The End of Slow Networks: It's Time for a Redesign

Carsten Binnig Andrew Crotty Alex Galakatos Tim Kraska Erfan Zamanian
Department of Computer Science, Brown University
{firstname_lastname}@brown.edu

ABSTRACT

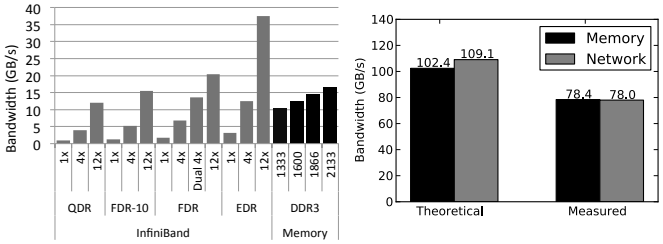
The next generation of high-performance networks with remote direct memory access (RDMA) capabilities requires a fundamental rethinking of the design of distributed in-memory DBMSs. These systems are commonly built under the assumption that the network is the primary bottleneck and should be avoided at all costs, but this assumption no longer holds. For instance, with InfiniBand FDR 4×, the bandwidth available to transfer data across the network is in the same ballpark as the bandwidth of one memory channel. Moreover, RDMA transfer latencies continue to rapidly improve as well. In this paper, we first argue that traditional distributed DBMS architectures cannot take full advantage of high-performance networks and suggest a new architecture to address this problem. Then, we discuss initial results from a prototype implementation of our proposed architecture for OLTP and OLAP, showing remarkable performance improvements over existing designs.

1. INTRODUCTION

We argue that the current trend towards high-performance networks with *remote direct memory access* (RDMA) capabilities like InfiniBand will require a complete redesign of modern distributed in-memory DBMSs. These systems are built on the assumption that the network is the main bottleneck [7] and consequently aim to avoid communication between nodes, using techniques such as locality-aware partitioning schemes [49, 45, 17, 62], semi-reductions for joins [51], and complicated preprocessing steps [47, 53]. Yet, with the nascent modern network technologies, the assumption that the network is the bottleneck no longer holds.

Even today, the bandwidth available to transfer data over the network with InfiniBand FDR 4× [6] is in the same ballpark as the bandwidth of one memory channel. DDR3 memory bandwidth currently ranges from 6.25GB/s (DDR3-800) to 16.6GB/s (DDR3-2133) [1] per channel, whereas InfiniBand has a specified bandwidth of 1.7GB/s (FDR 1×) to 37.5GB/s (EDR 12×) [6] per NIC port (see Figure 1(a)).

Proceedings of the VLDB Endowment, Vol. 9, No. 7



(a) Specification (b) Experimental

Figure 1: Memory vs Network Bandwidth: (a) specification, (b) for a Dual-socket Xeon E5v2 server with DD3-1600 and two FDR 4× NICs per socket

Moreover, future InfiniBand standards (e.g., HDR, NDR) promise bandwidths that will dramatically exceed memory channel bandwidth.

However, modern CPUs typically support four memory channels per socket. For example, a machine with DDR3-1600 memory has 12.8GB/s per channel, with a total aggregate memory bandwidth of 51.2GB/s; four dual-port FDR 4× NICs could provide roughly the same bandwidth.[1] Furthermore, data transfer between the CPU and memory is half-duplex, while InfiniBand and PCIe are full-duplex, such that *only* two NICs could saturate the memory bandwidth of a read/write workload. With the newer InfiniBand EDR standard, only a single dual-port NIC is needed. Figure 1(b) shows the theoretical (left) and measured (right) total memory and network throughput for a dual-socket machine with DDR3-1600 memory and two FDR 4× NICs per socket (four total). This microbenchmark shows that the network transfer on our hardware is limited by the available memory bandwidth. For measuring the local memory bandwidth, we used a 1:1 read/write ratio, which best simulates a distributed workload where two machines simultaneously write data to each other. While these measurements are for InfiniBand, we expect that Ethernet networks will become similarly advanced [57, 24].

