Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs

MATTHIAS JASNY*, Technical University of Darmstadt, Germany
LASSE THOSTRUP*, Technical University of Darmstadt, Germany
SAJJAD TAMIMI, Technical University of Darmstadt, Germany
ANDREAS KOCH, Technical University of Darmstadt, Germany
ZSOLT ISTVÁN, Technical University of Darmstadt, Germany
CARSTEN BINNIG, Technical University of Darmstadt & DFKI, Germany

In this paper, we present a novel communication scheme called zero-sided RDMA, enabling data exchange as a native network service using a programmable switch. In contrast to one- or two-sided RDMA, in zero-sided RDMA, neither the sender nor the receiver is actively involved in data exchange. Zero-sided RDMA thus enables efficient RDMA-based data shuffling between heterogeneous hardware devices in a disaggregated setup without the need to implement a complete RDMA stack on each heterogeneous device or the need for a CPU that is co-located with the accelerator to coordinate the data transfer. As such, we think that zero-sided RDMA is a major building block to make efficient use of heterogeneous accelerators in future cloud DBMSs. In our evaluation, we show that zero-sided RDMA can outperform existing one-sided RDMA-based schemes for accelerator-to-accelerator communication and thus speed up typical distributed database operations such as joins.

CCS Concepts: • **Hardware → Networking hardware**; • **Networks → Programmable networks**; • **Information systems → Relational parallel and distributed DBMSs**.

Additional Key Words and Phrases: Heterogeneous Compute; RDMA; GPU; FPGA; Communication Scheme

ACM Reference Format:
Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. 2024. Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 36 (February 2024), 28 pages. https://doi.org/10.1145/3639291

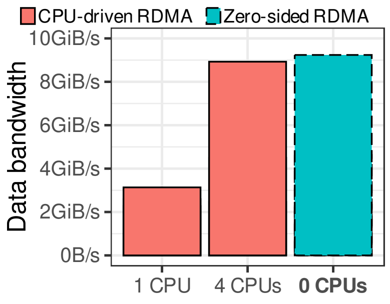
1 INTRODUCTION

Disaggregation and the need for RDMA. In the past decade, cloud computing and the landscape of database systems have undergone a significant transformation. Notably, there has been a shift towards disaggregation, separating compute and storage or even accelerator pools from traditional compute [5, 9, 25, 43] because disaggregation offers improved resource utilization, as each resource can be scaled independently based on demand. This trend has been transformative for cloud-native DBMSs, which all build on top of such disaggregated architectures. Since, in disaggregated

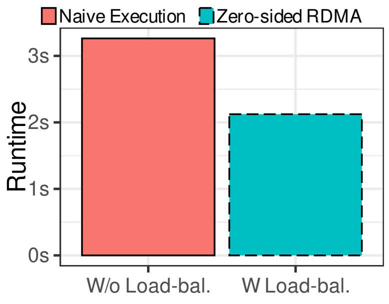
*Authors with equal contribution.

Authors' addresses: Matthias Jasny, Technical University of Darmstadt, Germany; Lasse Thostrup, Technical University of Darmstadt, Germany; Sajjad Tamimi, Technical University of Darmstadt, Germany; Andreas Koch, Technical University of Darmstadt, Germany; Zsolt István, Technical University of Darmstadt, Germany; Carsten Binnig, Technical University of Darmstadt & DFKI, Germany.

ACM 2836-6573/2024/2-ART36
https://doi.org/10.1145/3639291



(a) GPU-to-GPU transfers, CPU-driven vs. network-driven transfers.



(b) Perfect utilization in heterogeneous execution environments.

Fig. 1. Motivating examples of using zero-sided RDMA for a GPU-to-GPU shuffle of 4KiB data items. (a) Zero-sided RDMA provides equal or better performance than traditional CPU-based schemes without the need of co-locating a CPU per accelerator. (b) Zero-sided RDMA enables seamless load balancing of data transfers in case accelerators consume data at different speeds.

architectures, the rate at which data can be moved across nodes is one of the most important factors to performance, there has been considerable attention on developing fast network solutions. Remote Direct Memory Access (RDMA) is deployed by major cloud vendors to enable efficient resource disaggregation [2, 37]. In database use cases, RDMA has successfully improved the performance for distributed join operators [3, 41], reduced the cost of concurrency control [9], and made storage access [51] possible at unprecedented speeds.

The need for heterogeneous compute. Another significant trend in cloud data centers is that, in response to the performance stagnation of CPUs due to the end of Moore's Law and Dennard Scaling, heterogeneous accelerators are becoming commonplace as an alternative to CPU-based compute resources [31]. Accelerators have been shown to provide significant speed-ups for DBMS workloads [22, 27]. However, looking ahead, a major question for future cloud DBMSs is how to efficiently use disaggregated heterogeneous compute resources such as GPUs or FPGAs. Similar to disaggregated CPU-based compute resources, these accelerators should be deployed as network-attached accelerator pools, but today this is not the case. Accelerators depend on the presence of a host CPU for both control and moving data in and out of the accelerator. As we show in this work, if we continue to keep the CPU on the critical path of data movement, we will not be able to take advantage of fast networks and RDMA-based communication. Instead, we propose zero-sided RDMA as a solution to remove the CPU from the critical path and enable efficient accelerator-to-accelerator data movement powered by a smart switch.

The pitfalls of RDMA today for accelerators. To explain how zero-sided RDMA works, it is important to see how RDMA involves CPUs today. In addition to two-sided operation, where the CPUs of both the sender and receiver nodes are involved, it already offers one-sided primitives, namely READ and WRITE operations. These primitives solely involve the CPU of the initiator node rather than the target node. As a result, nodes can read from and write to remote memory locations without requiring the remote CPU to be actively involved, as the remote NIC handles the read and write operations. This offloading of network overhead to the NIC enables efficient scaling and improved performance [53]. However, if we want to fully utilize accelerators in the cloud, the involvement of CPUs, even if just on the sender side, will lead to bottlenecks (as seen in Figure 1a, multiple CPU cores need to be dedicated to coordinating data transfers).

Accelerator-driven RDMA as an alternative? One way to remove the CPU would be to implement RDMA-based operations directly on the accelerators. Even though this is technically possible [21], it is challenging for the following reasons: (1) RDMA primitives, such as one-sided

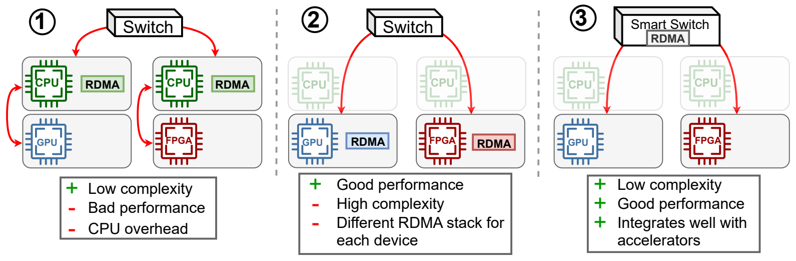
RDMA verbs, might not be available for a given accelerator, requiring a full RDMA stack to be implemented per accelerator type. Beyond the engineering cost, the additional problem is that executing communication logic on specialized hardware devices consumes compute resources that could otherwise be utilized for processing. (2) Implementing advanced RDMA-based communication schemes, such as many-to-many data shuffling or multicast, on top of the RDMA stack on each accelerator type requires re-implementing features and has different challenges and limitations on each hardware type.

The case for network-driven RDMA. To remove the CPU from the critical path of performance and to reduce the engineering effort in disaggregated accelerators, in this work, we propose a novel network-driven scheme where neither a CPU co-located with an accelerator is actively needed nor the GPU/FPGA has to implement the RDMA stack. Our approach fully offloads the RDMA stack and the RDMA-based communication logic between devices to the network, particularly to a programmable switch. The main idea of offloading the communication logic to the network is that the switch acts as a coordinator of data transfers; i.e., it issues an RDMA READ to a sender and rewrites the read response into an RDMA WRITE for the receiver using the programmable data plane of the switch. Since this does not involve the sender or receiver actively, we term this communication scheme *zero-sided RDMA*.

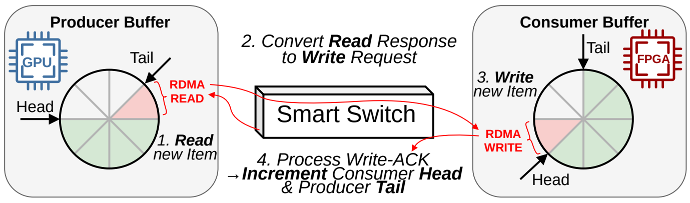
Why use a programmable switch? To implement zero-sided RDMA, a programmable switch presents an excellent choice due to its placement as an intermediate in the network, which allows it to support complex data transfer operations, such as distributed data shuffling used in databases, and optimally schedule data transfers. Important is that zero-sided RDMA is not limited to point-to-point communication (1:1) and, as we show later in this paper, we can support various communication flows for common distributed data management use cases, namely N:M shuffles, as well as complex distributed operations such as network-driven reliable multicast or seamless load-balancing. From a complexity perspective, this is much more efficient than implementing the RDMA stack and shuffling logic in each instance of an accelerator (i.e., FPGA, GPU). Finally, switches can process at the aggregated line-rate of all connected devices [17] and thus provide a scalable solution to zero-sided RDMA, as we show in our evaluation.

Motivating examples. Figure 1 shows the main benefits of our novel switch-driven scheme that uses zero-sided RDMA for data transfers. First, compared to the CPU-driven scheme (Figure 1a), zero-sided RDMA can achieve close to the maximum network bandwidth without requiring a co-located CPU per accelerator. In fact, the CPU-driven scheme requires multiple CPU cores to achieve the same bandwidth for transferring messages. Moreover, zero-sided RDMA provides many other benefits, such as seamless load-balancing (Figure 1b). Load-balancing is important if multiple (heterogenous) receivers consume data while one of the receivers is slower than the others. As we later discuss, our switch-driven scheme can detect such cases by monitoring the progress of a data shuffle and seamlessly redistributing data.

Relevance for DBMSs & contributions. The database community continuously seeks ways to adapt to the complexities of disaggregation and the integration of RDMA with diverse accelerators. In this context, zero-sided RDMA presents a practical solution by enabling network-driven data transfers between heterogeneous accelerators in DBMSs. We evaluate our idea on specific DBMS use cases focusing on OLAP scenarios, including data shuffling (i.e., 1:1, N:M), a core task in distributed query processing. In addition to the chosen shuffle scenarios, zero-sided RDMA can be applied to many more, including data replication or efficient data transfers in disaggregated databases to and from compute. As our experimental results on latency indicate, it will be possible in the future to extend zero-sided RDMA to also support latency-critical distributed workloads beyond OLAP, such as OLTP or data streaming.



