Rethinking Stateful Stream Processing with RDMA

Bonaventura Del Monte[1] Steffen Zeuch[1,2] Tilmann Rabl[3] Volker Markl[1,2]
[1]Technische Universität Berlin [2]DFKI GmbH [3]HPI, Potsdam Universität
bdelmonte@tu-berlin.de steffen.zeuch@dfki.de tilmann.rabl@hpi.de volker.markl@tu-berlin.de

ABSTRACT

Remote Direct Memory Access (RDMA) hardware has bridged the gap between network and main memory speed and thus invalidated the common assumption that network is often the bottleneck in distributed data processing systems. However, high-speed networks do not provide "plug-and-play" performance (e.g., using IP-over-InfiniBand) and require a careful co-design of system and application logic. As a result, system designers need to rethink the architecture of their data management systems to benefit from RDMA acceleration.

In this paper, we focus on the acceleration of stream processing engines, which is challenged by real-time constraints and state consistency guarantees. To this end, we propose *Slash*, a novel stream processing engine that uses high-speed networks and RDMA to efficiently execute distributed streaming computations. Slash embraces a processing model suited for RDMA acceleration and scales out by omitting the expensive data re-partitioning demands of scale-out SPEs. While scale-out SPEs rely on data re-partitioning to execute a query over many nodes, Slash uses RDMA to share mutable state among nodes. Overall, Slash achieves a throughput improvement up to two orders of magnitude over existing systems deployed on an InfiniBand network. Furthermore, it is up to a factor of 22 faster than a self-developed solution that relies on RDMA-based data re-partitioning to scale out query processing.

CCS CONCEPTS

• **Information systems → Stream management**.

KEYWORDS

stream management, distributed and parallel databases

ACM Reference Format:
Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517826

1 INTRODUCTION

Over the last decade, the advancement in data center networking technology has bridged the gap between network and main memory data rates [9, 25]. It is possible today to purchase or rent servers that offer supercomputer-grade network bandwidths [45]. For instance, a high-speed Network Interface Controller (NIC) supports up to 25 GB/s as network throughput and 600 ns latency per port [43], while modern switches support up to 40 TB/s as overall network throughput [44]. Double Data Rate 4 (DDR4) modules support up to 19.2 GB/s per channel and 13 ns CAS latency, while main memory bandwidth reaches up to 204.8 GB/s [3]. This improvement is due to Remote Direct Memory Access (RDMA): a feature of high-speed networks that enables high throughput data transfer with microsecond latency. Thus, the common assumption that network is often the bottleneck in distributed settings no longer holds [60].

Research has shown that RDMA hardware does not provide a "plug-and-play" performance gain to existing data management systems [9]. Consequently, it is necessary to revise their architecture to use full bandwidth [9]. Recent research has proposed a number of architecture revisions to accelerate OLAP [9, 40], OLTP [69], indexing [74], and key-value stores [19, 31] using RDMA in rack-scale deployments. In this paper, we make the case that Stream Processing Engines (SPEs) also require architectural changes to truly benefit from RDMA hardware. To this end, we show that current scale-out SPEs are not ready for RDMA acceleration and existing RDMA solutions do not fit the stream processing paradigm. Thus, we propose an SPE architecture that natively integrates with RDMA to efficiently ingest and process data in rack-scale deployments.

Current SPEs, e.g., Apache Flink [11], Storm [58], TimelyDataflow [47], and Spark [68], cannot fully benefit from RDMA hardware. Their design choices fundamentally prevent them from processing data at full data-center network speed for the following reasons. First, *RDMA-unfriendliness,* current SPEs rely on socket-based networking, e.g., TCP/IP, to ingest and exchange data streams. Even though socket-based networking runs on RDMA hardware, it cannot fully exploit its potential, e.g., using IP-over-InfiniBand (IPoIB) [9]. Second, *costly message-passing*, current SPEs rely on message-passing to process data following a Map/Reduce-like paradigm [16]. This results in a performance issue and in an inefficient execution induced by sub-optimal data and code locality [70, 71]. In particular, message passing induces expensive queue-based synchronization among network and data processing threads [30]. Furthermore, Map/Reduce-like paradigms are network-bound on relatively slow socket-based connections, when considering data-intensive workloads. Yet, they become compute bound in the presence of fast networks [9, 60]. As a result, they do not benefit from the data rate of a high-speed network. Finally, *costly scale-out execution*, current scale-out SPEs rely on *operator fission* [27] to achieve data-parallel computations. This enables each SPE executor to process a disjoint partition of the stream and manage local state. However, fission involves continuous data re-partitioning, which is expensive [70].

Furthermore, previous RDMA solutions for data-intensive systems do not solve the above problems. An SPE requires to run stateful analytics on in-flight records and perform point updates and range

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517826

scans on operator state. However, RDMA-accelerated OLAP systems speed-up batch analytics on immutable datasets [7, 20, 40, 53]. RDMA-based key-value stores are design for transactional workloads comprising point lookups and insertions [19, 31]. As a result, the data-access patterns and processing model of SPEs need dedicated solutions for RDMA-acceleration.

To enable stateful stream processing at full network bandwidth with very low latency, we propose Slash, our RDMA-accelerated SPE for rack-scale deployments. We design a new architecture to enable a processing model that omits data re-partitioning and applies stateful query logic on ingested streams. Slash comprises the following building blocks: the RDMA Channel, the stateful query executor, and the Slash State Backend (SSB). First, we design an RDMA-friendly protocol to support streaming among nodes via dedicated RDMA data channels. This enables Slash to perform data ingestion and data exchange among nodes at full RDMA network speed, by leveraging the aggregated bandwidth of all NICs. Second, we devise a stateful executor that omits message passing and runs queries following late merge technique [70]. Finally, we replace re-partitioning with the SSB that enables consistent state sharing across distributed nodes. This enables multiple nodes to concurrently update the same key-value pair of the state (for instance, a group of a windowed aggregation). To ensure consistency, we introduce an epoch-based protocol to lazily synchronize state updates using RDMA.

Slash executors scale out computation by eagerly applying stateful operators on data stream to compute partial state. Executors store partial state into the SSB, which ensures a consistent view of the state for all Slash executors. Our evaluation on common streaming workloads shows that Slash outperforms baseline approaches based on data re-partitioning and is skew-agnostic. In particular, we compare Slash against a scale-out SPE (Apache Flink) on an IPoIB network, a scale-up SPE called LightSaber, and a self-developed straw-man solution called RDMA UpPar, which scales out query execution via RDMA-based data re-partitioning. Slash achieves up to 25x and 11.6x higher throughput than RDMA UpPar and LightSaber, respectively. Furthermore, Slash outperforms Flink by achieving an order of magnitude higher throughput. In sum, this shows that RDMA alone cannot achieve peak performance without redesigning the SPE internals.

In this paper, we make the following contributions.

- To natively integrate high-speed RDMA networks, we propose Slash, a novel RDMA accelerated SPE.
- We design a stateful query executor to make Slash scale-out data stream processing over an RDMA network.
- We define an RDMA streaming protocol for Slash to transfer data at line rate using the RDMA channel.
- We architect the SSB that enables a distributed consistent state over RDMA interconnects.
- We validate Slash's design on common streaming benchmarks on a high-end RDMA cluster and show up to 25x throughput improvement over our strongest baseline.

We structure this paper as follows. In Sec. 2, we present background concepts about RDMA and data stream processing. In Sec. 3, we make the case of RDMA acceleration for an SPE and list challenges and opportunities. In Sec. 4, we present the system architecture of Slash and provide an overview of each component. After that, we describe the stateful executor (Sec. 5), the RDMA Channel (Sec. 6), and the SSB (Sec. 7). Afterwards, we conduct an extensive evaluation of Slash in Sec. 8. We describe related works in the realm of SPEs and RDMA-enabled database systems in Sec. 9. Finally, we summarize the findings of this paper and discuss ideas for future work in Sec. 10.

2 BACKGROUND

In this section, we provide the background for our paper. We describe RDMA in Sec. 2.1 and provide an overview of current approaches to stream processing in Sec. 2.2.

2.1 Remote Direct Memory Access

RDMA is a communication stack provided by Infiniband (IB), RoCE (RDMA over Converged Ethernet), and iWarp (Internet Wide Area RDMA Protocol) networks [32]. RDMA enables access to the main memory of a remote node with minimal involvement of the remote CPU. As a result, RDMA achieves high bandwidth (up to 200 Gbps per port [43]) and low latency (up to 2µs per round-trip [32]). RDMA offers bidirectional data transfer via *zero-copy*, which bypasses the kernel network stack. In contrast, socket-based protocols, such as TCP, involve costly system calls and data copies between user- and kernel-space [9]. RDMA-capable NICs also support socket-based communication via IP-over-InfiniBand (IPoIB). However, this approach results in lower efficiency [9]. RDMA provides two APIs (so-called *verbs*) for communication: one-sided and two-sided verbs APIs [32]. Besides, RDMA provides *reliable*, *unreliable*, and *datagram* connections. A reliable connection enables one-sided verbs and in-order packet delivery, while unreliable and datagram connections may drop packets. With two-sided verbs (Send-Recv), sender and receiver are actively involved in the communication. The receiver polls for incoming message, which requires CPU involvement. In contrast, one-sided verbs involves one active sender and one passive receiver (RDMA WRITE) or a passive sender and an active receiver (RDMA READ). They enable more efficient data transfer, but need synchronization to detect inbound messages.

RDMA provides two major benefits: it 1) enables fast data transfer and 2) shares memory areas among nodes [19]. However, RDMA-enabled systems need careful design, as RDMA does not offer coherence between local and remote memory. Instead, this is offloaded to the application. Besides, coherence among NIC memory, main memory, and the CPU is vendor-dependent [9]. The choice of verbs and parameters, such as message size, is application-sensitive and requires careful tuning [32].

2.2 Stateful Stream Processing Engines

Recent SPEs use either scale-up or scale-out processing model. Scale-up SPEs focus on single-node efficiency, whereas scale-out SPEs target cluster scalability. Scale-up SPEs, such as LightSaber [56], BriskStream [72], and Grizzly [23], target single-node deployments with multi-socket, multi-core CPUs. Scale-out SPEs, such as Flink [11], Storm [58], Spark Streaming [68], Millwheel [4], Google Dataflow [5], and TimelyDataflow [47], parallelize queries on shared-nothing architectures. Scale-up and scale-out SPEs assume common data and query models, yet they execute queries differently. We summarize their data, query, and processing models in the following.

