Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems

FEILONG LIU, LINGYAN YIN, and SPYROS BLANAS, The Ohio State University, USA

The commoditization of high-performance networking has sparked research interest in the RDMA capability of this hardware. One-sided RDMA primitives, in particular, have generated substantial excitement due to the ability to directly access remote memory from within an application without involving the TCP/IP stack or the remote CPU. This article considers how to leverage RDMA to improve the analytical performance of parallel database systems. To shuffle data efficiently using RDMA, one needs to consider a complex design space that includes (1) the number of open connections, (2) the contention for the shared network interface, (3) the RDMA transport function, and (4) how much memory should be reserved to exchange data between nodes during query processing. We contribute eight designs that capture salient tradeoffs in this design space as well as an adaptive algorithm to dynamically manage RDMA-registered memory. We comprehensively evaluate how transport-layer decisions impact the query performance of a database system for different generations of InfiniBand. We find that a shuffling operator that uses the RDMA Send/Receive transport function over the Unreliable Datagram transport service can transmit data up to 4× faster than an RDMA-capable MPI implementation in a 16-node cluster. The response time of TPC-H queries improves by as much as 2×.

CCS Concepts: • **Information systems → Relational parallel and distributed DBMSs**; *Query operators*;

Additional Key Words and Phrases: Data shuffling, RDMA, parallel database systems

ACM Reference format:
Feilong Liu, Lingyan Yin, and Spyros Blanas. 2019. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. *ACM Trans. Database Syst.* 44, 4, Article 17 (December 2019), 45 pages.
https://doi.org/10.1145/3360900

1 INTRODUCTION

Fast networking is no longer exclusive to high-end supercomputers. Database servers today commonly ship with 100-Gbps EDR InfiniBand, while 200-Gbps HDR InfiniBand devices have appeared in the higher-end segment of the server market. High-performance network protocols such as InfiniBand, RoCE, and iWARP offer low-latency, high-bandwidth communication and provide remote memory access (RDMA) capabilities that allow applications to directly access memory in remote computers.

This research was partially supported by the National Science Foundation under grants SHF-1816577, III-1422977, III-1464381, and CNS-1513120 and by a Google Research Faculty Award.
Authors' addresses: F. Liu, L. Yin, and S. Blanas, 395 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43210, United States of America; emails: {liu.3222, yin.387, blanas.2}@osu.edu.

0362-5915/2019/12-ART17 $15.00
https://doi.org/10.1145/3360900

To use RDMA, parallel database systems need to choose between message-passing mechanisms or shared memory abstractions for data transfer. Message-oriented communication is cooperative: The receiver initiates the communication and specifies a location in its memory space that will be changed; then the sender determines what to change in the receiver's memory space and completes the data transfer. A shared-memory abstraction removes this synchronization hurdle by allowing one of the two sides to remain completely passive. For this reason, one-sided communication primitives such as RDMA Read and RDMA Write have generated substantial research excitement. At the algorithmic level, prior work has proposed new join algorithms that use RDMA [6, 15, 16, 44]. At the systems level, prior work has redesigned the database kernel for fast networks for analytical [45], transactional [9, 57], and hybrid workloads [26].

This article designs and evaluates a bespoke data shuffling operator for analytical query processing in parallel database systems that exchanges data between query fragments via RDMA operations. Compared to prior work, our approach is unobtrusive, as it does not require extensive modifications to the database kernel. In addition, by designing an operator for the InfiniBand Verbs interface, our solution keeps the database system in charge of memory and communication management, unlike libraries such as Accelio [1], MPI [32], and `rsocket` [10]. This article is an extended version of our work in RDMA-aware data shuffling [29] that adds endpoint implementations for the RDMA Write primitive and an adaptive RDMA buffer management algorithm.

The article first introduces the communication endpoint abstraction to decouple the mechanics of data transmission from the repartitioning operation. Different endpoints can transmit data either through the RDMA Send/Receive message-passing abstraction or through the RDMA Read and RDMA Write shared-memory abstraction. The endpoint abstraction is oblivious to the RDMA transport service and hence supports both reliable transport that offloads communication management to hardware and guarantees message delivery, as well as unreliable communication that requires error handling and flow control in software. Our design assumes a network such as InfiniBand, where unreliable transport may deliver packets out of order but is lossless under congestion. We find that database systems can uniquely benefit from an unreliable transport service, because relational algebra operators are set based, which alleviates the need to store messages in a re-order buffer for many query plans.

This article introduces eight different designs of the data shuffling operator that represent tradeoffs between (1) the number of open connections, (2) the contention for the shared network interface, (3) the RDMA transport function, and (4) how much memory should be reserved to shuffle data between nodes during query processing. We adopt the popular pull-based operator interface to permit database systems to use the proposed techniques without radically redesigning their existing analytical processing engine. We propose an adaptive memory management algorithm to dynamically manage the RDMA buffer queue according to the data processing latency of each query fragment. The throughput of the adaptive algorithm is as good as or better than that of the fixed buffer algorithm for query fragments with both high and low processing latencies. We have open-sourced our prototype implementation for further scrutiny and research by the community [51].

The experimental evaluation compares the performance of eight possible designs for clusters with up to 16 nodes. The algorithms are evaluated on 56-Gbps FDR InfiniBand and the newer 100-Gbps EDR InfiniBand. The evaluation demonstrates that transport layer decisions (as exposed via RDMA) can significantly impact the analytical performance of a parallel database system. The RDMA Send/Receive message-passing abstraction over an unreliable transport layer achieves robust performance across all configurations, despite the overheads of performing coordination, flow control, and error handling in the database system. Overall, the data shuffling operator that is introduced in this article outperforms MVAPICH [32], an RDMA-capable MPI implementation, by as much as 4× with throughput-intensive tests and by as much as 2× with TPC-H queries.

2 RDMA BACKGROUND

RDMA allows applications to directly access remote memory. One needs to pin a page in physical memory and register it with the network adapter before accessing it through RDMA operations. We use the InfiniBand verbs programming interface (*ibv_** functions) throughout the article. The IB verbs interface is supported either natively or through emulation for InfiniBand, RoCE, and iWARP.

2.1 RDMA Transport Functions

Before communicating over RDMA, one first creates and initializes a Queue Pair (QP). A Queue Pair consists of a Send Queue (SQ) and a Receive Queue (RQ) and is associated with a Completion Queue (CQ). The depth of these queues is limited by the hardware. Communication requires posting Work Requests (WRs) to the Queue Pair. Work Requests consist of a pointer to registered memory and a request, which can be Send, Receive, or Read. Work Requests are processed asynchronously. When a Work Request is serviced, the network adapter populates the associated Completion Queue with a completion event. The application then retrieves completion events from the Completion Queue and reclaims the memory that each event points to.

RDMA Send and Receive are used in two-sided communication. Two-sided communication starts at the receiver: The receiver first posts an RDMA Receive Work Request into the Receive Queue that points to a free memory buffer. This free buffer will be used to store the data from a Send request. After the receiver has posted the RDMA Receive request, the sender then posts an RDMA Send Work Request to the Send Queue that points to the buffer to be transmitted. When the Send request is received, it will be matched and consume one Receive request. The application needs to ensure that there are sufficient Receive requests in the Receive Queue to match all incoming Send requests, else Send requests will be dropped.

RDMA Read and RDMA Write are one-sided communication primitives. In RDMA Read, the receiver will read the data from the sending node by posting an RDMA Read request into the Send Queue. The request specifies the remote address of the data to read from and a local buffer to store the data into. The sender remains completely passive in the communication. The local network adapter asynchronously performs the remote read operation, populates the corresponding buffers, and posts a completion event to the local Completion Queue when the operation has been completed. In RDMA Write, the sender posts an RDMA Write request to write data to the receiving node. The request specifies the local buffer, which contains the data to be written to the receiver and the remote address that the data will be written to.

RDMA offers two versions of the RDMA Write operation. One is RDMA Write without Immediate Data and the other is RDMA Write with Immediate Data. RDMA Write without Immediate Data offers no notification mechanism when an RDMA Write request completes, which means that the receiver is totally passive. When writing with Immediate Data, the completion of the request will consume one RDMA Receive request in the receiver and generate a completion entry in the Completion Queue of the receiver. An RDMA Write request with Immediate Data includes a four-byte integer that is included in the completion entry. When writing with Immediate Data, the receiver cannot be passive, as it needs to post RDMA Receive requests and poll the Completion Queue to detect the completion of incoming data transfers.

2.2 RDMA Transport Service Types

Our design considers two transport service types for RDMA: Reliable Connection (RC) and Unreliable Datagram (UD).

The Reliable Connection service is connection-oriented. Packets sent over the Reliable Connection service will be acknowledged and are guaranteed to be delivered once and in order. The Reliable Connection service supports the RDMA Send, RDMA Receive, RDMA Read, and RDMA Write transport functions. The maximum message size in Reliable Connection transport is hardware-specific and can be as large as 1GiB. Because Reliable Connection is connection oriented, each Queue Pair can only communicate with exactly one other Queue Pair. Hence, point-to-point communication between n nodes requires $\Theta(n^2)$ Queue Pairs with the Reliable Connection service.

The Unreliable Datagram service is connectionless and does not acknowledge the delivery of packets. Packets may thus be dropped or arrive out of order. The Unreliable Datagram service only supports the two-sided RDMA Send and RDMA Receive operations and the maximum message size is 4KiB. Because the Unreliable Datagram service is connectionless, one Queue Pair can communicate with any other Queue Pair. Thus, point-to-point communication between n nodes will require only $\Theta(n)$ Queue Pairs with the Unreliable Datagram service.

2.3 Programming Interface

The first step for an application to use RDMA is to create Queue Pairs using the *ibv_create_qp*() function, register the memory, exchange routing information, and build the connections between Queue Pairs. After building the connection, the application uses RDMA verbs to issue RDMA requests. In particular, *ibv_post_send()* is used for RDMA Read, RDMA Write, or RDMA Send to post requests to the Send Queue, and *ibv_post_recv()* is used for RDMA Receive to post requests to the Receive Queue. RDMA Send requests specify the memory address with the data to be sent; RDMA Receive requests specify the memory address, which will store the received data; while RDMA Read and RDMA Write specify the address of the respective operation. The hardware will then process the RDMA request. Once the RDMA request has been posted, the memory associated with this request cannot be reused. When the operation finishes, the hardware generates a completion entry in Completion Queue. The application retrieves completion entries by polling the Completion Queue using *ibv_poll_cq()*. The completion entry informs the application that RDMA request has completed so that the corresponding memory can be reused.

3 DESIGN TRADEOFFS FOR RDMA-BASED DATA TRANSFER

Different combinations of RDMA transport functions and service types pose different challenges in implementing RDMA-aware data shuffling algorithms. A summary of the design space is shown in Figure 1, in which we classify the design choices into three dimensions. Note that not all the points in the space are permissible; in particular, the Unreliable Datagram transport service only supports the Send/Receive transport functions.

Parameter 1. Number of QPs per node: The number of Queue Pairs (QPs) per node is a tradeoff between hardware resource consumption and parallelism. As QP data are cached in the Network Interface Card (NIC), the NIC cache will run out of space if there are too many Queue Pairs per node. Prior work [12] has shown that this can degrade performance by up to 5×. At the same time, more QPs means more concurrency, as there is less contention between threads.

Assume that a cluster has n nodes and t CPU cores per node and that one allocates each thread to a separate CPU core. With the Reliable Connection transport service, each QP can only communicate with one other QP. For a node to communicate simultaneously with n other nodes, at minimum n QPs per node are needed. With this design all threads will share the same QP when communicating with a specific node, which may cause thread contention. If each CPU core uses a distinct set of QPs to communicate to remote CPU cores to avoid contention, then $n \times t$

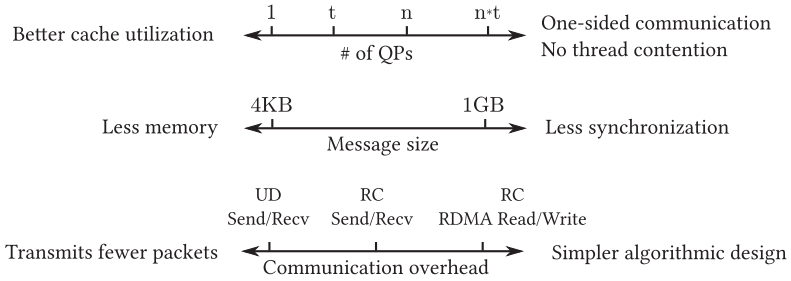


Fig. 1. RDMA design space for data shuffling algorithms for n nodes with t CPU cores per node.

connections per node are needed for communication.[1] This can easily overflow the NIC cache in larger clusters. With the Unreliable Datagram transport service, a QP can communicate with any other QP. Point-to-point communication between all n nodes is possible with just one QP per node; the QP will be shared by all CPU cores regardless of the destination. Thread contention can be eliminated by using t QPs per node.

Parameter 2. The message size for RDMA communication. The message size in RDMA communication trades between memory consumption and communication efficiency. The maximum message size with the Unreliable Datagram transport service is the MTU, which is 4KiB in many platforms, including our own. The maximum message size with the Reliable Connection service can be as high as 1GiB per the InfiniBand specification [3].

A smaller message size means that applications should post more requests to transmit the same volume of data. Thus, small sizes lead to more CPU overhead during communication. However, large message sizes require the application to pin and register substantially more memory for RDMA communication. To overlap communication with computation, at least $2 \times n$ message buffers are needed to communicate with n nodes. If one uses the extreme setting of 1GiB in a 16-node cluster, then at least 30GiB should be pinned in memory for communication in each node—which may be beyond what a parallel database system can comfortably allocate to a single query fragment.

Parameter 3. Overhead of communication. Different RDMA transport service types and functions pose different synchronization requirements and thus have different overheads.

Error handling: The Reliable Connection transport service guarantees the ordered and reliable delivery of every message. As every packet that is transmitted requires an acknowledgment, this leads to more traffic in the network but a simpler algorithm. The Unreliable Datagram transport service sends no acknowledgement packet, which leads to less traffic in the network. However, the delivery of the message may fail and messages can arrive in any order. Thus, the application needs to perform error handling and carefully handle state transitions despite packets arriving out of order or not at all. These considerations complicate the design of RDMA-aware algorithms.

Synchronization and flow control: RDMA Send and RDMA Receive are two-sided verbs. The application is responsible for posting an RDMA Receive request on the receiving side before an RDMA Send request arrives, else the RDMA Send request will be dropped. Synchronization is needed to tally the transmitted messages and continuously communicate the number of posted Send and Receive requests between the sender and the receiver.

RDMA Read is a one-sided verb. During communication, the sender remains passive while the receiver posts the RDMA Read request. The challenge is to ensure that the passive side (the sender)

[1]Note that $n \times t^2$ connections per node are needed if one wants to allow arbitrary communication between any two CPU cores in the cluster. We do not consider this communication pattern in this article.

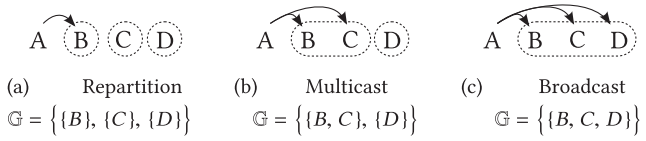


Fig. 2. The transmission group abstraction encapsulates the repartitioning, multicast, and broadcast data transmission patterns in database systems. The arrows show the pattern when node A transmits to the first transmission group in $\mathbb{G}$, denoted as $\mathbb{G}[0]$.

does not write memory that is currently being read by the active side (the receiver). Synchronization is needed to inform the sender when the buffer space can be safely reclaimed.

RDMA Write is another one-sided verb. During communication, the sender posts the RDMA Write request. Whether the receiver is passive or active depends on the specific verb that is used. If the sender posts RDMA Write without Immediate Data requests, then the receiver remains passive. When the sender posts RDMA Write with Immediate Data requests, the receiver is active: The receiver must to post sufficient RDMA Receive requests to match the RDMA Write with Immediate Data requests posted by the sender, and the receiver must poll the Completion Queue for completion events for the RDMA Receive Requests. Synchronization is necessary for either RDMA Write request type to ensure that the sender only writes to the remote buffer after the buffer has been consumed by the receiver. In addition, if the request type is RDMA Write with Immediate Data, then the receiver needs to coordinate with the sender to post a sufficient number of RDMA Receive requests to match each incoming RDMA Write request.

4 RDMA-AWARE DATA SHUFFLING ALGORITHMS

This section describes high-performance data shuffling algorithms that use InfiniBand verbs directly from user space and bypass the operating system's networking stack. Section 4.1 introduces the *transmission group* abstraction to support the repartition, multicast, and broadcast data transmission patterns. Section 4.2 introduces the *communication endpoint* that hides RDMA-specific complexities from other components, and Section 4.3 presents the SHUFFLE and RECEIVE operators.

4.1 Supported Data Transmission Patterns

The communication pattern during relational query processing is dynamic and data transfers may be issued to one or multiple recipients. Our RDMA-aware shuffling algorithms support the repartition, multicast, and broadcast patterns through the *transmission group* abstraction. Nodes can be arbitrarily assigned to zero, one, or more transmission groups. Figure 2 shows the three data communication patterns in a four-node cluster where node A is the sender. When the transmission group $\mathbb{G}$ contains singletons, as in Figure 2(a), node A will repartition the data. Figure 2(b) shows a multicast pattern, where data sent from A to transmission group $\mathbb{G}[0]$ will reach both B and C. When $\mathbb{G}$ contains a single set with every other node in the cluster, as in Figure 2(c), node A will broadcast data to all other nodes.