(a) CPU- vs. Accelerator- vs. Network-driven (zero-sided RDMA).



(b) 1:1 Flow with zero-sided RDMA.

Fig. 2. Network-driven communication with zero-sided RDMA vs. alternative communication schemes. (a) In a CPU-driven scheme ①, the CPU is responsible for carrying out the communication. In the accelerator-driven scheme ②, the RDMA stack is realized directly on the accelerator. With zero-sided RDMA ③, the RDMA stack and communication scheme are fully offloaded into the network, removing the need for CPUs, while only one RDMA stack (e.g., on the switch) is needed. (b) In zero-sided RDMA, producers/consumers interact only with their buffers in local memory. The switch reads items from the sender buffer (step 1) and moves them to the receiver buffer (step 3) by converting RDMA READS into WRITES (step 2). The switch coordinates the communication using head/tail pointers (step 4).

In summary, our work on zero-sided RDMA brings the following contributions: (1) First, we introduce our novel network-driven communication scheme — zero-sided RDMA — which allows efficient accelerator-to-accelerator communication without CPU involvement. (2) We use zero-sided RDMA to implement a network-driven data shuffling operator for distributed DBMS use cases. We demonstrate that it achieves line-rate performance for different communication patterns (i.e., 1:1, N:M shuffles). (3) We showcase how complex communications schemes, such as globally-ordered replication and load-balancing for multi-producer to multi-consumer, can be implemented efficiently in zero-sided RDMA. (4) Finally, we evaluate zero-sided RDMA through a set of DBMS-centric scenarios, such as a distributed TPC-H query, shuffling, and a distributed join, as well as using zero-sided RDMA for two different types of accelerators: GPUs and FPGAs.

2 BACKGROUND

2.1 Remote Direct Memory Access (RDMA)

RDMA has become the state-of-the-art communication method for distributed data-processing systems over high-speed networks [5, 47, 49, 50, 54]. Its main benefit is that it removes the overhead of traditional kernel-space network stacks such as TCP/IP. Major cloud vendors have already adopted RDMA in their pursuit of faster networking with little CPU overhead. An example of this is Microsoft Azure which reports that already around 70% of internal ToR traffic is RDMA [2].

Communication schemes in RDMA can be categorized as one-sided (READ / WRITE) or two-sided (SEND / RECEIVE) operations, which refer to the involvement of the sender and receiver in the communication. For one-sided operations, only the sender is actively involved and thus has to decide where the data should be placed on the remote node. With two-sided operations, also the receiver is actively involved in the communication and decides where to place data by issuing RECEIVE requests before SEND requests can be issued on the sender side. This simplifies remote memory management.

Especially the one-sided RDMA operations have seen high adoption in distributed data processing systems since they allow sender nodes to write into remote memory directly fully bypassing CPU cores of the receiving nodes [4, 53]. However, even with one-sided RDMA, achieving direct accelerator-to-accelerator communication is often not possible or infeasible. As such, the communication control flow must be relayed over the CPU [42].

Finally, to understand our zero-sided RDMA contribution, which directly processes RDMA traffic in the network layer, we want to mention that RDMA communication can run over different transport types, such as Reliable Connection or Unreliable Datagram. The most commonly used transport is the Reliable Connection which requires stateful connections between any end-points. The reliability is provided through acknowledgments and re-transmissions. While Unreliable Datagram has the lowest complexity and overhead, most systems require reliable data transfers and need to implement it manually in the application with added CPU overhead [18, 19]. In addition, Unreliable Datagram only supports two-sided operations. In this paper, we exclusively use Reliable RDMA Connections, as one-sided operations are essential to achieving network-initiated communication, and reliability is vital for most systems.

2.2 Programmable Switches

Programmable switches have gained traction in data centers for many years. Unlike traditional fixed-function switches, they offer flexible, line-rate packet processing capabilities of up to billions of packets per second. These are primarily attributed to the programmable packet processing pipeline. In networking, the control plane and data plane are two fundamental components that handle different responsibilities. The control plane, usually running on a CPU, is the brain of the network device, making decisions about where traffic should be directed. In contrast, the data plane is responsible for to carry out actual operations on network packets and routing traffic to different destinations.

The programmability of the data plane is given by utilizing a reconfigurable architecture based on match-action tables, extending the utility of switches beyond data routing to include the offloading of application logic via customized match-action rules. The programming of this data plane predominantly employs P4 (Programming Protocol-Independent Packet Processors) [6], a high-level language that allows users to draft match-action rules to define packet processing in the data plane. Initially conceived for programmable network switches, P4 is now applicable to numerous systems that process packets, including SmartNICs and FPGAs. P4 programs, in general, manipulate packet headers and specify their rewriting while adopting a C-like syntax without allowing complex constructs such as pointers, floating-point numbers, or loops for line-rate processing. Besides these constraints, however, many new opportunities can be used when engineering new algorithms for this platform. The most important thing to note is that every compiled P4 program can run at line-rate in the switch, which is an important aspect of implementing zero-sided RDMA in the switch.

3 OVERVIEW OF ZERO-SIDED RDMA

3.1 Why Network-driven Communication?

Hardware accelerators are increasingly used for data-intensive processing tasks as they provide very high processing power compared to the stagnating performance of CPUs. As such, a lot of work is put into incorporating accelerators such as GPUs and FPGAs into DBMS [11, 38, 39]. However, when considering the trend of disaggregation and the adoption of fast RDMA-capable networks in data centers, the challenge of bringing these two trends together is of growing importance.

CPU-driven data shuffling. When looking at how distributed accelerator communication is typically done today, we see that it is predominantly CPU-driven. We illustrate this in ① in Figure 2a. Here, the CPU handles communication control flow while processing tasks are offloaded to the accelerators. This design makes sense because accelerators are typically built with specific processing tasks in mind and do not cope well with the control-flow-heavy type of communication handling, and in addition, the RDMA library stack is readily available and built for the CPU. However, this configuration causes a tighter coupling between the CPU and accelerator, which can hinder accelerator utilization due to stalling and CPU-to-accelerator communication overhead. At the same time, it increases the CPU load, which can be substantial considering the task of saturating fast networks and accelerator processing.

Accelerator-driven data shuffling. A common solution in related work to overcome the limitation of design ① is to implement the RDMA stack (i.e., `libibverbs` library) directly in the accelerators as illustrated in ② in Figure 2a. This allows the accelerators to directly execute RDMA and communicate over the network without CPU involvement [1, 8, 21]. While this approach has strong merits as it removes the strong coupling between the CPU and accelerator, we argue that in many cases, such a configuration might be suboptimal or even impossible for the following reasons: (1) As previously mentioned, accelerators are built with a specific processing architecture, e.g., the massively parallel SIMT execution model by GPUs. Therefore, they might perform very poorly on the control-heavy and often non-parallelizable task of coordinating the communication. (2) In many cases, the RDMA stack might not even be available for a particular type of accelerator. Developing a new RDMA stack requires high efforts that not only slow down the adoption of accelerators in data centers but might also consume a significant fraction of the available hardware resources on accelerators. This is in particular problematic for FPGAs since this limits how much resources are available for the actual application. (3) Realizing distributed operations, such as shuffling, with one-sided RDMA on an accelerator is non-trivial and requires the re-implementation of the same logic for each accelerator type leading to increased system complexity.

Network-driven data shuffling. For these reasons, it is a highly complex task to develop a communication system that relies on implementing the communication logic directly in the heterogeneous accelerators. We take a different approach by separating the data processing and communication logic by offloading the whole communication scheme into the network. This overcomes the challenges mentioned above by taking away the burden on the accelerator side and providing a clean separation of concerns. Moreover, with programmability in the network, CPUs can be removed from the critical path of communication between accelerators. Programmable network components typically come in the form of SmartNICs or programmable switches. While both types can be used to drive communication on behalf of the accelerator, a programmable switch has many benefits over SmartNICs. Among these reasons is guaranteed line-rate processing for all connected nodes and the possibility of realizing coordination-free communication schemes due to the centralized position of the switch. We further discuss this in Section 6.3.

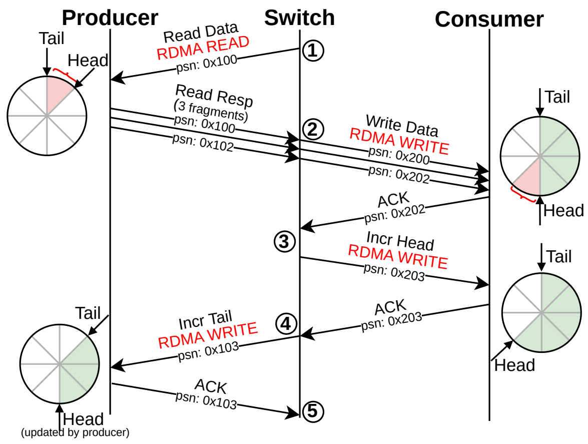


Fig. 3. Sequence of network packets for a switch-driven 1:1 data flow. The switch transfers data by reading from the producer and converting the READ response into a WRITE to persist the data in the consumer memory.

3.2 How Does Zero-sided RDMA Work?

Our approach to achieving zero-sided RDMA builds on the ability of the centralized programmable switch to initiate and direct data traffic directly in the data-plane at the aggregated line rate of all connected accelerators — called processing units (PUs) in the sequel. By placing the communication scheme in the switch data-plane, the switch can initiate one-sided RDMA operations (READ & WRITE) to PUs. At the same time, the PUs are completely oblivious to the network communication scheme, which not only has the benefit that the PUs do not have to issue communication primitives actively but also removes the distributed coordination for communication flows such as data shuffling or even replication.

Communication abstractions. On the side of the PUs (i.e., accelerators), a circular buffer is the core abstraction to participate in zero-sided RDMA data transfers. The design goal of the buffer is to allow PUs to push and pop items with only simple local memory operations while the switch transfers data fully asynchronously without any sender/receiver PU involvement. The coordination between the switch and PU is handled with two pointers to the buffer, a head and tail pointer. The head pointer indicates where the next data item can be written, and the tail pointer indicates where the next data item can be read.