Data and query model. We follow the definitions introduced by Fernandez et al. [12] and assume a data stream to consist of an

immutable, unbounded set of records. A record contains a timestamp t, a primary key k, and a set of attributes. Timestamp are strict monotonically increasing and used for windowing related operations as well as progress tracking. Streaming queries are modelled as directed acyclic graphs with stateful operators as vertices and data flows as edges. The output of a streaming operator depends on content, timestamp or arrival order of input records, and its intermediate state. In general, an operator must output no result at a timestamp t that is computed using records bearing timestamps greater than t.

Scale-up execution. Scale-up SPEs rely on task-based parallelization, compilation-based operator fusion [27], and late merge [70] to fully utilize available hardware resources. Logical operators are fused together and compiled to machine code, which the SPE executes on inbound data buffers using task-based parallelization. Tasks may concurrently update a global operator state or eagerly update local state, which the SPE eventually merges and ensure its consistency.

Scale-out execution. Scale-out SPEs use operator-to-thread parallelism and data re-partitioning [27] to scale out. Each logical operator consists of p physical operators, which the SPE runs in parallel on a cluster of nodes. Scale-out SPEs re-partition input streams so that each physical operators applies stateful transformations on a disjoint partition of the data. This enables consistent stateful computations via local mutable state [10]. Parallel instances receive records from upstream operators via in-memory or network-based *data channels* following an exchange pattern.

3 THE CASE FOR RDMA-ACCELERATED STATEFUL STREAM PROCESSING

In this section, we make the case for RDMA-based acceleration of stateful stream processing workloads. To this end, we analyze options to co-design an SPE with RDMA networks (Sec. 3.1) and derive design challenges to be tackled (Sec. 3.2). Afterwards, driven by our analysis, we propose the system architecture of Slash (Sec. 4).

3.1 RDMA Integration

Previous research proposes three general approaches for the integration of RDMA into a general-purpose data management system [9]. In the following, we analyze their applicability to an SPE and the implication behind these design decisions.

Plug-and-play integration. In this approach, the deployment of a shared-nothing system occurs on an IPoIB network. Previous research has shown that it does not necessarily result in performance improvements [9]. In particular, IPoIB does not saturate network bandwidth and introduces CPU overhead for small messages [9]. In our evaluation, we show that query execution of current SPEs improves only slightly using IPoIB.

Lightweight integration. This approach involves the replacement of socket-based networking with RDMA verbs. Although it benefits from high network bandwidth, it still suffers from bottlenecks that are present in the original design [9, 32, 70]. We validate this claim by building a straw-man solution that implements the lightweight approach. In particular, we implement and evaluate a data re-partitioning component that uses RDMA QPs instead of sockets [40]. Note that several SPEs, such as Apache Flink and TimelyDataflow, leverage data re-partitioning to parallelize operators and thus scale-out computation. We refer to this approach in the remainder of this paper as *RDMA UpPar*. Note that the performance regression induced by lightweight integration does not depend on the underlying runtime or language. Thus, we implement RDMA UpPar in C++ to omit managed runtime overhead and show that the performance regression is due to the overall SPE design.

Native integration. Previous research indicates that the most-effective option for the integration is to co-design software components with RDMA [9, 32, 69, 74]. For a data management system, this approach involves a re-design of its internals, e.g., storage management and query processing, to remove the network bottleneck. Notably, the data re-partitioning component of a scale-out SPE is network bound in the presence of a high data rates and slow networks [60]. Furthermore, recent works have shown that data re-partitioning is responsible for performance regressions in scale-out SPEs due to its sub-optimal CPU utilization in the presence of high bandwidths [70, 71]. This suggests that an SPE must re-think its scale-out computational model and evaluate alternatives to data re-partitioning, to fully benefit from RDMA acceleration. We refer to this approach in the remainder of this paper as *Slash*.

In this paper, we focus on the native approach and discuss the necessary architectural and algorithmic changes when designing an RDMA-accelerated SPE. In addition, we assess the performance of the native solution against plug-and-play and lightweight integrations.

3.2 Design Challenges

In the previous section, we make the case for native RDMA integration, which promises peak performance, but requires changes to a scale-out SPE. In this section, we analyze the design challenges and the implications that come with native RDMA acceleration.

C1: Efficient Streaming Computations. Besides high-throughput data transfer, RDMA-based systems enable a highly-efficient decoupled processing model: storage and compute nodes access each other's memory at byte-level granularity [9, 31, 40, 69, 74]. In this model, costly data re-partitioning for data-parallel processing is not necessary, as each node can access data from any remote memory location. As a result, we identify as first design challenge to support an efficient processing model for distributed streaming computations that profits from memory access at byte-level granularity. This involves replacing re-partitioning strategy with a performance-friendly, RDMA-based processing strategy.

C2: Efficient Data Transfer. In an SPE, data channels must enable efficient data transfer among operators over the network. However, RDMA provides several network transfer capabilities, which induce many design options and trade-offs between throughput and latency. For instance, one-sided verbs achieve lower network latency than two-sided verbs, especially on the most recent NICs [19, 30, 48]. An RDMA READ involves a full network round-trip and has thus higher latency than an RDMA WRITE [32]. Furthermore, techniques, such as huge-pages, pipelining, header-only messages, and selective signaling, further improve performance [9, 32]. To the best of our knowledge, there is no comprehensive analysis of RDMA capabilities on stream processing workloads [65, 70]. As a result, we identify as second design challenge the selection of RDMA capabilities to achieve high-throughput stream processing with low latency.

C3: Consistent Stateful Computation. SPEs needs to ensure consistent stateful computation. Thus, an RDMA-accelerated SPE

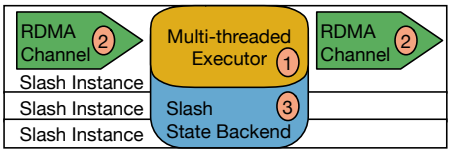


Figure 1: The Architecture of Slash.

must consistently manage state, while processing incoming records. This involves keeping track of the distributed computation, while ensuring exactly-once state updates. As a result, we identify as third design challenge to achieve consistency guarantees for stateful streaming operators that access shared data structures using RDMA.

In sum, there is no system that fully solves the above design challenges, to enable stateful stream processing at RDMA speed. Therefore, we propose Slash in the next section as a solution to bridge the gap between RDMA acceleration and stream processing.

4 SYSTEM DESIGN

The architecture of Slash comprises three components to enable robust stream processing: the *stateful executor* (1), the *RDMA channel* (2), and the *Slash State Backend* (3). Fig. 1 shows that each node executes an instance (a process) of a Slash stateful executor, which reads and writes stream records using RDMA channels, applies operator logic, and stores intermediate state into the state backend.

Stateful executor. A guiding design principle for the stateful executor of Slash (Sec. 5) is to make the common case fast when leveraging RDMA acceleration. To this end, Slash executors follow a relaxed processing model based on lazy merging of eagerly computed partial states. This is similar to the execution model of scale-up SPEs, which is based on late merge. However, Slash performs merging at cluster level using RDMA. To avoid costly data repartitioning, Slash executor eagerly compute partial state in parallel on physical data flows of a stream Furthermore, Slash executors lazily merge partial, distributed state and output consistent result using our RDMA-based components: the SSB and the RDMA channels. We use RDMA acceleration to design efficient distributed algorithms and data structures that enable fast, coherent memory access at byte-level granularity.

RDMA Channel. Slash RDMA channels (Sec. 6) are data channels that enable sending and receiving records at full line rate with sub-millisecond latency, via an RDMA-shared circular queue. Sender and receiver read/write from/to the queue using RDMA semantics. In contrast to socket-based RDMA channels, our RDMA channels enable higher throughput transfer and zero-copy semantics. We use RDMA channels to implement data repartitioning in RDMA UpPar as well as communication primitive for Slash.

Slash State Backend. The SSB is a distributed state backend (Sec. 7) that maintains operator state across the aggregated memory of multiple nodes. We design our distributed state backend for common stream processing use-cases: update-intensive workloads, and quick triggering and post-processing of in-flight windows [13]. In contrast to traditional approaches, which allow for local mutable state co-partitioned with input stream, our state backend enables distributed mutable state using RDMA. As a result, Slash executors can consistently and concurrently update key-value pairs of the distributed state, which in turn enables lazy execution.

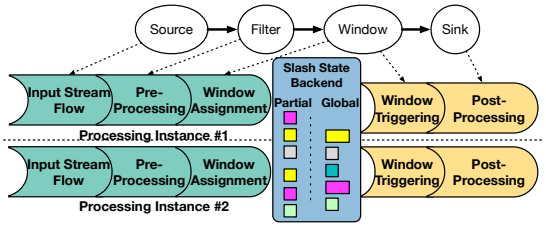


Figure 2: Slash translates a query comprising filter and a time-based windowing operators into pipelines (green and violet shapes). Slash executes each pipeline on its instances. Left pipelines update the distributed window state. Right pipelines trigger the window and read the distributed state.

Overall, the Slash stateful executor tackles **C1** and enables highly efficient yet consistent processing model using RDMA-based building blocks. RDMA channels enable fast network transfer and address **C2**. Finally, Slash's distributed state backend solves **C3** via consistent state management, by a coherence protocol tailored to RDMA.

5 SLASH STATEFUL EXECUTOR

In this section, we present the Slash processing model (Sec. 5.1) and discuss execution-related aspects: supported stream operators (Sec. 5.2) and parallel execution (Sec. 5.3).

5.1 Processing Model

Slash's stateful executor applies data-parallel transformations to physically partitioned data flows of a data stream. Thus, Slash runs in parallel multiple instances of the same operator across the nodes of a cluster. Slash does not assume that data flows are logically partitioned on the primary key, thus, a key may appear in multiple data flows.