4.2 The Communication Endpoint Abstraction

InfiniBand imposes unique design constraints for different combinations of transport modes and communication verbs. In addition, initializing the communication is more involved than setting up a TCP/IP socket, as one needs to pin and register memory with the network adapter and then build the RDMA connection. The time to setup an RDMA connection has been shown to be up to 1000× longer than setting up a TCP/IP-based connection [14].

We introduce the *communication endpoint* abstraction to hide such transport-level intricacies from the high-level communication logic. The endpoints implement the SHUFFLE and RECEIVE operators that are used by database systems for communication. Section 4.3 describes how the SHUFFLE and RECEIVE operators use the endpoints. A communication endpoint contains all RDMA-specific resources and the related data transmission logic. We adopt the pull-based operator design [18] in the endpoints. The endpoint is initialized by calling the OPEN function that registers memory for RDMA operations and builds the RDMA connections. The OPEN function also creates a unique integer to identify this endpoint during query processing, which is used similarly to a port and address pair in a TCP/IP connection. The endpoint owns the memory for RDMA operations and is responsible for managing it during regular processing. The endpoint is terminated with the CLOSE function that closes its RDMA connections and unregisters its memory. Implementations of the communication endpoint conform to the same interface but support different RDMA transport functions and service types. All functions of an endpoint are thread-safe.

The SEND endpoint transmits data using the following interface:

- PUTDATA (void* *buf*, int[]*dest*, int *state*)
  This function schedules to transmit the buffer *buf* to the endpoints in the *dest* array. The buffer cannot be used after PUTDATA returns. The binary parameter *state* signals if this is the last buffer to be sent (`Depleted`) or more data are available (`MoreData`). PUTDATA does not block.
- void* *buf* ← ACQUIRE()
  This function returns an RDMA-registered buffer *buf* that can be used in a subsequent PUTDATA call. ACQUIRE may block if all transmission buffers are in use.

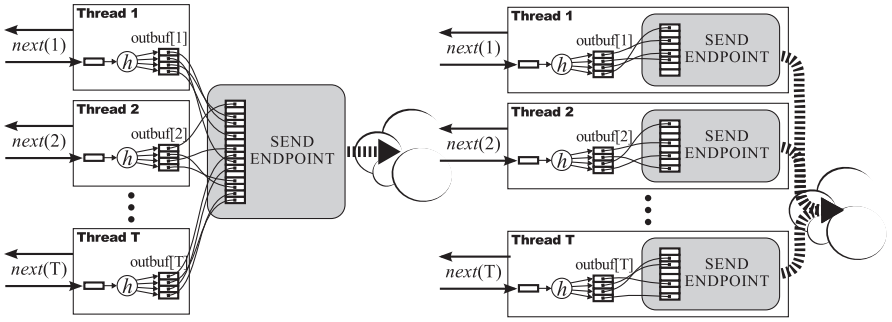
The RECEIVE endpoint has the following interface:

- <int *state*, int *src*, void* *remote*, void* *local*> ← GETDATA()
  This function returns data in the RDMA-registered transmission buffer *local*. The binary variable *state* denotes if this is the last buffer from this endpoint; *src* is the unique identifier of the endpoint that sent this buffer; *remote* is the address of this buffer in the remote endpoint. (See Section 5.4 for more details on how *remote* is used.) GETDATA will block if all buffers are in use.
- RELEASE (void* *remote*, void* *local*, int *src*)
  This function returns the RDMA-registered buffer *local* to the endpoint. The buffer *local* cannot be used after RELEASE returns. If the communication primitive is one-sided, then RELEASE also notifies the remote endpoint *src* that buffer *remote* has been consumed. (See Section 5.4 for more details.) RELEASE does not block.

4.3 The SHUFFLE and RECEIVE Operators

Our implementation uses the vector-at-a-time processing model, where a vector of tuples is returned in the NEXT function call. We parallelize the pull-based operator by adding a thread identifier as a parameter to the NEXT call. Every operator consists of its state and a set of output buffers; both are thread-partitioned to avoid cache interference. Threads are exclusively bound to CPU cores. Although the algorithms are described in the context of the pull-based operator model, they can be easily adapted for push-based execution models that commonly rely on query compilation [40, 48].

We now describe the implementation of the SHUFFLE and the RECEIVE operators using the endpoint interface described in Section 4.2. Figure 3 shows two different configurations of the SHUFFLE operator. A single endpoint configuration (SE) is shown in Figure 3(a), where all threads share one



(a) Single endpoint (SE). (b) Multi-endpoint (ME).

Fig. 3. Configurations for the SHUFFLE operator.

SEND endpoint. This uses less resources at the expense of contention for the shared resource—the endpoint. The multi-endpoint configuration (ME) is shown in Figure 3(b), where a SEND endpoint is dedicated to every thread. This avoids inter-thread contention but increases resource consumption significantly in modern many-core processors: Memory registration and connection time rise proportionally with the number of CPU cores. Likewise, the RECEIVE operator also supports single- and multi-endpoint configurations.

One way to avoid thread contention in the single endpoint configuration is to have one thread be exclusively responsible for communication with one remote node. Consider an approach that first shuffles data locally between threads so that data for a specific destination is sent to the corresponding thread, and this thread then transmits the data to the destination. The disadvantage is that shuffling data among threads incurs additional overhead: If all threads populate one contiguous RDMA buffer, then this needs fine-grained synchronization between threads. If each thread populates its own buffer, then this introduces a data copy when the sending thread reads and copies data from each buffer into its own RDMA-registered buffer.

4.3.1 The SHUFFLE *Operator.* This implementation of the data-transmitting SHUFFLE operator is shown in Algorithm 1. The SHUFFLE operator owns a thread-partitioned array of output buffers; output buffer i is used for transmitting data to the transmission group $\mathbb{G}[i]$. First, the SHUFFLE operator hashes every tuple t in the output of the next operator in the pipeline (Algorithm 1, line 8) and appends the tuple to the output buffer for the transmission group $\mathbb{G}[\textsc{Hash}(t)]$ (line 10). When the output buffer is full, the SHUFFLE operator schedules the entire output buffer for transmission in one RDMA operation (line 12) and requests a new, empty RDMA-registered transmission buffer from the endpoint (line 13). This process continues until the data source is depleted. To shutdown cleanly, the SHUFFLE operator needs to propagate the `Depleted` state to all RECEIVE endpoints. In the multi-endpoint configuration every thread sets the `Depleted` state for its own endpoint when its input is depleted (line 16). In the single-endpoint configuration multiple threads share the endpoint, so only the last thread sending out data needs to propagate the end-of-transmission status to the remote endpoint (line 18).

A design choice is whether to copy the tuples into RDMA-registered buffers or directly perform RDMA operations on the input (often referred to as the *zero copy* optimization). With *zero copy*, one tuple is one RDMA message. The message size is small when the record size is small. While with copy, tuples are reorganized into the RDMA-registered buffers and a batch of tuples are included in one RDMA message. This reduces the number of RDMA messages and the number of RDMA requests posted. Our experiments confirm the findings of Kesavan et al. [24] that zero copy shows

ALGORITHM 1: The SHUFFLE operator

```
   state mode: either SingleEndpoint or MultiEndpoint
         nextop: reference to the next operator in the pipeline
         endpoint: the endpoint object array
         outbuf: the output buffers array (see Figure 3)
         𝔾: the user-defined communication groups
   output state: either MoreData or Depleted
          batch: a data buffer
   function NEXT(tid)
1      if mode is SingleEndpoint then
2          target ← endpoint[0]
3      else if mode is MultiEndpoint then
4          target ← endpoint[tid]
5      repeat
6          <state, batch> ← nextop.NEXT(tid)
7          foreach tuple in batch do
8              dest ← HASH(tuple)
9              curbuf ← outbuf[tid][dest]
10             append tuple to curbuf
11             if curbuf is full then
12                 target.PUTDATA(curbuf, 𝔾[dest], MoreData)
13                 outbuf[tid][dest] ← target.ACQUIRE()
14     until state is Depleted;
15     if mode is MultiEndpoint or tid is last thread then
16         target.PUTDATA(curbuf, 𝔾[dest], Depleted)
17     else if mode is SingleEndpoint then
18         target.PUTDATA(curbuf, 𝔾[dest], MoreData)
19     return <Depleted, EmptyBatch>
```

little benefit when the record size is small (128 bytes). We thus choose to always copy based on the observation that tuple sizes are typically small for both column-oriented and row-oriented database systems. In column-oriented main-memory database systems, the tuple size is as little as 16 bytes. In row-oriented disk-based database systems, the tuple size is typically less than a few hundred bytes; for example, the biggest table (LINEITEM) of the TPC-H database is 204 bytes wide when loaded in PostgreSQL.

4.3.2 The RECEIVE *Operator.* The implementation of the RECEIVE operator is shown in Algorithm 2. Each thread will clear its output buffer and then ask for data from the endpoint (Algorithm 2, line 7). The thread then appends the data from the RDMA-registered buffer to the output buffer (line 8) and returns the buffer to the endpoint to be reused (line 9). If the output buffer is full, then the thread returns it to the parent operator (line 11). This process stops when the `Depleted` signal is received that marks the end of the data transmission.

5 IMPLEMENTING THE COMMUNICATION ENDPOINT

This section describes the implementation of the endpoints. The section is organized as follows: Section 5.1 discusses the challenges in the design choices of the endpoints and gives an overview of the tradeoffs. Four implementations of the communication endpoint are then described that use

ALGORITHM 2: The RECEIVE operator

```
   state mode: either SingleEndpoint or MultiEndpoint
         endpoint: the endpoint object array
         outbuf: the output buffers array
   output state: either MoreData or Depleted
          batch: a data buffer
   function NEXT(tid)
1      if mode is SingleEndpoint then
2          target ← endpoint[0]
3      else if mode is MultiEndpoint then
4          target ← endpoint[tid]
5      clear outbuf[tid]
6      repeat
7          <state, src, remote, local> ← target.GETDATA()
8          append local into outbuf[tid]
9          target.RELEASE(remote, local, src)
10         if outbuf[tid] is full then
11             return <MoreData, outbuf[tid]>
12     until state is Depleted;
13     return <Depleted, outbuf[tid]>
```

different RDMA transport functions and service types: the RDMA Send/Receive function with the Reliable Connection service (Section 5.2), the RDMA Send/Receive function with the Unreliable Datagram service (Section 5.3), the RDMA Read function with the Reliable Connection service (Section 5.4), and the RDMA Write function with the Reliable Connection service (Section 5.5). Section 5.6 describes how thread-safe interfaces for endpoints are implemented.

5.1 Overview

The implementation choices for endpoints can be classified into two dimensions: the choice of RDMA primitives (two-sided RDMA Send/Receive, one-sided RDMA Read, or one-sided RDMA Write) and the choice of RDMA transport types (Reliable Connection, or Unreliable Datagram). As Unreliable Datagram only supports RDMA Send/Receive, these two dimensions result in four implementation choices. An orthogonal consideration is how many endpoints should be associated with each shuffle operator: Does a single endpoint suffice or are multiple endpoints needed to use the full network bandwidth? Overall, we consider eight possible implementations for the shuffle operator.

5.1.1 Choice of RDMA Primitives. This section compares the three RDMA primitives, RDMA Send/Receive, RDMA Read, and RDMA Write.

RDMA Send/Receive. As described in Section 2.1, for every RDMA Send request posted in the sender, there must be one matching RDMA Receive request in the receiver. One challenge of using RDMA Send/Receive is to synchronize the senders and receivers to match the RDMA requests posted. We use a credit mechanism to match the RDMA requests. The receivers record the number of RDMA Receive requests posted and transmit this number to the senders with RDMA Write. The senders post RDMA Send requests only when there are sufficient outstanding RDMA Receive requests in the corresponding receiver. The credit mechanism is described in detail in Section 5.2.

RDMA Read. In the RDMA Read implementation, the receiver posts RDMA Read requests to read data from the sender. Two synchronization points are needed. First, the receiver needs to know whether a buffer in the sender is ready to be read. Thus, the sender needs to notify the receiver after it fully populates a buffer. Second, the sender can only reuse a buffer after it has been consumed by the receiver. However, as RDMA Read is one-sided operation, the sender is oblivious to RDMA Read requests by the receiver. Hence the receiver needs to notify the sender when it completes reading a buffer. The RDMA Read implementation uses RDMA Write to implement a message queue (which is presented in detail in Section 5.4) for the two synchronization points described above.

RDMA Write. In the RDMA Write implementation, the sender posts RDMA Write to write data to the receiver. Two synchronizations points are needed. First, the receiver needs to notify the sender which buffer can be written, because a buffer can be written by the sender only after the data has been consumed by the receiver. Second, because of the one-sided nature of RDMA Write, the receiver is not aware of the RDMA Write requests that are transmitted by the sender. The receiver thus needs to know which buffer transmissions have completed. Section 5.5 considers three mechanisms that the receiver can use to determine which buffers are ready to be consumed.

5.1.2 Choice of RDMA Transport. This section compares the two RDMA transport service types, Reliable Connection and Unreliable Datagram.

Reliable Connection. As described in Section 2.2, the message size for the Reliable Connection transport service can be up to 1GiB. One question is how to choose the message size. RDMA can only access registered memory that is pinned in physical memory, so large message sizes means less available memory for the database system. However, a small message size means more messages as more RDMA requests need to be posted to transmit the same amount of data. We evaluate how the message size affects performance in Section 7.1.3. Another challenge is that with Reliable Connection one Queue Pair can only communicate with only one other Queue Pair. Therefore, the number of Queue Pairs needed for communication increases linearly with the number of nodes in the cluster, which impacts the performance of data shuffling in larger clusters. The scalability of the different algorithms is evaluated in Section 7.1.4.

Unreliable Datagram. One challenge of the Unreliable Datagram transport service is that message delivery is not guaranteed and may be out of order. We describe how an endpoint implementation can deal with this in Section 5.3. An advantage of using the Unreliable Datagram transport service is that one Queue Pair can communicate with more than one Queue Pairs; therefore, the number of Queue Pairs is fixed regardless of the number of nodes in the cluster.

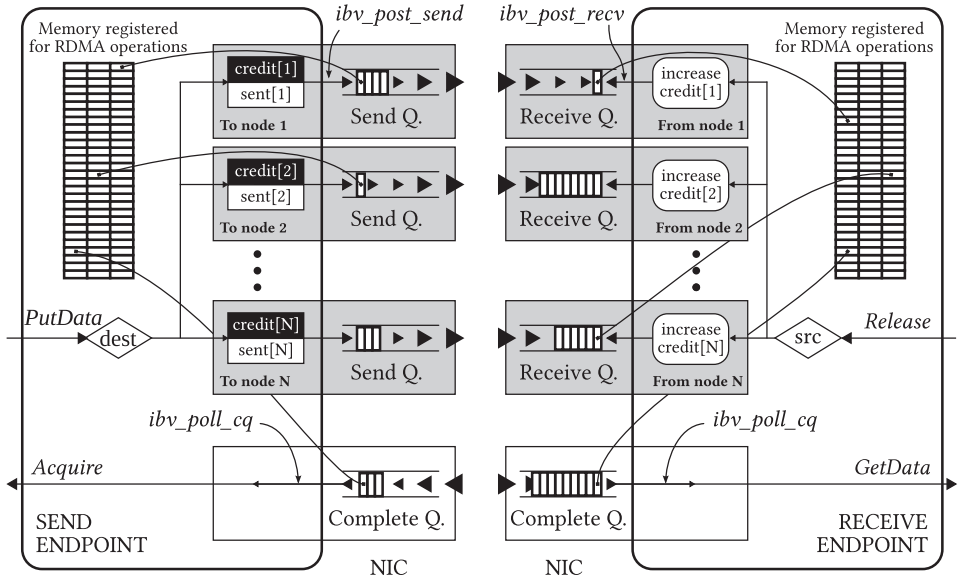
5.1.3 Design Alternatives for High-performance Data Shuffling. This article considers eight endpoint designs that make different design choices for RDMA primitives, RDMA transport types, and number of endpoints in the shuffle operator. Table 1 summarizes the tradeoffs associated with each of the eight designs in a cluster with n nodes and t threads per query fragment. We name each design by concatenating the number of endpoints, single (SE) or multiple (ME), with the implementation of the endpoint (SQ/SR, MQ/SR, MQ/RD, MQ/WR). For example, MESQ/SR refers to the multi-endpoint (ME) implementation that uses the Unreliable Datagram transport service (single Queue Pair, or SQ) using the Send/Receive primitive (SR).

5.2 RDMA Send/Receive with Reliable Connection

We now describe how to implement the communication endpoint using the message-passing semantics of the RDMA Send/Receive transport functions and the Reliable Connection transport

Table 1. Alternative Designs for a Cluster with n Nodes and t Threads per Node

RDMA Read, Write	RDMA Send/Receive	QPs per node	Thread contention	Messages	Transport
One-sided	Two-sided				
Periodic coordination to manage buffers	Continuous coordination on every transfer				
Hardware flow control	Software flow control				
MEMQ/RD, MEMQ/WR	MEMQ/SR	$n \cdot t$	None	Round-trip, up to 1GiB	Reliable Connection, hardware error control
SEMQ/RD, SEMQ/WR	SEMQ/SR	n	Moderate		
Not supported by InfiniBand	MESQ/SR	t	None	Half-trip, up to 4KiB	Unreliable Datagram, software error control
	SESQ/SR	1	Excessive		



(a) Send endpoint (b) Receive endpoint

Fig. 4. Endpoint implementation for the RDMA Send/Receive transport function and the Reliable Connection service.

service. Communication over RDMA Send/Receive requires that every arriving Send request (that contains the data) is matched to a posted Receive request (that specifies where the data will be stored). A Send request that cannot be matched to a Receive request will be dropped, as the network card does not know where to write the incoming data. The main technical challenge in implementing a high-performance communication endpoint with the RDMA Send/Receive function is synchronizing the sender and the receiver to ensure that a Receive request has been posted before a Send request arrives.

