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

Feilong Liu Lingyan Yin Spyros Blanas
The Ohio State University
{liu.3222, yin.387, blanas.2}@osu.edu

Abstract

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 paper 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 six designs that capture salient trade-offs in this design space. 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×.

1. Introduction

Fast networking is no longer exclusive to high-end supercomputers. Database servers today ship with 10Gbps Ethernet and are commonly upgraded to 56Gbps FDR InfiniBand, while 100Gbps EDR 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.

Parallel database systems can use either 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. One-sided communication primitives (such as RDMA Read) have thus generated substantial research excitement. At the algorithmic level, recent work has proposed join algorithms that use RDMA [3, 11, 12, 36]. At the systems level, recent work has redesigned the database kernel for fast networks for analytical [37], transactional [6, 43] and hybrid workloads [21].

Database systems architects, however, prefer unobtrusive implementations that neither modify the database kernel, nor delegate control of memory and communication management to libraries such as MPI [25], Accelio [1], or `rsocket` [39]. This paper 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.

The paper first introduces the communication endpoint abstraction to decouple the mechanics of data transmission from the data shuffling operator. Different endpoints can transmit data either through the RDMA Send/Receive message-passing abstraction or through the RDMA Read shared-memory abstraction. This abstraction is compatible with the reliable RDMA transport service that offloads communication management to hardware and guarantees message delivery, as well as unreliable communication that requires error handling and flow control in software. We assume a network that is lossless under congestion but may deliver packets out of order, such as InfiniBand. The enabling insight is that database systems can uniquely benefit from the Unreliable Datagram transport service because relational algebra operators are set-based. Hence, it often suf-

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064202

fices to count the number of messages that were transmitted without storing them in a re-order buffer.

We contribute six designs of the data shuffling operator that represent different trade-offs 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 implementors to use the proposed techniques without radically redesigning their existing analytical processing engines. We have open-sourced our prototype implementation for further scrutiny and research by the community.

Our experimental evaluation compares the performance of the six designs for cluster sizes up to 16 nodes. We evaluate our algorithms on 56Gbps FDR InfiniBand and the newer 100Gbps EDR InfiniBand. We find that using the RDMA Send/Receive message-passing abstraction over an unreliable transport layer achieves robust performance across all configurations, despite the overheads of coordination, flow control and error handling in software. Overall, the choice of the shuffling algorithm affects throughput by as much as $5\times$. The data shuffling operator that is tailored to database processing outperforms MVAPICH [25], an RDMA-capable MPI implementation, by as much as $2\times$ for TPC-H queries. To the best of our knowledge, this is the first paper that demonstrates that architectural decisions about the transport layer (as exposed via RDMA) can significantly impact the analytical performance of a parallel database system.

2. Background

In this section, we first introduce the data shuffling in parallel database systems and then we give an overview of the Remote Directory Memory Access (RDMA) capability of modern networks.

2.1 Data shuffling in parallel database systems

Database systems internally convert a SQL statement into an executable query plan. The query plan is a tree of operators that synthesizes individual algorithms, such as sort, join, etc. to produce the correct answer. The pull-based execution model [13] is a widely used abstraction to execute query plans in database systems. In this model, each algorithm is represented as an operator that implements a NEXT function which will return ("pull") data to the parent node. Each operator is vectorized and returns a batch of tuples in the NEXT function call [5, 20]. Figure 1 shows an example of a pipeline with parallel, vectorized pull-based operators.

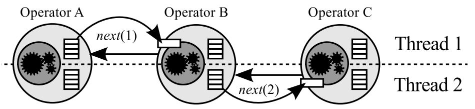


Figure 1. Example of a parallel, vectorized pull-based pipeline. Every worker thread passes its ID to the *next()* call to access thread-specific operator state and output.

In a parallel database system, the query plan is divided into query fragments which are replicated across the cluster. Different query fragments are connected with shuffling operators (also known as exchange operators in the Gamma system [7]). The shuffling operator is the only operator that will transmit and receive data over the network. Slow networks can be a bottleneck for parallel database systems [33] and data shuffling has been shown to be a significant contributor to the end-to-end query response time [2, 3, 37].

2.2 RDMA overview

Remote Direct Memory Access (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 interface *(ibv_*)* throughout the paper. The IB verbs interface is available either natively or through emulation for InfiniBand, RoCE, and iWARP.

2.2.1 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 can safely reuse the memory that each event points to.

The RDMA Read request is a one-sided communication primitive that allows the data sender to remain completely passive. The receiver will read the data from the remote 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 network adapter asynchronously performs the remote read, populates the local buffer and posts a completion event to the Completion Queue.

RDMA Send and Receive are used in two-sided communication. The receiver first posts a 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. The sender follows and posts a 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.

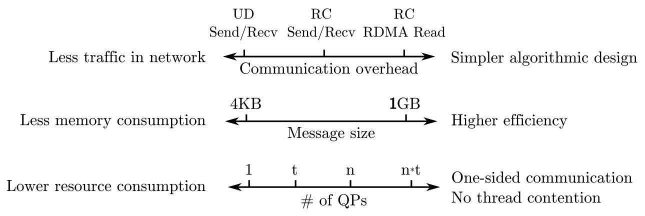


Figure 2. RDMA design space for data shuffling algorithms.

2.2.2 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 Send, Receive and Read transport functions. The maximum message size in Reliable Connection transport is hardware-specific and can be as large as 1 GiB. Because Reliable Connection is connection-oriented, each Queue Pair can only communicate with exactly one other Queue Pair. Hence, with the Reliable Connection service point-to-point communication between n nodes requires $\Theta(n^2)$ Queue Pairs.

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

2.2.3 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 process uses RDMA verbs to issue RDMA requests. In particular, *ibv_post_send()* is used for RDMA Read 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. An RDMA Send request specifies the buffer which contains the data to be sent, while RDMA Read and RDMA Receive requests specify the buffer which will be used to store the received data. These buffers cannot be reused after they have been posted. The requests will be then processed by the hardware, which will generate a completion entry in Completion Queue when the operation finishes. The application retrieves completion entries by polling the Completion Queue using *ibv_poll_cq()*. The completion entry informs the application which RDMA request has completed so that the corresponding buffer is reused.

3. Design trade-offs for RDMA data transfer

Different combinations of RDMA transport service types and transport functions pose different challenges in implementing RDMA-aware data shuffling algorithms. A summary of the design space is shown in Figure 2, 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.