Fig. 2 depicts the stateful processing model of Slash. Slash supports stateless and stateful continuous operators as operator pipelines, in line with recent scale-up approaches to stream processing [23, 36, 56, 70]. Each pipeline terminates with a soft pipeline breaker, such as a window trigger operator [23, 56], as shown in Fig. 2. However, Slash extends the late-merge scale-up approach to support scaling out. To this end, Slash shares operator state among a set of nodes (via RDMA) and omits data re-partitioning. Slash performs no data re-partitioning, e.g., no hash-based re-partitioning. Thus, operators immediately update the shared mutable state, which is kept consistent across Slash instances by the SSB. Furthermore, the degree of parallelism of a pipeline is bound by the number of its input data flows.

State and computation consistency involves two properties. *P1)* Slash must not output any result at timestamp t that is computed using records bearing timestamps greater than t. *P2)* A distributed computation over a data stream D in Slash must result after lazy merging in the same output that a sequential computation would produce processing D. We discuss below how we achieve these properties.

Progress Tracking. Scale-out SPEs rely on re-partitioning and in-band or out-of-band progress tracking to trigger event-time windows on a key basis [10, 47]. Slash omits data re-partitioning, which introduces a challenge in the progress tracking of the overall distributed computation. In fact, instances of a scale-out SPE that omits data re-partitioning must coordinate to detect window termination and merge partial windows. To satisfy *P1*, Slash relies on *vector*

clocks [39]. Every Slash executor e tracks the lowest watermark $l_{e,w}$ for each window w: the greatest event-time timestamp of the records that update the window. Upon lazy merging, Slash executors share among each other their low watermarks via RDMA to build a vector clock $V_w = \{l_{1,w},\ldots,l_{m,w}\}$, where m is the number of Slash executors. Through the vector clock, executors observe each other's progress and coordinate window triggering in event-time. Triggering occurs when a Slash executor determines a timestamp entry in the vector clock to be greater than the end timestamp of a pending window.

Consistency. Slash ensures computation consistency using an *epoch-based coherence protocol* [14, 22] and *conflict-free replicated data types* (CRDTs) [54]. While we discuss our coherence protocol in Sec. 7.2.2, we describe CRDT-related aspects in the following.

The state backend represents the partial state of a window as a CRDT. As a result, the window bucket (or the window slice) in Slash need to be represented as a CRDT. CRDTs enable merging partial state while guaranteeing consistent results as follows. A CRDT for a non-holistic window computation, such as an aggregation, relies on commutativity of the aggregation. A CRDT for a holistic window computation, such as a join, relies on join-semilattice and delta updates [64]. For instance, the CRDT for a sum-based window stores the partial sums of each parallel summation. Upon merging, the CRDT computes the final result as the sum of all partial values.

5.2 Stateful Operators

Slash provides two common stateful operators: hash-based aggregations and hash-based joins on event-time windows. Slash assumes windowing techniques that rely on window buckets [38] or general slicing [59], with the following modifications.

Windowing. Slash executes windowed operators as part of an operator pipeline that consists of a window bucket (or slice) assigner and a window trigger. The window assigner determines the bucket (or slice) to which a record belongs and updates it accordingly. In Slash, a window assigner does not assume pre-partitioned data, but offloads state consistency to the state backend. The window trigger outputs the window content based on event-time. In Slash, a window trigger requires a vector clock to evaluate the triggering condition and relies on the state backend to provide consistent state.

Windowed Aggregation. Slash provides hash-based aggregation that follows the *late merge* approach [70]. Each Slash executor thread eagerly computes its own local state, i.e., a partial hash-based aggregate for each in-flight window. In Slash, we scale-out *late merge* using RDMA acceleration and distributed CRDT-based aggregations.

Windowed Join. Slash offers a windowed streaming join based on a hash join. For every in-flight window, Slash eagerly builds a hash-table for the two streams based on the key and event-time of incoming records. When a window terminates, Slash probes the hash-tables to output per-key pairwise combinations of stored records. Slash ensures state consistency as the underlying distributed hash-table lazily concatenates all partial values with the same key.

5.3 Parallel Execution

The Slash executor parallelizes operators using worker threads across a number of nodes. For each physical operator, Slash assigns its RDMA channels to a worker thread. Each thread polls each channel for incoming data buffers to process. Slash uses an event-driven

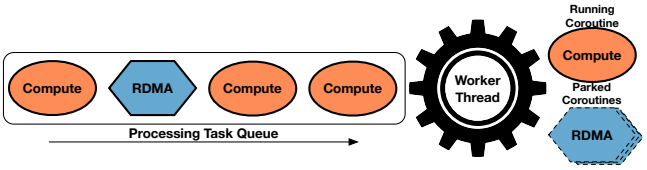


Figure 3: The coroutine-based event-driven scheduler of Slash.

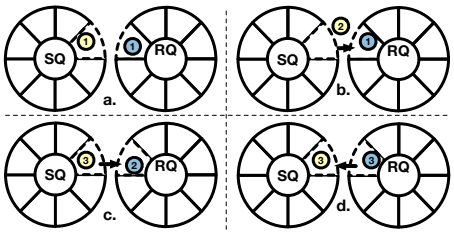


Figure 4: The protocol behind an RDMA channel: a.) the sender (SQ) acquires the dotted buffer, b.) begins the transfer, c.) waits for credit, and d.) acknowledge the completion on the receiver (RQ).

scheduler based on coroutines and a push-based processing model (Fig. 3). Coroutines are lightweight threads that enable cooperative multitasking [46]. The Slash scheduler interleaves compute coroutines with RDMA coroutines. RDMA coroutines execute RDMA-related tasks, such as polling buffers. Compute coroutines perform push-based processing on polled buffers. In the case of an empty RDMA channel, the scheduler parks the related RDMA coroutine and executes available ready compute tasks. Thus, empty RDMA channels do not stall the execution of pending compute coroutines.

We select coroutines, as they enable context switch with 10-20 ns of latency [31] and the interleaving of compute and I/O-tasks [26]. Current SPEs perform network-related operation on dedicated threads [10, 23, 56, 72]. However, performing RDMA operations on dedicated threads needs synchronization with processing threads, which wastes up to 400 cycles on common x86 CPUs [30]. The Slash scheduler hides network latency by executing compute tasks while RDMA packets are in-flight. This enables fine-grained control on RDMA and compute operations, which results in higher CPU efficiency.

Overall, Slash's processing model is inspired by LightSaber [56] and Grizzly [23], as it relies on task-based parallelism and late merging of partial state. However, Slash differs from them as follows. First, it extends the processing model of the above systems to target scale-out execution. To this end, it introduces eager, distributed computation of partial results and their lazy merge through RDMA. Second, it extends task-based parallelism using coroutines to interleave RDMA-related operations with processing tasks. Finally, it is agnostic to the execution strategy, as it supports compilation-based and interpretation-based strategies. Slash's worker threads have their own queues of coroutines to execute, whereas LightSaber and Grizzly share a single task-queue among their worker threads and focus on compilation-based execution.

6 RDMA FOR DATA STREAMING

In this section, we present our RDMA-based transfer protocol that enables Slash to stream data with high throughput and low latency.

We provide an overview of our protocol (Sec. 6.1), describe its phases (Sec. 6.2), and discuss details of our RDMA-channels (Sec. 6.3).

6.1 RDMA Data Transfer Protocol

Our protocol determines the record exchange between a producer and a consumer via an RDMA channel. It defines the pipelined access to an RDMA-capable circular queue that guarantees FIFO delivery of records. Our protocol is consumer-driven: the consumer (based on its processing capabilities) requires the producer to adjust its sending rate to avoid back-pressure. Furthermore, it defines a coherence model to access the queue: a producer cannot overwrite unread buffers in the memory of the consumer. To this end, we schedule writes/reads to/from the queue via *credit-based flow control* (CFC) [35]. CFC is widely used in RDMA protocols to ensure coherence of RDMA-based data structures [30, 31]. In this work, we use CFC to ensure FIFO delivery of records and avoid back-pressure.

6.2 Phases of the Protocol

Our protocol has two phases: a *setup* phase and a *transfer* phase. The setup phase defines the initialization of an RDMA channel, while the transfer phase defines the handling of data transfer at runtime.

Setup phase. This phase consists of 1) the initialization of a circular queue of RDMA-capable memory on the sender and receiver side and 2) setting up of a reliable RDMA connection between the two parties. The circular queue has c slots, which is the initial number of credits. Each slot is an RDMA-capable, fixed-size buffer, which are allocated in this phase. The value of c is fixed throughout query execution, as its selection is hardware-sensitive and determines the level of pipelining, which depends on the NIC capabilities [32].

Transfer phase. Fig. 4 shows the steps of the transfer phase. In this phase, a producer is permitted to: (1) acquire the next buffer from the circular queue and write to it, (2) post a write request for a buffer to an RDMA NIC, and (3) poll for credit from the consumer. A consumer is permitted to: (1) poll for an incoming data buffer, (2) mark the buffer for processing, and (3) send a credit to the producer.

Using pipelining, a producer that follows our protocol can send up to c buffers before it must wait for credit [30]. In particular, write requests do not overtake each other and result in a data buffer to be readable on the consumer upon their completion. To guarantee that a producer does not overwrite unread data, the consumer must notify the producer about writables buffer in its queue.

Properties. Based on the operations above, our protocol ensures three invariants. First, a producer decreases its number of credits by one after a write request. Second, a consumer transfers a credit to the producer after processing a buffer. This notifies the producer that the buffer is writable. Finally, a producer with no credit cannot pick buffers from the queue. As a result, it cannot push further write requests and has to wait for new credit from the receiver. Overall, producers and consumers that follow our protocol are guaranteed to consistently exchange records in FIFO order, at a self-adjusting data rate.

6.3 RDMA Channels

An RDMA channel consists of a QP, a circular queue, and a credit counter. RDMA channels enable zero-copy transmission and reception of buffers using RDMA. Fundamental choices behind this component are a flat memory layout of the circular queue and a push-based transfer model via RDMA WRITEs. The choices influence design regarding data structure, RDMA verbs, and message layout.

Data structure. The circular queue consists of an RDMA-capable memory area of $c \times m$ bytes, with c as number of credits and m as the size of a single buffer. As a result, buffers are contiguously stored, which induces a flat memory layout. Each buffer comprises of contiguous payload and metadata, such as a flag for polling. A flat layout is beneficial for three reasons. First, it avoids expensive pointer chasing operations [70]. Second, contiguously-stored payload and metadata enable data transfer via a single RDMA request, whereas decoupled data region and metadata would require two RDMA requests. Finally, it allows for cacheline alignment and huge-pages allocation, which reduce CPU cache misses and NIC TLB misses [32].