In our implementation, we synchronize senders and receivers through a *stateless credit mechanism*, where the receiver issues credit to the sender only after a Receive request has been posted. The SEND endpoint, shown in Figure 4(a), keeps the available credits (*credit*) and the consumed credits (*sent*) for each connection. By design, we store the absolute credit (that is, the number of Receive requests that have been posted in this connection so far) rather than the relative credit (that is, how many additional Receive requests have been posted) to keep the credit protocol

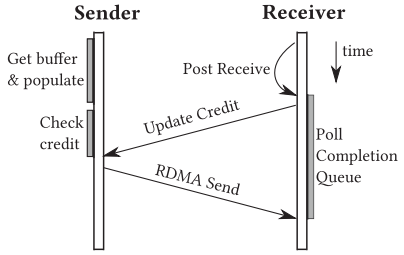


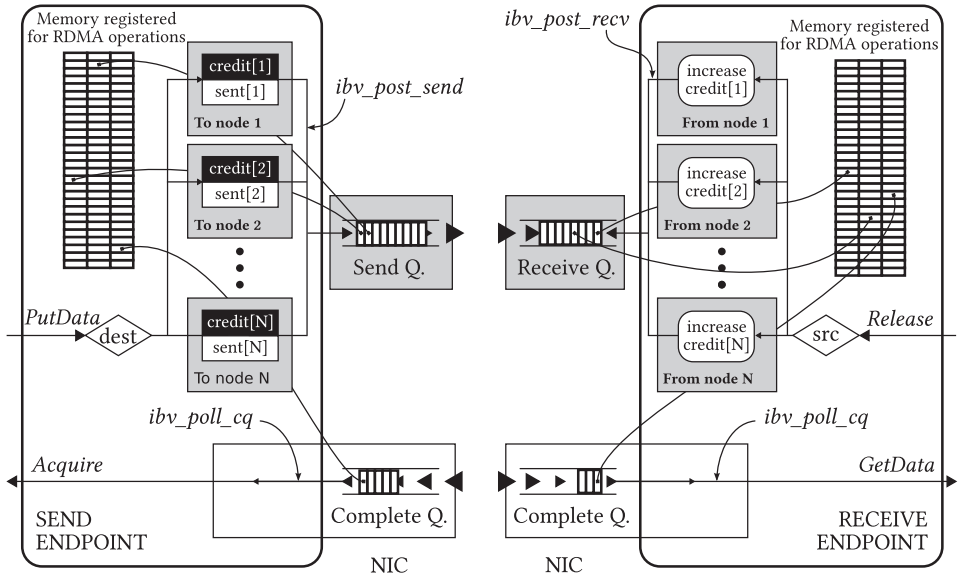
Fig. 5. Steps involved in one message transfer for the RDMA Send/Receive algorithm.

stateless. To end the transmission, the sender must consume all the credits and issue the same number of Send requests (which may involve sending empty buffers) to ensure that there is Send/Receive request parity. In our implementation, we inline the credit value in each request to save one DMA request as done in prior work [22]. One influential configuration parameter for the credit mechanism is the frequency at which the RECEIVE endpoint will write back the credit. One can amortize this credit write-back overhead over multiple Receive requests at the risk of starving the SEND endpoint for credit. We study this tradeoff experimentally in Section 7.1.1.

Figure 5 shows the steps involved in a data transfer using RDMA Send/Receive and Algorithm 3 shows the pseudo code. The sender first requests a free buffer by calling the ACQUIRE function. ACQUIRE polls the local Completion Queue for buffers that have been retired from completed RDMA operations, as shown in Figure 4, and blocks until an empty buffer is located. Note that a buffer may

ALGORITHM 3: Send/Receive with Reliable Connection

```
   function PUTDATA(buffer, destarr, state)
1      encode (destarr, state, source) as metadata in buffer
2      foreach node in destarr do
3          while credit[node] <= sent[node] do
4              wait
5          increment sent[node]
6          call ibv_post_send to enqueue buffer for transmission
   function void* ACQUIRE( )
7      do
8          buffer ← call ibv_poll_cq to poll for completions
9          gid ← the transmission group buffer was sent to
10     until |𝔾[gid]| completion events have been received ;
11     return buffer
   function RELEASE(remote, local, src)
12     call ibv_post_recv to enqueue local for data delivery
13     increment credit[src]
14     if enough credit has been accumulated then
15         send credit[src] to src
   function <state, src, remote, local> GETDATA( )
16     buffer ← call ibv_poll_cq to poll for completions
17     decode (state, source) from metadata in buffer
18     return <state, source, ⊥, buffer>
```



(a) Send endpoint (b) Receive endpoint

Fig. 6. Endpoint implementation for the RDMA Send/Receive transport function and the Unreliable Datagram service.

be referenced in multiple RDMA Send requests to different destinations in the same transmission group, and hence Acquire returns the buffer when completion events from all nodes in the group have been received (Algorithm 3, line 10). At the same time, the receiver retires consumed buffers by posting RDMA Receive requests on them using the Release function (Figure 4). Release first posts a Receive request using *ibv_post_recv* (Algorithm 3, line 12) and then notifies the sender by adding credits for this connection: It increments and remotely writes *credit* to the corresponding sender using the RDMA Write transport function (Algorithm 3, line 15).

Once a buffer has been populated, the sender uses the PutData function to transmit it. PutData blocks until sufficient credits are available: For every node in the transmission group, the algorithm tries to consume one credit, if available, by incrementing the *sent* counter (Algorithm 3, line 5). PutData then issues one RDMA Send request per destination to schedule the buffer transmission. In the meantime, the receiver polls the local Completion Queue in the GetData function with *ibv_poll_cq* (Algorithm 3, line 15) for new messages. GetData returns a *buffer* with data from the associated RDMA Send request for local processing.

5.3 RDMA Send/Receive with Unreliable Datagram

Whereas the Reliable Connection service requires n Queue Pairs to communicate with every other node in an n-node cluster, an endpoint implementation that uses the Unreliable Datagram transport service allows a single Queue Pair to communicate with any other Queue Pair on any node. This permits an endpoint implementation that has been designed for the Unreliable Datagram service to drastically cut down its RDMA-related memory consumption from $\Theta(n)$ to $\Theta(1)$, which has been shown to improve performance, as it avoids expensive page table fetches across the PCI bus [12].

Figure 6 sketches the endpoint implementation. With the Unreliable Datagram transport service, we only need a single Queue Pair in the endpoint to communicate with every other Queue

Pair (cf. the Reliable Connection implementation in Figure 4). We use the same stateless credit mechanism that was introduced in Section 5.2 to synchronize the sender and the receiver, with the only distinction being that now all destinations share one Queue Pair.

There are two challenges with using the Unreliable Datagram transport service:

(1) **Out of order packet delivery.** The first challenge is that the delivery of packets can be reordered. One limitation caused by out of order delivery is that it is necessary to handle the end of data transmission carefully, as the final message may arrive prematurely. Specifically, the message tagged as `Depleted` may arrive at the receiving endpoint before messages tagged with `MoreData` because of out of order delivery. The data receiver thus needs to ensure that a transition to the `Depleted` state is not premature. We handle this with a message counting algorithm. The send endpoint maintains an additional counter in the Unreliable Datagram implementation that records the total number of packets sent to each destination. Likewise, the receiving endpoint records the number of packets received from every source node. At the end of transmission, the sender communicates the total number of messages sent, and the receiver compares this number with the number of messages it has already received. The shuffling is complete only after the receiver has received the same number of packets. A consequence of unordered delivery is that the Unreliable Datagram transport service cannot be used for query plans or operations that have strict requirements on ordering, such as a sort-merge join.

(2) **Packet loss.** The second challenge with unreliable message delivery is that packets may be lost. The message count algorithm described above would block the receiver indefinitely if a packet gets lost. We solve this problem by setting a limit on the time the receiver waits for outstanding packets. If the totals still do not match after waiting, then we treat this as a network error and re-start the query. Re-starting the query can be expensive, but thankfully this occurs rarely in practice in InfiniBand: As Kalia et al. point out, InfiniBand has lossless link-level flow control and packets are never lost due to buffer overflows [21, 23]. Packet loss happens due to bit errors on the wire and hardware failures, which are rare events.

The above discussion only considers packet loss. Checking the number of messages cannot handle the case where one packet is dropped and another packet is duplicated in the same transmission stream. This problem could be solved by computing a hash signature of the entire data stream and having the receiving side calculate and compare the hash signature at the end of the transmission. Computing this signature has non-negligible overhead but may be necessary with networks with higher error rates such as RoCE. Another possible solution is to assign a sequence number to every message sent out from the sender and check the sequence number in the receiver. However, checking the number sequence for gaps has non-negligible overhead as the receiver needs to record sequence numbers for every sender. The CPU overhead can be substantial given that the message size limit is 4KiB with the Unreliable Datagram transport service, as the endpoint will need to track sequence numbers very frequently.

5.4 RDMA Read with Reliable Connection

We now present the endpoint implementation that uses RDMA Read, a one-sided communication primitive, to retrieve data from the sender to the receiver. In this implementation the buffers are owned by the sender and are transmitted ("pulled") to the receiver using RDMA Read. Data shuffling using RDMA Read needs two synchronization points. The first identifies when the receiver can read a buffer from the memory space of the sender. The second identifies when the sender can

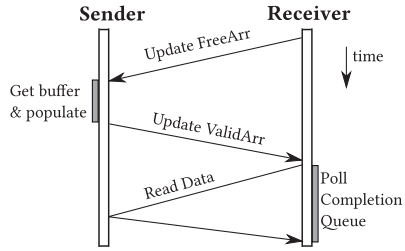
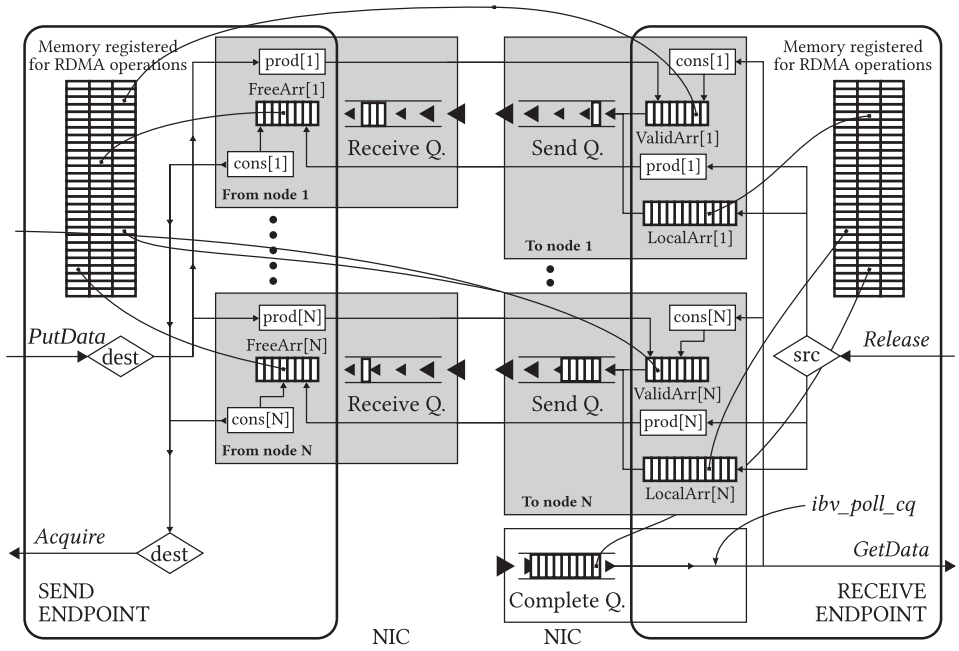


Fig. 7. Steps involved in one message transfer for the RDMA Read algorithm.



(a) Send endpoint (b) Receive endpoint

Fig. 8. Endpoint implementation for the RDMA Read transport function under the Reliable Connection service.

reuse a local buffer, because the receiver has finished processing the data in the buffer. Figure 7 shows the steps involved in one message transfer in the RDMA Read implementation.

The receiver uses a circular queue *FreeArr* to notify the sender whether a buffer can be reused and populated. As shown in Figure 8(a), *FreeArr* is maintained in the SEND endpoint and is filled by the RELEASE function in the RECEIVE endpoint. The pseudo code of the RELEASE function is shown in Algorithm 4. The RELEASE function signals that a *buffer* can be reused by adding it in the *FreeArr* queue using an RDMA Write request to the originating SEND endpoint (Algorithm 4, line 16).

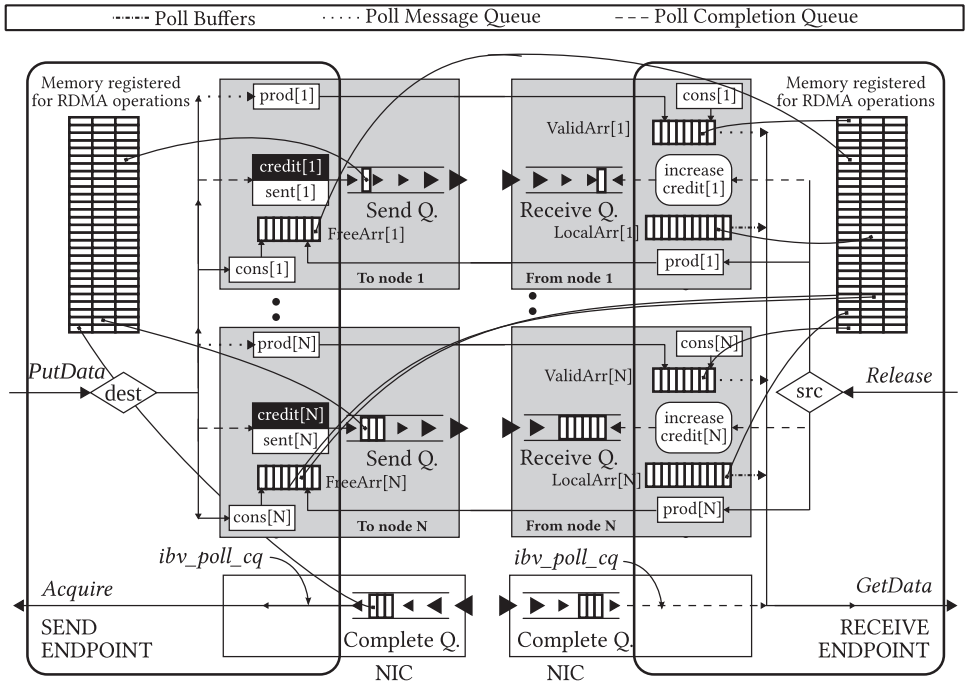
Before transmission, the sender locates a free buffer from *FreeArr* using the ACQUIRE function in the SEND endpoint. The ACQUIRE function looks for free buffers in the *FreeArr* of any incoming link but returns the buffer only if all destinations in the transmission group have notified that *buffer* can be reused (Algorithm 4 line 13).

ALGORITHM 4: RDMA Read with Reliable Connection

```
   function PutData(buffer, destarr, state)
1      addr ← address of buffer
2      encode (destarr, state, source, addr) as metadata in buffer
3      foreach node in destarr do
4          ValidArr[node][prod[node]] ← addr
5          increment prod[node]
   function void* Acquire( )
6      while true do
7          for i ∈ [1, N] do
8              while FreeArr[i] is set do
9                  buffer ← FreeArr[i][cons[i]]
10                 increment cons[i]
11                 mark notification for buffer
12                 gid ← the transm. group buffer was sent to
13                 if |𝔾[gid]| notifications received then
14                     return buffer
15         wait
   function Release(remote, local, src)
16     FreeArr[src][prod[src]] ← remote
17     increment prod[src]
18     push local into LocalArr[src]
   function <state, src, remote, local>GetData( )
19     for i ∈ [1, N] do
20         while ValidArr[i] is set and LocalArr[i] is set do
21             remote ← ValidArr[i][cons[i]]
22             increment cons[i]
23             local ← pop from LocalArr[i]
24             call ibv_post_send to read remote into local
25     buffer ← call ibv_poll_cq to poll for completions
26     decode (state, source, addr) from metadata in buffer
27     return <state, source, addr, buffer>
```

After the buffer has been populated with data, the sender writes into the circular queue *ValidArr* to notify the receiver that this buffer can be processed with RDMA Read. As shown in Figure 8(b), *ValidArr* is stored in the RECEIVE endpoint and is filled by the PutData function in the SEND endpoint. The PutData function signals that the *buffer* can be read by adding its address in the *ValidArr* queue of every RECEIVE endpoint in the transmission group using an RDMA Write request (Algorithm 4, line 4).

In the last step, the receiver locates one buffer in the sender that is ready for reading. The GetData function finds the address of a valid buffer from *ValidArr* (Algorithm 4, line 21) and issues an RDMA Read request to read the data (Algorithm 4, line 24). GetData then continuously polls the local Completion Queue for the completion of one request and returns the associated *buffer* for processing.