Another important factor is that network latency is rapidly improving with major advances in RDMA. For example, our microbenchmarks show that the network transfer latency with InfiniBand FDR 4× using RDMA to send 1KB of data is $\approx 1\mu$s, compared to $\approx 0.08\mu$s for the CPU to read the same amount of data from memory. With messages of only 256KB, there is virtually no difference between the access times, since the bandwidth becomes the dominant factor.

[1]We do *not* assume that the PCIe bus is a bottleneck since current Xeon E5 boards have 39.4GB/s of total bandwidth.

Yet, we do not argue that network latency will ever meet or exceed memory latency; rather we believe that efficient use of CPU caches and local memory will play an even more important role for small data requests (e.g., a hash table lookup) because performance is no longer dictated by network transfers.

At the same time, InfiniBand is becoming increasingly affordable for smaller deployments. For example, a small cluster with eight servers, each with two Xeon E5v2 CPUs and one 2-port InfiniBand FDR 4× NIC, and a total of 2TB of DDR3-1600 memory costs under $80K, with the switch and NICs representing roughly $20K of the total cost. In this configuration, the bandwidth for sending data across the network (13.6GB/s) is close to the bandwidth of one memory channel (12.8GB/s). Furthermore, memory prices continue to drop such that even large datasets can fit entirely in memory on just a few machines, thereby removing disk I/O and creating a more balanced system.

However, it is *wrong to assume that a high-performance network changes the cluster to a NUMA architecture* because: (1) RDMA-based memory access patterns are very different from local memory access patterns; (2) random access latency for remote requests is still significantly higher; and (3) hardware-embedded coherence mechanisms that ensure data consistency in a NUMA architecture do not exist for RDMA. We therefore believe that clusters with RDMA should be regarded as hybrid shared-memory and message-passing architectures: they are neither a shared-memory system (several address spaces exist) nor a pure message-passing system (data can be directly accessed via RDMA).

Consequently, we argue that it is time for a complete redesign of traditional distributed DBMS architectures in order to fully leverage the next generation of network technologies. For example, given the fast network, it is no longer obvious that avoiding distributed transactions is always beneficial.Similarly, distributed algorithms (e.g., joins) should no longer be designed to minimize network communication [47]; instead, they should carefully consider multi-core architectures and CPU caching effects. While our proposal is not the first attempt to leverage RDMA for distributed DBMSs [59, 54, 38], existing work does not fully recognize that next generation networks create an architectural inflection point.

In summary, this paper makes the following contributions:

- We present microbenchmarks to assess performance characteristics of one of the latest InfiniBand standards, FDR 4× (Section 2).
- We present alternative architectures for a distributed in-memory DBMS over fast networks and introduce a novel Network-Attached Memory (NAM) architecture (Section 3).
- We show why the common wisdom that says "2-phase-commit does not scale" no longer holds for RDMA-enabled networks and outline how OLTP workloads can take advantage of the network by using the NAM architecture (Section 4).
- We analyze the performance of distributed OLAP operations (joins and aggregations) and propose new algorithms for the NAM architecture (Section 5).

2. BACKGROUND

Before making a detailed case concerning why and how distributed DBMS architectures need to fundamentally change in order to take advantage of the next generation of network technologies, we first provide some background information and microbenchmarks that describe the characteristics of InfiniBand and RDMA.

2.1 InfiniBand and RDMA

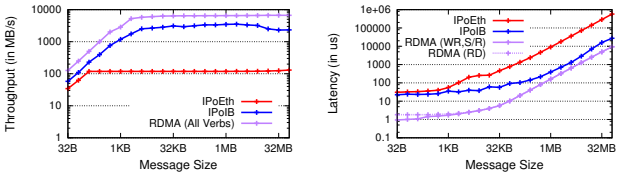
In the past, InfiniBand was a very expensive, high bandwidth, low latency network found only in high-performance computing settings. However, InfiniBand has recently become cost-competitive with Ethernet and thus a viable alternative for networking in small clusters.