RDMA verbs. We select a push-based transfer approach using RDMA WRITEs instead of RDMA READs for the following reasons. First, an RDMA READ involves a round-trip per message, which leads to higher latency and CPU utilization [31]. In contrast, an RDMA WRITE needs a single trip per message. Second, RDMA WRITEs enable push-based transfer: the producer writes into the memory of the consumer, which polls its local memory. In contrast, RDMA READs allow for pull-based transfers: the consumer continuously reads the producer's remote memory until the requested data is available. Thus, an RDMA pull-based model induces extra network traffic, as polling occurs over RDMA. Overall, our push-based approach requires only one network access per message and efficiently polls local memory.

Message layout. Slash transfers buffers as messages via RDMA WRITEs. This needs a detection mechanism of inbound messages at the receiver. To this end, we divide the buffer into a data region for the payload and a footer for metadata. We use the final byte of the footer for polling, which has two benefits compared to polling on the header. Polling on the footer guarantees full data transfer, as RDMA WRITE transfers buffers from lower to higher memory addresses. Polling on the header does not ensure full reception of a buffer, as the transfer might still be in progress. The consumer can safely process the data region when it detects the change on the last byte.

7 SLASH STATE BACKEND

The Slash State Backend (SSB) is a concurrent key-value store for in-memory operator state. It provides state management techniques to build global operator state shared across multiple nodes using RDMA. In this section, we describe our approach to RDMA-accelerated state management (Sec. 7.1) and the components of our SSB (Sec. 7.2).

7.1 RDMA-accelerated State Management

In this section, we describe our approach to accelerate state management of a scale-out SPE using RDMA. Our state management leverages RDMA to enable the nodes of an SPE to consistently read and write each other's state with high bandwidth and low latency. To this end, we first present requirements for a state management component (Sec. 7.1.1) and then discuss the design of the SSB (Sec. 7.1.2).

7.1.1 Requirements. State management defines how operators access and modify state. To speed-up state access via RDMA and enable running operators to concurrently modify shared state, we analyze requirements on workloads and RDMA semantics.

Workload. A state backend for SPEs has three strict design requirements [13]. First, a state backend must support update-intensive

workloads as stateful operators concurrently perform point updates of the state on a record basis. Point updates consists of *read-modify-write* (RMW) operations that change a key-value pair based on the previous value and the record content. Second, a state backend must enable efficient scans of its content, for example, to timely trigger and post-process a window. Finally, it must allow for arbitrary state sizes, which may exceed single-node memory boundaries.

RDMA semantics. A state backend must efficiently handle concurrent updates among nodes. Nodes may concurrently update the same key-value pair and thus the state backend must ensure consistent update semantics. This is a two-fold challenge: 1) it needs a coherence protocol among nodes to achieve consistency and 2) it involves careful design of memory access patterns from the local CPU as well as remote RDMA NICs, as they are not coherent.

7.1.2 Design of the Slash State Backend. Based on the above requirements, we consider the following design choices to achieve RDMA acceleration for consistent state management. As discussed in Sec. 5, Slash does not perform re-partitioning but uses shared mutable state for stateful operators. Shared mutable state enables concurrent reads and writes on the same key-value pair. However, this requires expensive coordination among readers and writers, which we avoid with our SSB as we show in the following.

Partial State. SSB maintains on every executor a partial state for each locally-running operator. Operators eagerly update partial state locally, which is in line with common scale-up principles [23, 56, 70]. With our approach, the common operation is the per-record update of partial state, which neither induces queueing among operators nor suffers from skew-sensitive hash partitioning [70]. In contrast, the common operation for traditional SPEs is the per-record partitioning and the update of co-partitioned state.

State Maintenance. The SSB divides the key-value space into disjoint partitions and assigns each partition to an executor. Each executor is the *leader* for only one partition, which we call *primary* partition. Slash does not perform re-partitioning thus each executor potentially maintains state that belong to the partition of another leader. Thus, an executor stores a *fragment* of each remote primary partition and becomes *helper* of their respective leader executors. For each partition, its leader and helpers synchronize their content based on the following coherence protocol. This leads to a space amplification for key-value pairs proportional to the number of nodes. However, the value size depends on the semantics of the window computation. Thus, non-holistic window results in an aggregate per node, whereas holistic windows results in disjoint sets of values per node.

Coherence Protocol. The SSB lazily synchronizes partitions between leader and helper executors using an epoch-based coherence protocol. An epoch is a time span between two synchronization points. At the end of an epoch, helper nodes of a partition send its content to its leader node, which merges key-value pairs. Furthermore, this enables arbitrary sizes of state as it is scattered across aggregated memory of the cluster and is materialized only at the end of an epoch.

Update conflicts. The SSB needs to support concurrent updates of the same key-value pair from multiple executor. To this end, Slash relies on CRDT to merge conflicting key-value pairs (see Sec. 5.2). Our SSB enables a processing model that omits data re-partitioning and makes common case operation fast. The common case operation of Slash is the eager computation of partial state, while current SPEs partition records prior to update state.

7.2 Components of Slash State Backend

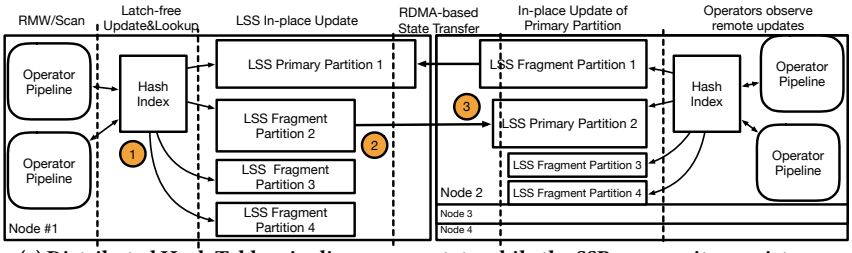
The SSB is our state storage layer, which provides distributed hash tables to consistently manage operator state. In the following, we describe the techniques behind our SSB: our distributed hash table (Sec. 7.2.1) and our epoch-based coherence protocol (Sec. 7.2.2).

7.2.1 Distributed Hash Table. The SSB uses a distributed hash table based on separate chaining and log-structuring. The hash table consists of a *hash index* and a *log-structured storage* (LSS) [37, 49, 50] of key-value pairs for each partition. We show the architecture of our distributed hash table in Fig. 5a. In the following, we provide the rationale behind our design and describe the LSS.

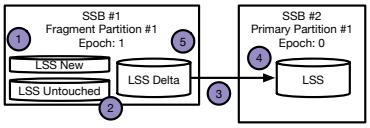
Rationale. An update of a key-value pair requires a lookup in the hash index to find the position of the pair in one of the LSSs. Decoupling indexing from storage has two advantages over techniques such as open addressing [52]. First, it enables one index for each partition that points to multiple LSSs (1). Second, log-structuring induces temporal locality for the updates, i.e., frequently accessed key-value pairs are in the same portion of the log. This enables quick detection of changes in the LSS so that a helper can send them to a leader using RDMA without involving pointer chasing. As a result, a helper moves to a leader only the last modified pairs to avoid redundant network transfer. In contrast, open addressing induces a scattered memory layout that requires a full scan to detect update. Finally, we do not assume a particular design for the hash index. Instead, we use the hash index of FASTER [13] in the remainder of this paper.

Log-structured storage. Our LSS is an RDMA-capable circular buffer that stores dense key-value pairs. We partially follow the design of FASTER [13] and consider its in-memory capabilities. However, we extend its design to enable RDMA acceleration in a distributed setting but skip disk spilling, as it is out of our scope. The LSS acts as a hybrid log that enables concurrent append and in-place update operations on key-value pairs (2). We extend FASTER's design as follows. We rethink its design for distributed execution and introduce leader and helper nodes. Helper nodes transfer delta changes to leader executors in chunks using dedicated RDMA channels. Slash interleaves reception and merging of delta changes with query processing. Furthermore, we enable the circular buffer to adaptively resize as partitions vary in size over time due to frequency shifts in key distributions. Consequently, our state backend adapts to shifts in workloads size. Overall, our state backend enables incremental state synchronization via our epoch-based coherence protocol (3).

7.2.2 Epoch-based coherence protocol. Slash slices infinite streams into finite chunks of records based on epochs. Epoch-based concurrent systems safely execute global operations at epoch boundaries. Many systems rely on epoch-based synchronization for diverse goals, such as checkpointing [10, 13]. In Slash, we extend the concept of epochs to merge distributed shared partitions lazily (stored on helper nodes) into their respective primary partition. The SSB follows an epoch-based coherence protocol that enables nodes to synchronize state and ensures consistency. In the following, we present the setup and synchronization phases as well as the properties of our protocol.



(a) Distributed Hash Table: pipelines access state while the SSB ensures its consistency.



(b) The Epoch Protocol: SSB#1 sends the delta of the fragment partition #1 at epoch 1 to SSB#2 over RDMA.

Figure 5: The Slash State Backend: an overview of its architecture and a detail of the epoch protocol for state consistency.

Setup Phase. Consider a Slash deployment of n Slash Executors and n primary partitions, where n is the number of nodes. Each partition has an epoch counter to version its content. In the setup phase, each leader executor connects to all possible executors. Overall, Slash creates n^2 RDMA channels for state synchronization during this process. Note that our RDMA channels for state transfer use the LSS memory instead of dedicated circular queues to avoid data copies. Slash assumes epoch duration to be agnostic to window size. However, a Slash instance signals the ahead-of-time termination of an epoch upon window triggering.

Synchronization Phase. We assume that each stateful operator receives records as well as tokens that notify system-wide events, such as punctuations. This is a common technique used in several SPEs to make operators perform operations, e.g., trigger windows or take a state snapshot [10]. In Slash, the arrival of a synchronization token to an operator makes helpers perform the following steps, which we show in Fig. 5b.