Flow of data transfers. The overall flow of transferring a data item between one producer and one consumer using zero-sided RDMA is illustrated in Figure 2b. The switch mirrors the state of the producer and consumer buffers (head and tail pointers) to know when a producer has an item to send and whether the consumer has free space. To transfer an item, (1) the switch first issues an RDMA READ on data in the producer buffer. (2) The READ response is then converted on the switch into an RDMA WRITE and (3) written into the next free slot at the consumer. When converting a READ response into a WRITE request, the switch does not need to buffer or modify the data payload. (4) After the remote NIC acknowledges the RDMA WRITE, the head pointer of the consumer is incremented to indicate the new item. The tail pointer is incremented on the producer, which frees up the item in the buffer for reuse. We cover the detailed design in Section 4 to enable zero-sided data transfers.

Hardware requirements. Zero-sided RDMA can easily be used by many heterogeneous devices since only a few requirements must hold for participating in zero-sided RDMA: (1) the device must have memory in which to store the buffer data structure (2x 4 bytes for head/tail pointers and memory for data items), (2) the memory must be accessible by an off-the-shelf RDMA-enabled NIC and (3) the memory consistency model must ensure that a write to an item and the subsequent update to the head pointer is executed in order. In the case of weaker consistency models (like GPUs and ARM CPUs), memory fences can be used to ensure the ordering of writes. (4) In addition, to avoid stale reads, accelerators with caches must ensure that local reads and writes to the head and tail pointers of the buffer are directed to the RDMA-registered memory. For Nvidia GPUs this is achievable with the `volatile` keyword. We support accelerators with both little and big endianness. The switch processing pipeline operates in big-endian and as such to support little-endian systems, we perform endian conversion (if needed) on the switch in order to process the head and tail pointer values.

Software requirements. In order to enable zero-sided RDMA from the software side, it must be possible to register the accelerator memory of the zero-sided buffers to the RDMA-enabled NIC and establish an RDMA connection between the switch and PU. This is not a performance-critical task and only has to be done once in the setup phase. Hence, a rather small CPU is sufficient to carry out these management-related tasks. Important is that this CPU is not involved in the actual data transfers.

As an example, we discuss the details to enable zero-sided RDMA for an FPGA (or any other PCIe device): The memory of an FPGA can be exposed through a PCIe bar register and registered to the RDMA NIC. Registering external memory of PCIe devices as RDMA-enabled memory is supported by the `libibverbs` library using PeerDirect [32]. This requires a kernel driver that implements `peer_memory_client` [28] for the given device. With that, the RDMA NIC can directly access the FPGA's memory through PCIe without the involvement of the CPU or operating system using peer-to-peer DMA.

3.3 Integration into a DBMS

We now cover the basic steps needed to integrate zero-sided RDMA into DBMSs. The most important question is when a zero-sided communication flow between accelerators is set up and torn down in a DBMS. Moreover, other important aspects of cloud DBMS are the support for elasticity of flows as well as the use of zero-sided RDMA in different DBMS architectures.

Setting up and tearing down zero-sided flows. Here, different variants are possible. One variant is that the communication flows are instantiated when a DBMS cluster is launched, and different buffers are reserved, allowing a cluster node to send data to any other node. However, we also support variants where flows are instantiated and deployed ad hoc. In such a scheme, before query execution, a coordinator node (e.g., using a CPU) has to initialize the communication flow by setting up the buffers on the processing units. Subsequently, it initializes the communication flow by sending the buffer locations to the switch. During execution, we stress that no involvement is needed by the coordinator (i.e., no CPU is involved anymore). The switch handles the tear-down of flows. Once it detects that all producers are finished, it propagates the information to consumers and closes the RDMA connections.

Supporting elasticity in zero-sided flows. In order to support elasticity, we additionally allow producers and consumers to seamlessly be added or removed anytime from the communication flow during execution. We show an experiment for this in our evaluation in Section 7. The benefit of switch-driven communication is that changes to the communication flow will not incur any connection changes to already connected processing units. In fact, in e.g., a load-balancing communication flow adding another consumer at runtime will be completely oblivious to the producers.

Support for different DBMS architectures. Zero-sided RDMA integrates well into common DBMS architectures such as shared-nothing and shared-storage. For shared-nothing architectures, data is predominantly shuffled between nodes storing partitioned tables. Zero-sided RDMA can effortlessly interconnect processing units through the provided communication schemes while saving communication-related overhead and complexity at the connected nodes. Shared-storage architectures are gaining traction in the cloud due to their flexibility to scale in and out by placing tables on specific storage devices. This architecture can also benefit a lot from zero-sided RDMA in the way that it facilitates direct storage-to-accelerator communication.

4 SWITCH-DRIVEN DATA TRANSFERS

To realize zero-sided RDMA, we have to enable a switch-driven data transfer scheme that transfers data from producers to consumers without their active involvement.

Core challenges of realizing zero-sided RDMA. A core challenge of realizing zero-sided RDMA is to map the data transfer logic to the pipelined execution model of a switch, which provides a limited set of instructions and memory per stage. In addition, since all connected PUs are not directly connected to each other but are connected to the switch, the switch must adhere to the exact protocol to be compliant with off-the-shelf RDMA NICs. This includes managing stateful RDMA connections with reliability and correctly propagating congestion- and flow-control information from the consumers to the producers within the switch's data-plane to guarantee execution at line-rate.

The following explains how zero-sided RDMA can be implemented on the switch for a 1:1 data transfer. In subsequent sections, we will explain how to extend zero-sided RDMA to more complex flows (e.g., N:M flows or load balancing).

Overview of a 1:1 flow. The overall sequence of network messages for a data transfer from a single producer to a single consumer is depicted in Figure 3. The switch logic can be grouped into five steps: in ①, the switch evaluates whether it is possible to initiate a data transfer, next ②, the response of the read, which can be in multiple fragments, is rewritten into an RDMA WRITE to the consumer, in ③ the ACK from the data write triggers a new RDMA WRITE to increment the head pointer of the consumer buffer. In ④, the tail pointer of the producer is incremented, and the subsequent ACK packet ⑤ is recirculated back to the head of the switch processing pipeline to step ①. With that, the switch continuously evaluates whether it is possible to issue a data transfer for a given producer-consumer pair. Recirculation is a technique that virtually connects an output port in the switch with an input port, such that packets can pass through the pipeline multiple times. This resembles a looping construct. We initiate this main control loop per flow upon successful completion of the connection setup of the PUs. In the following, we present how we initialize the data transfer in step ① and rewrite the RDMA READ into an RDMA WRITE in step ②. Furthermore, we discuss which state is needed on the switch for achieving RDMA communication.

① Initiating data transfers. As a first step (① in Figure 3), the switch evaluates whether it is possible to initiate a data transfer. The control flow for this step is illustrated in Figure 4. The switch uses a pipelined processing model which operates on a per-packet basis and executes logic in a sequence of stages. By accessing registers and applying simple match-action-rules (i.e., actions can be arithmetic operations in each stage), we can express the logic needed to decide whether a data transfer can be initiated by letting a packet traverse the pipeline.

In the ingress pipeline, the switch first checks whether the producer has items in the buffer to send and whether the consumer has space. If neither the producer has data nor the consumer space available, the packet is recirculated. Recirculating the packet back to the ingress is necessary because while-loops are not supported in the pipelined processing model. We use unique indexes for all producers and consumers to identify a connection between a specific producer & consumer and to access their respective registers, e.g., pointer values and packet-sequence numbers (PSNs). PSNs are used to detect missing or duplicate packets. With that information, we can construct communication schemes between multiple producers and consumers. We apply adaptive batching

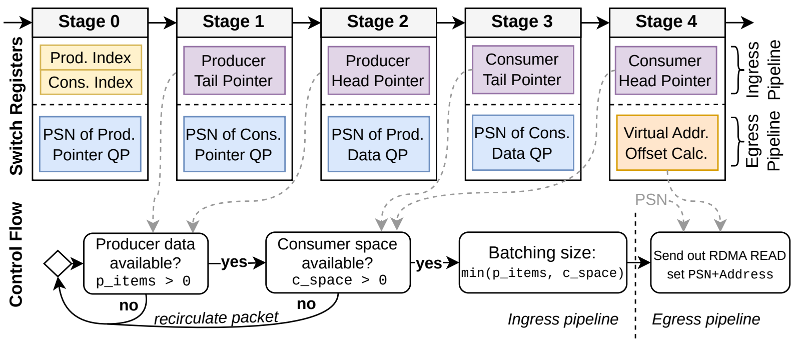


Fig. 4. Switch state needed to initialize a data transfer using a pipelined execution model in the switch. The steps of reading head/tail pointers of the producer and consumer, deciding on the batch size of elements to transfer, and sending the data need to be mapped to a sequence of stages in the switch. All steps are executed sequentially.

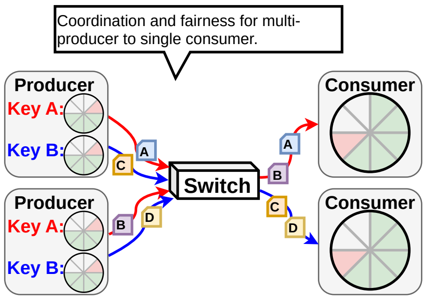
such that if a producer has generated multiple items that fit into the consumer, all the items will be transferred in one big RDMA READ. This optimization yields better efficiency as the overhead from control messages is relatively reduced. If these checks pass, the packet is forwarded to the egress pipeline, where necessary header fields are set to create an RDMA READ request which adheres to the RDMA protocol.

② **Data payload rewrite.** In the second step (② in Figure 3), the read response, containing the data payload to be transferred to the consumer, is streamed through the switch. Here the switch rewrites the response of the RDMA READ into an RDMA WRITE. Rewriting the RDMA operation type entails updating the relevant header fields and making it adhere to the correct packet sequence numbers and virtual address expected by the consumer. In terms of bandwidth consumption of the payload transfer from the producer to the consumer, rewriting the RDMA READ into an RDMA WRITE consumes the exact same bandwidth as a CPU-driven transfer would. Figure 3 step ② shows this rewrite by the switch for three fragment transfers of a payload.

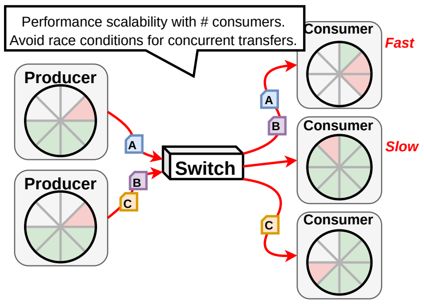
The read response might be fragmented into several packets if the payload exceeds 1024 bytes, as this is the default maximum payload size (MTU) for RoCEv2. The challenge here is that each packet from the read response must be assigned a new packet sequence number (PSN) according to the consumer connection. Instead of naively assigning new PSNs to the RDMA WRITE packets in the order they arrive, we calculate an offset between the producer-side and consumer-side sequence numbers such that our protocol is resilient to any network reordering of fragments (e.g., a fragment with PSN `0x101` arrives on the switch after fragment `0x102`). As such, the assigned PSNs on the consumer side will correctly map to the right fragments, allowing the target consumer-side NIC to assert the right order for the memory write.

