and Ryan Marcus. 2025. Global Hash Tables Strike Back! An Analysis of Parallel GROUP BY Aggregation. *arXiv preprint arXiv:2505.04153* (2025). <https://doi.org/10.48550/arXiv.2505.04153>
 - [19] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220. <https://doi.org/10.14778/2735508.2735511>