1. Increment the epoch counter for each shared partition.
2. Identify the portion of the circular buffer that contains the latest changes in the LSS of each modified partition. Prior to the transfer, mark the changes as read-only to prevent inconsistency between DMA reads and CPU writes.
3. Transfer the changes in the circular buffer via RDMA channels.
4. Incrementally merge the transferred content in the local LSS.
5. After the transfer, invalidate the content of the transferred portion of the storage so that it can serve further RMW operations.

Properties. In response to the above steps, leader executors lazily receive updates for the state they manage. We piggyback vector clock updates with state updates so that a leader executor can observe the progress of helpers. A leader node can trigger a per-key window at timestamp t only if the vector clock guarantees the occurrence of no record nor state update that bears an event-time timestamp smaller than t. Note that a local epoch counter induces an order on the arrived updates such that state updates cannot skip each other. Furthermore, discarding transferred content is safe, as RMW operations restart from a zero value.

Distributed instances of the SSB that follow this protocol are guaranteed to converge to a consistent state at the end of each epoch. Window operators benefit from this approach as the triggering of a window occurs at the end of an epoch. As a result, the window state becomes consistent upon triggering, which ensures correct results.

8 EVALUATION

In this section, we experimentally validate the system design of Slash through a set of end-to-end experiments and micro-benchmarks. First, we describe the setup of our evaluation in Sec. 8.1. Second, we compare Slash against RDMA UpPar, LightSaber, and Apache Flink on end-to-end queries (see Sec. 8.2). Third, we perform a drill-down analysis on Slash and RDMA UpPar to understand the implications behind our design choices (see Sec. 8.2). Finally, we sum up our key findings of our evaluation in Sec. 8.4.

8.1 Experimental setup

In the following, we introduce our hardware and software configurations (Sec. 8.1.1) as well as the selected workloads (Sec. 8.1.2).

8.1.1 Hardware and Software. In our experiments, we use the following hardware and software configurations.

Hardware Configuration. We run the experiments on an in-house, 16-node cluster. Each node is equipped with a 10-core, 2.4 Ghz Intel Xeon Gold 5115 CPU, 96 GB of main-memory, and a single-port Mellanox Connect-X4 EDR 100Gb/s NIC. Each NIC is connected to a 100 Gbits InfiniBand EDR switch by Mellanox. Every node runs Ubuntu Server 16.04. We disable hyper-threading and pin each thread to a dedicated core. Unless stated otherwise, every hardware component is configured with factory settings.

Software Configuration. In our evaluation, we use Slash, RDMA UpPar, LightSaber [56], and Apache Flink 1.9 [11] as Systems under Test (SUTs). We select Apache Flink as a representative of production-ready, scale-out SPEs based on managed runtimes, whereas we choose LightSaber as representative of scale-up SPEs. Flink provides queue-based partitioning to scale-out query processing. To configure Flink, we follow its configuration guidelines [6]. On each node, we allocate half of the cores for processing and the other half for network I/O. We reserve 50% of the OS memory to Flink and allocate the remaining memory to store the input dataset that we stream via main memory. We configure Flink to use IPoIB on our RDMA cluster.

We build Slash with `O3` compiler optimization and native CPU support using `gcc 9.3`. We configure Slash to use all physical cores and 48 GB of memory (using 2MB hugepages) for RDMA-related operations. Unless stated otherwise, we run Slash with the best configuration parameters that we present in Sec. 8.3 and configure the epoch of SSB to end every 64 MB of data. We follow similar configuration steps for LightSaber and RDMA UpPar. Note that we use Slash's RDMA channel to implement RDMA UpPar.

8.1.2 Workloads. To experimentally validate our system design, we select the *Yahoo! Streaming Benchmark* (YSB) [70], the *NEXMark benchmark suite* (NB) [61], and the Cluster Monitoring benchmark (CM) [63]. We choose YSB and NB, as they are commonly-used benchmarks that represent real-world scenarios [17, 70]. We select CM,

as it is based on a publicly-available, real-world dataset provided by Google. Furthermore, we introduce a self-developed *Read-Only* (RO) benchmark for our drill-down analysis.

YSB. The YSB assesses the performance of windowed aggregation operators. A record is 78-bytes large and stores an 8-bytes primary key and an 8-bytes creation timestamp. YSB consists of a filter, projection, and a time-based, per-key window. Following YSB specifications, we use a 10m event-time, tumbling count window.

NB. The NB simulates a real-time auction platform with three logical streams: an auction stream, a bid stream, and a seller event stream. Records are 206 (seller), 269 (auction), and 32 (bid) bytes large. Each record stores an 8-bytes primary key and an 8-bytes creation timestamp. The NB contains queries with stateless and stateful operators.

We use queries 7 (NB7), 8 (NB8), and 11 (NB11) to cover a wide range of scenarios. Based on these queries, we define three workloads to assess our SUTs. NB7 contains a window aggregation with a window of 60s on the bid stream. We select NB7, as it features small state sizes and an RMW state update pattern. NB8 consists of a 12h tumbling window join in event time over the auction and seller streams. We choose NB8, as it reaches large state sizes due to its append pattern for state update and large tuple sizes. NB11 consists of a session window join in event time over the bid and seller streams. We choose NB11 to assess the effect of small tuple size on the join implementation. We omit the other queries in the suite, as they are either stateless (NB1-2) or evaluate aggregations and joins (NBQ3-14), which we cover already with the selected queries.

CM. The CM benchmark executes a stateful aggregation over a stream of timestamped records containing the traces from a 12.5K-nodes cluster at Google. Each record is 64 bytes large and stores an 8-bytes primary key and an 8-bytes timestamp. The stateful aggregation is a 2s tumbling window that computes the mean CPU utilization of each executed job.

RO. The RO benchmark is a stateful query that counts the number of occurrences of items in a stream. We implement RO to investigate I/O bottlenecks, as data flows throughout the system without any costly computation. Each record stores an 8-bytes primary key and an 8-bytes creation timestamp. A stateful operator maintains the count of occurrences of each key. Keys are drawn following uniform distribution from a 100M-wide range.

Experiment Overview. We structure our evaluation as follows. First, we execute end-to-end queries to compare Slash, RDMA UpPar, LightSaber, and Flink (Sec. 8.2). To this end, we scale the input data size up to the number of nodes to perform weak scaling experiments [24]. Second, we run a series of micro-experiments to reason in detail about the performance behavior of Slash components (Sec. 8.3). Specifically, we breakdown the execution time of Slash and RDMA UpPar to perform a micro-architecture analysis [66] to reveal how well Slash uses hardware resources. Finally, we summarize the key findings of our evaluation (Sec. 8.4).

8.2 End-to-end Queries

In this section, we focus on the execution of end-to-end queries by the SUTs. We present our evaluation methodology in Sec. 8.2.1. We evaluate queries with windowed aggregations (YSB and NB7) in Sec. 8.2.2 and queries with windowed join (NB8 and NB11) in Sec. 8.2.3. Finally, we conduct in Sec. 8.2.4 a COST (Configuration that Outperforms a Single Thread) analysis [42] and compare Slash against LightSaber [56], which is an SPE optimized for single-node execution.

8.2.1 Methodology. In our end-to-end evaluation, we follow the benchmark methodology proposed in earlier research [70]. We pregenerate the dataset to stream data from main memory, to omit record creation and ingestion overhead. Thus, the upper bound for the input rate is the main-memory bandwidth. In every run, source operators consume data in real-time, which SUTs process. During execution, we measure query processing throughput, which we define as the number of records the SUT can process in one second. We repeat each experiment multiple times and compute average measurements. Through our experiments, we evaluate the efficiency of the SUTs while they execute a query.

8.2.2 Queries with Windowed Aggregations. In this section, we focus on the performance comparison between the SUTs while they perform windowed aggregations using YSB, CM, and NB7.

Workload. In YSB and NB7, each executor thread processes a partition of 1 GB of input data. In CM, each executor thread processes a partition of the provided input dataset. Each YSB partition contains records with a primary key drawn uniformly from a 10M-wide range. Each NB7 partition contains bid records with a primary key generated following a Pareto distribution that induces a long-tail due to heavy-hitters. Partitions of YSB, NB7, and CM are non-disjoint: the same key can occur multiple times in multiple partitions. In addition, we scale the number of executor threads and nodes. We configure each SUT to use 10 threads per node and up to 16 nodes. However, RDMA UpPar and Flink need to partition the input stream before the window operator. As a result, they use half the threads to execute the filter and projection and the second half for the window operator. Instead, Slash runs filter, projection, and windowing on all threads.

Result. In all YSB, CM, and NQB7 experiments, Slash outperforms all SUTs (Fig. 6a, 6b, and 6c). It achieves up to 12x and 25x higher throughput on the YSB compared to RDMA UpPar and Flink, respectively. Slash attains up to 22x and 104x higher throughput on the NQB7 compared to RDMA UpPar and Flink, respectively. Similarly, Slash achieves up to two order of magnitude higher throughput than RDMA UpPar and Flink, while executing CM. Overall, Slash is the only SUT to achieve throughput of up to 2 billion records/second and almost linear weak scaling in YSB , CM, and NB7.

Discussion. In this experiment, Slash achieves almost linear weak scaling, whereas other SUTs result in sub-optimal performance. The reasons for this superior performance for windowed aggregations are two-fold. First, as shown by previous research [70], queue-based partitioning of input records introduce a significant bottleneck in single-node setups. This also applies in the distributed case, as network transfer is mediated by software queues. As a result, SUTs that use queue-based partitioning to scale-out incur an inherent bottleneck regardless of the network hardware. Second, Slash efficiently computes local partial states and consistently merges them using point-to-point RDMA transfers among nodes. In contrast, Slash is not affected by a performance regression, when processing workloads that have a skewed distribution of partitioning keys. Overall, Slash attains higher throughput due to better utilization of underlying hardware resources, as we further explain in Sec. 8.3.