Memory requirements on the switch. The programmable switch in zero-sided RDMA does not need to buffer data items in its memory. This is because data transfers are only initiated when there is enough free space in the consumer buffer in step ①. In fact, the switch only rewrites the headers of incoming packets in a streaming manner (at line rate) such that data items are immediately transferred to the consumer's buffer.

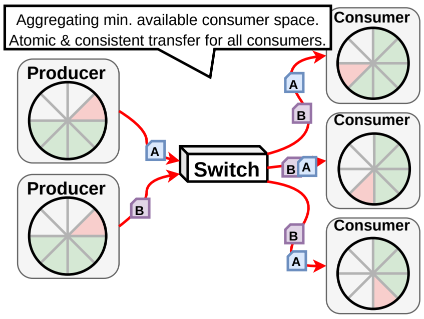
To enable zero-sided RDMA on the switch, we need to maintain the necessary metadata information in the switch. We can group the state needed on the switch into the static connection state which does not change during communication and the dynamically changing state. For RDMA, the static connection state includes a remote key and destination queue-pair number to identify the remote RDMA queue-pair, along with its IP and MAC addresses. Since none of these header fields change during the course of communication, we store this information in the static switch tables



(a) N:M Shuffle


Performance scalability with # consumers. Avoid race conditions for concurrent transfers.
Producer
Producer
Switch
Consumer
Fast
Consumer
Slow
Consumer
A
B
C

(b) N:M Load Balancing


Aggregating min. available consumer space. Atomic & consistent transfer for all consumers.
Producer
Producer
Switch
Consumer
Consumer
Consumer
A
B

(c) N:M Replication with Global Order

Fig. 5. Communication flows supported with zero-sided RDMA. All flows are completely switch-driven and require no processing or coordination by connected processing units. The core challenges per flow are highlighted in the text boxes.

upon connection setup. However, for the dynamically changing state, such as packet sequence numbers (PSNs) and virtual remote addresses, we need to change these values on a per-packet basis, and they need to be stored directly in registers on the data-plane. Since the switch is in the middle of the communication channel between the PUs, the switch must maintain PSNs and addresses for all connected end-points. In addition to the RDMA-specific state, we also store the buffers' head and tail pointer values in switch registers. This sums up to around 50 bytes in register memory and around 700 bytes of static table data for one producer-consumer pair connection. Concretely, more than 1500 concurrent flows can fit on the first-generation Tofino switch which is typically sufficient for most single-rack use cases. We estimate that with more recent programmable switches with longer pipelines and bigger state, the number of supported concurrent flows will be substantially higher.

5 COMPLEX FLOWS & USE CASES

5.1 N:M Data Shuffling

Data shuffling between multiple senders and receivers (i.e., an N:M shuffle) is very common in typical distributed DBMSs, due to the need for re-partitioning of tables in, e.g., distributed joins. However, these types of communication flows can be challenging due to the connection complexity of many sender/receiver combinations, which typically can even result in RDMA queue-pair cache thrashing on the NIC [9].

With our centralized switch design, instead, we can reduce the connection complexity significantly as each communication end-point is only connected to the switch. Specifically, the switch can connect multiple producers to one consumer by handling the coordination of multiple writers into the same consumer-side memory directly in the data-plane of the switch. As such, at the consumer side, only one connection is needed for all incoming data, which both reduces memory overhead and also has the benefit that the PU only has to poll one memory location for incoming messages for multiple producers. In a key-based shuffle flow with N producer end-points and M consumer end-points, where each producer needs to send to multiple destinations based on, e.g., a join-key, we need $N \times M$ producer-side buffers and connections. In contrast, on the consumer-side, only M buffers are needed. The composition of client-side buffers and connections is shown in Figure 5a.

To handle this coordination between multiple producers connected to one consumer, we implement switch-side logic, which initiates data transfers for each producer separately in a round-robin manner. While a round-robin scheme might seem inefficient at first glance, the switch can iterate over a single producer in a matter of nanoseconds. As such, the latency overhead of iterating each producer round-robin to detect when a data transfer can be initiated is negligible, considering normal RDMA microsecond network latency. Moreover, this scheme has the benefit that producers are treated fairly and ensures a fair bandwidth share among producers.

5.2 Advanced Flows & Features

Switch-driven data transfers can also provide more advanced flows (i.e., load balancing, replication) and features (i.e., fine-grained quality of service guarantees), which open up interesting applications in cloud DBMSs.

Load balancing. Distributing work evenly across processing units in a distributed DBMS is non-trivial, given unforeseen network congestion or processing contention. Multiple schemes have been devised to overcome this [10, 24, 52], which require additional coordination overhead, e.g., through work-stealing or a centralized server-side dispatcher.

With zero-sided RDMA, we provide a communication flow between N producers and M consumers, providing automatic load balancing across all consumers. This flow is realized without any form of producer- or consumer-side coordination since the switch will transparently initiate the data transfers between producers and consumers. The overall buffers and connections needed for this are only $N + M$ as illustrated in Figure 5b.

The key to achieving this communication flow is introducing consumer-specific thread-like processing on the switch, where it is evaluated when a data transfer can be initiated from any of the producers, essentially adapting the data transfer rate from all producers to each consumer independently. We provide more details on how we realized this in Section 6.1.

Replication with ordering. Lastly, zero-sided RDMA also provides a replication-based communication flow in which items from each producer are multicasted out to all consumers.

Multicast has many applications in distributed DBMSs, such as replicated joins [41], or state replication [13, 26] for providing availability. While RDMA already has multicast capabilities, it is only supported through the Unreliable Transport and two-sided verbs. As such, it comes with the cost of higher CPU overhead at the communication endpoints due to the two-sided communication and the cost of ensuring reliability. With zero-sided RDMA, we realize replication without any coordination or computational involvement on the processing units by initializing, steering, and multicasting the data directly in the data-plane of the switch.

Traditionally, effects like reordering of packets in the network can cause the received data at each consumer to observe a different order. However, as our zero-sided approach transfers data sequentially with separate acknowledgments from each consumer, we can ensure globally ordered data transfers to all consumers without introducing any overhead at the processing units.

Fine-grained quality of service. Finally, another important aspect is that many DBMSs wish to prioritize certain processing jobs or queries over unimportant background jobs, e.g., to provide certain SLAs. However, ensuring a certain quality of service (QoS) or prioritization of parts of the network traffic (e.g., for a time-critical query) is hard to achieve as the available Priority-based Flow Control (PFC) requires network reconfiguration and only provides a means of prioritizing different classes of traffic. As such, PFC does not allow setting fine-grained prioritization (e.g., for each zero-sided RDMA flow) and adapting these settings at runtime.

To address this, we allow systems to adjust the QoS on per-flow granularity. The key to achieving this is to throttle down any zero-sided data transfers to a desired rate through congestion control primitives native to RDMA over Converged Ethernet (RoCE). With this feature, splitting up the available bandwidth for contending flows into any ratio desired is trivial. We show the ability to do fine-grained flow prioritization later in Section 7.2.

5.3 Use Cases

Zero-sided RDMA offers DBMSs with easy-to-use flow abstractions that integrate well into typical DBMSs use cases. We iterate a set of common use cases that can be trivially realized with zero-sided RDMA over various hardware accelerators.

Distributed joins (OLAP). In analytical data processing, distributed joins account for a substantial amount of the runtime as they have to shuffle big tables over the network. Due to their popularity, they have seen a lot of work to accelerate with hardware devices such as GPUs and FPGAs in the last decade [7, 27, 39]. Through our key-based shuffle communication flow, we simplify the join implementation by abstracting away all communication complexity from the processing unit. As such, the only task for the processing unit is to execute the actual join logic, as the producer and consumer buffer abstraction only has a few simple reads and writes into local memory for pushing or popping items.

Scan operators (OLAP). Table scans with filters are an interesting operator to accelerate due to their computational simplicity but high bandwidth demands, making them a great fit for specialized accelerators that meet these demands. In disaggregated compute and storage setups, it is often desirable to push down the table scan as close to the data to reduce the amount of data to ship over the network. We argue that zero-sided RDMA fits this use case well since the CPU can be completely avoided on storage nodes, making it possible to directly interface with Near Data Processing devices which carry out the table scan on the storage servers [44].

Replication (OLTP & OLAP). Lastly, with the zero-sided replication flow, we trivially support use cases involving data replication. These entail database replication [20], state-machine replication [13, 23, 26, 45], or even joins that rely on replication of tables.

6 IMPLEMENTATION DETAILS

In this section, we take a closer look at how we realized the more complex communication flows involving load balancing or replication in N:M scenarios. We conclude with advanced challenges and a discussion of alternative zero-sided implementations.

6.1 Load Balancing and Replication

In the following, we take a look at a set of unique technical challenges when realizing load balancing and replication on the switch.

Many-to-many load-balancing. First, we present how we realize load-balancing between multiple producers to multiple consumers. This communication design aims to enable an even distribution of data to the consumers based on their individual processing speeds, without requiring any additional coordination or overhead at the processing units. We cannot simply extend the 1:1 scheme by alternating between different consumers because a single data transfer done by a switch thread can only saturate the bandwidth of a single consumer. Therefore we identify two challenges associated with this flow from the switch's perspective: *(1) ensuring efficient scaling in line with the number of processing units*, and *(2) preventing race conditions that could occur due to simultaneous data transfers*.

To provide efficient scaling, we extend the switch threading mechanism (i.e., continuous packet recirculation after data transfers) to not only transfer data between a specific producer-consumer combination but instead we spawn a switch thread packet per consumer which each iterate over all producers for the communication flow. With this, we can concurrently issue data transfers for each consumer. However, as previously mentioned, it comes with the challenge of avoiding race conditions in the producer-side switch registers. The reason for this is that the switch threads per consumer might try to issue a data transfer from the same producer at the same time, which would result in a race condition and inconsistent register state. To avoid this, we introduce a lock per producer such that only one switch thread can initiate a data transfer from a producer at any given time. We introduce this lock in a register of the switch pipeline and let each switch thread test-and-set the producer lock. If a switch thread fails, it will recirculate and try the next producer. If it succeeds, the lock will be taken, and a data transfer will be initiated for the producer and consumer pair.