Communication Stacks: InfiniBand offers two network communication stacks: IP over InfiniBand (IPoIB) and remote direct memory access (RDMA). IPoIB implements a classic TCP/IP stack over InfiniBand, allowing existing socket-based applications to run without modification. As with Ethernet-based networks, data is copied by the application into OS buffers that the kernel then processes by sending packets over the network. While providing an easy migration path from Ethernet to InfiniBand, our experiments show that IPoIB cannot fully leverage the network's capabilities. On the other hand, RDMA provides a *verbs* API, which enables data transfer using the processing capabilities of an RDMA NIC (RNIC). With verbs, most of the processing is executed by the RNIC without OS involvement, which is essential for achieving low latencies.

RDMA provides two verb communication models: one-sided and two-sided. One-sided RDMA verbs (write, read, and atomic operations) are executed without involving the CPU of the remote machine. RDMA WRITE and READ operations allow a machine to write (read) data into (from) the remote memory of another machine. Atomic operations (fetch-and-add and compare-and-swap) allow remote memory to be modified atomically. Two-sided verbs (SEND and RECEIVE) enable applications to implement an RPC-based communication pattern that resembles a socket-based approach. Unlike one-sided verbs, two-sided verbs involve the CPU of the remote machine.

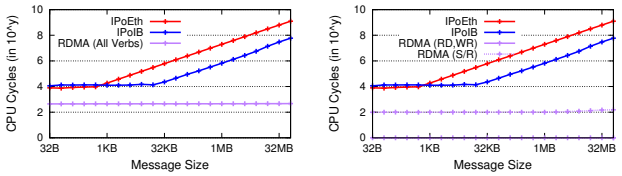
RDMA Details: RDMA connections are implemented using queue pairs (i.e., send/receive queues). The application creates queue pairs on both the client and the server, and the RNICs handle the state of the queue pairs. To communicate, a client creates a Work Queue Element (WQE) by specifying a verb and parameters (e.g., a remote memory location). The client then puts the WQE into a send queue and informs the local RNIC via Programmed IO (PIO) to process the WQE. WQEs can be sent either signaled or unsignaled. For a signaled WQE, the local RNIC pushes a completion event into a client's *completion queue* (CQ) via a DMA WRITE once the WQE has been processed by the remote side. With one-sided verbs, WQEs are handled by the remote RNIC without interrupting the remote CPU using a DMA operation on the remote side (called server). However, one-sided operations require memory regions to be registered to the local and remote RNIC a priori in order to be accessible by DMA operations (i.e., the RNIC stores the virtual to physical page mappings of the registered region). For two-sided verbs, the server must additionally put a RECEIVE request into its receive queue to handle a SEND request from the client.

Since queue pairs process WQEs in FIFO order, a typical pattern to reduce the overhead on the client side and mask latency is to use selective signaling. That is, for send/receive queues of length n, the client can send $n-1$ WQEs



(a) Throughput (b) Latency

Figure 2: Network Throughput and Latency



(a) Client (b) Server

Figure 3: CPU Overhead for Network Operations

unsignaled and the n-th WQE signaled. Once the completion event (i.e., the acknowledgment message of the server) for the n-th WQE arrives, the client implicitly knows that the previous $n-1$ WQEs have also been successfully processed. In this scenario, computation and communication on the client can be efficiently overlapped without expensive synchronization mechanisms.

Another interesting aspect is how RDMA operations of an RNIC interfere with CPU operations if data is concurrently accessed. Recent Intel CPUs (Intel Sandy-Bridge and later) provide a feature called Data Direct I/O (DDIO) [5]. With DDIO the DMA executed by the RNIC to read (write) data from (to) remote memory places the data directly in the CPU L3 cache if the memory address is resident in the cache to guarantee coherence. On other systems without DDIO, the cache is flushed/invalidated by the DMA operation to guarantee coherence. Finally, non-coherent systems leave the coherency problem to the software. These effects must be considered when designing distributed RDMA-based algorithms. Note that this only concerns coherence between the cache and memory, not the coherence between remote and local memory, which is always left to the software.