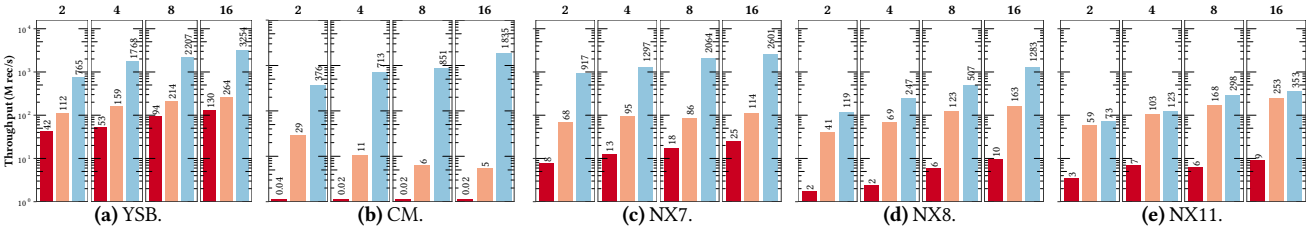


Figure 6: Throughput for YSB, CM, and NB (in records/s) of Flink (●), RDMA UpPar (●), and Slash (●) on 2, 4, 8, and 16 nodes.

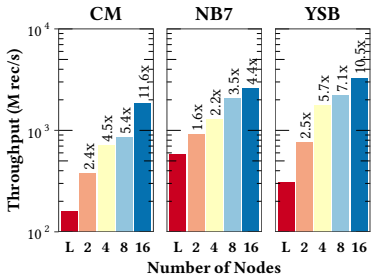


Figure 7: COST comparison against LightSaber (L).

8.2.3 Queries with Windowed Joins. In this section, we focus on the performance of our SUTs as they run windowed joins of NB8-11.

Workload. We follow the same setup of NB7 except for the following aspects. Each executor thread processes a partition of 1 GB of input stream, in which the ratios between auction and seller and between bid and seller are 4 to 1, according to the benchmark. Note that every bid has always a valid seller.

Result. In all NQB8 and NB11 runs, Slash achieves higher throughput compared to the other SUTs (Fig. 6d and 6e). In NQB8, it reaches up to 8x and 128x higher throughput than RDMA UpPar and Flink, respectively. In NQB11, it shows up to 1.7x and 40x higher throughput than RDMA UpPar and Flink, respectively.

We observe that Slash achieves almost linear weak scaling on queries with join operators, similarly to windowed aggregation. In contrast, Apache Flink and RDMA UpPar exhibit a severe loss in throughput. In sum, Slash does not achieve the same performance gain of aggregations, although it outperforms the other SUTs.

Discussion. The main performance differences for windowed joins among the SUTs result from the following characteristics. First, a windowed join operator is more memory-intensive than a windowed aggregation. A hash-based streaming join operator appends every record to intermediate state including the records that have no matching join partner yet. As a result, append operations in Slash do not benefit from CPU cache temporal locality. In contrast, RMW-based aggregations benefit from CPU caches, as the RMW operations induce temporal relations to cache accesses. Second, partitioning in RDMA UpPar and Flink induces performance regression as in the case of windowed aggregation. This becomes more severe while increasing the number of nodes. In contrast, Slash does not show performance regression when scaling out.

In sum, hash-based streaming joins do not show the same performance gain as windowed aggregations, as join are limited by compute resources. We plan to conduct further studies on the RDMA-based acceleration of streaming joins as future work.

8.2.4 COST Analysis. In this experiment, we compare the processing performance of Slash against a scale-up SPE, following the COST metric proposed by McSherry et al. [42]. We select LightSaber as the latest proposed scale-up SPE, which does not run on a managed runtime (as BriskStream [72]). We choose CM, NB7, and YSB as workloads supported by both SUTs, as LightSaber does not support joins. In Fig. 7, we show the throughput of LightSaber (L) and of Slash (on 2, 4, 8, and 16 nodes) on the selected workloads.

We observe that Slash outperforms LightSaber in each run as it improves its performance when doubling the number of nodes. Furthermore, Slash achieves almost linear speedup on YSB and CM (up to 11.6x throughput increment using 16 nodes) and sub-linear scaling on NB7 compared to LightSaber (up to 4.4x throughput increment using 16 nodes), respectively. The improvement of Slash over LightSaber is smaller compared to the gain over RDMA UpPar, as LightSaber's execution is agnostic to data re-partitioning. Overall, the key insight of this experiment is that LightSaber offers a valid alternative to SPEs that rely on data re-partitioning as long as the workload 1) is sustainable on a single node, 2) does not need RDMA ingestion, and 3) does not involve join operators. For highly-demanding workloads, Slash offers robust scale-out performance, as our evaluation shows.

8.3 Performance Drill-down

In the previous section, we have analyzed the performance result of Slash and Flink on end-to-end queries. In this section, we reveal the reasons behind the improvement in performance of Slash over RDMA UpPar. We omit Flink in this evaluation, as its partitioning approach suffers from runtime and IPoIB overhead [70]. In the following, we first describe the methodology for our drill-down evaluation in Sec. 8.2.1. After that, we assess in Sec. 8.3.2 the maximum achievable throughput of RDMA UpPar and Slash in our RDMA evaluation setup. In this setup, we consider application-related aspects, such as parallelism and data skewness. In Sec. 8.3.3, we break down the execution time of both SUTs to analyze the impact of each CPU component. Finally, in Sec. 8.3.4, we analyze the resource utilization of each SUT on a stateful workload using hardware performance counters.

8.3.1 Methodology. In our performance drill-down, we analyze workload-related and hardware-related aspects of Slash and RDMA UpPar. Workload-related aspects provide a high-level identification of bottlenecks and include data characteristics and application settings. Hardware-related aspects consist of hardware performance counters that we use to conduct micro-architecture analysis. These metrics allow us to derive the CPU components that stall the execution and the saturation point of the RDMA links. In the following, we consider the RO and YSB benchmarks and provide a brief description of the sampled metrics for each class of experiments.

8.3.2 Analysis of workload-related aspects. In this section, we compare the performance of RDMA UpPar and Slash with a focus on RDMA-based data transfer. We consider the RO query, which is primarily I/O bound, to evaluate the impact on performance of data re-partitioning. We analyze the effect on throughput and latency during query processing of application-related knobs, such as parallelism and buffer size, as well as data characteristics

Workload. We setup two Slash instances on two servers connected by a single RDMA NIC to measure the impact of buffer size on throughput and latency. The producer instance streams the input data to the consumer instance via our RDMA channels. The consumer instance polls the RDMA channels and applies stateful operator logic based on the benchmark. Each instance uses up to 10 threads for the executor. Every producer thread on the first node sends buffers of records via RDMA to one consumer thread on the other node in Slash. In RDMA UpPar, every producer thread sends buffers of records to any consumer via hash-partitioning. To measure the impact of parallelism on throughput, we use up to 8 nodes. Note that we configure our RDMA channels to use $c = 8$ (credits). Other configurations, such as $c = 8$ and $c = 16$ decrease throughput by up to 3%, whereas $c = 64$ leads to a performance regression by up to 10%.

Results. In this experiment, we assess the impact of application-related aspects on the performance of both SUTs. First, we show the impact of buffer size on throughput for both SUTs (Fig. 8a) and latency (Fig. 8b). Second, we measure the effect of parallelism (Fig. 8c) by scaling the number of threads and nodes. In Fig. 8a and 8c, we mark in red the maximum achievable network bandwidth (11.8 GB/s), which we measure using the *ib_write_bw* tool [33]. Finally, we analyze the impact on throughout of a skewed distribution of the partitioning key (Fig. 8d). To this end, we generate partitioning keys following a Zipfian distribution using $z = 0.2 \ldots 2.0$.

`Throughput.` We observe that Slash outperforms RDMA UpPar in all configurations as it utilizes up to 95% of the available network bandwidth (11.2 GB/s out of 11.8 GB/s) using two threads. Slash almost saturates the theoretical bandwidth limit of one RDMA NIC using two threads and 32 KB buffer size. In contrast, RDMA UpPar utilizes up to 50% of the available network bandwidth, i.e., 5.9 GB/s.

`Latency.` Slash achieves latencies below 100 μs for buffers sizes below 128 KB, while it achieve up to 1 ms of latency with 1 MB buffer size and above. In contrast, latencies of RDMA UpPar for each buffer size are about 10% higher than Slash.

`Parallelism.` Slash achieves the highest aggregated throughput for query processing (Fig. 8c). Slash achieves 11.2 GB/s on the RO benchmark using two threads. In contrast, RDMA UpPar requires 10 thread to saturate up to 91% of the available network throughput.

`Data Skewness.` Slash shows robust performance in the presence of a skewed distribution of the partitioning keys. Interestingly, we observe that the throughput of Slash increases, when the partitioning keys in the data stream are highly skewed. In contrast, the throughput of RDMA UpPar decreases by up to 68% (RO) and 110% (YSB), while the skewness in the distribution increases.

Discussion. Overall, our experiments show three interesting aspects. First, Slash becomes network bound with a lower number of threads compared to RDMA UpPar (i.e., 2 vs. 10). This induces an important benefit: increasing the number of threads and RDMA NICs per node results in higher processing throughput. We cannot derive the same conclusion for RDMA UpPar, as it requires an higher number of threads to achieve almost full line rate. As a result, Slash exhibits a higher per-thread efficiency compared to RDMA UpPar. Second, buffer size plays an interesting role also for data stream processing on RDMA hardware. It enables the highest (or lowest) throughput (or latency) based on workloads and service constraints. In particular, both SUTs achieve micro-second latency, which is one order of magnitude lower than the latencies measured on Flink (not shown in the figures). Finally, Slash offers more robust performance than RDMA UpPar, in the presence of skewed data. RDMA UpPar suffers a performance regression, as hash-partitioning causes load imbalance due to the data-dependent selection of the consumer induced by data skewness. In contrast, Slash achieves constant throughput on RO, regardless of the skewness, as the transfer performance of RDMA channels is not data-dependent. When executing a stateful query, such as YSB, skewness results in higher throughput for Slash, as it reduces the number of key-value pairs of the state to be merged by the SSB.