1. Number of QPs per node: Choosing the appropriate number of Queue Pairs (QPs) per node is a trade-off between hardware resource consumption and parallelism. As QP data is cached in the Network Interface Card (NIC), the NIC will run out of space if there are too many Queue Pairs per node. Prior work [8] 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 point-to-point communication, at minimum n connections per node are needed. This design assumes that all threads will share one 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 other nodes to avoid contention, $n \times t$ connections per node are needed for point-to-point 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 queue pair per node; the QP will be shared by all CPU cores regardless of the destination. Thread contention can be eliminated by using t connections per node.

2. The message size for RDMA communication. Choosing the message size for RDMA communication requires a delicate balance between memory consumption and communication efficiency. For the Unreliable Datagram transport service, the maximum message size is the MTU (which is 4 KiB in many platforms, including our own). With the Reliable Connection service, the maximum message size can be as high as 1 GiB per the InfiniBand specification. A smaller message size means that applications should post more requests to transmit the same volume of data. Thus, small messages lead to more CPU overhead during communication.

However, large message sizes require the application to pin and register substantially more memory for RDMA communication. In order to overlap communication with computation, $2 \times n$ message buffers are needed to communicate with any node in a cluster with n nodes. If one uses the extreme setting of 1 GiB in a 16-node cluster, at least 32

[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 paper.

GiB memory should be allocated for communication in each node—which may be beyond what a parallel database system can comfortably allocate to a single query fragment.

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. This leads to more traffic in the network but a simpler algorithm, as every packet that is transmitted requires an acknowledgment. The Unreliable Datagram transport service sends no acknowledgement packet, which leads to less traffic in the network. However, the delivery of the message is unordered and unreliable—the application needs to perform error handling and carefully handle state transitions despite packets arriving out of order. These considerations complicate the design of RDMA-aware algorithms.

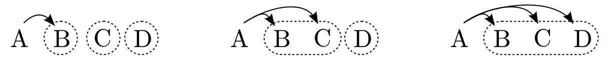
Synchronization and flow control: RDMA Send and RDMA Receive are two-sided verbs. The application is responsible for posting an RDMA Receive request on the receiving side before an RDMA Send request arrives, else the RDMA Send request will be dropped. Coordination is needed to tally the transmitted messages 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 the communication, the sender remains passive while the receiver posts the RDMA Read request. The coordination challenge is to ensure that the passive side (the sender) does not read memory that is currently being modified. Coordination is needed to inform the sender when the buffer space can be safely reclaimed to be overwritten.

RDMA Write is another one-sided verb that shares many technical similarities to RDMA Read, such as that both require a reliable transport and allow for messages up to 1 GiB big. Both RDMA Read and RDMA Write have been used in prior work [8, 28, 43]. This paper uses RDMA Read as it is semantically closer to the pull-based model (see Figure 1) and is thus more intuitive to implement.

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. Section 4.4 describes implementations of the communication endpoint that use different RDMA transport functions and service types. Finally, Section 4.5 enumerates all algorithms and summarizes their properties.



(a) Repartition $\mathbb{G} = \{\{B\},\{C\},\{D\}\}$ (b) Multicast $\mathbb{G} = \{\{B,C\},\{D\}\}$ (c) Broadcast $\mathbb{G} = \{\{B,C,D\}\}$

Figure 3. 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 $\mathbb{G}[0]$.

4.1 Supported data transmission patterns

The communication pattern during relational query processing is dynamic as 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 3 shows the three data communication patterns in a 4-node cluster. When the transmission group $\mathbb{G}$ contains singletons, as in Figure 3(a), node A will repartition the data. Figure 3(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 3(c), node A will broadcast data to the entire cluster.

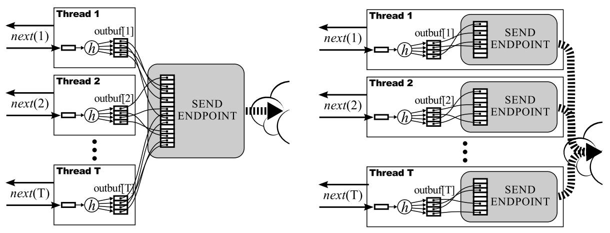
4.2 The communication endpoint abstraction

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 three orders of magnitude greater than TCP/IP [10]. We introduce the *communication endpoint* abstraction in order to hide such transport-level intricacies from the high-level communication logic.

A communication endpoint consists of RDMA-specific resources and data transmission logic. Every endpoint that participates in a query plan is assigned a unique integer for identification. (This identifier is used similarly to a port and address pair in a TCP/IP connection.) 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 endpoint owns and registers the memory for RDMA operations and is responsible for managing this memory.

The SEND endpoint transmits data using the following interface:

- SEND (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 SEND returns. The binary parameter *state* signals if this is the last buffer to be sent (`Depleted`) or more data is available (`MoreData`). SEND does not block.



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

Figure 4. Configurations for the SHUFFLE operator.

- void* *buf* ← GETFREE()
  This function returns an RDMA-registered buffer *buf* that can be used in a subsequent SEND call. GETFREE 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 which sent this buffer; *remote* is the address of this buffer in the remote endpoint. (See Section 4.4.3 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, RELEASE also notifies the remote endpoint *src* that buffer *remote* has been consumed. (See Section 4.4.3 for more details.) RELEASE does not block.

4.3 The SHUFFLE and RECEIVE operators

Our implementation vectorizes each operator by returning a batch of tuples 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 [31, 40].

We now describe the implementation of the SHUFFLE and the RECEIVE operators using the endpoint interface described in Section 4.2. Figure 4 shows two different configurations of the SHUFFLE operator. A single endpoint configuration (SE) is shown in Figure 4(a), where all threads share one 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 4(b), where a SEND endpoint is dedicated to every thread. This avoids inter-thread contention but increases resource consumption by more than an order of magnitude 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.

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 4)
       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             write tuple to curbuf
11             if curbuf is full then
12                 target.SEND(curbuf, 𝔾[dest], MoreData)
13                 outbuf[tid][dest] ← target.GETFREE()
       until state is Depleted;
14     if mode is MultiEndpoint or tid is last thread then
15         target.SEND(curbuf, 𝔾[dest], Depleted)
16     else if mode is SingleEndpoint then
17         target.SEND(curbuf, 𝔾[dest], MoreData)
18     return <Depleted, EmptyBatch>
```

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 (Alg. 1, line 8) and copies the tuple to the output buffer of the transmission group $\mathbb{G}[\text{HASH}(t)]$ (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 another 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 status for its own dedicated endpoint (line 15), whereas in the single-endpoint configuration the last thread

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         copy local into outbuf[tid]
9         target.RELEASE(remote, local, src)
10        if outbuf[tid] is full then
11            return <MoreData, outbuf[tid]>
      until state is Depleted;
12    return <Depleted, outbuf[tid]>
```

sending out data will propagate the end-of-transmission status to the remote endpoint (line 15).