By employing this design, each switch thread, specific to a consumer, will aim to read from all producers in a round-robin fashion. This ensures a fair distribution of data from all producers to each consumer. Additionally, it allows each consumer to process and remove data from its buffer at its own pace, independent of the other consumers.

Many-to-many replication. For the replication communication flow, the goal is to do switch-driven multicast to all consumers without any overhead or coordination at the processing units. The challenge here is twofold: *(1) how to accumulate the global minimum space available at all consumers* and *(2) how to ensure atomic & consistent transfer for all consumers*. In our implementation, the replication of packets to be multicasted happens between the ingress and egress of packets in the programmable switch. Therefore any logic and state needed to modify packets specifically for each consumer have to be placed in the stages of the egress pipeline.

Computing the global minimum buffer space available for all consumers cannot be trivially implemented using a for-loop over the head and tail registers because a packet can only access a specific register (e.g., all consumer head values) once per pipeline. To work around this constraint, we recirculate a packet $N - 1$ times to accumulate the minimum space over all consumers in the communication flow. The recirculating packet contains a temporary header that stores the current consumer index and the minimum space value between the pipeline passes. Once the head and tail values for the last consumer have been read, the switch asserts that the current producer has something to send and sends out the RDMA READ request. Each response fragment of the RDMA READ is then multicasted N times (step ② in Figure 3) and rewritten to RDMA WRITEs that target the appropriate consumer buffer memory. These N write requests generate N acknowledgments coming from each consumer connection. The switch then counts all acknowledgments before it increases the consumer-heads. We accumulate all acknowledgments to ensure that either all consumers see a new item or none of them in case a drop occurred for a data transfer. Upon receiving the last ACK from write data, the switch multicasts N times the Incr-Head RDMA WRITE to each consumer. After receiving N acknowledgments as before, the switch increases the tail from the current producer and starts the sequence again.

6.2 Further Challenges

The key to achieving zero-sided RDMA is that the PUs do not directly communicate; instead, all communicate with the switch. This introduces unique challenges such as handling of queue-pair state, ensuring reliability, and congestion control, which we will address in this section.

Buffer-state updates. One challenge of zero-sided RDMA is how to propagate the local buffer changes of PUs (i.e., when a producer pushes a new item or a consumer frees an item) to the switch. The switch uses this information to determine when new data transfers can be initiated. Here, there are two different mechanisms to consider: poll-based or doorbell-based switch state updates. In the poll-based mechanism, the switch polls the head and tail pointers of the PUs with RDMA READ operations issued by the switch. This has the benefit that neither the producer nor the consumer has to execute any RDMA operations actively. However, it naturally comes with the downside of added latency. To this end, we expose the polling interval as a configurable parameter as it is very application specific what the requirements for latency or throughput are. However, for PUs that already have the possibility of issuing RDMA WRITEs themselves (e.g. a network-enabled FPGA or CPU), the doorbell-based mechanism allows PUs to directly send out an update to the switch to indicate changes in their buffer state (i.e., new items or free space). One could contend that now the PUs could potentially handle all communication, taking charge of the communication scheme and data transfers. However, doing so would negate the advantages of network-driven data transfers, which include high-level communication flows such as load-balancing, replication, and the distinct separation between control and data paths. While the introduction of an additional doorbell mechanism might increase the complexity of the PUs it can significantly decrease communication latency, as demonstrated in Section 7.1.

Ensuring reliability. While the RDMA Reliable Connection transport already provides network reliability, since the switch is operating on the Data Link Layer (L2), any packet can be dropped in the network, and it is up to the switch to adhere to the RoCEv2 protocol [34] when this happens. In essence, any of the network packets illustrated in Figure 3 can be dropped, which, without any further action, would cause the data transfer to halt. If a drop occurs, the switch either detects it through a configurable timeout or gaps in the sequence numbers. If the drop happened during data transfers (i.e., step ②), the switch reissues the data transfer request (i.e., RDMA READ on the producer device). However, if the drop happened after receiving the ACK from the data transfer from the consumer, the transfer was successful, and it only reissues the subsequent RDMA WRITEs for the head and tail pointers.

Congestion control. Furthermore, to handle incast scenarios (e.g., two producer nodes with 2x bandwidth and a consumer node with 1x bandwidth), we integrate congestion control into our zero-sided communication scheme on the switch. The switch emits explicit congestion notifications whenever a link becomes congested, similar to RoCEv2. When the NIC of the producer receives the congestion notification, it throttles down the rate of outgoing packets, removing the congestion bottleneck.

6.3 Discussion

Offloading the network communication from the hardware processing units can be realized in different ways. We now discuss the centralized or decentralized approaches, compare their strengths and weaknesses, and subsequently touch on the possibility and challenges of single vs. multi-rack setups.

Decentralized SmartNIC-driven approach. Instead of using the main CPU to carry out the communication, an approach is to use a SmartNIC to issue RDMA operations on behalf of the accelerator, such as the CPU-based Nvidia BlueField or FPGA-based Mellanox Innova SmartNICs, or even FPGA-based SmartNICs that implement the RDMA stack directly in the FPGA [21, 46].

The benefit is that a specialized compute unit embedded in the NIC carries out the communication instead of the main CPU. Concrete examples of this approach are Lynx [42] and FpgaNIC [46]. Lynx and FpgaNIC aim to enable direct GPU communication by letting the GPU communicate without any CPU involvement or need for RDMA operations directly on the GPU.

However, a decentralized SmartNIC-driven approach has several downsides. First, realizing one-sided RDMA communication schemes is highly complex due to the distributed coordination for remote memory accesses. Adding to the fact that typical CPU-based SmartNICs are too weak to saturate line-rate throughput for many scenarios properly [40], SmartNIC-driven communication schemes are best realized on more performance-efficient compute architectures such as FPGAs. This, in turn further adds to the complexity when considering decentralized one-sided communication schemes. Second, since a dedicated SmartNIC is needed per server, the hardware cost is directly proportional to the number of servers. As the cost of a SmartNIC can be up to 10× more expensive than a normal RDMA-enabled NIC for the same link speeds, the added cost for a disaggregated solution is non-negligible.

Centralized programmable switch-driven approach. We argue that a switch-driven design of direct accelerator communication is preferable. The reasons for this are: First, a programmable switch provides the unique possibility of doing line-rate processing for all connected nodes. In terms of realizing a zero-sided RDMA communication scheme, the switch can natively scale out to full link throughput for all connected devices without hitting any performance bottlenecks. Switch-driven communication is different from, e.g., CPU-driven communication, where with increasingly faster networks, saturating the throughput of just one link becomes increasingly difficult. Second, the centralized position of the switch in the network allows us to realize communication flows (as covered in Section 5) without expensive distributed coordination. Lastly, offloading the communication onto a programmable switch with many ports results in a cheaper overall hardware cost for typical scale-out solutions compared to decentralized per-server solutions.

Data center deployments. In many cloud deployments, the jobs for query processing are scheduled on compute nodes within a single rack connected by a Top-of-the-Rack (ToR) switch to increase locality [15, 48]. This is crucial for distributed operators (e.g., joins) to have high all-to-all shuffle bandwidth.

In this paper we focus on a single-switch setup to mirror a typical rack setup but cross-rack communication is supported by zero-sided RDMA out of the box. Regular RDMA communication between end hosts (and initiated by end hosts) can be used across racks, traversing multiple switches. This is the same for switch-initiated RDMA traffic. Nontheless, there are several aspects left for future investigation: e.g., the added latency between the switch and connected accelerator devices, asymmetric bandwidth, and possible implications to congestion control. Moreover, with multi-switch setups, we envision the possibility of graceful failover between switches in case of failures. This is possible since the runtime state stored on each switch, e.g. fill-levels of the buffers and connection state, can be recovered by reading them out from participating PUs.

Finally, a limiting factor for many in-network processing ideas is that data streams are often encrypted in data centers, making payloads opaque to the processing element. This is however not an issue for zero-sided RDMA: the switch does not need to read or modify the payload of the transferred data items, but merely modifies the header and forwards the packets.

7 EXPERIMENTAL EVALUATION

With our zero-sided RDMA contribution, we mainly target distributed analytical workloads, which is the focus of the evaluation. We first evaluate the efficiency of zero-sided RDMA through a set of microbenchmarks that vary in different performance-related parameters. Subsequently, we investigate the benefit of switch-driven data transfers in the context of distributed database systems by evaluating two DBMS use cases (i.e., a distributed join and a full TPC-H query) using a disaggregated accelerator setup.

Setup and implementation. We evaluate our zero-sided RDMA communication scheme in a cluster of 4 nodes running Ubuntu 18.04 LTS with Linux kernel 4.15.0. Each node is equipped

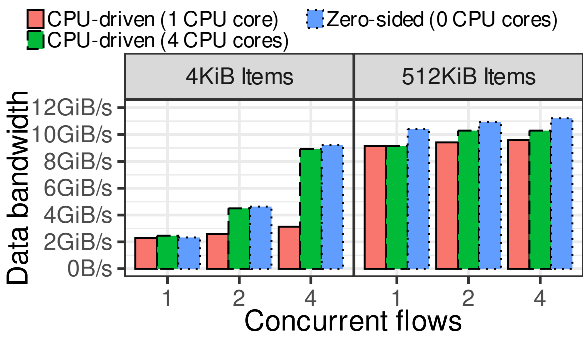


Fig. 6. GPU-to-GPU transfer: Zero-sided RDMA vs. a CPU-driven one-sided RDMA baselines for 1:1 GPU data transfer with persistent GPU kernels. The CPU-driven baselines use 1 or 4 CPU cores; our approach does not use the CPU.

with an Intel(R) Xeon(R) Gold 5120 2.2GHz CPU, an Nvidia V100 GPU, and a Mellanox ConnectX-5 that is connected to an Intel Tofino switch [30] via 100G RoCEv2. Our code written in C++20 is compiled with gcc-12 and CUDA-12. The switch's control-plane logic is implemented in C++, and the switch's data-plane logic is implemented in P4 and compiled using Intel SDE-9.11.0 [14]. For all experiments, unless otherwise stated, the polling-based mechanism is used for detecting new data items on producers and fill-grade on consumers. The source-code is available at [16].

7.1 Efficiency of Zero-sided RDMA

Zero-sided vs. CPU-driven transfer (GPU-to-GPU). In this experiment, we focus on accelerator-to-accelerator communication across two nodes. As accelerators in this experiment, we use a homogeneous setup (GPU-to-GPU) to provide the same execution speeds on all senders and receivers. At the end of the evaluation, we also show heterogeneous setups with GPUs and FPGAs.