(a) Send endpoint (b) Receive endpoint

Fig. 9. Endpoint implementation for the RDMA Write transport function under the Reliable Connection service.

5.5 RDMA Write with Reliable Connection

This section describes the endpoint implementation that uses RDMA Write. In this implementation the buffers are owned by the receiver and the sender transmits ("pushes") data to the receiver using RDMA Write.

The RDMA Write algorithm has two synchronization points. Synchronization is needed for the sender to identify free buffers in the receiver, given that the sender must never overwrite a remote buffer while the receiver is consuming data in the buffer. As shown in Figure 9(a), we use a circular queue *FreeArr* in the SEND endpoint to communicate which buffers are free. Entries in *FreeArr* are populated by the Release function in the RECEIVE endpoint: When the receiver completes consuming data in a buffer, it will call the Release function, which writes the address of the buffer into *FreeArr* with RDMA Write (Algorithm 5, line 20). Entries in *FreeArr* are consumed by the PutData function in the SEND endpoint: Before writing data to the receiver, the sender fetches a remote buffer from *FreeArr* (Algorithm 5, line 7), and uses it as the destination buffer for the RDMA Write request (Algorithm 5, line 9).

Design tradeoff: polling cost vs. number of remote RDMA Write requests. A notification mechanism needs to be implemented in the RECEIVE endpoint to detect which buffer transmissions have been completed. There are a number of design alternatives here that can reduce the polling cost.

One option is to continuously poll the receive buffers to check whether new messages have been received in their entirety. This keeps the number of RDMA Write requests that need to be transmitted to the minimum of two (namely, one to materialize the buffer and another to notify that it has been consumed), but it requires continuously accessing every buffer in the RECEIVE

ALGORITHM 5: RDMA Write with Reliable Connection

```
   Global: mode // {pollbuff: poll buffers, pollmq: poll message queue, pollcq: poll completion queue}
   function PutData(buffer, destarr, state)
1      addr ← address of buffer
2      encode (destarr, state, source, addr) as metadata in buffer
3      foreach node in destarr do
4          if mode is pollcq then
5              while credit[node] <= sent[node] do
6                  wait
7          remote ← FreeArr[node][cons[node]]
8          increment cons[node]
9          PutDataData(mode, remote)
   function void* Acquire( )
10     do
11         buffer ← call ibv_poll_cq to poll for completions
12         gid ← the transmission group buffer was sent to
13     until |𝔾[gid]| completion events have been received;
14     return buffer
   function Release(remote, local, src)
15     if mode is pollcq then
16         call ibv_post_recv for remote RDMA Write with Immediate Data
17         increment credit[src]
18         if enough credit has been accumulated then
19             send credit[src] to src
20     FreeArr[src][prod[src]] ← local
21     increment prod[src]
   function <state, src, remote, local> GetData( )
22     buffer ← PollData(mode)
23     decode (state, source, addr) from metadata in buffer
24     return <state, source, addr, buffer>
```

endpoint. If a cluster has n nodes and the queue length is k buffers per node, then a total of $n \times k$ buffers will need to be accessed.

One way to curtail the number of memory accesses is to make the sender notify the receiver which buffer has been written at the end of the transfer. This allows the receiver to only check a single location per node, or poll n memory locations in total, but requires senders to post three RDMA Write requests to complete a transfer.

Finally, one can use RDMA Write with Immediate Data and receive a notification in the Completion Queue of the RECEIVE endpoint on every completed write. (Recall that there are two versions of the RDMA Write operation and they differ based on whether Immediate Data can be included in the RDMA Write request or not.) However, using RDMA Write with Immediate Data requires posting an RDMA Receive request to match every incoming RDMA Write request, which can be achieved through a credit mechanism. Using RDMA Write with Immediate Data requires polling a single memory location, the Completion Queue, but it requires posting three remote RDMA Writes in the worst case when the credit is updated on every operation. In addition, the receiver can no longer remain passive.

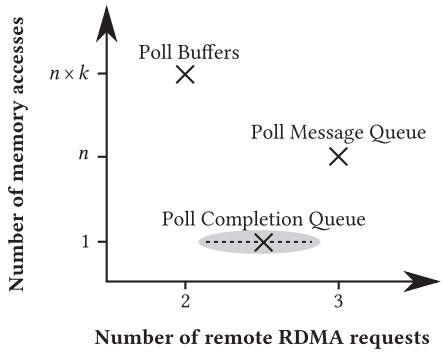
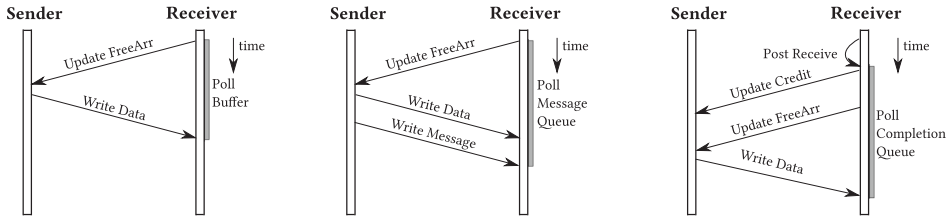


Fig. 10. Comparison of the three variants of the RDMA Write algorithm, where n is the number of SEND endpoints per RECEIVE endpoint and k is the number of buffers in the RECEIVE endpoint.



(a) Poll Buffers (b) Poll Message Queue (c) Poll Completion Queue

Fig. 11. Steps involved in one buffer transfer for the RDMA Write algorithm.

This design tradeoff is shown on Figure 10. No notification mechanism is strictly better than the others. We propose three alternative poll-based methods through which the RECEIVE endpoint can detect the successful transmission of new buffers and describe each in turn.

5.5.1 Poll Buffers. One solution is to continuously poll the buffers themselves for completion. In this method, the SEND endpoint uses RDMA Write without Immediate Data to transmit data to the RECEIVE endpoint. The last byte in the RDMA buffer acts as the "seal" flag that indicates whether the buffer has valid data. This flag is set by the SEND endpoint in the same RDMA Write operation that writes the data. Because RDMA Writes are performed in increasing address order [12], placing the "seal" at the end of the buffer means that it becomes visible at the end of the RDMA Write operation.

Figure 11(a) shows the steps involved in one buffer transfer using the Poll Buffers method. The RECEIVE endpoint polls the "seal" bit of all buffers to detect the completion of an incoming message in the GetData function, which is shown as dash-dotted lines in Figure 9. The sender reads the location of empty buffers from the *FreeArr* and posts RDMA Write requests to the remote buffer using the SendData function (Algorithm 6, line 8). When the RDMA Write requests complete, the PollData function in the receiver returns a buffer to the query engine for consumption (Algorithm 6, line 22). The seal bit is reset by the RECEIVE endpoint after the data has been consumed.

In total, two RDMA requests are posted using the Poll Buffers synchronization method. During polling, the RECEIVE endpoint needs to access $n \times k$ memory locations, where k is the number of buffers in each remote SEND endpoint and n is the number of SEND endpoints in the cluster.

5.5.2 Poll Message Queue. One can improve the memory efficiency of polling by consolidating the "seal" flags that indicate completed transmissions into a single array. Having one compact data structure allows for more efficient polling, as it keeps all the flags contiguously in memory.

Figure 11(b) shows the steps involved in one buffer transfer. The RECEIVE endpoint keeps a circular queue $ValidArr$, shown in dotted lines in Figure 9(b), that keeps track of the buffers that have been populated by the sender. After the receiver notifies the sender of free buffers by writing into $FreeArr$, the receiver continuously polls the message queue $ValidArr$ for new buffers with data in the PollData function (Algorithm 6, line 14). The SEND endpoint first performs an RDMA Write without Immediate Data to transmit the buffer to the RECEIVE endpoint. The sender then performs an additional RDMA Write in the SendData function to set $ValidArr$ for the same buffer (Algorithm 6, line 5). Because RDMA requests are guaranteed to be delivered in order in Reliable Connection, the RECEIVE endpoint will see the notification in the $ValidArr$ queue only after the buffer transmission has completed.

Polling the message queue entails three RDMA requests: one to notify the sender of empty buffers to reuse, one to transmit the buffer, and one to update the queue after the buffer has been transmitted. One round of polling accesses n memory locations for a cluster with n SEND endpoints.

ALGORITHM 6: Functions used in the RDMA Write algorithm.

```
   function SendData(mode, remote)
1      if mode is pollcq then
2          call ibv_post_send to writebuffer into remote with remote to be the Immediate Data
3      else if mode is pollmq then
4          call ibv_post_send to write buffer into remote without Immediate Data
5          ValidArr[node][ prod[node] ] ← remote
6          increment prod[node]
7      else if mode is pollbuff then
8          call ibv_post_send to write buffer into remote without Immdeidate Data
   function void* PollData(mode)
9      if mode is pollcq then
10         buffer ← call ibv_poll_cq to get immediate data
11         return buffer
12     else if mode is pollmq then
13         for i ∈ [1, N] do
14             if ValidArr[i] is set then
15                 buffer ← ValidArr[i][cons[i]]
16                 increment cons[i]
17                 return buffer
18     else if mode is pollbuff then
19         for i ∈ [1, N] do
20             for buffer ∈ LocalArr[i] do
21                 if seal flag of buffer is set then
22                     return buffer
```

5.5.3 Poll Completion Queue. Another design choice is to use the RDMA Write with Immediate Data primitive to offload synchronization to the network: The SEND endpoint will use the address of the remote buffer that is being populated by the RDMA Write request as the Immediate Data. Then, the RECEIVE endpoint polls for a completion event from the Completion Queue and the Immediate Data field of the completion entry will point to the buffer written by the RDMA request.

One challenge is that the RECEIVE endpoint needs to post sufficient RDMA Receive requests to be matched with incoming RDMA Write requests. We use a credit mechanism, shown as dashed lines in Figure 9, to ensure that a sufficient number of RDMA Receive requests have been posted for incoming RDMA Write requests. The credit mechanism is similar to the one used in the RDMA Send/Receive implementation described in Section 5.2. One subtle difference of the RDMA Write credit mechanism is that the RDMA Receive requests do not need to specify buffer addresses, as these will be populated by the sender during the RDMA Write. Hence, the RDMA Write implementation can overextend credit by posting more RDMA Receive requests than the available number of buffers, as there is no longer a one-to-one correspondence between the two.

Figure 11(c) shows the steps involved in one buffer transfer. The receiver first posts a RDMA Receive requests, then updates the corresponding credit in the Release function (Algorithm 5, lines 16–20). The sender posts an RDMA Write with Immediate Data request after retrieving empty buffers from $FreeArr$ (Algorithm 6, line 2). In the PollData function, the receiver polls the local Completion Queue to get new buffers (Algorithm 6, line 10).

Overall, relying on the Completion Queue for synchronization requires posting at most three RDMA requests to the remote NIC for updating the credit, modifying the free buffer queue and transmitting the buffer, respectively. The three RDMA Write requests that need to be transmitted in the network is a worst-case analysis, as credit updates can be amortized over multiple requests. As the Completion Queue is shared by all the Queue Pairs of the RECEIVE endpoint, one round of polling involves checking only one Completion Queue.

5.6 Making Endpoints Thread-safe

As described in Section 4.3, the endpoint will be shared by multiple threads in the single endpoint configuration. Hence, the implementation of the endpoints should be thread-safe. The interfaces provided by the RDMA, such as posting RDMA requests and polling Completion Queue are thread-safe, and can be directly called by the threads. In the implementation of endpoints, we protect the shared data structures by using atomic operations. For example, one data structure that is updated by multiple threads concurrently is the "credit" in the RDMA Send/Receive implementation. Every thread increments the credit for a specific node after posting an RDMA Receive request, and we use the atomic increment operation to do the update. The interfaces of the endpoints are thread-safe.

6 BUFFER QUEUE MANAGEMENT IN RDMA-AWARE DATA SHUFFLING

The memory buffers that are used for RDMA transmission are pinned in physical memory. Unlike TCP/IP-based communication, the depth of the message queues needs to be managed by the data shuffling operator itself to achieve good performance and reasonable memory consumption.

There is an inherent tradeoff in deciding how much memory should be registered for RDMA operations. Using too little memory for communication will lead to starvation if there are insufficient buffers to overlap computation and communication. However, using too much memory for communication limits the memory available for other uses, since the RDMA buffers are pinned in physical memory. Given the speeds of high-performance networks, a buffer that keeps data for a fraction of a second would require many gigabytes of pinned memory.

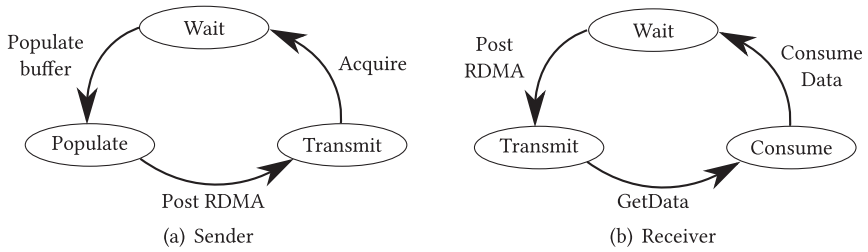


Fig. 12. Buffer states in our algorithms.

This section introduces two mechanisms to manage the RDMA buffer queue depth. Section 6.1 introduces the bandwidth-delay product (BDP), which is widely used for buffer size tuning in networking [42]. Section 6.2 introduces our adaptive buffering algorithm.

6.1 Query-oblivious Buffering Using the Bandwidth-delay Product

The bandwidth delay product defines the amount of data that should be in transmission at any time to fully utilize the available channel capacity. The bandwidth-delay product (BDP) is formulated as follows:

$$BDP = B_{available} \times t_{latency}, \tag{1}$$

where $B_{available}$ denotes the available bandwidth and $t_{latency}$ denotes the latency of the network. The bandwidth-delay product is a fixed value that reflects how much memory the receiver needs to process pending messages at line rate until the sender reacts to a message sent by the receiver. Hence, the bandwidth-delay product offers a lower bound on the buffer queue depth that will be necessary to fully utilize the network bandwidth.

The drawback of the bandwidth-delay product is that it does not consider the latency of data processing at the receiving end of the query pipeline. As a consequence, the bandwidth-delay product will be accurate for "shallow" pipelines (that is, query fragments with few operators after the receiver), but it will underestimate the optimal queue depth for "deep" pipelines with many operators. Motivated by this observation, we introduce a dynamic algorithm that can capture the rate of data production and the data consumption latency.

6.2 Query-aware Buffer Management

Buffer management can adapt to the query by changing the number of RDMA buffers that are used during query execution. A shared pool of buffers is registered when the database system starts. When an operator needs more buffers, it fetches buffers from the shared pool. When the operator has more buffers than necessary, it returns buffers to the pool. The pool registers additional RDMA buffers if there is demand and de-registers RDMA buffers if there is memory pressure.

The query-aware buffer management technique classifies buffers in one of three states. In the sender, shown in Figure 12, the three states are as follows: $Wait$, in which the buffer is free and can be used or retired; $Populate$, in which the buffer is being populated by local threads; and $Transmit$, in which the buffer is being accessed by the NIC. A buffer transitions from $Wait$ to $Populate$ before a local thread begins writing to the buffer; it transitions from $Populate$ to $Transmit$ when the buffer is posted for RDMA transmission; and it transitions from $Transmit$ to $Wait$ when the associated completion event has been consumed by the sender. Similarly, there are three states in the receiver, $Wait$, $Transmit$, and $Consume$, in which the data in the buffer are being processed by the local threads. A buffer transitions from $Wait$ to $Transmit$ when the buffer is posted for receiving data; it transitions from $Transmit$ to $Consume$ when the buffer is ready to be processed by a local thread;

and it transitions from *Consume* to *Wait* when the local thread has completed processing the data in the buffer. The number of buffers in each state changes during query execution.

The decision whether to return buffers to the pool or request more buffers from the pool is based on the number of buffers in the *Wait* state. In particular, the goal is to always keep a few buffers in the *Wait* state. Intuitively, if there are no buffers in the *Wait* state and calling the Acquire function does not return new buffers, the sender is starved for buffer space. The sender should then add a new buffer in circulation from the buffer pool to avoid waiting. (Ditto for the receiver.) But when does the sender have too many buffers? We empirically found that a good heuristic is to check how many buffers can transition to the *Wait* state and release all but one buffer back to the buffer pool. The intuition behind this limit is that returning exactly one buffer into the *Wait* state is exactly how many buffers the calling thread will take out of the *Wait* state; hence, this does not increase the number of buffers in the *Wait* state.

The implementation of the adaptive buffer management algorithm builds on the endpoint interface that was introduced in Section 4.2. The Acquire and GetData functions are modified to be non-blocking and instead return as many buffers that are ready (which may be zero if no buffer is ready). We refer to the modified functions as AdaptiveAcquire and AdaptiveGetData, respectively. The AdaptiveAcquire function in the SEND endpoint checks the number of buffers in the *Wait* state and how many are returned by calling Acquire. If the number is zero, then this means the sender has a shortage of buffers. AdaptiveAcquire fetches a buffer from the buffer pool instead and returns it. If Acquire returned more than one buffers, then the AdaptiveAcquire function will retire the additional buffers to the buffer pool and return only one buffer to the caller. Similarly, the AdaptiveGetData function in the RECEIVE endpoint checks the number of buffers in the *Wait* state and how many are returned by GetData to determine whether to add or mark the buffer for removal from circulation. The buffer is returned to the buffer pool only after the data in the buffer have been consumed.