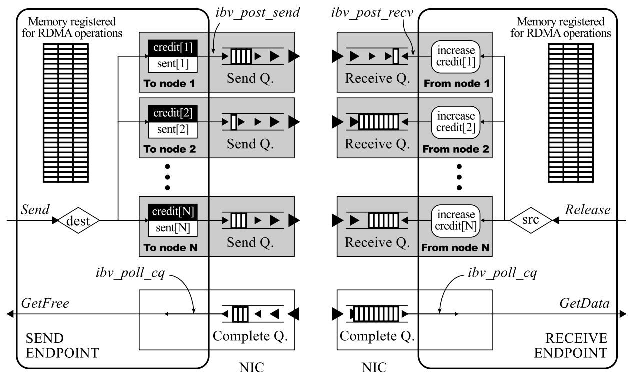
A design choice is whether to copy the tuples into RDMA-registered buffers or directly perform RDMA operations on the input (often referred to as the *zero copy* optimization). Our experiments confirm the findings of Kesavan et al. [18] that zero copy shows little benefit when the record size is small (128 bytes). We thus choose to always copy based on the observation that tuple sizes are typically less than a few hundred bytes for row-oriented disk-based database systems, and as little as 16 bytes in column-oriented main-memory database systems. 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 (Alg. 2, line 7). It then copies the data from the RDMA-registered buffer to the output buffer (line 8) and returns the buffer to the endpoint (line 9). If the output buffer is full, 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.

4.4 Implementing the communication endpoint

We now describe three implementations of the communication endpoint that use different RDMA transport functions and service types, namely the RDMA Send/Receive function with the Reliable Connection service (Section 4.4.1), the RDMA Send/Receive function with the Unreliable Datagram service (Section 4.4.2) and the RDMA Read function with the Reliable Connection service (Section 4.4.3).



(a) Send endpoint (b) Receive endpoint

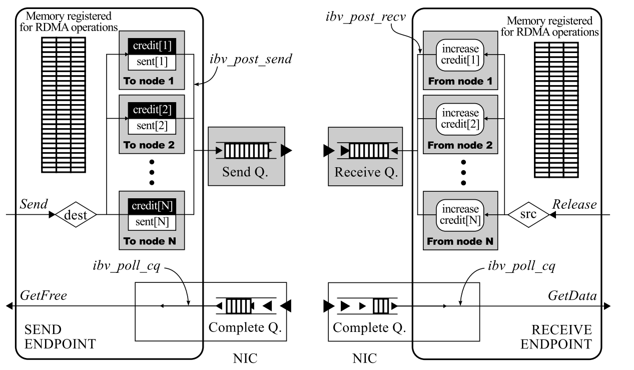
Figure 5. Endpoint implementation for the RDMA Send/Receive transport function and the Reliable Connection service.

4.4.1 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 service. The data delivery guarantee, however, requires that every arriving Send request (that contains the data) can be matched to a posted Receive request (that specifies where the data will be stored). A Send request that cannot be matched to a posted Receive request will be dropped, as the network card cannot deposit the incoming data in RDMA-registered memory. The main technical challenge in implementing a high-performance communication endpoint with the RDMA Send/Receive function and the Reliable Connection service 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. We transmit 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 stateless. The sender, in turn, is obligated to consume all the issued credits and to issue the same number of Send requests (which may involve empty buffers if the transmission is ending) 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 [16]. 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 eventually starving the SEND endpoint for credit. We experimentally study this trade-off in Section 5.1.1.

Figure 5 sketches the endpoint implementation that uses the RDMA Send/Receive transport function for communication. The credit for every connection is stored in the sending



(a) Send endpoint (b) Receive endpoint

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

Queue Pair, which is shown in Figure 5(a), in the variables *sent* and *credit*. Figure 5(b) shows the receiver who will increment the credit after a Receive request has been posted. With the Reliable Connection service, a Queue Pair on the local side can communicate with exactly one Queue Pair on the remote side. Both the SEND and RECEIVE endpoints have as many Queue Pairs as the cluster size to permit sending and receiving messages from any node. We associate all the Queue Pairs with a single Completion Queue to amortize the cost of polling, as we can poll all the Queue Pairs for completions with one invocation of the *ibv_poll_cq* function.

The implementation of the interface in Section 4.2 using RDMA Send/Receive with Reliable Connection is as follows. In the SEND function, the algorithm iterates over every node in the transmission group. If this connection has sufficient credit, it will increment the credit and call *ibv_post_send* to enqueue the *buffer* for transmission, else it will wait for credit. The GETFREE function repeatedly polls using *ibv_poll_cq* until a completion event has been received from each node in the transmission group, and then returns the *buffer* for reuse. On the receiver, the RELEASE function will post a Receive request using *ibv_post_recv*, then increments the *credit* value of the corresponding sender using the RDMA Write transport function. GETDATA polls for completion with *ibv_poll_cq* and returns the *buffer* associated with the completion request.

4.4.2 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 [8].

Figure 6 shows 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 5). We use the same stateless credit mechanism that was introduced in Section 4.4.1 to synchronize the sender and the receiver, with the only distinction being that now all destinations share one Queue Pair.

One challenge with the Unreliable Datagram transport service is that message delivery is unreliable and packets may be lost. Thankfully, this rarely occurs in practice. As Kalia et al. point out, InfiniBand has lossless link-level flow control and packets are never lost due to buffer overflows [15, 17]. Packet loss happens due to bit errors on the wire and hardware failures, which are rare events.

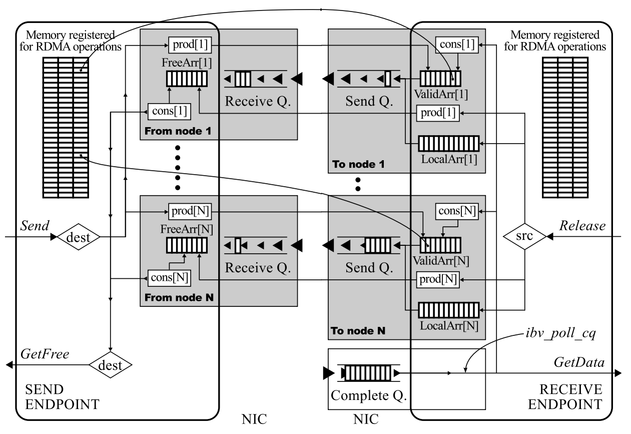
Although lost messages are rare, it is common for the Unreliable Datagram transport service to deliver packets out of order. This problem frequently arises at the end of the data transmission, when the receiving endpoint may see a message tagged as `Depleted` followed by multiple messages tagged with `MoreData`. The data receiver needs to ensure that a transition to the `Depleted` state is not premature.