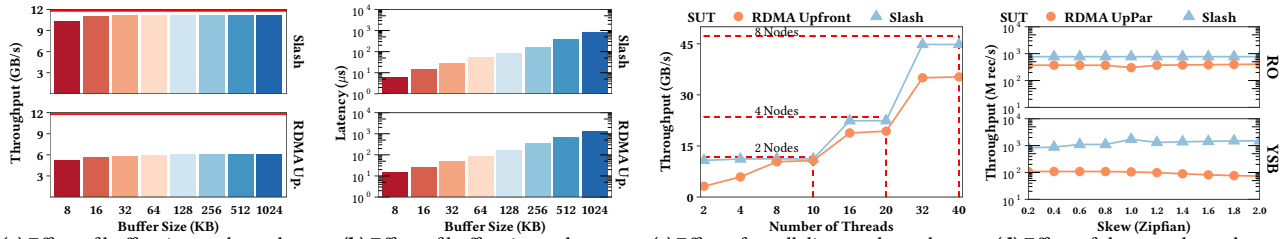
8.3.3 Execution Breakdown. In the previous section, we have analyzed the effect of application-related knobs on throughput and latency of query processing. In particular, we have shown that RDMA UpPar provides lower efficiency compared to Slash. In this section, we perform an execution breakdown to reveal the reasons behind the sub-optimal performance of RDMA UpPar in comparison to Slash. To this end, we carry out a micro-architecture analysis of Slash and RDMA UpPar on the RO benchmark.

Metrics. Before delving into our analysis, we provide a brief description of the considered metrics. On a high level, recent x86 CPUs consist of two pipelined components: a *front-end* and a *back-end*. The front-end decodes instructions into μ-*ops* and delivers up to four μ-ops per cycle to the back-end. The back-end processes μ-ops out-of-order by allocating execution units and loading data from memory. Completed μ-ops are defined as *retired* (R) and constitute the useful work performed by a CPU. Front-end stalls, back-end stalls, and branch mis-prediction are sources of inefficiency for a CPU. Upon a front-end stall, the back-end has no μ-ops to process, thus, the application is front-end bound (FeB). Upon a back-end stall, a μ-ops needs to wait for data from the memory subsystem or for an execution unit. In the former case, the execution is *Memory-bound* (MemB), whereas in the latter case, it is *Core-bound* (CoreB). Finally, *bad speculation* (BadS) due to branch mis-prediction results in the cancellation of μ-ops prior to their retirement.

Workload. We execute the RO benchmark with best configurations of both SUTs that we derive in the previous experiments and refer to the nodes as sender and receiver. Specifically, we choose 64 KB as buffer size and repeat the measurements using two and ten threads, as they induce the highest throughput.

Results. In Fig. 9, we show the execution breakdown in μ-ops for the RO benchmark. RDMA UpPar requires up to twice more μ-ops to execute the RO benchmark than Slash. Its senders incur in up to 7x more front-end stalls than the senders of Slash. In fact, the execution of its senders with two and ten threads are front-end bound (22 and 33% of total CPU cycles, respectively). In contrast, Slash's senders are essentially core-bound and its receivers are memory-bound.

Discussion. This experiment reveals the source of inefficiency of RDMA UpPar as we compare it to Slash. The key finding is that RDMA UpPar inefficiently use CPU resources due to more complex application logic. This has a two-fold impact on the execution. First,



(a) Effect of buffer size on throughput. **(b)** Effect of buffer size on latency. **(c)** Effect of parallelism on throughput. **(d)** Effect of skew on throughput.

Figure 8: Drill-down analysis of Slash and RDMA UpPar. We consider the impact on throughput and latency of buffer size (a, b), parallelism (c), and skewness in the distribution of the partitioning key (d).

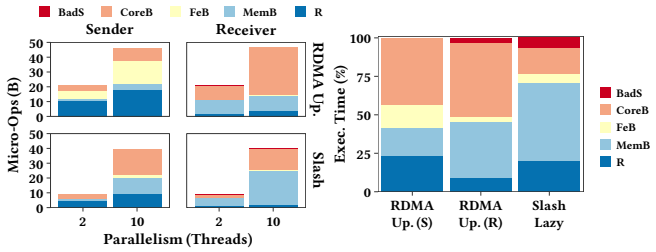


Figure 9: Execution Breakdown of RO.

Figure 10: Execution Breakdown of YSB.

the complex logic behind partitioning results in a large code footprint and thus high number of μ-ops to retire, compared to Slash. A large code footprint results in front-end stalls that slow down the sender and in turn the receiver needs to spend more time waiting. Furthermore, the implementation of partitioning requires branches, which lead to front-end stalls in the case of branch mis-prediction.

Second, the receivers of RDMA UpPar need to poll on multiple RDMA channels (and thus memory locations) depending on fan-out, which makes execution mainly core-bound. Core-bound execution is induced by the *pause* instruction [15] that RDMA channels use for polling (see Sec. 6). This occurs when an RDMA buffer is not fully transferred from remote to local memory. In contrast, Slash's senders are core-bound, as they saturate the network and thus must wait for data transmission through the *pause* instruction [15]. Its receivers are primarily memory-bound, which is the result of waiting for in-flight data to materialize in registers. Its receivers are also core-bound, as they must wait on senders. However, this differs from the execution of RDMA UpPar. In fact, the senders of Slash cannot send as the network is saturated, whereas the senders of Slash cannot send due to stalls. In sum, the discussion above suggests that the main bottleneck of Slash is the network, in line with our findings of Sec. 8.3.2.

8.3.4 Resource Utilization of Stateful Execution. In the following, we further analyze the usage of CPU resources of Slash and RDMA UpPar to understand the impact of our design choices on stateful queries. We use the best performing configuration as in Sec. 8.3.3 to conduct a micro-architecture analysis of YSB on Slash and the sender and receiver of RDMA UpPar. We collect execution-related hardware performance counters to analyze the following three aspects.

Micro-architectural Analysis. In Fig. 10, we consider the resource utilization of the CPU micro-architecture and observe that Slash is primarily memory-bound, whereas sender and receiver of RDMA UpPar are core-bound. RDMA UpPar's sender suffers from front-end stalls, as shown by prior findings [70], whereas Slash minimally suffers from branch mis-prediction. Finally, we note that Slash spends about 20% of its execution time performing retirement, whereas the receiver of RDMA UpPar - which computes results - spends 10% of its time retiring instructions.

The reason behind the above observations are as follows. On the sender side of RDMA UpPar, the partitioning logic results in front-end stalls. Besides, the data-dependent writes to fan-out RDMA buffers result in back-end stalls (memory-bound fraction). In total, this accounts for about 30% of the execution time, which slows down partitioning. Receiver threads of RDMA UpPar poll on multiple, inbound RDMA channels, which rely on the *pause* instruction, thereby execution becomes core-bound and slows down the receivers. Note that the sender adjusts its speed to the processing rate of the receiver, which results in waiting that makes the sender core-bound. In contrast, Slash shows a more efficient execution than RDMA UpPar, as it essentially performs RMWs to the in-memory area of the SSB. RMWs induce a mainly memory-bound execution due to the latency of atomic instructions, such as `compare-and-swap`. The epoch-based state synchronization of Slash results in streamlined RDMA accesses that negligibly impact performance.

Instruction Stream Analysis. We show in Tab. 1 that Slash requires up to 4x less instructions and up to 5x less cycles to process each single record compared to RDMA UpPar. Furthermore, Slash executes close to one *instruction per cycle* (IPC). RDMA UpPar requires up to 0.4 and 0.6 IPC on sender and receiver, respectively. Note that an optimal execution retires 4 instructions per cycle [67].

The difference between the two SUTs lays in the more complex partitioning logic of RDMA UpPar, which needs more instructions per record. Thus, the limiting factor for RDMA UpPar is partitioning, which slows down the receiver. As a result, the consumer essentially waits for inbound data to process, and is thus core-bound. In contrast, Slash executes a simple processing logic on a record basis, yet relies on a more complex logic upon state synchronization. With this trade-off, Slash attains fast execution on the common code path and induces negligible overhead upon synchronization.

Data Locality Analysis. We observe in the rightmost part of Tab. 1 that the execution of Slash induces about 1.5 misses per record on each cache level. In contrast, the producer of RDMA UpPar exhibits about 1.3 misses on each cache level, whereas its receiver minimally suffers from LLC misses. Additionally, Slash induces an aggregated memory throughput of 70.2 GB/s, which is about 52% of the aggregated memory bandwidth of the two nodes. In contrast, RDMA UpPar has a memory access rate of 4.1 (sender) and 4.2 (receiver) GB/s. The LLC misses for RDMA UpPar's sender are due to data dependent writes in RDMA fan-out buffers. Slash is affected by cache misses due to the updates of the SSB, which rely on atomic

	IPC	Instr./ Rec.	Cyc./ Rec.	L1d Miss/ Rec.	L2d Miss/ Rec.	LLC Miss/ Rec.	Aggr. Mem. Bw (GB/s)
RDMA	0.6	166	274	1.36	1.31	1.2	4.1
UpPar	0.4	78	276	1.74	1.42	0.4	4.2
Slash	0.9	42	53	1.75	1.52	1.3	70.2

Table 1: Resource utilization of RDMA UpPar (sender and receiver) and Slash on YSB using two nodes.

operations. Partitioning throughput induces a low memory access rate for RDMA UpPar, whereas Slash is mainly memory-bound.

Discussion. This experiment sheds light on the different performance of both Slash SUTs when executing stateful computations. This is due to the more efficient resource utilization of the SSB of Slash versus the data-partitioning approach of RDMA UpPar. In sum, RDMA UpPar is bound by partitioning throughput and ultimately by network bandwidth, whereas Slash is primarily limited by memory performance. This validates that our processing model based on eager computation of partial results and on their lazy merging is an alternative strategy to data re-partitioning of state-of-the-art SPEs.

8.4 Summary

In sum, the findings of our experiments validate our design choices. Based on them, we derive the following guidelines for system builders who seek to accelerate their stream processing workloads via RDMA.

1. Apply native RDMA acceleration. Native RDMA acceleration enables a system design that scales with the number of nodes and achieves higher throughput on common streaming workloads than partitioning-based approaches. In particular, our Slash prototype achieves up to 25x and 8x higher throughput than the strongest scale-out baseline on windowed aggregations and joins, respectively. Furthermore, Slash outperforms a state-of-the-art scale-up SPE called LightSaber by a factor of 11.6 on windowed aggregations.

2. Avoid data re-partitioning. Data re-partitioning induces a performance regression, as it makes SPEs be limited by partitioning throughput. To understand this performance regression, we run a performance drill-down of Slash and RDMA UpPar. We show that Slash achieves full line rate on RO benchmark using RDMA and minimal CPU resources. In contrast, RDMA UpPar needs a higher degree of parallelism to reach high throughput, as it is primarily CPU bound due to costly data re-partitioning. Furthermore, we demonstrate that Slash does not suffer from skewed data distributions.