This adaptive buffer management procedure is cognizant of the query workload. In a long pipeline, populating or consuming a buffer takes longer, which leaves fewer buffers in the *Wait* state. The algorithm reacts to this by adding additional buffers in circulation from the buffer pool. Conversely, populating or consuming a buffer is very fast in a short pipeline, which means that multiple buffers are transitioning to the *Wait* state per invocation. The algorithm corrects this overproduction by retiring buffers to the buffer pool.

7 EXPERIMENTAL EVALUATION

We have implemented all variants of the data SHUFFLE and RECEIVE operators in a prototype open-source in-memory query engine that we have written in C++ [51]. We evaluate data shuffling in two shared clusters. One cluster is connected by an FDR (56 Gb/s) InfiniBand network. Each node in the FDR cluster has 64-GiB memory across two NUMA nodes with Intel Xeon E5-2670v2 10-core processors. The other shared cluster is connected by an EDR (100 Gb/s) InfiniBand network. Each node in the cluster has 128-GiB memory across two NUMA nodes with two Intel Xeon E5-2680v4 14-core processors. We use eight nodes in the evaluation, unless otherwise specified. The questions we evaluate and the insights from the evaluation are as follows:

- What is the overhead of flow control when using the two-sided RDMA Send/Receive transport function? Section 7.1.1 shows that the cost of software flow control is negligible.
- Which transmission completion technique for the RDMA Write algorithm performs the best? Section 7.1.2 shows that polling the Completion Queue has the best performance despite its high messaging cost.

- What message size should one pick for the algorithms that use the Reliable Connection transport service? Section 7.1.3 shows that a message size of 64KiB offers a good balance between performance and memory consumption.
- How does the repartition and broadcast throughput scale as the cluster size increases? Section 7.1.4 shows that the MESQ/SR algorithm has good scalability in both FDR and EDR clusters and outperforms the MPI and IPoIB algorithms by as much as 4×.
- How does the number of Queue Pairs affect performance? Section 7.1.5 shows that the MESQ/SR algorithm uses fewer Queue Pairs and achieves higher throughput than the MQ algorithms.
- How significant is the connection setup time for RDMA? Section 7.1.6 shows that the setup cost for the MESQ/SR algorithm is stable and less than 40 ms as the cluster size increases.
- What is the performance with compute-intensive queries? Section 7.1.7 shows that all algorithms except SESQ/SR successfully overlap communication and computation as the queries become more compute-intensive.
- What is the performance of the adaptive buffering algorithm compared with the fixed buffering algorithm? Section 7.1.8 shows that the adaptive algorithm always has comparable or better performance than the fixed buffering algorithm.
- Does a faster network improve end-to-end query performance? Section 7.2.1 shows that MESQ/SR offers higher throughput than MPI for both the FDR and the EDR cluster. MESQ/SR successfully overlaps communication and computation.
- How does query response time scale as the database size grows proportionally to the cluster size? Section 7.2.2 shows that MESQ/SR has better scalability than MPI and outperforms MPI by up to 2×.

7.1 Evaluating Shuffling Throughput Using Microbenchmarks

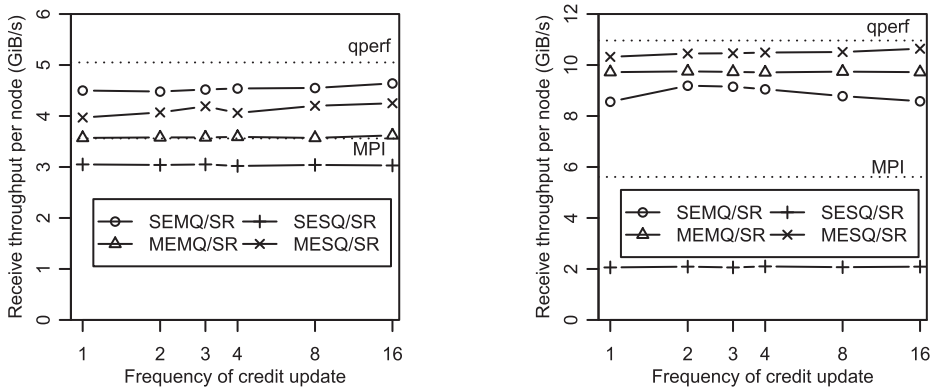
This section uses a synthetic workload to study the receive throughput per node with different data shuffling algorithms.

We generate a synthetic table R with two long integer (8-byte) attributes $R.a$ and $R.b$ for evaluation. $R.a$ is uniformly distributed. The table has 1 billion tuples and the size of the table is 16GiB. This table is replicated in each node of the cluster and each node randomly permutes the local fragment of R before the experiment starts.

We evaluate the throughput of the data shuffling operation with a synthetic query. In this query, all nodes scan the local fragment of table R and repartition R using $R.a$ as the key. The communication pattern corresponds to repartitioning data that is uniformly and randomly distributed. We calculate the total throughput as the reciprocal of the query response time and the total number of nodes in the cluster. This underestimates peak throughput as all shuffling operations will not complete at the same time. To amortize transient fluctuations in network performance, the measurement window is at least 15s long for all experiments in this section. This is accomplished by scanning and transmitting the R table 10 times such that 160GiB per node are transmitted. We do not take the time to build RDMA connections and close them into account when calculating the throughput. This does not meaningfully impact the throughput calculation as the setup overhead is in the millisecond range (see Section 7.1.6).

As we are not aware of any RDMA-capable parallel query engine to use for direct comparison, we revert to three performance baselines.

(1) The first comparison baseline is `qperf` [17], a bandwidth benchmarking tool. The sender in `qperf` registers a single buffer for data transfer and keeps posting RDMA Send requests. The receiver in `qperf` continuously posts RDMA Receive requests in an infinite loop.



(a) 56 Gbps FDR InfiniBand (b) 100 Gbps EDR InfiniBand

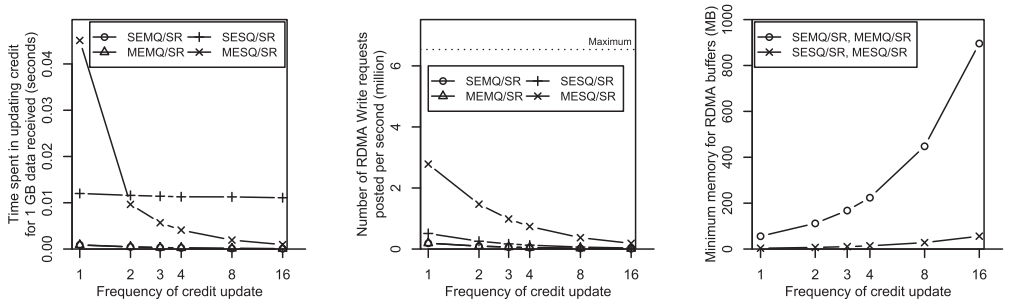
Fig. 13. Performance of the MQ/SR and SQ/SR algorithms when changing the credit write back frequency.

Neither the sender nor the receiver access the transmitted buffer. Results from `qperf` reflect the peak networking capability of the cluster.

(2) The second baseline we compare with is TCP/IP-based communication over InfiniBand ("IPoIB"). This reflects the performance of a network upgrade without any changes in software. The sender in the IPoIB algorithm uses the *send*() function to transmit data through a socket. The receiver uses *select*() to monitor all sockets for activity and calls *recv*() on any socket that has data. We fix the message size to 128KB as the algorithm has the best performance with this message size.

(3) Finally we compare with the MPI [38]. MPI is an interface specification widely used in HPC applications. MPI defines high level primitives such as send, receive, reduce, broadcast for the application to use. Low level details, such as memory management, are hidden from the application. MPI has different implementations and we compare with the MVAPICH [32] implementation that uses RDMA for communication. The MVAPICH [32] implementation communicates with the Reliable Connection transport service type. Small messages and control messages use the hybrid of RDMA Send/Receive and RDMA Write primitives and large messages use the RDMA Write primitive for communication. We have implemented an endpoint using MPI. In the repartition algorithm, the sender uses *MPI_Send* to send data while the receiver calls the *MPI_Irecv* function to retrieve data. The MPI implementation uses the *MPI_Ibcast* primitive for the broadcast algorithm. We use 64KB as the message size in MPI, which is also the message size used in our MQ algorithm when communicating with the Reliable Connection transport.

7.1.1 Flow Control Overhead in RDMA Send/Receive. The two-sided RDMA Send/Receive implementation synchronizes the sender and the receiver through the credit protocol (described in Section 5.2) so that the receiver does not drop messages because it is overwhelmed by the sender. This section evaluates what is the overhead of the credit mechanism and how frequently credit should be written back. This experiment uses eight nodes and each thread registers 16 RDMA buffers per remote node. Here the size of each message buffer is 64KB in MQ algorithms and 4 KB for SQ algorithms. Figure 13 shows the throughput at the receiver for all RDMA Send/Receive algorithms for different credit update frequencies. The horizontal axis shows the credit write back frequency, measured as the number of RDMA Receive requests the receiver posts before it updates the credit value in the sender. The vertical axis shows the receive throughput for each algorithm.



(a) Time the receiver spends updating credit for every 1 GB of data received.

(b) RDMA Write requests posted for credit update per second.

(c) Minimum memory needed for different credit update frequencies, for an 8-node cluster with 14 threads per node.

Fig. 14. Profiling of the performance when credit update frequency changes.

The results show that the performance impact of the credit mechanism is not very significant when compared to the "qperf" result that does not include a credit mechanism to prevent buffer overruns at the receiver. In addition, the frequency of the credit update has very little effect on the achieved throughput.

Analysis. A deeper analysis reveals that changing the credit update frequency significantly impacts resource utilization, although this is not discernible if one solely considers the achieved throughput. We now consider four more detailed metrics:

- *The time the receiver spends updating credit:* Figure 14(a) shows the time receivers spend to update credit for every 1GB of data received. With the exception of updating credit for MESQ/SR on every message, the time to update credit is negligible.
- *The total number of RDMA Write requests posted for the credit mechanism:* Figure 14(b) shows the number of RDMA Write requests posted to update credit. We show the maximum of RDMA Write requests one QP can post from a microbenchmark as a dotted line. The result shows that the RDMA Write requests posted for updating credit do not come near the saturation point of the network even when updating the credit on every message using a single QP ("SQ" algorithms).
- *The minimum amount of memory required for communication:* It is not desirable to update the credit very infrequently. Less-frequent credit updates requires more RDMA buffers to be registered. For example, if the credit update frequency is set to be 16, then each thread must have at least 16 buffers for each remote destination; otherwise, the remote destination node will never accumulate 16 credits—the threshold for a credit update. Figure 14(c) shows the minimum amount of RDMA memory needed for an eight-node cluster with 14 threads per node when the credit update frequency changes. The memory needed to support a credit update frequency of 16 can be nearly 1GB across the entire cluster for a single data transfer operation inside one query. Hence, frequent credit updates are preferred.
- *The time a sender is blocked for credit to send data:* We observed that the time senders are blocked for credit is comparable between all algorithms. Furthermore, no statistically significant effect was found between the time a sender is blocked for credit when changing the frequency of the credit update. We omit the detailed results for brevity.

Based on these results, we configure the receiver in all RDMA Send/Receive algorithms to write back the credit after posting 2 RDMA Receive requests for the remainder of the evaluation.

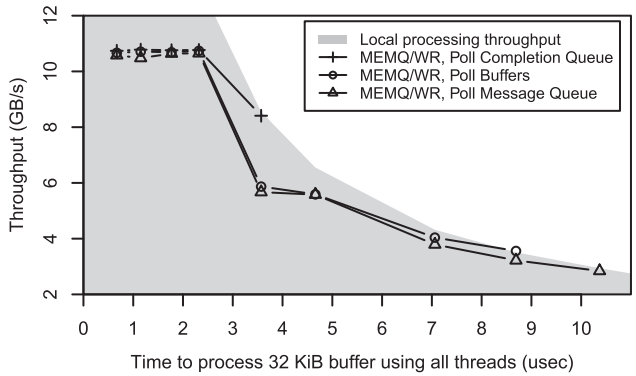


Fig. 15. All notification mechanisms for the RDMA Write implementation perform comparably with network-bound queries. Polling the Completion Queue has the best network performance when the query becomes compute-bound, as it completely overlaps computation and communication. Polling the Completion Queue results in up to 1.5× faster data transfer for compute-bound queries.

7.1.2 Evaluation of Different Notification Mechanisms for the RDMA Write Algorithm. This experiment evaluates which of the three variants of the RDMA Write implementation achieves the best performance. As discussed in Section 5.5, each variant differs in terms of the number of posted RDMA requests and the number of memory accesses it performs.

Figure 15 compares the `Poll Buffer`, `Poll Message Queue`, and `Poll Completion Queue` variants of the Multiple Endpoint RDMA Write (MEMQ/WR) algorithm. (Although we only show data from the Multiple Endpoint (ME) algorithm, the performance of the Single Endpoint (SE) algorithm is similar.) In this experiment, the receiving plan fragment continuously receives data in batches of 32KiB—the L1 data cache size of our system—and sums all the received values together. We change the compute intensity of the receiving fragment by forcing the compiler to compute the sum multiple times. This increases the instruction path length on the receiving side and thus simulates more CPU-intensive query fragments.

The horizontal axis of Figure 15 shows the average time it takes to process a 32KiB batch of data. When the plan fragment is more compute intensive (moving right on the horizontal axis) the receiving query fragment takes longer to process the data and thus the time to retrieve the next batch increases. (Note that the horizontal axis does not directly correspond to the processing time per batch: All threads process data concurrently in the receiving query fragment and any thread can "snatch" the next batch for processing.)

The vertical axis of Figure 15 shows the local processing throughput of the receiving fragment (in gray) and the throughput of each variant of the RDMA Write algorithm. The local processing throughput of the receiving fragment was measured by placing the data in local memory and changing the receiving query fragment to scan locally instead of retrieving data remotely; the local processing throughput is identical for all three variants. Each line plots the network throughput of a different RDMA Write variant. When a mark is within the gray area, this means that this configuration is network-bound, as network throughput is less than the local processing throughput. When the network throughput reaches the throughput of local processing, this means that the query is now compute-bound. Once a variant becomes compute-bound we stop plotting it to not clutter the graph.

Analysis. The results in Figure 15 show that polling the Completion Queue fully overlaps computation and communication even for fragments with low compute intensity (as low as 3.5μs per

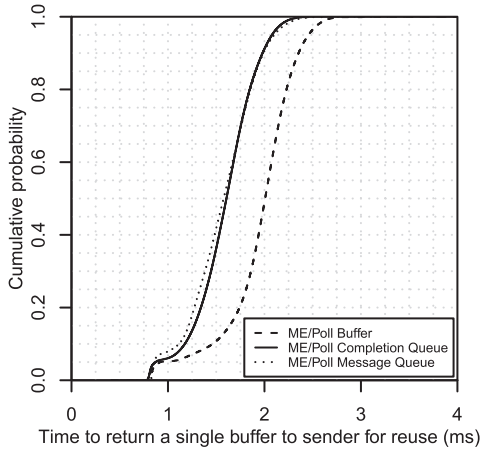


Fig. 16. The cumulative distribution function of the time it takes a sender using RDMA Write to reuse a remote buffer with different notification mechanisms. In this experiment, the query is network-bound (x = 2.3μs in Figure 15). Detecting message completion by polling the buffers is more CPU-intensive and takes up to 0.5 ms longer than the other two methods to return a free buffer to the sender.

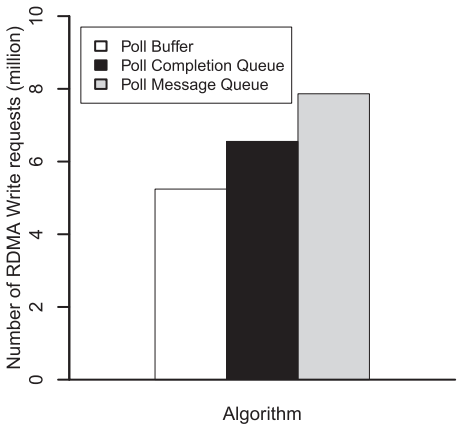
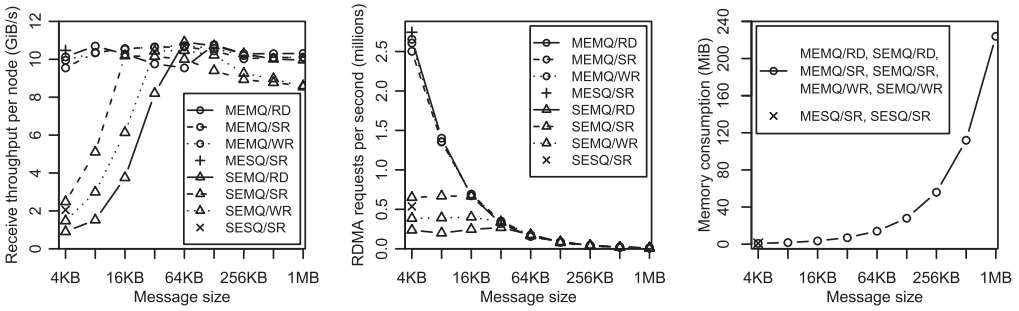


Fig. 17. Number of RDMA Write requests posted in each algorithm. In this experiment, the query is network-bound (x = 2.3μs in Figure 15). Polling the Completion Queue produces fewer RDMA Write requests because the credit update can be amortized over multiple data transfers.

batch). This is because the `Poll Completion Queue` variant requires only a single memory access when polling (cf. Figure 10). Polling the Completion Queue results in data transfer speeds that are up to 1.5× faster than the other two variants for compute-bound queries.