We handle unreliable data transmission and out-of-order deliver by maintaining an additional counter in the Unreliable Datagram implementation of the endpoint. The data sender records the total number of packets sent to each destination, while the data receiver 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 will compare this number with the number of messages it has already received. If the number does not match, then the receiver has not received all the packets yet, either due to (rare) packet loss or (commonly) because some packets are still in transit. We set a limit on the time the receiver waits for outstanding packets. If the totals still do not match after waiting, we treat this as a network error and restart the query.

4.4.3 RDMA Read with Reliable Connection

We now present an endpoint implementation that uses a one-sided communication primitive, namely RDMA Read. During communication the SEND endpoint remains completely passive, while the RECEIVE endpoint will issue the RDMA Read request to transfer remote data into local memory.

The technical challenge is to identify when a buffer is ready to be consumed by the receiver, and when a buffer is available to be reused by the sender. The RECEIVE endpoint can issue an RDMA Read request only if it knows the address of an RDMA-registered buffer in the SEND endpoint. To facilitate this, the SEND endpoint must share addresses of full data buffers with the RECEIVE endpoint. Conversely, because RDMA Read is one-sided, the SEND endpoint cannot tell when the RDMA Read operation has completed and the buffer is no longer in use. Thus, the RECEIVE endpoint must share addresses of free buffers with the SEND endpoint.



(a) Send endpoint (b) Receive endpoint

Figure 7. Endpoint implementation for the RDMA Read transport function under the Reliable Connection service.

Figure 7 sketches the endpoint implementation that uses the RDMA Read transport function with the Reliable Connection service. The notification mechanism to mark buffers as free or full consists of the circular queues *FreeArr* and *ValidArr*. The circular queues are used as message queues and remote nodes send messages by updating the message queues. The SEND endpoint keeps the *FreeArr* queue while the RECEIVE endpoint keeps the *ValidArr* queue. Entries in both *FreeArr* and *ValidArr* point to RDMA-registered memory in the SEND endpoint. The consume pointer *cons* points to the local queue, while the produce pointer *prod* points to the remote queue. That is, the SEND endpoint consumes free buffers from the local *FreeArr* queue and produces full buffers into the remote *ValidArr* queue, while the RECEIVE endpoint consumes full buffers from the local *ValidArr* queue and produces empty buffers in the remote *FreeArr* queue. The RECEIVE endpoint uses the *LocalArr* stack to retrieve unused RDMA-registered destination buffers for the outgoing RDMA Read requests; this buffer will contain the remote data when the RDMA operation completes.

Algorithm 3 shows more implementation details. The SEND function will signal that the *buffer* can be read by adding it in the *ValidArr* queue using an RDMA Write request to every RECEIVE endpoint in the transmission group. The GETFREE 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. On the receiver, the RELEASE function will signal that the *buffer* can be reused by adding it in the *FreeArr* queue using an RDMA Write request to the originating SEND endpoint. The GETDATA function first issues any pending RDMA Read requests until all requests or available buffers are depleted. It then blocks until at least one completion is received and returns the *buffer* for use.

Algorithm 3: RDMA Read with Reliable Connection

```
function SEND(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* GETFREE( )
6     while true do
7         for $i \in [1,N]$ do
8             while FreeArr[i] is not empty 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 $|\mathbb{G}[gid]|$ 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 $i \in [1,N]$ do
20        while ValidArr[i] is not empty
          and LocalArr[i] is not empty 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>
```

4.5 Putting it all together: Design alternatives for high-performance data shuffling

So far Section 4 has introduced two orthogonal design choices: (a) the number of endpoints per operator, and (b) different endpoint implementations. The endpoint abstraction was introduced in Section 4.3 as a configurable parameter that can balance shared resource contention with excessive resource consumption. The two extremes of this design dimension are a single endpoint design (SE) that assigns one endpoint to all threads, or a multi-endpoint design (ME) that dedicates one endpoint per thread. Section 4.4 has presented three different endpoint implementations that use a single Queue Pair and the message-passing RDMA Send/Receive primitive (SQ/SR), or multiple Queue Pairs with either the message-passing RDMA Send/Receive primitive (MQ/SR) or the shared-memory RDMA Read primitive (MQ/RD). These design dimensions are orthogonal and

RDMA Read	**RDMA Send/Receive**				
One-sided communication, periodic coordination needed to manage buffers	Two-sided communication, continuous coordination needed on every transfer				
Flow control in hardware	Flow control in software	**Open connections (QPs) per node**	**Thread contention**	**Messaging**	**Transport**
MEMQ/RD	MEMQ/SR	Excessive $n \cdot t$	None	Round-trip, up to 1 GiB	Reliable Connection (RC), error control in hardware
SEMQ/RD	SEMQ/SR	Moderate n	Moderate		
Not supported by InfiniBand	MESQ/SR	Moderate t	None	Half-trip, up to 4 KiB	Unreliable Datagram (UD), error control in software
	SESQ/SR	Minimal 1	Excessive		

Table 1. Alternative data shuffling operator designs for a cluster with n nodes and t threads per query fragment.

can be combined to produce six design alternatives for a data shuffling operator for high-performance networks. We refer to each design by concatenating the number of endpoints (SE or ME) with the implementation of the endpoint (SQ/SR, MQ/SR, MQ/RD). Table 1 summarizes the tradeoffs associated with each of the six algorithms in a cluster with n nodes and t threads per query fragment.

5. Experimental evaluation

We have implemented all variants of the data SHUFFLE and RECEIVE operators in Pythia, a prototype open-source in-memory query engine that is written in C++ [34]. 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 8 nodes in our evaluation, unless otherwise specified.

This section evaluates the following questions:

- What is the overhead of flow control when using the two-sided RDMA Send/Receive transport function? (§ 5.1.1)
- How does the message size affect performance with the Reliable Connection transport service? (§ 5.1.2)
- How does the repartition and broadcast throughput scale as the cluster size grows? (§ 5.1.3)
- How does the number of Queue Pairs affect performance? (§ 5.1.4)
- How significant is the setup cost for RDMA? (§ 5.1.5)
- What is the performance with compute-intensive queries? (§ 5.1.6)
- Does a faster network improve query performance? (§ 5.2.1)
- How does query response time scale as the database size grows proportionally to the cluster size? (§ 5.2.2)

5.1 Evaluating receive throughput

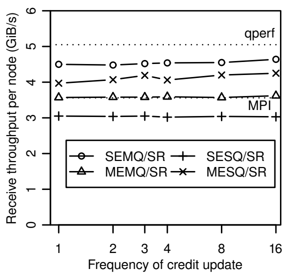
In this section, we use a synthetic workload to study the receive throughput per node with different data shuffling algorithms. We generate a synthetic table R with two long integer attributes $R.a$ and $R.b$ for evaluation. The table contains 1 billion tuples and the size of the table is 16 GiB. $R.a$ is uniformly distributed and randomized. This table is replicated in each node of the cluster.