The data transfers for all variants in this experiment (i.e., CPU-driven and zero-sided RDMA data transfers) are all directly from GPU-to-GPU without intermediate copies over the main memory. Moreover, in both the CPU-driven baseline and our zero-sided communication, we execute the exact same GPU kernels and use the same buffer abstraction on the GPUs. However, for the CPU-driven baseline, we let the CPU detect new items in the buffer and issue the RDMA data transfers on behalf of the GPU. The GPU kernels are executed as persistent kernels to minimize any kernel launch overhead for both variants.

Figure 6 shows the bandwidth between CPU-driven and network-driven communication in our experiment. The CPU-driven baseline issues asynchronous RDMA data transfers and utilizes persistent GPU kernels to reduce synchronization overhead between the CPU and the GPU. We report the data bandwidth for varying item sizes. In addition, we test with a different number of parallel producer and consumer communication flows to evaluate different communication loads of realistic scenarios (e.g., multiple concurrent queries running at the same time or one query using multiple flows for intra-query parallelization).

For small item sizes (4 KiB), we can see that multiple CPU cores are necessary to achieve comparable performance to the zero-sided approach when increasing the concurrent flows. In fact, with small item sizes, one CPU core is not sufficient to handle all flows (see the red bar in Figure 6, left-hand-side). This indicates that even for persistent GPU kernels that overcome the GPU kernel launch overhead data transfers are inherently CPU bound. In our zero-sided approach, the switch instead handles all flows completely independently and as such scales perfectly with the number of flows without the need to use a dedicated CPU per flow as in the CPU-driven scheme (red and green bars). Moreover, for the CPU-driven scheme and zero-sided RDMA, another effect is

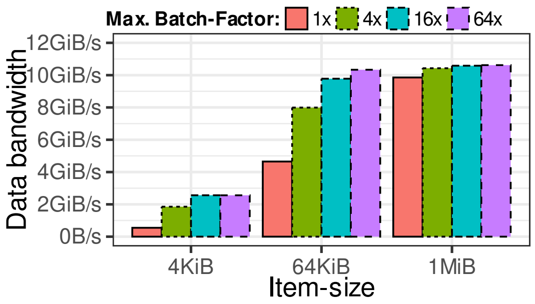


Fig. 7. GPU-to-GPU transfer: 1:1 GPU data transfer with adaptive batching for different tuple sizes. Bigger batch sizes yield more efficient data transfers.

that the communication overhead is reduced for larger item sizes, which enables higher transfer bandwidths with fewer concurrent transfers.

Adaptive batching. In the next experiment, the main objective is to understand how adaptive batching implemented in the switch can improve the performance of zero-sided RDMA. This optimization mitigates the overhead of smaller transfers by seamlessly grouping multiple items into a single transfer. We measured throughput across a variety of batching factors and three distinct item sizes: 4 KiB, 64 KiB, and 1 MiB. The results are depicted in Figure 7 and show how the data bandwidth is influenced by item-size and batch-size. The results reveal that the use of adaptive batching can minimize the overhead of smaller transfers. However, the effect of batching becomes less pronounced as the item-size increases. This is because a data transfer only with one large item is enough to almost saturate the network. The performance still increases slightly with larger batches as the overhead of buffer head and tail pointer updates are amortized further. However, choosing a too-large batch size can have detrimental effects on data transfer latency, especially in scenarios with network congestion. The reason for this is that larger data transfers will consist of a higher number of fragments, increasing the probability that one fragment is lost and the data transfer has to be reissued. We found that a batching size no larger than 4 MiB can help to saturate the available network bandwidth even for just one producer and consumer, but at the same time performs very stable under congestion.

Comparison with CPU-less shuffling (GPU-to-GPU). Shuffling data between nodes is crucial for many distributed database operators such as joins or aggregation. In Figure 8, we report the data bandwidth for a GPU-to-GPU shuffle scenario where we compare against NVSHMEM (version 4.0.2) [33]. NVSHMEM is the only commercial library that supports GPU-initiated data transfers to remove the CPU from the control-path and is as such a comparable baseline to zero-sided RDMA. For the experiment, we use the non-blocking PUT primitive of NVSHMEM for efficient one-sided data transfers. We use item sizes of $512KiB$ and execute 2 shuffle kernels on each GPU node. After launching the GPU kernels, no CPU cycles of the host system are spent on communication primitives for either zero-sided RDMA or NVSHMEM. Both setups use RDMA (RoCE). For simplicity, we compare against NVSHMEM without any distributed coordination and use a non-skewed workload and thus report the best-case bandwidth for NVSHMEM.

As can be seen in Figure 8, the reported bandwidth is almost equal since both NVSHMEM and zero-sided RDMA manage to fully utilize the link bandwidth. However, an important difference between zero-sided RDMA and NVSHMEM is that while zero-sided RDMA can integrate many different types of heterogeneous accelerators, NVSHMEM works exclusively for GPU-to-GPU transfers. Furthermore using NVSHMEM requires a decentralized protocol between GPUs to coordinate data transfers which is provided in zero-sided RDMA by the switch along with other functions for load balancing not available in NVSHMEM.

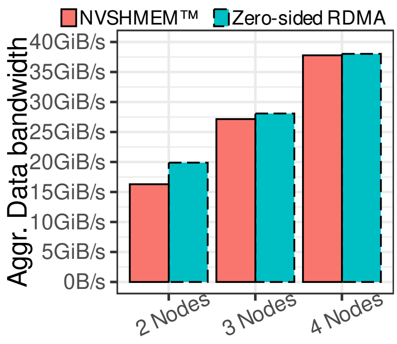


Fig. 8. GPU-to-GPU shuffle: Comparison of zero-sided RDMA GPU-to-GPU shuffle bandwidth with NVSHMEM [33], that only allows GPU-to-GPU communication and requires custom coordination.

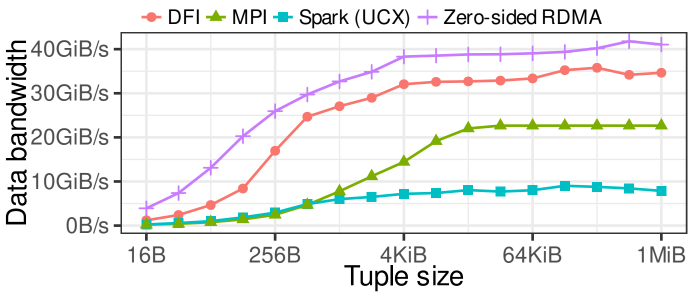


Fig. 9. CPU-to-CPU shuffle: Comparison of RDMA-based data shuffling with zero-sided RDMA vs. CPU baselines (DFI, MPI, Spark UCX) for various tuple sizes.

Comparison with existing CPU baselines (CPU-to-CPU). In this experiment, we shift our attention from a GPU-to-GPU to a CPU-to-CPU shuffle (i.e., data resides in CPU memory), in order to show that zero-sided RDMA can also be beneficial in such a setting. In particular, we compare zero-sided RDMA against three CPU-based baselines for CPU-to-CPU data shuffling: (1) Spark (UCX) [35] which is a baseline for RDMA-based data shuffling in a production-ready system (Apache Spark v3.0 with UCX Plugin), and two isolated CPU-based data shuffling baselines - (2) a DFI-based shuffle [41] as well as (3) an MPI-based shuffle that uses MPI collectives (Nvidia HPC-X v2.17) for shuffling [12, 29]. DFI and MPI are both state of the art open-source libraries that enable data shuffling using RDMA. In this experiment, we report the bandwidth of a data shuffle between 4 nodes for varying tuple sizes. On each node, we use four worker threads (cores) as data producers/consumers for the shuffle for zero-sided RDMA and all baselines.

As shown in Figure 9, zero-sided RDMA achieves constantly higher throughput than the others. The reason for this is twofold: First, zero-sided RDMA frees up sender/receiver threads from issuing RDMA primitives and thus CPUs can use their cycles solely for producing/consuming data. Second, zero-sided RDMA applies adaptive batching on the network level leading to transfers that are efficiently larger than the tuple sizes, which further improves the bandwidth usage.

Ping-pong latency (polling vs. doorbell). Finally, in the last experiment of this section, we focus on transfer latency. We first analyze on the impact of switch polling intervals in the context of buffer updates (i.e., new items pushed or popped at the PUs). As these updates are detected on the switch via polling the buffers at the PUs, the duration of polling intervals can potentially influence the transfer latency of items. Understanding this relationship can assist in determining the optimal polling intervals for various applications with different requirements for throughput and latency. In the experimental setup, we report half round-trip time on the y-axis across various

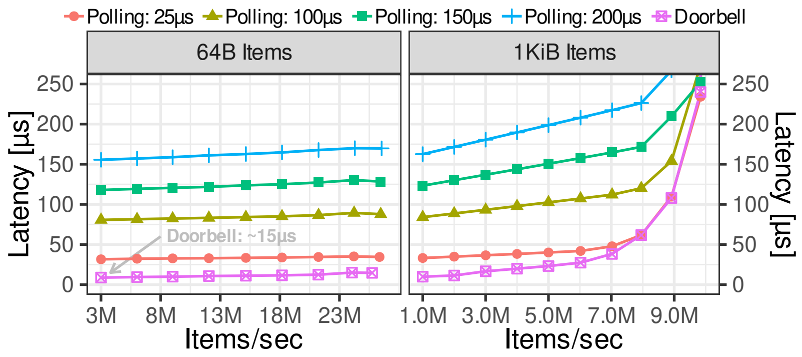


Fig. 10. One-way median latency over different throughput and polling intervals. In "Doorbell" mode PUs actively signal the switch for available data.

polling intervals, denoted by different line symbols. We use two CPU nodes as PUs, each with both a producer and consumer, which the switch connects to form a bi-directional communication flow to measure the round-trip time.

Additionally, we analyze the effect of the doorbell mechanism in zero-sided RDMA. Clearly, polling for state updates using the switch enables data transfers without any involvement from the PUs. However, this potentially comes with the cost of increased latency. Zero-sided RDMA thus also allows a PU to actively signal the switch when it has data available for transfer, referred to as doorbell mode. To send out a doorbell message, the PU simply sends out a network packet, containing the new buffer state, signaling an update to the switch. If the PU lacks this capability, a small co-located FPGA could also be used to perform this task.