Additional insights can be gleaned if one considers the time it takes for a free buffer to be returned to the sender for reuse. For compute-bound queries, this time would be dominated by the compute cost of consuming the data on the receiving side. Figure 16 shows the cumulative distribution of the time for a network-bound query ($x = 2.3\mu$s in Figure 15). The result shows that the `Poll Buffer` algorithm takes about 0.5ms longer for a buffer to be returned, as it takes more memory accesses to identify which transmissions have been completed. Polling the buffers is more CPU intensive, as it requires $n \times k$ memory accesses, where n is the cluster size and k is the buffer queue depth. The other two methods, polling the Completion Queue and polling the Message Queue, have statistically indistinguishable delay for network-bound queries. We hypothesize that this is due to caching effects from polling 1 and n memory locations, respectively, for small clusters like shown here (where $n = 8$). This experiment shows that the cost of polling all receive buffers to check for completions is substantial, even for network-bound queries that leave abundant CPU cycles for polling.

When comparing the `Poll Completion Queue` and `Poll Message Queue` variants, one finds that polling the Completion Queue has a decisive advantage with respect to the number of RDMA Write messages that need to be transmitted. Figure 17 shows the number of posted RDMA Write requests for the same network-bound query. Both methods require one RDMA Write to notify the sender that the buffer is free and one to transmit the data. Polling the Message Queue always requires an additional message (or three messages per transmission) to notify the receiver which buffer is free. In contrast, polling the Completion Queue requires updating the credit to ensure



(a) Throughput

(b) Average number of RDMA messages sent out by one node per second.

(c) Memory consumption

Fig. 18. Effect of message size for EDR InfiniBand cluster.

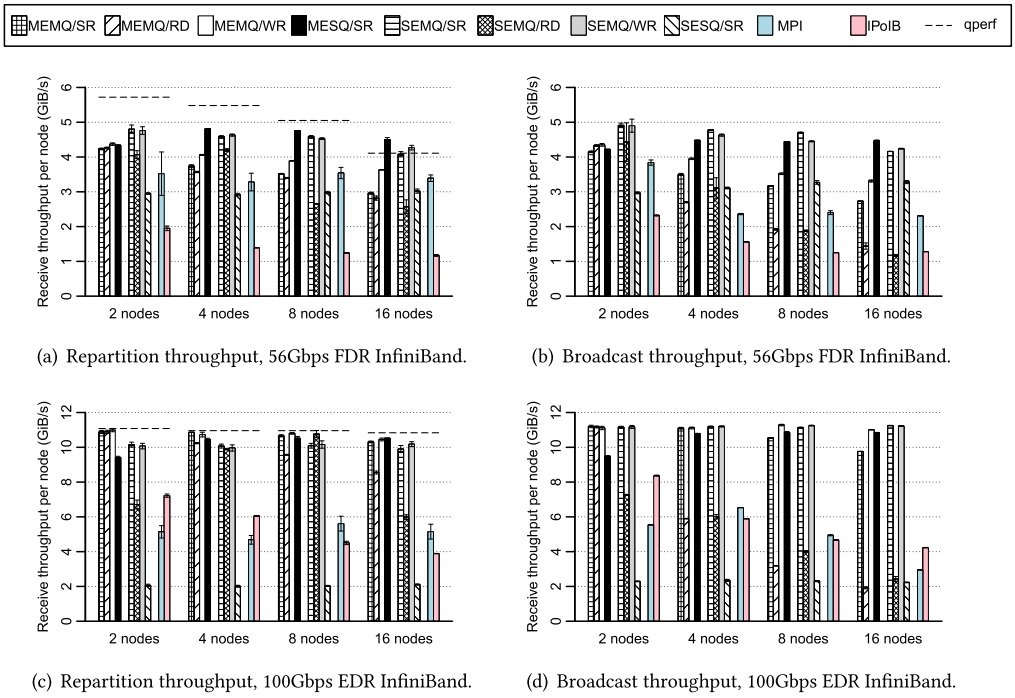
that the receiver expects the RDMA Write. This credit update is batched over $b = 2$ operations, thus bringing the number of messages down to $2 + \frac{1}{b} = 2.5$ messages per transmission.

We use the `Poll Completion Queue` variant of the RDMA Write implementation in all the experiments that follow because of its higher performance with compute-intensive queries and its frugal use of CPU cycles and RDMA Write messages.

7.1.3 Effect of Message Size in Reliable Connection. All shuffling algorithms accumulate tuples in an RDMA-registered buffer and send buffers out as one RDMA message. In our hardware the Unreliable Datagram transport only supports messages that are up to 4KiB big; however, the Reliable Connection transport supports messages as big as 1GiB [3, 36]. One thus needs to tune the message size for all the algorithms that use the Reliable Connection transport, which are the algorithms that include "MQ" in their name (cf. Table 1). This experiment runs on eight nodes in the EDR cluster and uses double buffering, i.e., every thread will register two RDMA buffers for each destination. Figure 18 shows the throughput per node of each algorithm for message sizes between 4KiB and 1MiB. For the algorithms with a single endpoint (i.e., starting with "SE"), small message sizes (up to 16KiB) significantly underutilize the network. For the algorithms with multiple endpoints (i.e., starting with "ME"), the performance stays close to the peak throughput regardless of the message size.

Analysis. The throughput results in Figure 18(a) show that small message sizes underutilize the network for all algorithms with a single endpoint (i.e., starting with "SE"). The reason for this underutilization is because a single endpoint hits the limit of how many small RDMA messages can be posted per second. Figure 18(b) plots the number of RDMA messages sent out per second by one node. Because small message sizes lead to more messages, the SE algorithms hit the limit of the number of RDMA messages when the message size is small. This is not a problem for ME algorithms as there are multiple endpoints per operator. Hence, the performance of the SE algorithms is limited by the number of RDMA messages one endpoint can post per second.

Very large message sizes are undesirable in practice, however, due to the need to allocate and pin a substantial amount of memory for RDMA accesses. Figure 18(c) shows the memory registered for RDMA communication (vertical axis) as the message size changes (horizontal axis) when running on eight nodes in the EDR cluster. As the message size approaches 1MiB, the pinned RDMA memory can be more than 100MiB for a single shuffle operator, as the operator needs to keep buffers for all open connections to other nodes of the cluster. If one considers that query plans consist of multiple shuffle operators and parallel database systems may execute dozens of query fragments



(a) Repartition throughput, 56Gbps FDR InfiniBand.

(b) Broadcast throughput, 56Gbps FDR InfiniBand.

(c) Repartition throughput, 100Gbps EDR InfiniBand.

(d) Broadcast throughput, 100Gbps EDR InfiniBand.

Fig. 19. Throughput when changing the number of nodes in the cluster.

concurrently, then a message size around 1MiB may translate into a pinned memory footprint of 10GiB or more. In comparison, the RDMA Send/Receive algorithms (ending in "SR") that use the Unreliable Datagram protocol have a decisive space advantage, as they require under 1 MiB of pinned memory to reach their peak throughput.

The optimal message size value should be set to achieve peak throughput while using as little memory as possible. Based on these results, we fix the message size to be 64KiB for the algorithms that use the Reliable Connection transport (i.e., have "MQ" in their name) and use double buffering for all algorithms.

7.1.4 Throughput When Scaling Out. This section studies the performance of the eight algorithms when increasing the number of nodes. In this experiment, we run the eight algorithms using 2, 4, 8, and 16 nodes. In the repartition experiments, we use the modulo hash function to repartition data such that every node receives the same amount of data. Figure 19 shows the average receiving throughput with bar plot and their standard deviations as error bars. In addition to the eight RDMA-aware algorithms, we also run experiments with MPI and IPoIB. The vertical axis shows the receive throughput per node. The dashed lines represent the throughput reported by `qperf`, while bars show the throughput of each algorithm. Since `qperf` does not support the broadcast pattern, we omit the throughput measurement for `qperf` in the broadcast result.

Figure 19(a) and (c) show performance for the repartition pattern in the FDR and the EDR cluster, respectively. First, our RDMA-aware algorithms outperform the MPI algorithm by as much as 2× (see Figure 19(c), MESQ/SR vs. MPI with 16 nodes in the EDR cluster) and outperform the IPoIB algorithm by as much as 3× (see Figure 19(a), MESQ/SR vs. IPoIB with 8 nodes in the FDR cluster). Looking at the FDR cluster in Figure 19(a), the MESQ/SR algorithm has comparable performance to all other algorithms when the cluster size is small but exhibits better scalability than other

algorithms as the cluster size increases. This is because of the high number of open connections: As shown in Table 1, the number of open connections per node for the MESQ/SR algorithm is constant while the number of open connections per node increases proportionally to the cluster size for all the MQ algorithms. For 16 nodes, the MESQ/SR algorithm even outperforms `qperf`. This is because `qperf` runs using the Reliable Connection transport and thus requires $O(n)$ connections per node for a cluster with n nodes. Looking at the EDR cluster in Figure 19(c), the MESQ/SR algorithm has good performance when scaling out. Unlike the results from the FDR cluster, the performance of the MQ/SR and MQ/WR algorithms does not degrade as the cluster grows. This is because the EDR hardware can cache more QP data for more point-to-point connections [23]. We anticipate that the degradation that was observed in the FDR cluster will manifest in larger clusters where the larger cache size will not suffice to cache all connection data.

The results for the broadcast pattern are shown in Figures 19(b) and (d). The RDMA-aware algorithms outperform MPI by as much as 4× (see Figure 19(d), SEMQ/SR vs. MPI with 16 nodes in the EDR cluster) and outperform IPoIB by as much as 3× (see Figure 19(b), MESQ/SR vs. IPoIB with 16 nodes in the FDR cluster). As also seen in the repartition pattern, the MESQ/SR algorithm shows good scalability. In contrast to the repartition results, the performance of MEMQ/RD and SEMQ/RD degrades significantly in the broadcast communication pattern. This is because in the broadcast pattern the RDMA Read algorithms reuse a buffer only when all the nodes finish reading its data. Receivers using the RDMA Read algorithm will starve for free buffers if there is some load imbalance or a transient network degradation.

Analysis. A deeper analysis reveals that the bottlenecks during execution are different for each algorithm. When profiling a run on 8 nodes of the EDR cluster with the repartition pattern, we found that the IPoIB algorithm spends about $^2/_3$ of all cycles in the *send* and *recv* functions. For the RDMA-based algorithms, the most CPU-intensive activity on the sender is hashing the individual tuples and copying them to RDMA-registered memory. One can further reduce this overhead using vectorization [5]. The SESQ/SR algorithm is bottlenecked due to thread contention in the *ibv_post_send* function. The MEMQ/SR and MESQ/SR algorithms are blocked for credit, while the remaining RDMA-based algorithms are blocked on the completion of pending RDMA operations. Overall, the best-performing algorithms leave the sender idle for about 30% of the cycles. On the receiving side, all RDMA algorithms are blocked on the completion of prior RDMA operations and up to 90% of the cycles are idle.

We conclude that the RDMA shuffling algorithms achieve throughput close to the line rate for FDR and EDR InfiniBand. The MESQ/SR algorithm, in particular, shows good scalability in both. Overall, the RDMA-aware data shuffling algorithms outperform MPI and IPoIB by as much as 4×.

7.1.5 Effect of Multiple Queue Pairs per Node. This section shows the throughput of the RDMA algorithms when using a different number of Queue Pairs. Prior work has shown that the number of Queue Pairs significantly impacts performance [22, 23]. In this experiment, all algorithms repartition data on 16 nodes of the EDR cluster using a different number of endpoints. The "SE" corresponds to the configuration when one operator has only one single endpoint. The "ME" corresponds to the configuration when each thread in the operator has one endpoint. The result is plotted in Figure 20. The horizontal axis is the number of Queue Pairs and the vertical axis is the receiving throughput per node. The result shows that the MESQ/SR algorithm achieves higher throughput with fewer Queue Pairs than the algorithms that rely on the Reliable Connection transport ("MQ"). In a larger cluster all the MQ algorithms would use proportionally more Queue Pairs per operator, whereas the SQ/SR algorithm would use the same number of Queue Pairs per operator.

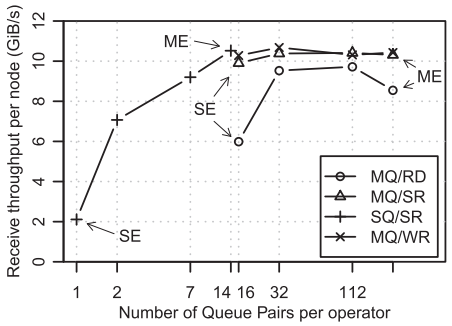


Fig. 20. Effect of many Queue Pairs.

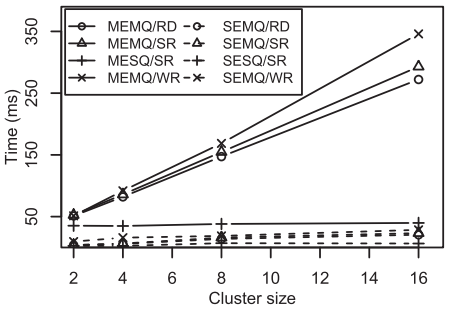


Fig. 21. Time to build RDMA connection.

7.1.6 Startup Cost of Setting up RDMA Communication. Some queries do not shuffle a lot of data. For such queries, the communication initialization time matters more than the peak throughput. One question is whether a database system can afford to build the RDMA connections at runtime. To answer this question, we show the time to build RDMA connections in Figure 21 as the cluster size changes. The reported time on the vertical axis is the time spent creating the connection, registering memory, and de-registering memory in the EDR cluster. The horizontal axis is the number of nodes and the vertical axis is the time to build the RDMA connections.

The multi-endpoint ("ME") algorithms take longer to initialize as they construct many more connections than than the single-endpoint ("SE") algorithms. The connection time increases linearly with the cluster size for all MQ algorithms and stays stable for the SQ algorithms as the cluster size increases. (This is because the connection time is proportional to the number of Queue Pairs; see Table 1.) Overall, the set up time for the MESQ/SR algorithm is about 40ms and does not depend on the cluster size. Query fragments that shuffle as little as 250MB of data using the MESQ/SR algorithm will complete the transmission faster when building connections at runtime than when using pre-allocated connections with IPoIB or MPI. This means that building RDMA connections at runtime outperforms MPI and IPoIB when the size of data shuffled exceeds 250MB.

7.1.7 Performance with Compute-intensive Queries. The experiments so far have compared all algorithms with a network-bound query. This section studies how the different shuffling algorithms perform when the query becomes compute-intensive. This experiment adjusts the compute intensity of the receiving query fragment to simulate different compute demands of real queries. The methodology and the presentation of the result in Figure 22 is the same as in Section 7.1.2.

Figure 22 shows that all algorithms are network-bound if the receiving fragment does minimal processing. At the leftmost point, the throughput of the data shuffling algorithm (~11GiB/s) is about 20% of the throughput of the receiving query fragment (~50GiB/s). As the receiving query fragment becomes more compute-intensive, the MQ/SR, MQ/WR, and MESQ/SR algorithms reach relative peak throughput earlier than the MQ/RD algorithms. The relative throughput of MQ/SR and MESQ/SR even exceeds 100%. This is because of TLB caching: The baseline experiment that measures the local processing throughput reads 160GiB of raw data per node from local memory, whereas the RECEIVE endpoint in the RDMA experiments only polls the registered RDMA buffers (7MB per node). Interestingly, MPI and IPoIB fail to completely overlap communication and computation even for compute-intensive queries. Overall, all RDMA algorithms except SESQ/SR outperform MPI and IPoIB for both network-bound and compute-bound queries.