We evaluate the throughput of the data shuffle operation with a synthetic query. In this query, all the nodes scan the local fragment of table R and repartition R using $R.a$ as the key. The communication pattern corresponds to repartitioning data which is uniformly and randomly distributed. We calculate the total throughput as the reciprocal of the query response time and divide by the total number of nodes in the cluster. This slightly underestimates peak throughput as the shuffling operation may not complete at the same time. To amortize transient fluctuations in network performance, the R table is scanned and transmitted ten times (in a sequence) such that 160GiB per node are transmitted.

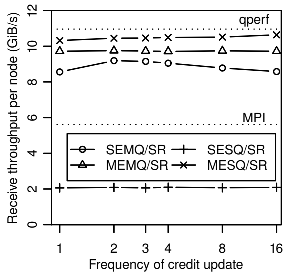
As we are not aware of any open-source RDMA-capable parallel database system, we revert to three performance baselines. One comparison baseline is the MVAPICH2 [25] implementation of the ubiquitous MPI library [29] that uses RDMA for communication. We have implemented an endpoint using MPI. In the repartition 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. Second, we report results from `qperf` [35], a bandwidth benchmarking tool. The sender in `qperf` only 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 and never accesses the transmitted data. Although `qperf` gives some measure of "peak" networking capability for each cluster, these design assumptions preclude any direct comparison. We also compare with TCP/IP-based communication over InfiniBand ("IPoIB"). This reflects the performance from a network upgrade without any changes in software. In the IPoIB algorithm, we use the *send*() function in the sender. In the receiver, we use *select*() to monitor the sockets created for communication, and we call *recv*() to receive data when one socket is ready.

5.1.1 Flow control overhead in RDMA Send/Receive

The two-sided RDMA Send/Receive transport functions require flow control in software, else packets will be dropped.



(a) 56 Gbps FDR InfiniBand



(b) 100 Gbps EDR InfiniBand

Figure 8. Performance of the MQ/SR and SQ/SR algorithms when changing the credit write back frequency.

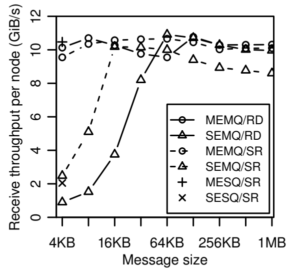
We synchronize the sender and the receiver through a credit protocol that is described in Section 4.4.1. The question is what is the overhead of the credit mechanism and how does the credit write back frequency affect performance. In this experiment, we use 8 nodes and each thread registers 16 RDMA buffers per remote node.

We study the performance of all RDMA Send/Receive algorithms when we change the frequency of the credit update and we measure the throughput at the data receiver. The result is shown in Figure 8. The horizontal axis is the credit write back frequency, which reflects how many RDMA Receive requests the data receiver needs to post before it writes back the credit. The vertical axis is the receive throughput for each algorithm. From the evaluation, we can see that the performance degradation due to the credit mechanism is not very significant. We thus fix the write back frequency to two requests for all RDMA Send/Receive algorithms.

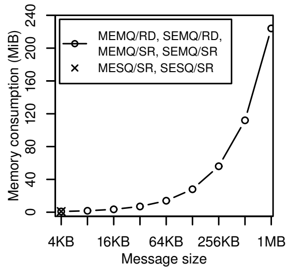
5.1.2 Effect of message size in Reliable Connection

In our shuffling algorithm, tuples are accumulated in an RDMA-registered buffer and are sent out as one RDMA message. While the Unreliable Datagram transport only supports messages that are up to 4 KiB big, the Reliable Connection transport supports messages as big as 1 GiB. One thus needs to tune the message size for the MQ algorithms that use the Reliable Connection transport. In this experiment, we use double buffering, i.e. every thread will register two RDMA buffers for each destination, and we change the message size from 4 KiB to 1 MiB. The experiment runs on eight nodes in the EDR cluster. The result is shown in Figure 9. The horizontal axis is the message size and the vertical axis is the receive throughput for each algorithm. For the SEMQ/RD and SEMQ/SR algorithms, the throughput first increases when increasing the message size, then slightly drops after reaching the peak throughput. For the MEMQ/RD and MEMQ/SR algorithms, the performance stays stable as the message size changes.

Message sizes around 1 MiB are unrealistic configuration choices in practice, however. Figure 9(b) shows the memory registered for RDMA communication (vertical axis) as the



(a) Throughput



(b) Memory consumption

Figure 9. Effect of message size for EDR InfiniBand.

message size changes (horizontal axis) when running on 8 nodes in the EDR cluster. As the message size approaches 1 MiB, the pinned RDMA memory can be 100 MiB or more for a single shuffle operator. If one considers that query plans consist of multiple shuffle operators and parallel database systems may execute dozens of query fragments concurrently, a message size around 1 MiB may translate into a pinned memory footprint of 10 GiB or more. The RDMA Send/Receive algorithm in the Unreliable Datagram protocol has a decisive space advantage here, as it requires under 1 MiB of pinned memory to achieve its peak throughput.

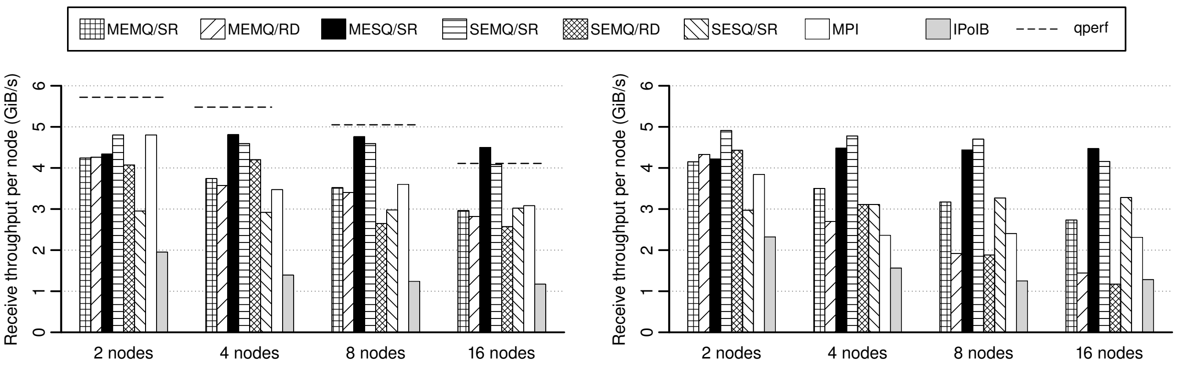
Based on these results, we fix the message size to be 64 KiB for the algorithms that use the Reliable Connection transport and fix double buffering for all algorithms.