2.2 Microbenchmarks

This section presents microbenchmarks that compare the throughput and latency of: (1) a TCP/IP stack over 1Gb/s Ethernet (IPoEth), (2) IPoIB, and (3) RDMA. These results form the basis of our proposals for the redesign of distributed DBMSs in order to fully leverage high-performance networks.

Experimental Setup: Our microbenchmarks used two machines, each with an Intel Xeon E5-2660 v2 processor and 256GB RAM, running Ubuntu 14.04 with the OFED 2.3.1 RNIC driver. Both machines were equipped with a Mellanox Connect IB FDR 4× dualport RNIC, and each RNIC port has a full-duplex bandwidth of 54.54Gb/s (6.8GB/s). Additionally, each machine had a 1Gb/s Ethernet NIC (one port) connected to the same Ethernet switch. We used only one port on each RNIC to ensure a fair comparison between InfiniBand and Ethernet, and all microbenchmarks used single-threaded execution in order to isolate low-level network properties.

Throughput and Latency (Figure 2): For this experiment, we varied the message size from 32B up to 32MB to simulate the characteristics of different workloads (i.e., OLTP and OLAP), measuring the throughput and latency for IPoEth, IPoIB, and RDMA send/receive and read/write. Additionally, we measured the RDMA atomic operations but omitted the results from the figure, since they only support a maximum message size of 8B and provide the same throughput/latency as 8B READs.

While all RDMA verbs saturate the InfiniBand network bandwidth (≈ 6.8GB/s) for message sizes greater than 2KB, IPoIB only achieves a maximum throughput of 3.5GB/s despite using the same network hardware. Moreover, the latency of sending a message (i.e., 1/2 RTT) over IPoIB is also higher than the latency of RDMA verbs. In fact, for small message sizes, the latency of IPoIB is much closer to the latency of the 1Gb/s Ethernet network (IPoEth). For example, for a message size of 8B, the latency is 20μs for IPoIB and 30μs for IPoEth, while an RDMA WRITE takes only 1μs. This is because the TCP/IP stack for IPoIB has a very high CPU overhead per message for small messages, as discussed in following section. For larger message sizes ($\geq$ 1MB), the latency of IPoIB is closer to the latency of RDMA but still a factor of 2.5× higher. For example, sending a 1MB message has a latency of 393μs over IPoIB compared to 161μs for RDMA.

An interesting observation is that RDMA WRITE/SEND operations take only 1μs for message sizes less than 256B, while RDMA READ operations take 2μs, since payloads of less than 256B can be inlined into the PIO to avoid a subsequent DMA read [40].

CPU Overhead (Figure 3): We also measured the overhead in CPU cycles for messages over different communication stacks on both the client and server, varying the message sizes as in the previous experiment. RDMA has a constant overhead on both the client and server that is independent of message size because of the constant cost of registering a WQE on the RNIC. The actual data transfer is executed by the RNIC, which acts as a coprocessor to handle a given WQE. On the client side, the overhead is ≈ 450 cycles regardless of the RDMA verb used, including atomic operations. On the server side, only the RECEIVE verb incurs CPU overhead, as expected. All other one-sided verbs (i.e., READ/WRITE and atomic operations) do not incur any overhead on the server side.

The CPU overhead of IPoIB is very different from that of RDMA and is in fact much closer to the Ethernet-based TCP/IP stack (IPoEth). Unlike RDMA, the CPU overhead per message grows linearly with the message size after exceeding the TCP window size for both IPoEth and IPoIB. In our experiments, the default TCP window size was 1488B for IPoEth and 21888B for IPoIB. For small message sizes, the CPU overhead per message for IPoIB is even higher than for IPoEth. For example, an 8B message requires 13264 cycles for IPoIB compared to only 7544 cycles for IPoEth.