The experimental results of our polling and the doorbell mechanism are illustrated in Figure 10 for 64B and 1KiB item sizes. The transfer rate, specified in items per second, is fixed to certain values denoted on the x-axis. As we see in Figure 10 for the polling scheme, the median latency is slightly less than the polling interval. This is because, on average, an update to the producer buffer is detected in half of the polling interval plus the latency of the data transfer. Enabling the doorbell mechanism, the median latency is reduced to $\sim 15\mu s$ since the producer can immediately signal the switch that an item can be transferred. This optimization is, however, only possible on PUs with RDMA capabilities as discussed before in Section 6.2.

Important is that latency remains stable across various throughputs, demonstrating consistency in performance irrespective of the transfer rate. For 64 B item sizes, the throughput scales up to around 27 million items per second due to the adaptive batching optimization. However, for the larger 1 KiB sizes, the latency stays stable up to around 10 million items per second which almost matches the link's capacity resulting in full buffers and high latency. Interestingly, for the slower polling intervals, e.g., $200\mu s$, the higher rate of pushing items to the producer buffer results in more items being batched for each data transfer resulting in a slightly higher latency.

7.2 Benefits of Switch-driven Data-transfers

Utilizing network-driven data transfers from a centralized switch provides significant advantages. Given its overarching view of all network traffic, the switch allows for streamlined management of data transfers. In particular, we enable fine-grained Quality of Service (QoS) via flow prioritization, facilitate load-balancing, and promote elasticity for efficient resource scaling. These benefits, challenging to achieve in decentralized data-flow solutions, underscore the value of centralized, network-driven data transfers.

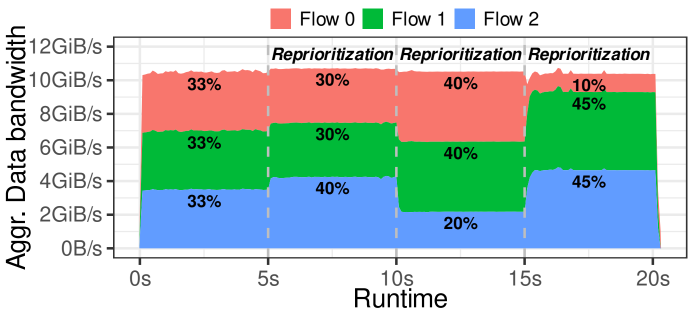


Fig. 11. QoS flow prioritization between two PUs with 3 concurrent flows. Prioritization changes every 5 seconds.

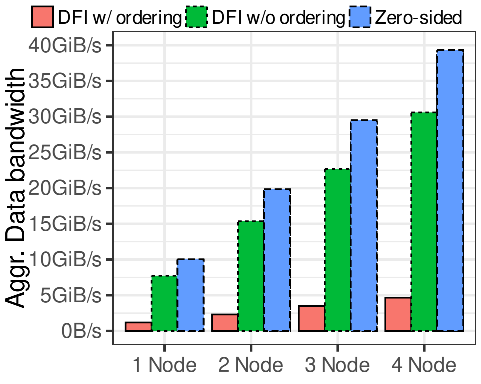


Fig. 12. 1:N zero-sided replication compared to DFI [41] with or without global ordering. Zero-sided replication always ensures a global order observed by all consumers.

Fine-grained flow prioritization. In this experiment, we investigate the fine-grained (per flow) prioritization capabilities provided by our zero-sided communication scheme. Figure 11 shows the individual data bandwidth for three active concurrent flows between two nodes over 20 seconds, with the ratio altered every 5 seconds. Our system demonstrates a swift and efficient response to changes in the prioritization of different flows, with the processing units remaining oblivious to the adjustments. Centralized initiated data transfers allow fine-grained prioritization of different flows, which is not supported natively within RDMA.

Replication with ordering. The subsequent experiment, presented in Figure 12, centers on the effectiveness of zero-sided RDMA replication which by design ensures global ordering for each consumer. For the baseline, we use the Data Flow Interface (DFI) [41], a state-of-the-art library that enables replication using RDMA multicast with two-sided SEND/RECEIVE operations. We present results for DFI both with and without software-based ordering. For the experiment, we use CPUs as PUs to be able to compare against the CPU-only DFI baseline.

The results show the aggregated data bandwidth of each consumer for a different number of nodes that receive data from a single producer. Across all scenarios, zero-sided replication with ordering consistently demonstrates superior performance, nearly saturating the link bandwidth at each consumer node. In addition, zero-sided replication does not infer any overhead at the PUs. In stark contrast to native RDMA multicast that only supports two-sided SEND/RECEIVE operations and, as such, introduces communication overhead at not only the sender but also the receiver.

Elasticity of flows (TPC-H Query 1). In this experiment, we demonstrate the elasticity of our system. For this experiment, we execute TPC-H Query 1 in a disaggregated setup. In particular,

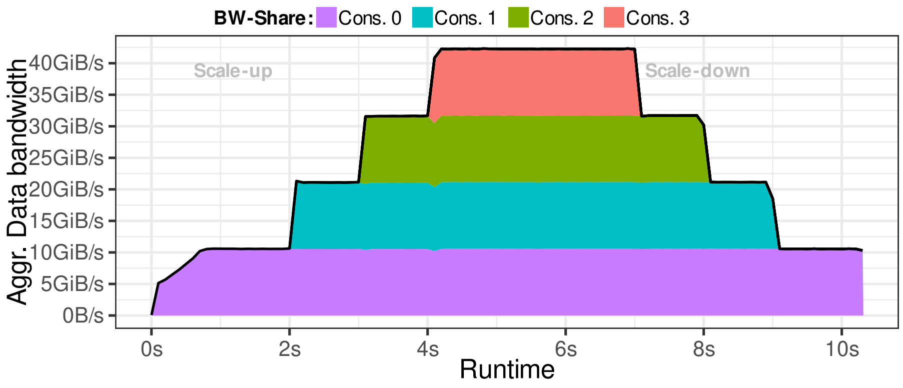


Fig. 13. Elasticity for 4:N Load balancing flow - TPC-H Query 1 with SF 100 executed on GPUs. Each producer PU is unaware of the elasticity.

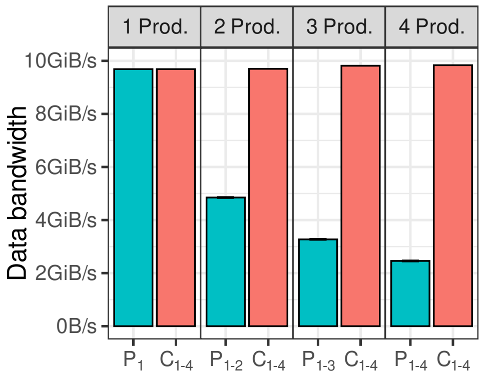


Fig. 14. Per producer & consumer bandwidth for multicast with an increasing number of producers (from left-to-right) and four consumers (C1-4). P1-2 refers to a scenario with two producers.

storage nodes (using CPUs) stream data to GPUs as processing units in the compute layer. We scale up the compute layer from one to four nodes (each having one GPU).

Figure 13 represents the aggregated bandwidth of each consumer in an area plot, with each consumer distinguished by different colors. This experiment shows that the processing capability can dynamically be scaled up from a single node to four nodes and scaled down when required. Important is that the producer nodes (i.e., the storage) remain completely unaware of the compute-layer elasticity. The entire coordination of this scaling operation is managed by the switch. This experiment illustrates the ability of our system to efficiently adjust and manage resource allocation dynamically, offering significant potential for enhanced scalability and efficiency in distributed accelerator environments.

Fairness between producers. In this next experiment, we explore the fairness of bandwidth distribution among multiple producers, each producing data to the same multicast flow. Producer fairness is crucial to ensure producers are given an equal share of the consumer-side available link bandwidth. In Figure 14, we report the bandwidth separately for producers and consumers. Important to see is that in every scenario (1-4 producers from left to right), all consumers receive a fair share of the bandwidth. When more producers are added to the system, the bandwidth is divided evenly among them. The barely noticeable error bars in the figure for the producers denote the low degree of unfairness, emphasizing the fairness between multiple producers.

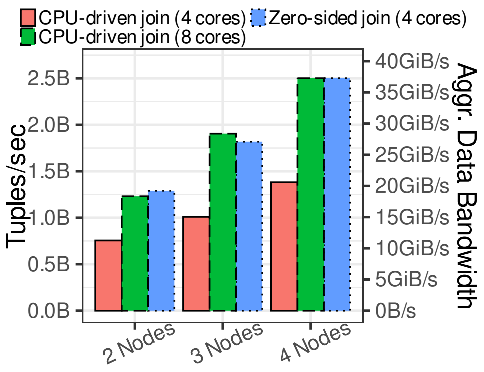


Fig. 15. Distributed Join. Tables are shuffled from CPU-based storage nodes to GPU nodes for joining. Zero-sided RDMA achieves better performance with fewer CPU cores compared to [39].

Distributed join. Distributed joins represent a fundamental and often performance-critical operation in DBMSs. In light of this, we evaluate the performance of a distributed join using zero-sided RDMA. We mirror a typical disaggregated setup, where tables are residing on CPU-based storage nodes and are shuffled on the join key to GPUs for join processing. As a baseline, we use a state-of-the-art pipelined GPU join implementation where the shuffling is CPU-driven [39]. Both the zero-sided join and the baseline implement the same optimizations for shuffling such as software write combine buffers (SWWCBs) and non-temporal streaming hints [36**?**], We use two synthetic and uniformly distributed relations with 16-byte tuples of sizes 10M and 10B for the workload.

We apply an N:N shuffle flow and measure the throughput in tuples/sec for configurations with up to 4 nodes. The experiment includes both the build phase for the hash table and the probe phase. Similar to the baseline implementation, we overlap the shuffling with the building or probing such that the join is executed overlapped on the GPUs [39]. For the zero-sided join, we use 4 producers and 4 consumers per node, totaling up to 256 producer buffers and 16 consumer buffers for 4 nodes. Each producer pushes tuples based on the join key into separate buffers from which they are transferred to the corresponding consumer using zero-sided RDMA. In the CPU-driven GPU join baseline we report numbers for 4 and 8 active CPU cores shuffling to 4 GPU buffers per node.

Our results in Figure 15 showcase a near-linear speed-up for the zero-sided RDMA join as the number of nodes increases and performance that is on par with the baseline. As an important effect, we additionally see the benefit of offloading the communication scheme into the network and thereby freeing up CPU resources by letting the CPU-based storage nodes focus solely on table scanning and shuffling. This is apparent since the zero-sided RDMA join only needs 4 CPU cores to achieve comparable performance to the CPU-driven join baseline.