5.1.3 Throughput when scaling out

This section studies the performance of the six algorithms when increasing the number of nodes. In this experiment, we run the six algorithms using 2, 4, 8 and 16 nodes. The result is shown in Figure 10. In addition to the six 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.

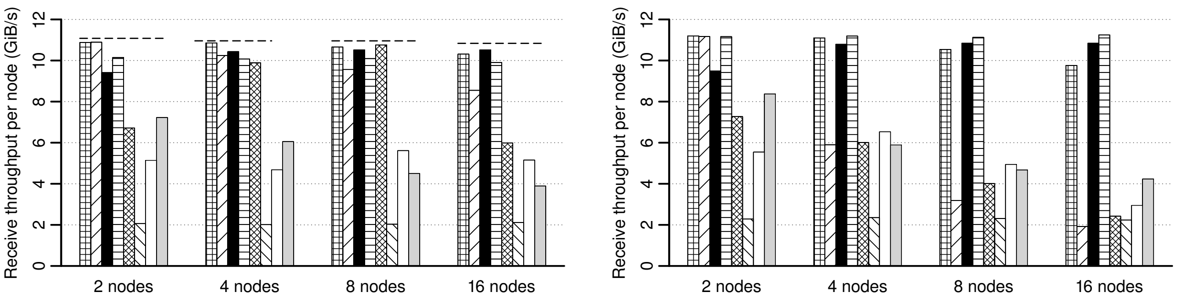
Figures 10(a) and 10(c) show performance for the repartition pattern in the FDR and the EDR cluster respectively. First, our RDMA-aware algorithms outperform the MPI algorithm by as much as 2× (see Figure 10(c), MESQ/SR vs. MPI with 16 nodes in the EDR cluster), and outperform the IPoIB algorithm by as much as 3× (see Figure 10(a) MESQ/SR vs. IPoIB with 8 nodes in the FDR cluster).

Looking at the FDR cluster in Figure 10(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 attributed to the 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 increases proportionally to the cluster size for all the MQ algorithms. For 16 nodes, the MESQ/SR



(a) Repartition throughput with 56Gbps FDR InfiniBand.

(b) Broadcast throughput with 56Gbps FDR InfiniBand.



(c) Repartition throughput with 100Gbps EDR InfiniBand.

(d) Broadcast throughput with 100Gbps EDR InfiniBand.

Figure 10. Throughput when changing the number of nodes in the cluster.

algorithm even outperforms the qperf which runs using Reliable Connection transport.

Looking at the EDR cluster in Figure 10(c), the MESQ/SR algorithm has good performance when scaling out. Unlike the results from the FDR cluster, the performance of the MQ/SR algorithms does not degrade as the cluster grows. This is because the EDR hardware can cache QP data for more point-to-point connections [17].

When profiling the repartition run on 8 nodes of the EDR cluster, we found that the IPoIB algorithm spends about 2/3 of the cycles in the *send* and *recv* functions. The SESQ/SR algorithm is bottlenecked due to contention for the *ibv_post_send* function. On the sender side of the RDMA algorithms, the most CPU-intensive activity is hashing the individual tuples and copying them to RDMA-registered memory. Still, about 30% of the cycles are idle and would be devoted to other activities in a well-designed database system. Barthels et al. [2] reduce this overhead further by using AVX instructions during partitioning. Among the RDMA algorithms, the MEMQ/SR and MESQ/SR algorithms are blocked for credit, while the rest are blocked on the completion of pending RDMA operations. 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.

The results for the broadcast pattern are shown in Figures 10(b) and 10(d). The RDMA-aware algorithms outperform MPI and IPoIB by as much as 3× (for MPI see Figure 10(d), SEMQ/SR vs. MPI with 16 nodes in the EDR cluster; for IPoIB see Figure 10(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 the FDR cluster while the MQ algorithms degrade. 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 a buffer is free and will be reused only when all the nodes finish reading its data. For the RDMA Read algorithms, the time to reuse one buffer depends on when the last node finishes reading the data. It is therefore possible for nodes to starve for free buffers if there is some load imbalance or there is a transient network degradation.

We conclude that the RDMA shuffling algorithms show throughput close to the line bandwidth for FDR and EDR InfiniBand. The MESQ/SR algorithm, in particular, shows good scalability in both. Compared with the MPI and IPoIB alternatives, our RDMA-aware data shuffling algorithms outperform by as much as 3×.

5.1.4 Effect of many Queue Pairs

In this section we show the throughput of the RDMA algorithms with different number of Queue Pairs. Prior work has shown that the number of Queue Pairs can significantly affect performance [16, 17]. In the experiment, the repartition

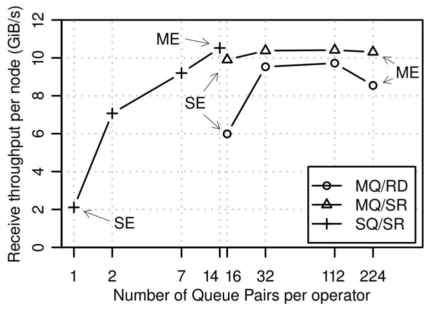


Figure 11. Effect of many Queue Pairs.

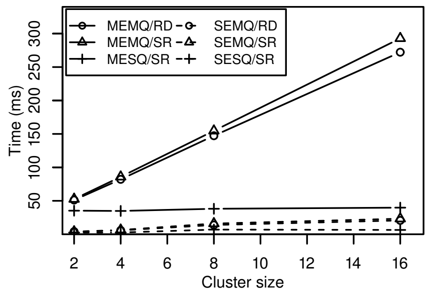


Figure 12. Time to build RDMA connection.

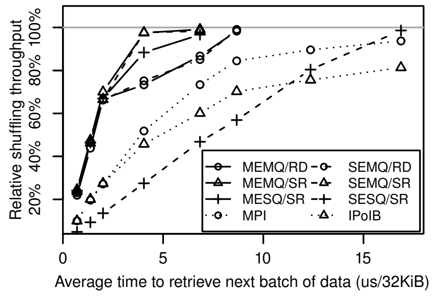


Figure 13. Performance for compute-intensive query.

algorithms ran on 16 nodes in the EDR cluster and with a different number of endpoints. (This controls the number of Queue Pairs, as shown in Table 1.)