7.1.8 Adaptive Buffering. This section compares the performance of the adaptive buffering algorithm (described in Section 6) with the algorithm that uses a fixed number of buffers. The

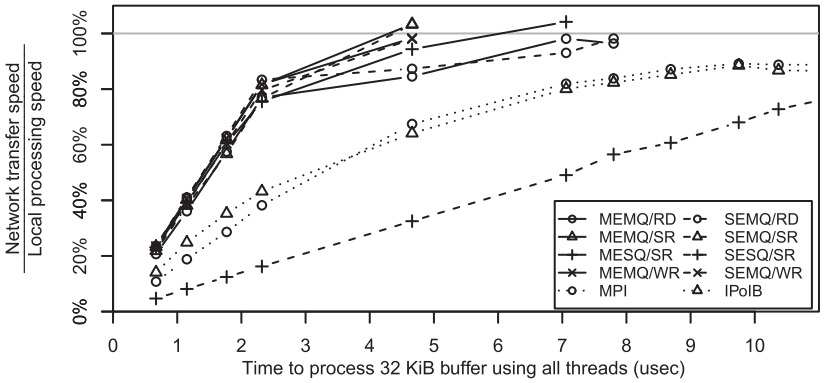


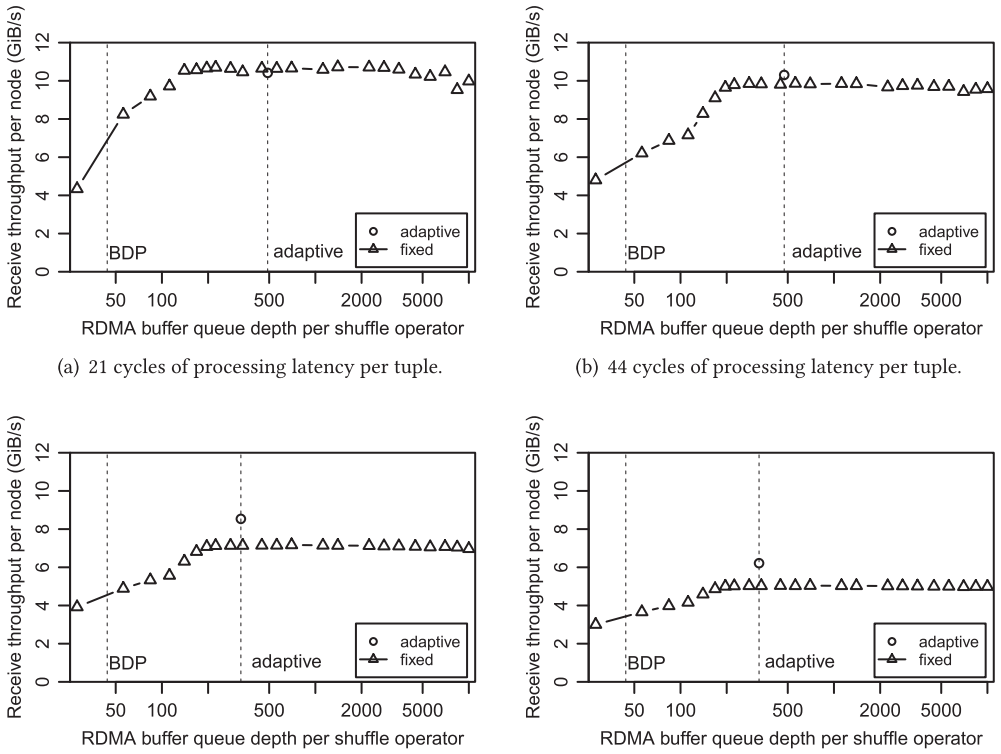
Fig. 22. Performance for compute-intensive queries.

adaptive buffering algorithm is implemented in the receiving side of the MESQ/SR algorithm. To capture the performance envelope of the fixed buffering algorithm, we vary the number of buffers from 28 (single buffering) to 10,000. In addition, the latency of data processing also influences the optimal number of buffers. We emulate different data processing latencies by changing the compute intensity of the receiving query fragment using the methodology described in Section 7.1.2. We use the average latency per tuple to quantify the processing latency, which can range from as little as 21 cycles (approximately an L2 cache miss or a branch misprediction) to 95 cycles per tuple (approximately an L3 cache miss).

Experiments with fixed processing latency. Figure 23 shows the throughput of the adaptive and static buffering algorithms when running on two nodes of the EDR cluster. Each figure corresponds to a different fixed processing latency. The horizontal axis is the number of buffers used by the algorithm while the vertical axis shows the receiving throughput. The dashed "adaptive" line reflects the most frequently picked number of buffers of the adaptive buffering algorithm. The dashed "BDP" line reflects the result from the bandwidth-delay product calculation, which is 44 buffers for the EDR cluster. The bandwidth and round-trip latency used in the BDP calculation are obtained from `qperf`; the calculation is done statically and it is oblivious to the query processing load.

Figure 23 shows that the adaptive algorithm has better performance than picking a fixed number of buffers using the bandwidth-delay product. The number of buffers computed using the bandwidth-delay product law does not achieve peak performance, because the calculation does not take the data processing latency into account. The result also shows that our adaptive algorithm uses fewer buffers when the CPU intensity of query fragments increases and it takes more CPU cycles to process one tuple. This is because for CPU intensive queries, CPU rather than network is the bottleneck. While for queries that are less CPU intensive, the network is the bottleneck and the adaptive algorithm will try to improve the network performance by using more RDMA buffers.

Interestingly, the adaptive buffering algorithm has better performance than the fixed buffering algorithm when the data processing latency is high (see Figure 23(c) and (d)). This is because the adaptive algorithm polls the completion queue for multiple entries and returns unused buffers in the pool. This does not make much difference when the data processing latency is low and the query is network bound, as most of the time polling returns zero or one entries. However, when the data processing latency increases, the query becomes CPU bound, thus it is more common for the adaptive algorithm to get more than one entries back from the completion queue and deposit unused buffers in the pool. This amortizes the cost of polling and improves overall performance.



(a) 21 cycles of processing latency per tuple.

(b) 44 cycles of processing latency per tuple.

(c) 63 cycles of processing latency per tuple.

(d) 95 cycles of processing latency per tuple.

Fig. 23. Performance of the fixed and adaptive RDMA buffer queue management algorithms.

Experiments with skewed processing latency. The next experiment evaluates how the adaptive algorithm compensates for skew during data transmissions. Skew naturally arises when database systems are deployed. The sources of skew are many, ranging from datacenter topology where communication across racks has a longer network path compared to communication within a rack, to compute skew where different fragments have different data processing latency. This experiment transmits data between two nodes using the adaptive buffering strategy, where one node takes 21 cycles to process one tuple and the other node takes 95 cycles to process one tuple. The result is shown in Figure 24. The adaptive buffering algorithm picks a different queue depth for each node based on the observed latency, while a fixed queue depth is suboptimal for one or both nodes.

In conclusion, the bandwidth-delay product law underestimates the number of buffers needed to achieve peak performance as it ignores additional compute delays that arise during query processing. The adaptive buffering algorithm has comparable or better performance than the optimally configured fixed buffering algorithm. A unique strength of the adaptive strategy is that it can adjust the queue depth to compensate for skew during data processing.

7.2 Evaluation with TPC-H Data

We now turn to the TPC-H data warehousing benchmark to evaluate query response time when using the MESQ/SR algorithm in comparison to MPI. We use the same configuration settings as in Section 7.1.4. We distribute each tuple of every table in TPC-H to a random node in the cluster,

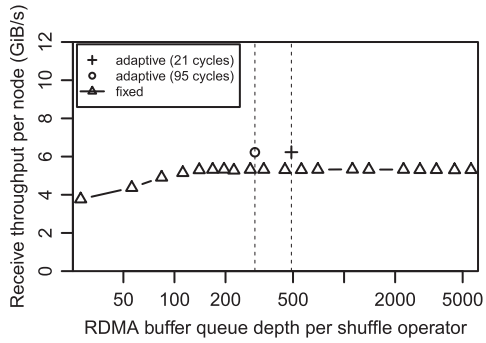
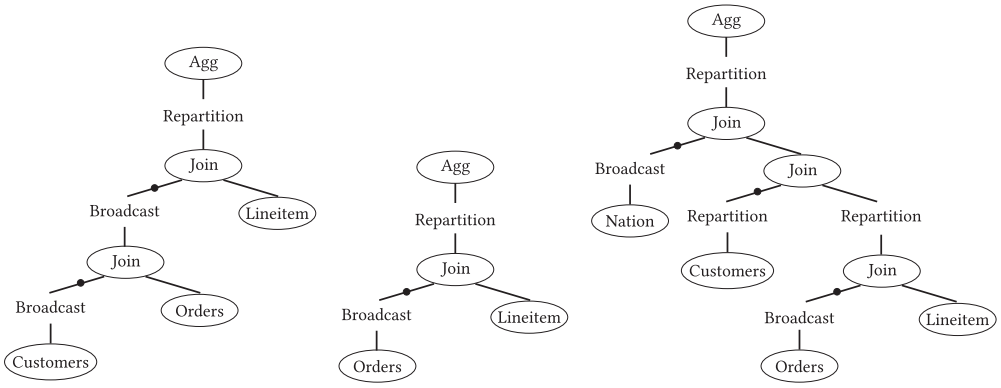


Fig. 24. Performance of the fixed and adaptive RDMA buffer queue management algorithms when plan fragments have different processing latencies. In this experiment one plan fragment has 21 cycles of processing latency while the other one has 95 cycles of processing latency. The adaptive buffering algorithm can adjust the queue depth to compensate for processing skew.



(a) The query plan for TPC-H Q3 (b) The query plan for TPC-H Q4 (c) The query plan for TPC-H Q10

Fig. 25. The query plans for TPC-H query Q3, Q4, and Q10.

except for the Nation and Region tables that we replicate to all nodes (as they contain only 25 and 5 tuples, respectively). This data distribution mimics the experimental setup used in prior work [16, 45, 50]. We use the modulo function in repartitioning data during query evaluation. We pre-project all unused columns as a column-store database would. We choose TPC-H queries Q3, Q4, and Q10 for the evaluation due to their data access locality [7]. The join operations in the queries are evaluated with the non-partitioned join algorithm [52].

We use the query optimizer of a commercial database system to obtain the parallel execution plan. The query plans for the three queries are in Figure 25. For each query, we profile the output and input cardinality of each operator to make sure that the plan that shuffles less data is chosen. Notice that this optimization undermines the performance improvement for our algorithm. The less data shuffled, the less performance improvement from fast shuffling algorithms. Figure 25(c) shows the execution plan for TPC-H Q10. The plan is optimized to reduce the amount of data shuffled in the network. For example, in the join of the Orders and Lineitem table, when the scale factor is 1, there are around 1.5 million tuples joined after pre-filtering for the Lineitem table, and about 60,000 tuples joined after pre-filtering for the Orders table. The tuple size of the

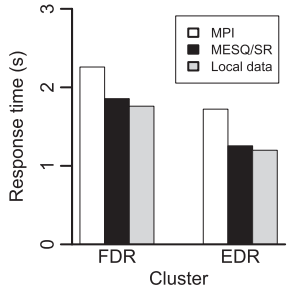


Fig. 26. Effect of EDR network on response time of TPC-H query 4, with eight nodes and scale factor 400.

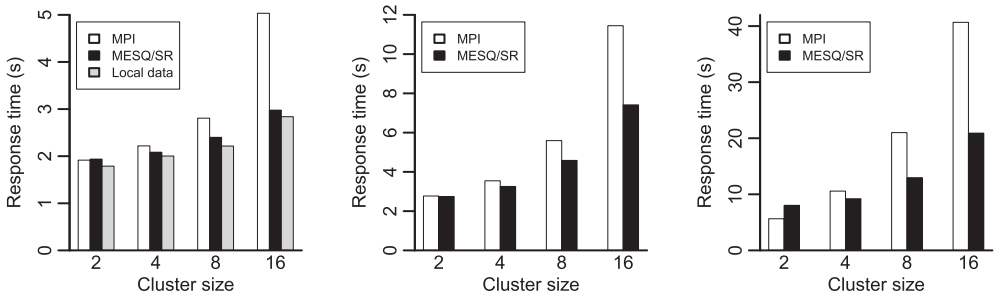
Orders after attribute filtering is 8 bytes, with the tuple size of the Lineitem to be 16 bytes after filtering. Hence we choose to broadcast Orders in the join execution so as to reduce the total network traffic. Similarly, we choose the query plan for Q3 (Figure 25(a)) and Q4 (Figure 25(b)) to minimize the amount of data shuffled.

7.2.1 Response time with Faster Network. We first investigate how query response time changes as one upgrades from the slower 56Gbps FDR InfiniBand to the faster 100Gbps EDR InfiniBand. In this experiment the same TPC-H database with scale factor 400 is distributed across the memory of eight nodes in both the FDR and EDR clusters.

Figure 26 shows the response time from TPC-H Q4. The "local data" bar shows the query response time if all data were stored locally and there were no data shuffling, i.e., all input tables are already co-partitioned. (Note that the local processing time is faster for the EDR cluster as the nodes have faster CPUs and faster memory.) We observe that the MESQ/SR algorithm outperforms MPI in both clusters by the same margin. The performance advantage of MESQ/SR can be traced back to the nearly 2× higher eight-node broadcast throughput of MESQ/SR over MPI in Figure 19(b) and (d). Second, we observe that the MESQ/SR algorithm has similar performance as the "local data" plan that does not shuffle any data. This indicates that the MESQ/SR can successfully overlap communication and computation, unlike MPI. More importantly, as the hardware is upgraded, the performance improvement of MESQ/SR is keeping pace with the improvement in local processing (about 50% for both from FDR to EDR), while MPI is lagging (about 30% gain from FDR to EDR).

7.2.2 Query Response Time When Scaling. We now investigate how query response time changes as the TPC-H database grows in proportion to the cluster size. We generated TPC-H databases with scale factors 200, 400, 800, and 1,600 and loaded them to 2, 4, 8, and 16 nodes, respectively, of the EDR cluster. We evaluate with TPC-H Q3, Q4, and Q10. While Q4 only joins two tables, Q3 and Q10 join three and four tables on different attributes. This makes co-partitioning without replication impossible; thus, we omit the "local data" experiment for Q3 and Q10.

The response time of TPC-H Q4, Q3, and Q10 is shown in Figure 27(a), (b), and (c). The "local data" bar in Figure 27(a), again, shows the performance of the query plan if all data were stored locally, i.e., the data were co-partitioned. (Note that the optimal scale-out line is increasing due to the broadcast communication pattern: As the cluster size increases, the database grows, hence every node receives proportionally more data.) We observe that the MESQ/SR algorithm scales better than MPI. For both Q3 and Q4, although both algorithms perform similarly with 2 nodes, MESQ/SR is nearly 70% faster for Q4 and 55% faster for Q3 than MPI for 16 nodes. For Q10, MESQ/SR is nearly 2× faster than MPI for 16 nodes.



(a) TPC-H query 4. (b) TPC-H query 3. (c) TPC-H query 10.

Fig. 27. TPC-H query response time when scaling the cluster size from 2, 4, 8, to 16. There is 100GiB of data per node in all experiment runs.

7.3 Discussion of Experimental Results and Insights on the Design of Database Systems

This section discusses the experimental results and offers insights on how the advent of RDMA-capable networks can impact the design of future database systems, with a particular emphasis on open research opportunities.

7.3.1 Synchronous Communication with Send/Receive Is Efficient for Tightly Coupled Processes. One-sided primitives such as RDMA Read and RDMA Write are appealing, because they allow one node to stay completely passive during communication. One-sided primitives are very effective when one can accumulate data in a large RDMA-registered buffer and then transmit the entire buffer in a single RDMA Read or RDMA Write operation. Therefore, one-sided primitives are a good fit for loosely coupled systems whose components can operate with a high degree of independence.

However, data shuffling during query processing is a tightly coupled process. In query processing individual query fragments form a pipeline of data producers and data consumers. Query fragments that are downstream cannot start if earlier fragments have not produced data. Delays and errors need to propagate to earlier stages of the pipeline to prevent overflows. In addition, the amount of data that needs to be transmitted is significant and no RDMA-registered buffer would be large enough to hold all the data, so buffers must be reused. Therefore, data shuffling needs periodic synchronization. Producers need to signal consumers to process buffers after a data transmission completes. Consumers in turn signal producers to reuse empty buffers. A database system that uses one-sided primitives would need to build a synchronization mechanism from scratch. Using a synchronous communication primitive (Send/Receive) naturally requires query fragments to coordinate on every message. For this reason, synchronous communication outperforms one-sided algorithms both at the operator level (compare the RDMA Send/Receive algorithms with the one-sided RDMA Read and RDMA Write algorithms in Figure 19) and also at the mechanism level (see the evaluation of the `Poll Completion Queue` notification mechanism in Figures 15–17).

7.3.2 Selective Use of Network Capabilities for Each Query. RDMA allows a database system to drop redundant features of the networking protocol during query processing and select different networking capabilities based on the workload. Existing network protocols, such as TCP/IP, do not permit this lower level of integration between software and hardware. One such example is the tradeoff between scalability and message ordering. The MESQ/SR algorithm uses the Unreliable Datagram transport service, which has better scalability, because it uses a fixed number of Queue Pairs regardless of the cluster size (see Section 7.1.5). Although the Unreliable Datagram transport

cannot guarantee in-order delivery, this is not a problem for query plans that are not order preserving and can process messages in any order. If ordering is desired for some data transmissions, then the database system can selectively switch to the more expensive SEMQ/SR algorithm that uses the Reliable Connection transport that guarantees message ordering, at the cost of using at least as many Queue Pairs as the number of nodes in the cluster. Finally, the choice of the implementation needs to consider the capabilities of the hardware and the current workload. For example, as shown in Section 7.1.4, the newer EDR network supports more active Queue Pairs without performance degradation than the FDR network, hence algorithms that use the Reliable Connection transport will scale better in an EDR network than in an FDR network.

7.3.3 Event-based Notification Mechanisms for Concurrent Query Processing. There are two ways an application can detect completed requests in the Completion Queue. One is polling, where the function returns immediately regardless of whether there are any entries in the Completion Queue. The other is an event-based mechanism where the function blocks and returns only when there are entries to process in the Completion Queue. Both methods have their pros and cons.

Busy polling has low latency as it returns immediately and will not lead to a context switch when there are no entries in the Completion Queue. This is especially important for the "SQ" algorithms, as the message size is only 4KB, which requires more RDMA messages and more context switching if one uses event-based polling. However, in busy polling, the thread holds the CPU that prevents sharing CPU cores among communication threads. In addition, polling very soon after posting a request will most likely find zero entries in the Completion Queue and waste CPU cycles.

Event-based polling enables sharing of CPU cores among threads as a thread will block and be switched out when there are no entries in the Completion Queue. However, event-based polling introduces higher latency in communication. The performance of modern networks has pushed this latency to the microsecond range, which turns out to be a latency region that is difficult to absorb in the application and the OS. This latency has been referred to as the "killer microsecond" in prior work for this reason [4].