7.3 Heterogeneous Communication

Scheduling processing jobs across a wide range of heterogeneous processing devices with different and varying throughputs is a challenging task that requires careful coordination between all participants. Therefore in this section, we evaluate the load-balancing communication flow in a setup that mimics a cloud data center with disaggregated storage and heterogeneous compute resources that consist of CPU, GPU, and FPGA.

Load balancing (TPC-H Query 1). In this experiment, depicted in Figure 16, we explore load-balancing for a disaggregated setup using a heterogeneous computing environment comprising a CPU, a GPU, and an Xilinx Alveo U55C FPGA as consumers. We expose the FPGA's local HBM memory to the RDMA NIC through the PCIe bar for peer-to-peer DMA (see Section 3.2). Data

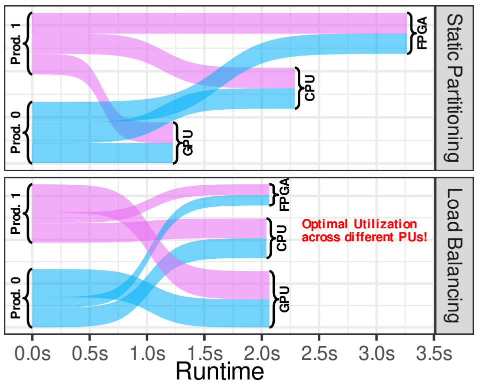


Fig. 16. Static partitioning vs. load balancing for TPC-H Query 1 with SF 100 executed on CPU, GPU and FPGA. The thickness of the lines indicates the speed of the consumers.

producers on the storage are spread out on 2 (CPU) nodes. The workload is based on TPC-H Query 1, as in the elasticity experiment before. The system's performance is assessed under two distinct scenarios: static partitioning and load balancing.

In the scenario with static partitioning, the input TPC-H table *lineorder* is partitioned into two equal sizes at both producers which send the data in equal portions of the data to three consumers: an FPGA, a GPU, and a CPU. Without load-balancing (Figure 16, upper part) in the switch, the GPU completes its tasks more rapidly than the CPU, which is faster than the FPGA as the slowest device. Overall, the FPGA takes in total 3.2 seconds, which dominates the end-to-end latency of the query.

In contrast, with load balancing (Figure 16, lower part), the *lineorder* table is dynamically divided to each consumer based on each device's processing speed. This arrangement enables the CPU, FPGA, and GPU to finish their tasks simultaneously, thus reducing the total runtime of the query to 2.1 seconds and ensuring optimal utilization with no idle time for either processing unit.

These results underscore the potential of load balancing in zero-sided RDMA: it allows for optimal utilization of different processing units without the need for complex partitioning or work-stealing schemes, leading to more simple query execution code.

8 CONCLUSIONS

In this paper, we presented zero-sided RDMA as a way to enable direct RDMA-based accelerator-to-accelerator communication, which does not require CPUs to coordinate the communication because the communication scheme is driven from the network. Moreover, with zero-sided RDMA, we enable efficient RDMA-based data shuffling between heterogeneous hardware devices without the need to implement a complete RDMA stack on each heterogeneous device. Our evaluation showed that zero-sided RDMA can outperform CPU-driven one-sided RDMA schemes for accelerators and in addition, provide a set of useful and efficient communication flows targeting disaggregated cloud DBMSs.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037) and the DFG Collaborative Research Center 1053 (MAKI). We also thank hessian.AI, 3AI, DFKI as well as Intel for their support.

REFERENCES

[1] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2018. GPUDirect Async: Exploring GPU synchronous communication techniques for InfiniBand clusters. *J. Parallel Distributed Comput.* 114 (2018), 28–45. https://doi.org/10.1016/j.jpdc.2017.12.007

[2] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert G. Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[3] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1463–1475. https://doi.org/10.1145/2723372.2750547

[4] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1463–1475. https://doi.org/10.1145/2723372.2750547

[5] 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 (2016), 528–539. https://doi.org/10.14778/2904483.2904485

[6] Mihai Budiu and Chris Dodd. 2017. The P416 Programming Language. *ACM SIGOPS Oper. Syst. Rev.* 51, 1 (2017), 5–14. https://doi.org/10.1145/3139645.3139648

[7] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA Useful for Hash Joins?. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[8] Feras Daoud, Amir Wated, and Mark Silberstein. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers, Kyoto, Japan, June 1, 2016*, Kamil Iskra and Torsten Hoefler (Eds.). ACM, 6:1–6:8. https://doi.org/10.1145/2931088.2931091

[9] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[10] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *Proc. VLDB Endow.* 12, 12 (2019), 2218–2229. https://doi.org/10.14778/3352063.3352137

[11] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2020. In-memory database acceleration on FPGAs: a survey. *VLDB J.* 29, 1 (2020), 33–59. https://doi.org/10.1007/s00778-019-00581-w

[12] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. 2014. *Using advanced MPI: Modern features of the message-passing interface.* MIT Press.

[13] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. 1999. The Performance of Database Replication with Group Multicast. In *Digest of Papers: FTCS-29, The Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999.* IEEE Computer Society, 158–165. https://doi.org/10.1109/FTCS.1999.781046

[14] Intel. 2023. Intel P4 Studio. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html.

[15] Virajith Jalaparti, Peter Bodík, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye (Eds.). ACM, 407–420. https://doi.org/10.1145/2785956.2787488

[16] Matthias Jasny and Lasse Thostrup. 2023. Zerosided RDMA Code. https://github.com/DataManagementLab/zerosided_rdma.

[17] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. 2022. P4DB - The Case for In-Network OLTP. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1375–1389. https://doi.org/10.1145/3514221.3517825

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[20] Bettina Kemme and Gustavo Alonso. 2010. Database Replication: a Tale of Research across Communities. *Proc. VLDB Endow.* 3, 1 (2010), 5–12. https://doi.org/10.14778/1920841.1920847

[21] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do OS abstractions make sense on FPGAs?. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 991–1010. https://www.usenix.org/conference/osdi20/presentation/roscoe

[22] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. 2022. Bandwidth-optimal Relational Joins on FPGAs. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). OpenProceedings.org, 1:27–1:39. https://doi.org/10.5441/002/edbt.2022.03

[23] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. 2021. RamCast: RDMA-based atomic multicast. In *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga (Eds.). ACM, 172–184. https://doi.org/10.1145/3464298.3493393

[24] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 743–754. https://doi.org/10.1145/2588555.2610507

[25] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 355–370. https://doi.org/10.1145/2882903.2882949

[26] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 467–483. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[27] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1633–1649. https://doi.org/10.1145/3318464.3389705

[28] Mailinglist. 2023. [RFC 6/7] IB/core: Peer memory client for IO memory. https://www.spinics.net/lists/linux-rdma/msg33298.html.

[29] MPICH. 2023. Manpage: MPI_Alltoall. https://www.mpich.org/static/docs/latest/www3/MPI_Alltoall.html.

[30] APS Networks. 2021. Intel Tofino APS Networks BF2556X-1T-A1F. https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF2556X-1T_V04.pdf.

[31] Joel Nider and Alexandra (Sasha) Fedorova. 2021. The last CPU. In *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, Sebastian Angel, Baris Kasikci, and Eddie Kohler (Eds.). ACM, 1–8. https://doi.org/10.1145/3458336.3465291

[32] NVIDIA. 2023. HowTo Implement PeerDirect Client using MLNX_OFED. https://enterprise-support.nvidia.com/s/article/howto-implement-peerdirect-client-using-mlnx-ofed.

[33] NVIDIA. 2023. Nvidia NVSHMEM. https://developer.nvidia.com/nvshmem.

[34] NVIDIA. 2023. RDMA Over Converged Ethernet (RoCE). https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=56986516.

[35] OpenUCX. 2023. SparkUCX ShuffleManager Plugin. https://github.com/openucx/sparkucx.

[36] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 351–362. https://doi.org/10.1145/1807167.1807207

[37] Amazon Web Services. 2023. Elastic Fabric Adapter. https://aws.amazon.com/hpc/efa/.

[38] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1617–1632. https://doi.org/10.1145/3318464.3380595

[39] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM Manag. Data* 1, 1 (2023), 29:1–29:26. https://doi.org/10.1145/3588709

[40] Lasse Thostrup, Daniel Failing, Tobius Ziegler, and Carsten Binnig. 2022. A DBMS-centric Evaluation of BlueField DPUs on Fast Networks. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–10. http://www.adms-conf.org/2022-camera-ready/ADMS22_thostrup.pdf

[41] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2022. DFI: The Data Flow Interface for High-Speed Networks. *SIGMOD Rec.* 51, 1 (2022), 15–22. https://doi.org/10.1145/3542700.3542705

[42] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 117–131. https://doi.org/10.1145/3373376.3378528

[43] Lluís Vilanova, Lina Maudlej, Shai Bergman, Till Miemietz, Matthias Hille, Nils Asmussen, Michael Roitzsch, Hermann Härtig, and Mark Silberstein. 2022. Slashing the disaggregation tax in heterogeneous data centers with FractOS. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 352–367. https://doi.org/10.1145/3492321.3519569

[44] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proc. VLDB Endow.* 15, 10 (2022), 1991–2004. https://www.vldb.org/pvldb/vol15/p1991-petrov.pdf

[45] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 94–107. https://doi.org/10.1145/3127479.3128609

[46] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 967–986. https://www.usenix.org/conference/atc22/presentation/wang-zeke

[47] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 87–104. https://doi.org/10.1145/2815400.2815419

[48] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, Christine Morin and Gilles Muller (Eds.). ACM, 265–278. https://doi.org/10.1145/1755913.1755940

[49] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2017. The End of a Myth: Distributed Transaction Can Scale. *Proc. VLDB Endow.* 10, 6 (2017), 685–696. https://doi.org/10.14778/3055330.3055335

[50] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020 (SIGMOD '20)*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 511–526. https://doi.org/10.1145/3318464.3389724

[51] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 685–699. https://doi.org/10.1145/3514221.3526187

[52] Tobias Ziegler, Carsten Binnig, and Uwe Röhm. 2019. Skew-resilient Query Processing for Fast Networks. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband (LNI, Vol. P-290)*, Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke (Eds.). Gesellschaft für Informatik, Bonn,

81–85. https://doi.org/10.18420/btw2019-ws-06

[53] Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2020. RDMA Communciation Patterns. *Datenbank-Spektrum* 20, 3 (2020), 199–210. https://doi.org/10.1007/s13222-020-00355-7

[54] 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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 741–758. https://doi.org/10.1145/3299869.3300081

Received July 2023; revised October 2023; accepted November 2023