The result is plotted in Figure 11. 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 than the MQ/SR and MQ/RD algorithms with fewer Queue Pairs. Note that in a larger cluster the SQ/SR algorithm would use the same number of Queue Pairs, while all the MQ algorithms would use proportionally more Queue Pairs.

5.1.5 Fixed costs of setting up RDMA communication

Some queries do not shuffle a lot of data. For such queries, the communication initialization time matters as much as the peak throughput. One question is whether a database system should build the RDMA connections at runtime. We report the time spent in building the connection, registering memory, and de-registering memory in the EDR cluster as the cluster size increases to answer this question.

The time to build RDMA connections is shown in Figure 12. (The time to register and de-register memory is negligible as it takes less than 5 ms and 1 ms respectively.) The horizontal axis is the number of nodes and the vertical axis is the time to build the RDMA connections. The ME algorithms take longer to build the connection than the SE algorithms as the ME algorithms need to connect multiple endpoints. The connection time increases linearly for the 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.) Prior work shows that RDMA connection setup takes about 200 ms [10], but our evaluation reveals that the set up time for the MESQ/SR algorithm stays stable at less than 40 ms when scaling out. The high throughput of the MESQ/SR algorithm can make up for the slow connection time: Queries which shuffle as little as 250 MB data using the MESQ/SR algorithm will outperform IPoIB when building connections at runtime.

5.1.6 Performance for compute intensive query

In the experiments so far we have compared all algorithms with a synthetic network-bound query. This section studies how the different shuffling algorithms perform when the query becomes compute intensive. In our experiment, we adjust the compute intensity of the receiving query fragment (pipeline) to simulate different compute demands in real queries. We evaluate the repartition algorithms using 8 nodes in the EDR cluster. The receiving plan fragment continuously fetches new data in batches of 32 KiB (the L1 data cache size in the EDR cluster) from the receive operator and then processes the data.

In Figure 13, the horizontal axis shows the average time to retrieve the next 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 correspond to the processing time per batch: all threads process batches concurrently in the receiving query fragment and any thread can "snatch" the next batch for processing.) The vertical axis is the shuffling throughput of the RDMA algorithm relative to the processing throughput of the receiving fragment. The relative shuffling throughput reaches 100% when computation and communication completely overlap.

As shown in Figure 13, all algorithms are network-bound if the receiving fragment does minimal processing. At the leftmost point, the throughput of the data shuffling algorithm ($\sim$ 11 GiB/sec) is only a fraction of the throughput of the receiving query fragment ($\sim$ 50 GiB/sec). As the receiving query fragment becomes more compute intensive, the MQ/SR and MESQ/SR algorithms reach peak throughput earlier than the MQ/RD algorithms. All RDMA algorithms except SESQ/SR outperform MPI and IPoIB for both network-bound and compute-bound queries. Interestingly, MPI and IPoIB fail to completely overlap communication and computation even for compute-intensive queries.

5.2 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 5.1.3. We distribute each tuple of every table in TPC-H to a random node in the cluster, except for the NATION and REGION tables which we replicate to all nodes (they contain only 25 and 5 tuples, respectively). This data distribution mimics the experimental setup used in prior work [12, 37, 41]. 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 [4], and we use the query optimizer of a commercial database system to obtain the query execution plan.

5.2.1 Response time with faster network adapter

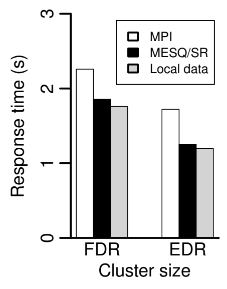
We first investigate how query response time changes as one upgrades from the slower 56Gbps FDR InfiniBand to the faster 100Gbps EDR InfiniBand. In this experiment the same TPC-H database with scale factor 400 is distributed across the memory of 8 nodes in both the FDR and EDR clusters.

Figure 14(a) 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 8-node broadcast throughput of MESQ/SR over MPI in Figures 10(b) and 10(d). Second, we observe that the MESQ/SR algorithm has similar performance as the "local data" plan that doesn't 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 further behind (about 30% gain from FDR to EDR).

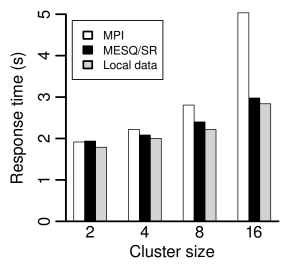
5.2.2 Query response time when scaling

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 1600 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.

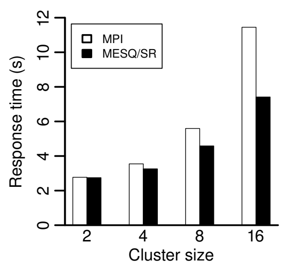
The response time of TPC-H Q4, Q3 and Q10 is shown in Figures 14(b), 14(c) and 14(d). The "local data" bar in Figure 14(b), again, shows the performance of the query plan if



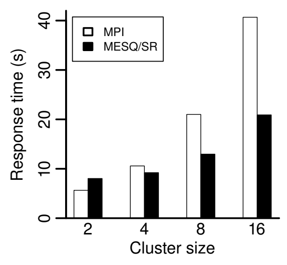
(a) Effect of EDR network on response time of TPC-H query 4, 8 nodes.



(b) TPC-H query 4.



(c) TPC-H query 3.



(d) TPC-H query 10.

Figure 14. TPC-H query response time, 100 GiB per node.

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.

6. Related Work

High performance networks. Foong et al. [9] show that about 1GHz in CPU performance is necessary for every 1Gb/s network throughput. In addition to being CPU intensive, Frey et al. [11, 12] show that TCP introduces traffic on the memory bus because of data copying. The zero-copy and CPU-bypass features of RDMA bypass that overhead. Frey et al. [10] show that communication can benefit from RDMA only when buffers are large and are reused.

RDMA is extensively studied in supercomputing. Liu et al. [23, 25] use Send/Receive and RDMA Write to transmit data in MVAPICH, while they use RDMA Write to transmit small messages and RDMA Read to transmit large messages in the MPICH2 implementation. Liu et al. [24] have also studied how to efficiently implement broadcast in MPI. MacArthur and Russell [27] compare the performance of different RDMA verbs. Koop et al. [19] 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. [28] use Send/Receive and Read to implement puts and gets, respectively, in key-value stores. Kalia et al. [15] use unreliable RDMA Write for client requests and use Send/Receive in unreliable datagram transport for server responses. Lu et al. [14, 26] improve the performance of Hadoop and HBase by using RDMA instead of TCP/IP communication. Dragojević et al. [8] designed FaRM, a computing platform which uses RDMA Read for data accesses and RDMA Write for messages. Wu et al. [44] have extended FaRM with graph processing capabilities. Chen et al. [6] and Wei et al. [43] implement a distributed in-memory transaction processing system with one-sided memory and atomic primitive provided in RDMA and hardware transactional memory.