7.3.4 Weaknesses of MPI. MPI has a number of weaknesses when used for data shuffling. First, MPI adopts a process-centric addressing and parallelization model, where a process (rank) is used as a destination address for communication. This is not a good fit for database systems that make heavy use of multi-threaded parallelism and share data structures across threads. We have experimentally observed that the performance of MPI is poor with multiple threads, while the MPI performance is very close to the line rate with multiple processes. Second, communication in MPI requires the creation of a fixed process group before any data transfer takes place. However, the communication pattern in data-intensive applications is data dependent and cannot be known in advance. Making matters worse, many MPI operations within a process group are collective and impose barrier-like syncrhonization between processes, which is a poor fit for ad-hoc query processing. Third, MPI does not allow applications to prioritize among different data transfers, which is necessary to meet service level objectives (SLOs) but also for quality of service (QoS) guarantees. For example, an MPI implementation could choose to schedule a short credit update message that unblocks the sender (see Section 5.2) after an unrelated large data transfer between the same nodes. More broadly, MPI does not have a fault tolerance mechanism, in part due to inherent technical challenges in providing fault tolerance to general applications. Without a clearly defined failure model, database systems cannot gracefully react to transmission failures.

7.3.5 Towards Semantically Richer Communication Interfaces for Data-centric Applications. Another challenge is that the one-sided RDMA Verbs interface exposes a limited programming surface that only consists of read, write, and single-word atomic operations to remote memory. Routine

database operations, such as atomically appending to a buffer, require multiple round-trips to complete. Furthermore, the conflict window for such operations is at least one network round-trip, which practically limits the utility of one-sided RDMA to uncontended data accesses. Database systems would benefit from conveying more complex operations, such as append, and issue them in a single message. The remote NIC would then evaluate some simple processing logic locally and return the answer in a single round-trip. This would reduce the number of transmitted messages and the latency by one order of magnitude. In addition, the conflict window would be shortened substantially, because memory operations are initiated locally, which would make distributed algorithms scale better under contention. DPI is a proposed interface description for modern networks that is tailored for data processing and aims to overcome some of these difficulties [2].

8 RELATED WORK

High-performance networks. Foong et al. [13] show that about 1GHz in CPU performance is necessary for every 1-Gb/s network throughput. In addition to being CPU intensive, Frey et al. [15, 16] show that TCP introduces traffic on the memory bus because of data copying. The zero-copy feature of RDMA bypasses that CPU overhead. Frey et al. [14] show that communication can benefit from RDMA only when buffers are large and are reused.

RDMA is extensively studied in supercomputing. MVAPICH uses Send/Receive and RDMA Write to transmit data, while MPICH2 uses RDMA Write to transmit small messages and RDMA Read to transmit large messages [30, 32]. Liu et al. [31] have studied how to efficiently implement broadcast in MPI. MacArthur and Russell [35] compare the performance of different RDMA verbs. Koop et al. [25] reduced the memory consumption in MPI by lowering the number of sent WQEs in RDMA connections and coalescing messages.

RDMA is also studied for key-value stores and "big data" processing. Mitchell et al. [37] use Send/Receive and Read to implement puts and gets, respectively, in key-value stores. Kalia et al. [21] use unreliable RDMA Write for client requests and use Send/Receive in unreliable datagram transport for server responses in their RDMA-aware key-value store. Lu et al. [20, 34] improve the performance of Hadoop and HBase by using RDMA instead of TCP/IP for communication. Dragojević et al. [12] designed FaRM, a computing platform that uses RDMA Read for data accesses and RDMA Write for messages. Wu et al. [58] have extended FaRM with graph processing capabilities. Gu et al. [19] design a remote memory paging system with RDMA that can run applications that do not fit in local memory without modification. Cai et al. [8] design GAM, a new distributed in-memory platform that provides cache coherence with RDMA.

New storage systems have been designed to leverage the direct memory access capability of RDMA. Trivedi et al. [54] build a DRAM-based data store with RDMA. Dinh et al. [11] implement a distributed data storage system, UStore, which leverages RDMA in communication. Tsai and Zhang [55] implement LITE, an RDMA communication kernel, and build a distributed storage system over LITE. Shan et al. [49] use RDMA to build a shared memory system for non-volatile memory.

RDMA has also been used to accelerate transaction processing. Chen et al. [9] and Wei et al. [57] implement a distributed in-memory transaction processing system that uses RDMA one-sided and atomic primitives along with hardware transactional memory. Yoon et al. [59] use RDMA to design a distributed lock manager for distributed transaction processing. Wang et al. [56] use RDMA to accelerate log shipping and replay in database systems to accelerate transaction processing. Zamanian et al. [60] use RDMA to accelerate concurrency control in distributed transaction processing.

New algorithms for fast networks. Li et al. [27] have studied the data shuffling problem in NUMA systems and proposed careful scheduling of the communication. Liu et al. [28] reduce the

network cost of aggregation with data distribution-aware aggregation scheduling. Polychroniou et al. [43] propose a new join algorithm, "track join," which tracks the distribution of data in a relation on a tuple-by-tuple basis and uses the data distribution to reduce the communication workload. Rödiger et al. [46] also take data distribution properties into account and propose an integer linear optimization program to find optimal communication schedules in their "neo-join" algorithm. Although both works are motivated by the high bandwidth of high-end networks, these algorithmic optimizations are orthogonal to the network-level optimizations of this article.

Prior work has also considered how RDMA interacts with database operations. Frey et al. [14, 16] design a new join algorithm, the "cyclo-join" algorithm, which uses RDMA to transfer data. Their results show that with a proper design, the memory bandwidth rather than the network becomes the bottleneck during a join. Tinnefeld et al. [53] study join operations over RAMCloud, which is a DRAM-based storage system connected via RDMA-enabled network adapters. They compare Grace join, distributed Block Nested Loop join and cyclo-join. They also consider three node allocation algorithms and three data distribution strategies. Muhleisen et al. [39] study the performance of a database system when using memory in remote nodes using RDMA. Barthels et al. [6] show how to scale the radix join algorithm to rack-scale computers using RDMA to transmit data during the partitioning phase of the join. Rödiger et al. [45] propose hybrid parallelism that distinguishes local and distributed parallelism and design a push-based multiplexer to shuffle data; all threads share the same multiplexer. In contrast, in this article we design pull-based endpoints and systematically explore the impact of transport-level design decisions as well as how to assign endpoints to threads. Rödiger et al. [44] also propose the "flow-join" algorithm that uses RDMA to ameliorate skew during the join. Li et al. [26] use RDMA to directly access the buffer pool of other nodes in Microsoft SQL Server. Barthels et al. [5] use MPI to explore the performance of distributed join algorithms in HPC systems with thousands of CPU cores. Loesing et al. [33] design Tell, a shared-data database system that runs over RAMCloud and uses RDMA for communication. Salama et al. [47] propose a prototype database system, I-Store, that uses RDMA for distributed query execution.

This manuscript extends previous work in RDMA-aware data shuffling [29] in two ways. The first contribution is a new endpoint implementation algorithm based on the RDMA Write primitive. Building a RECEIVE endpoint using RDMA Write is challenging due to the one-sided nature of the primitive, which means that the receiver is not notified of the arrival of new data for processing. We propose three different techniques to overcome this challenge in Section 5.5 and evaluate their performance in Section 7.1.2. The second contribution is an adaptive RDMA buffer management algorithm that adapts the buffer depth of the shuffling operation based on the needs of the query processing pipeline. Section 6 describes the adaptive RDMA buffer management algorithm and Section 7.1.8 evaluates its performance versus static approaches that use a fixed number of buffers.

9 CONCLUSIONS AND FUTURE WORK

This article studies the challenges and opportunities that arise when using RDMA to shuffle data among query fragments during query execution in parallel database systems. We propose eight algorithms that utilize both reliable and unreliable transport services as well as one-sided and two-sided RDMA transport functions. We find that the MESQ/SR algorithm that uses the Send/Receive message-passing abstraction over an unreliable transport layer exhibits robust performance across all configurations, despite the overheads of coordination, flow control, and error handling in software. Experiments with TPC-H queries show that the MESQ/SR algorithm completely overlaps computation and communication; this improves query response time by up to 2× over the MVAPICH RDMA-capable MPI implementation. We also propose an adaptive buffer management

that dynamically controls the number of RDMA buffers that are used for data transmission according to the data processing latency of the receiving query plan fragment. Experiments show that the adaptive buffering algorithm has comparable or better performance than the fixed buffer algorithm for queries with variable processing latencies.

There are two promising avenues for future work. First, the performance of the same algorithms in RoCE and iWARP networks is an open question. Second, one can specialize the MESQ/SR algorithm to use the native InfiniBand multicast primitive for broadcasting data. We hypothesize that the impact of using native InfiniBand multicast will be to reduce the CPU cost of broadcasting data, as MESQ/SR already transmits data at line rate. Looking further ahead, achieving high end-to-end analytical performance requires the careful interplay of many different algorithms. Faster data transfers naturally expose bottlenecks in other components of the analytical execution pipeline. Amdahl's law suggests that further performance gains will come from directly integrating RDMA capabilities within individual algorithms and from holistically rethinking query processing for RDMA-capable networks.

ACKNOWLEDGMENTS

The evaluation was conducted in part at the Ohio Supercomputer Center [41].

REFERENCES

[1] Accelio. 2019. Accelio. Retrieved from http://www.accelio.org/.

[2] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: The data processing interface for modern networks. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*.

[3] InfiniBand Trade Association. 2015. InfiniBand Architecture Specification Volume 1. retrieved rom https://cw.infinibandta.org/document/dl/7859.

[4] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (Mar. 2017), 48–54. https://doi.org/10.1145/3015146

[5] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed join algorithms on thousands of cores. *Proc. VLDB* 10, 5 (2017), 517–528.

[6] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data (SIGMOD'15)*. ACM, 1463–1475. https://doi.org/10.1145/2723372.2750547

[7] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden messages and lessons learned from an influential benchmark. In *Proceedings of the 5th TPC Technology Conference on Performance Characterization and Benchmarking (TPCTC'13)*. 61–76. https://doi.org/10.1007/978-3-319-04936-6_5

[8] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proc. VLDB* 11, 11 (2018), 1604–1617.

[9] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, New York, NY, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349

[10] Intel Corporation. 2016. rsocket. Retrieved from https://github.com/linux-rdma/rdma-core/blob/master/librdmacm/rsocket.h.

[11] Anh Dinh, Ji Wang, Sheng Wang, Gang Chen, Wei-Ngan Chin, Qian Lin, Beng Chin Ooi, Pingcheng Ruan, Kian-Lee Tan, Zhongle Xie, Hao Zhang, and Meihui Zhang. 2017. UStore: A Distributed storage with rich semantics. *CoRR* abs/1702.02799 (2017). arxiv:1702.02799.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. 401–414.

[13] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. 2003. TCP Performance re-visited. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03)*, pp. 70–79. 10.

[14] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the hidden cost of RDMA. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'09)*. 553–560. https://doi.org/10.1109/ICDCS.2009.32

[15] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. 2009. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN'09)*. ACM, New York, 27–33. https://doi.org/10.1145/1565694.1565701

[16] Philip Werner Frey, Romulo Goncalves, Martin L. Kersten, and Jens Teubner. 2010. A spinning join that does not get dizzy. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'10)*. 283–292. https://doi.org/10.1109/ICDCS.2010.23

[17] Johann George. 2012. qperf. Retrieved from https://www.openfabrics.org/downloads/qperf/.

[18] G. Graefe. 1994. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with infiniswap. In *Proceedings of the14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 649–667.

[20] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC'12)*. Article 35, 35 pages.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM'14)*, pp. 295–306. https://doi.org/10.1145/2619239.2626299

[22] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 437–450.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 185–201.

[24] Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. 2016. To copy or not to copy: Making in-memory databases fast on modern NICs. In *Proceedings of Data Management on New Hardware: The 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures (ADMS'16) and Proceedings of the 4th International Workshop on In-Memory Data Management and Analytics (IMDM'16)*. 79–94. https://doi.org/10.1007/978-3-319-56111-0_5

[25] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. 2007. Reducing connection memory requirements of MPI for InfiniBand clusters: A message coalescing approach. In *Proceedings of the IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid'07)*. 495–504. https://doi.org/10.1109/CCGRID.2007.92

[26] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating relational databases by leveraging remote memory and RDMA. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data (SIGMOD'16)*. ACM, 355–370. https://doi.org/10.1145/2882903.2882949

[27] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: The case of data shuffling. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*.

[28] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. 2018. Chasing similarity: Distribution-aware aggregation scheduling. *Proc. VLDB Endow.* 12, 3 (Nov. 2018), 292–306. https://doi.org/10.14778/3291264.3291273

[29] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and evaluation of an RDMA-aware data shuffling operator for parallel database systems. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. 48–63. https://doi.org/10.1145/3064176.3064202

[30] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K. Panda, David Ashton, Darius Buntinas, William Gropp, and Brian R. Toonen. 2004. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. https://doi.org/10.1109/IPDPS.2004.1302922

[31] Jiuxing Liu, Amith R. Mamidala, and Dhabaleswar K. Panda. 2004. Fast and scalable MPI-level broadcast using InfiniBand hardware multicast support. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'04)*. https://doi.org/10.1109/IPDPS.2004.1302912

[32] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. 2004. High performance RDMA-based MPI implementation over InfiniBand. *Int. J. Parallel Program.* 32, 3 (Jun. 2004), 167–198. https://doi.org/10.1023/B:IJPP.0000029272.69895.c1

[33] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 663–676. https://doi.org/10.1145/2723372.2751519

[34] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. 2013. High-performance design of Hadoop RPC with RDMA over infiniband. In *Proceedings of the International Conference on Parallel Processing (ICPP'13)*. 641–650. https://doi.org/10.1109/ICPP.2013.78

[35] Patrick MacArthur and Robert D. Russell. 2012. A performance study to guide RDMA programming decisions. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications and the IEEE International Conference on Embedded Software and Systems (HPCC-ICESS'12)*, pp. 778–785. https://doi.org/10.1109/HPCC.2012.110

[36] Mellanox. 2015. RDMA Aware Networks Programming User Manual. Retrieved from http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[37] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114.

[38] MPI. 2018. MPI Forum. Retrieved from http://www.mpi-forum.org/.

[39] Hannes Mühleisen, Romulo Gonçalves, and Martin Kersten. 2013. Peak performance: Remote memory revisited. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN'13)*. ACM, New York, Article 9, 7 pages. https://doi.org/10.1145/2485278.2485287

[40] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (Jun. 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[41] Ohio Supercomputer Center. 2015. Ruby Supercomputer. Retrieved from http://osc.edu/ark:/19495/hpc93fc8.

[42] Larry L. Peterson and Bruce S. Davie. 2007. *Computer Networks: A Systems Approach* (4th ed.).

[43] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. 2014. Track join: Distributed joins with minimal network traffic. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data (SIGMOD'14, 1483–1494)*. 12. https://doi.org/10.1145/2588555.2610521

[44] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *Proceedings of the 32nd IEEE International Conference on Data Engineering, (ICDE'16)*. 1194–1205. https://doi.org/10.1109/ICDE.2016.7498324

[45] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed query processing over high-speed networks. *Proc. VLDB* 9, 4 (2015), 228–239. https://doi.org/10.14778/2856318.2856319

[46] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2014. Locality-sensitive operators for parallel main-memory database clusters. In *Proceedings of the (ICDE'14)*. 592–603. https://doi.org/10.1109/ICDE.2014.6816684

[47] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. 2017. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.* 40, 1 (2017), 27–37.

[48] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data (SIGMOD'79)*. 23–34. https://doi.org/10.1145/582095.582099

[49] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. 323–337. https://doi.org/10.1145/3127479.3128610

[50] Ambuj Shatdal and Jeffrey F. Naughton. 1995. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*. ACM, New York, NY, 104–114. https://doi.org/10.1145/223784.223801

[51] Feilong Liu Spyros Blanas. 2018. Pythia. Retrieved from https://code.osu.edu/pythia/core.

[52] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*. IEEE Computer Society, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[53] Christian Tinnefeld, Donald Kossmann, Joos-Hendrik Böse, and Hasso Plattner. 2014. Parallel join executions in RAMCloud. In *Proceedings of the 30th International Conference on Data Engineering Workshops*. pp. 182–190. https://doi.org/10.1109/ICDEW.2014.6818325

[54] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R. Gross. 2015. RStore: A Direct-access DRAM-based data store. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS'15)*. 674–685. https://doi.org/10.1109/ICDCS.2015.74

[55] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 306–324. https://doi.org/10.1145/3132747.3132762

[56] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query fresh: Log shipping on steroids. *Proc. VLDB* 11, 4 (2017), 406–419.

[57] 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'15)*. ACM, New York, NY, 87–104. https://doi.org/10.1145/2815400.2815419

[58] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling graph computation to the trillions. In *Proceedings of the International System-on-Chip Conference (SoCC'15)*. 408–421. https://doi.org/10.1145/2806777.2806849

[59] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed lock management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data Conference (SIGMOD'18)*. 1571–1586. https://doi.org/10.1145/3183713.3196890

[60] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2017. The end of a myth: Distributed transaction can scale. *Proc. VLDB* 10, 6 (2017), 685–696.

Received August 2018; revised June 2019; accepted August 2019