3. Use lazy merging. We show that a processing model based on lazy merging of eagerly computed partial results attains the highest throughput in our evaluation. However, lazy merging requires careful synchronization among nodes to avoid overhead and inconsistency. We demonstrate that our SSB achieves lazy merging, is skew-agnostic, and induces minimum overhead on query processing.

9 RELATED WORK

In the following, we discuss the application of RDMA to database systems and the differences between our approach and existing SPEs.

RDMA for database systems. The database community has adopted RDMA to speed up OLAP and OLTP workloads. We identify three areas of adoption: distributed transactions, batch analytical query processing, and key-value stores. Distributed transactions techniques [8] profit from RDMA to scale out on large deployments [69]. Systems that use RDMA for OLTP are Oracle RAC [1], IBM pureScale [2], NAMDB [9, 69], and FaRM [18, 19]. OLAP operators, especially joins, benefit from RDMA to speed up partitioning [7, 9, 20, 21, 57]. Big data frameworks accelerate batch workloads via RDMA [28, 41, 62], while key-value stores use RDMA to increase throughput and reduce latency of value access [31, 51].

Our work is orthogonal to above research, as stream processing has different requirements than traditional database systems or key-value stores [29, 55]. SPEs require fast, stateful processing of inbound data streams. Their key requirement deals with the state management component that must support fast, concurrent point updates and range scans for analytics readiness over windows. An RDMA-based key-value store, such as FaRM, is not suitable to store state, as it targets transactional workloads with point lookups and updates. As a result, system designers need to devise algorithms and protocols to natively accelerate an SPE using RDMA. With Slash, we fill this gap as we provide SPE components that address RDMA acceleration by design.

Stream processing engines. We distinguish two classes of SPEs. Scale-out SPEs focus on scalability and rely on socket-based communication to distribute query execution on a large cluster of nodes [11, 12, 47, 58, 68]. A recent prototype provides lightweight RDMA integration for Apache Storm [73] and is thus equivalent to RDMA UpPar. Scale-up SPEs target single-node performance but neglect network-related aspects [23, 34, 56, 70, 72]. With Slash, we combine the best of both worlds. Our goal is scale-out stream processing using RDMA acceleration, while considering recent scale-up techniques for SPEs to achieve maximum performance and at rack-scale. To this end, we propose a system design that profits from scale-up techniques, such as omitting partitioning, late merge, shared mutable state, and a hardware-conscious execution. However, we rethink the above techniques to apply them to distributed stateful computation using RDMA and scale processing over multiple nodes. Thus, we devise the RDMA acceleration of late merging and enable consistent shared mutable state among the nodes. Overall, our approach attains higher performance than RDMA-based data re-partitioning approaches, such as RDMA UpPar, and almost linear weak scaling in comparison to a scale-up SPEs, such as LightSaber.

10 CONCLUSION

In this paper, we propose Slash, our novel SPE with native RDMA acceleration, which paves the way to a new class of SPEs for high-speed networks. Slash enables a processing model based on eager computation of distributed, partial state and its lazy merging into a consistent global state. We validate our prototype against RDMA UpPar that uses RDMA-based data re-partitioning, IPoIB-enabled Apache Flink, and a scale-up SPE called LightSaber [56]. Overall, we show that our approach achieves on common streaming workloads higher throughput than RDMA UpPar (up to a factor of 22), than Apache Flink (up to two orders of magnitude), and than LightSaber (up to a factor of 11.6).

Acknowledgement. This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and 01IS18037A) and by the German Research Foundation as FONDA (ref. 414984028).

REFERENCES

[1] 2010. Delivering Application Performance with Oracles InfiniBand Tech. https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf

[2] 2012. Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature. http://www.redbooks.ibm.com/redbooks/pdfs/sg248018.pdf

[3] Advanced Micro Devices, Inc. 2020. AMD EPYC® 7003 Series Processor. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Very Large Data Bases*. 734–746.

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB* (2015).

[6] Apache Foundation. 2015. Apache Flink Configuration. https://ci.apache.org/projects/flink/flink-docs-master.

[7] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *ACM SIGMOD*.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. https://doi.org/10.1145/568271.223785

[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[10] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[12] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 725–736. https://doi.org/10.1145/2463676.2465282

[13] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA* (2018 acm sigmod international conference on management of data (sigmod '18), houston, tx, usa ed.). ACM. https://www.microsoft.com/en-us/research/publication/faster-concurrent-key-value-store-place-updates/

[14] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[15] Intel Corporation. 2012. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-044US.

[16] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[17] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2471–2486. https://doi.org/10.1145/3318464.3389723

[18] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. *No Compromises: Distributed Transactions with Consistency, Availability, and Performance.* Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[19] Aleksandar Dragojeviundefined, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *USENIX ATC*.

[20] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1477–1488. https://doi.org/10.1109/ICDE48307.2020.00131

[21] Philip Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. 2010. A Spinning Join That Does Not Get Dizzy. *IEEE International Conference on Distributed Computing Systems.*

[22] Hector Garcia-Molina and Christos A. Polyzios. 1990. Two Epoch Algorithms for Disaster Recovery. In *Proceedings of the Sixteenth International Conference on Very Large Databases* (Brisbane, Australia). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 222–230.

[23] Philipp M. Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2487–2503. https://doi.org/10.1145/3318464.3389739

[24] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[25] S. Hauger, T. Wild, A. Mutter, A. Kirstaedter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. 2009. Packet Processing at 100 Gbps and Beyond - Challenges and Perspectives. In *2009 ITG Symposium on Photonic Networks*. 1–10.

[26] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 431–444.

[27] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[28] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2012.65

[29] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, and et al. 2018. Providing Streaming Joins as a Service at Facebook. *Proc. VLDB Endow.* (2018).

[30] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[33] The Linux Kernel and OpenFabrics Alliance. 2019. Open Fabrics Enterprise Distribution (OFED) Performance Tests. https://github.com/linux-rdma/perftest.

[34] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 555–569. https://doi.org/10.1145/2882903.2882906

[35] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications* (London, United Kingdom) *(SIGCOMM '94)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/190314.190324

[36] 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* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[37] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. In *Proceedings of the International Conference on Very Large Databases, VLDB 2013* (proceedings of the international conference on very large databases, vldb 2013 ed.). VLDB - Very Large Data Bases. https://www.microsoft.com/en-us/research/publication/llama-a-cachestorage-subsystem-for-modern-hardware/

[38] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44. https://doi.org/10.1145/1058150.1058158

[39] Barbara Liskov and Rivka Ladin. 1986. Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada) *(PODC '86)*. Association for Computing Machinery, New York, NY, USA, 29–39. https://doi.org/10.1145/10590.10593

[40] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-Aware Data Shuffling Operator for Parallel Database Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA,

48–63. https://doi.org/10.1145/3064176.3064202
[41] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. 2013. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *2013 42nd International Conference on Parallel Processing*. 641–650. https://doi.org/10.1109/ICPP.2013.78
[42] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry
[43] Mellanox Technologies. 2019. Mellanox ConnectX®6 VPI Card. https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf.
[44] Mellanox Technologies. 2019. Mellanox Quantum® HDR Modular Switch CS8700. https://www.mellanox.com/related-docs/prod_ib_switch_systems/PB_QM8700.pdf.
[45] Microsoft Azure. 2021. Microsoft Azure HBv3-series Specification. https://docs.microsoft.com/en-us/azure/virtual-machines/hbv3-series.
[46] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages. https://doi.org/10.1145/1462166.1462167
[47] Derek Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (proceedings of the 24th acm symposium on operating systems principles (sosp) ed.). ACM. https://www.microsoft.com/en-us/research/publication/naiad-a-timely-dataflow-system-2/
[48] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. 2019. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/3319647.3325827
[49] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048
[50] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/2043556.2043560
[51] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, and et al. 2011. The Case for RAMCloud. *Commun. ACM* (2011).
[52] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 96–107. https://doi.org/10.14778/2850583.2850585
[53] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed Query Processing over High-speed Networks. *Proc. VLDB Endow.* 9, 4 (2015), 228–239. https://doi.org/10.14778/2856318.2856319
[54] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.
[55] Michael Stonebraker and Stan Zdonik. 2005. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Rec.* (2005).
[56] Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-core Processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. ACM, Portland, OR, USA.
[57] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *to appear in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA.
[58] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@Twitter. In *ACM SIGMOD*.
[59] Jonas Traub, Philipp Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing. In *22th International Conference on Extending Database Technology (EDBT)*.
[60] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On the [Ir]Relevance of Network Performance for Data Processing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing* (Denver, CO) *(HotCloud'16)*. USENIX Association, USA, 126–131.
[61] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2004. NEXMark - A Benchmark for Queries over Data Streams DRAFT. (2004).
[62] Md. Wasi-ur Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. DK Panda. 2013. High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 1908–1917. https://doi.org/10.1109/IPDPSW.2013.238
[63] John Wilkes. 2011. More Google Cluster Data. https://github.com/google/cluster-data.
[64] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044
[65] Seokwoo Yang, Siwoon Son, Mi-Jung Choi, and Yang-Sae Moon. 2019. Performance improvement of Apache Storm using InfiniBand RDMA. *The Journal of Supercomputing* (2019), 1–27.
[66] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459
[67] Ahmad Yasin, Jawad Haj-Yahya, Yosi Ben-Asher, and Avi Mendelson. 2019. A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors. *ACM Trans. Archit. Code Optim.* 16, 4, Article 46 (Dec. 2019), 25 pages. https://doi.org/10.1145/3369383
[68] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 423–438.
[69] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335
[70] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing Efficient Stream Processing on Modern Hardware. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 516–530. https://doi.org/10.14778/3303753.3303758
[71] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 659–670. https://doi.org/10.1109/ICDE.2017.119
[72] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 705–722. https://doi.org/10.1145/3299869.3300067
[73] Ziyu Zhang, Zitan Liu, Qingcai Jiang, Junshi Chen, and Hong An. 2021. RDMA-Based Apache Storm for High-Performance Stream Data Processing. *International Journal of Parallel Programming* (2021), 1–14.
[74] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 741–758. https://doi.org/10.1145/3299869.3300081