New algorithms for fast networks. Li et al. [22] have studied the data shuffling problem in NUMA systems and proposed careful scheduling of the communication. Polychroniou et al. [33] 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. [38] 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 that is available in high-end networks, these algorithms are orthogonal to the selection of the network.

More recent work has also considered how RDMA interacts with database operations. Utilizing the features of RDMA, Frey et al. [10, 12] 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. Tinnefeld et al. [42] 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. [30] study the performance of database when using memory in remote nodes using RDMA. Barthels et al. [3] study how to scale the radix join algorithm to rack-scale using RDMA to transmit data during the partitioning phase of the join. Rödiger et al. [37] propose hybrid parallelism to distinguishes local and distributed parallelism and design a push-based multiplexer to shuffle data; all threads share the same multiplexer. In contrast, in our work, we design pull-based endpoints and systematically explore both one-sided shared memory primitives and two-sided message-passing primitives as well as different endpoint assignments to threads. Recently, Rödiger et al. [36] propose the "flow-join" algorithm that uses RDMA to ameliorate skew during the join. Li et al. [21] uses RDMA to directly access the buffer pool of other nodes in Microsoft SQL Server. Barthels et al. [2] used MPI to explore the performance of distributed join algorithms in HPC systems with thousands of CPU cores.

7. Conclusions and future work

This paper studies the challenges and opportunities in utilizing RDMA to shuffle data among query fragments in parallel database systems. We propose six algorithms which utilize both a reliable and an unreliable transport service 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 an RDMA-capable MPI implementation.

Our future work will explore three avenues. First, we plan to implement an endpoint based on the RDMA Write primitive to evaluate its performance. Second, we plan to investigate how performance changes when using the same algorithms in RoCE and iWARP networks. Third, we plan to specialize the MESQ/SR algorithm to use the native InfiniBand multicast primitive for broadcasting data. We hypothesize that this will reduce the CPU cost during analytical query processing, as MESQ/SR already achieves throughput close to the line bandwidth.

Analytical performance remains an end-to-end concern that requires the interplay of many different algorithms. Faster data transmission will naturally expose bottlenecks in other components of the analytical execution pipeline. Amdahl's law suggests that future performance gains will come from directly integrating RDMA capabilities within individual algorithms and from holistically rethinking query processing for RDMA-capable networks.

Source code

We have implemented all data shuffling algorithms in the open-source Pythia query engine. Our code can be found at `https://code.osu.edu/pythia/core`.

Acknowledgements

We would like to thank the anonymous reviewers and Jens Teubner, our shepherd, for their insightful comments that greatly improved this paper. This research was partially supported by the National Science Foundation under grants III-1422977, III-1464381, CNS-1513120 and by a Google Research Faculty Award. The evaluation was conducted in part at the Ohio Supercomputer Center [32].

References

[1] `http://www.accelio.org/`.

[2] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[3] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD '15*, pages 1463–1475. ACM, 2015.

[4] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, pages 61–76, 2013.

[5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[6] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 26:1–26:17, New York, NY, USA, 2016. ACM.

[7] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, Mar. 1990.

[8] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. NSDI'14, pages 401–414, 2014.

[9] A. P. Foong, T. R. Huff, H. H. Hum, J. R. Patwardhan, and G. J. Regnier. TCP performance re-visited. ISPASS '03, pp. 70–79, 2003.

[10] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. ICDCS 2009, pages 553–560, 2009.

[11] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: High-speed networks for distributed join processing. DaMoN '09, pages 27–33, New York, NY, USA, 2009. ACM.

[12] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. ICDCS 2010, pages 283–292, 2010.

[13] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[14] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance RDMA-based design of HDFS over InfiniBand. SC '12, 2012.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. SIGCOMM'14, pp.295–306, 2014.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, GA, Nov. 2016. USENIX Association.

[18] A. Kesavan, R. Ricci, and R. Stutsman. To copy or not to copy: Making in-memory databases fast on modern nics. In *ADMS-IMDM 2016*, Nov. 2016.

[19] M. J. Koop, T. Jones, and D. K. Panda. Reducing connection memory requirements of MPI for InfiniBand clusters: A message coalescing approach. CCGrid 2007, pages 495–504, 2007.

[20] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD '14, pages 743–754. ACM, 2014.

[21] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. SIGMOD '16, pages 355–370. ACM, 2016.

[22] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[23] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. R. Toonen. Design and implementation of MPICH2 over InfiniBand with RDMA support. *CoRR*, cs.AR/0310059, 2003.

[24] J. Liu, A. R. Mamidala, and D. K. Panda. Fast and scalable MPI-level broadcast using InfiniBand hardware multicast support. IPDPS, 2004.

[25] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[26] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of Hadoop RPC with RDMA over InfiniBand. ICPP '13, pages 641–650, 2013.

[27] P. MacArthur and R. D. Russell. A performance study to guide RDMA programming decisions. HPCC-ICESS, pp. 778–785, 2012.

[28] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. USENIX ATC'13, pages 103–114, 2013.

[29] `http://www.mpi-forum.org/`.

[30] H. Mühleisen, R. Gonçalves, and M. Kersten. Peak performance: Remote memory revisited. DaMoN '13, pages 9:1–9:7, 2013.

[31] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[32] Ohio Supercomputer Center. Ruby Supercomputer. `http://osc.edu/ark:/19495/hpc93fc8`, 2015.

[33] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. SIGMOD'14, 1483–1494, 2014.

[34] `https://code.osu.edu/pythia/core`.

[35] `https://www.openfabrics.org/downloads/qperf/`.

[36] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. ICDE'16, 2016.

[37] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4):228–239, 2015.

[38] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. ICDE 2014, pages 592–603, 2014.

[39] `https://linux.die.net/man/7/rsocket`.

[40] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. SIGMOD '79, pages 23–34, 1979.

[41] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 104–114, New York, NY, USA, 1995. ACM.

[42] C. Tinnefeld, D. Kossmann, J. Böse, and H. Plattner. Parallel join executions in RAMCloud. In *Workshops Proceedings of the 30th International Conference on Data Engineering*, pp. 182–190, 2014.

[43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[44] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling graph computation to the trillions. SoCC '15, pages 408–421, 2015.