3. RETHINKING THE ARCHITECTURE

In this section, we discuss why the traditional shared-nothing architecture for distributed in-memory DBMSs is suboptimal for high-performance networks and present novel alternatives that directly leverage RDMA. We then discuss research challenges that arise for these new architectures.

3.1 Architecture Types

Distributed DBMSs face two primary challenges: (1) distributed control-flow (e.g., synchronization), and (2) dis-

tributed data-flow (e.g., data exchange between nodes). We describe how three existing distributed DBMS architectures handle these challenges and then propose a new architecture designed specifically for fast networks.

3.1.1 Traditional Shared-Nothing

Figure 4(a) shows the shared-nothing (SN) architecture for distributed in-memory DBMSs. Data is partitioned across each of the nodes, and each node has direct access only to its local partition. In order to implement distributed control-flow and data-flow, nodes communicate with each other using socket-based send and receive operations.

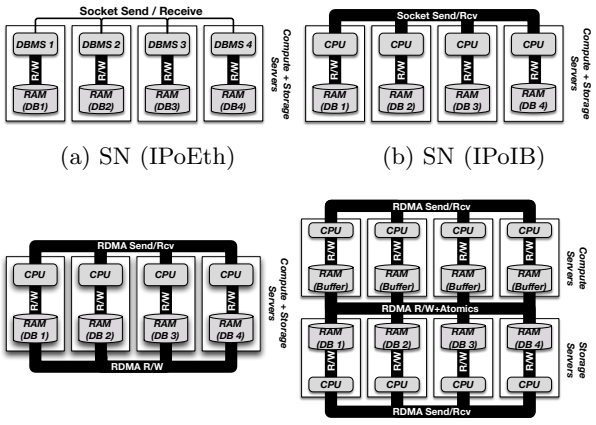
Efficient distributed query and transaction processing aims to maximize data-locality for a given workload by applying locality-aware partitioning schemes or employing strategies to avoid communication (e.g., semi-joins). In the extreme case, no communication would need to occur between nodes. For many real-world workloads, however, network communication cannot be entirely avoided, resulting in large performance penalties for slow networks. For example, even using the best techniques for co-partitioning tables [18, 45], it is not always possible to avoid expensive distributed join operations or distributed transactions, leading to high communication costs [47]. Furthermore, workloads change over time, making it difficult to find a good static partitioning scheme [21], while dynamic strategies often require moving huge amounts of data, further restricting the bandwidth for the actual work. As a result, the network limits the throughput of the system as well its scalability; that is, the more machines that are added, the more of a bottleneck the network becomes.

3.1.2 Shared-Nothing for IPoIB

An easy way to migrate a traditional shared-nothing architecture to a high-performance network is to simply use IPoIB as shown in Figure 4(b). A big advantage of this architecture is that almost no change to the DBMS is required, but the system can still benefit from the higher bandwidth. In particular, data-flow operations that send large messages (e.g., data re-partitioning) will benefit tremendously from this change. However, as shown in Section 2, IPoIB cannot fully leverage the network. Perhaps surprisingly, for some types of operations, upgrading the network and using IPoIB can actually decrease performance, particularly for control-flow operations which send many small messages. Figure 3 shows that the CPU overhead of IPoIB is greater than the CPU overhead of IPoEth for small messages. In fact, as we will show in Section 4, these small differences can have a negative impact on the overall performance of distributed transaction processing.

3.1.3 Distributed Shared-Memory

Obviously, to better leverage the network we have to take advantage of RDMA. RDMA not only allows the system to fully utilize the bandwidth (see Figure 2(a)), but also reduces network latency and CPU overhead (see Figures 2(b) and 3). Unfortunately, changing an application from a socket-based message passing interface to RDMA verbs is not trivial. One possibility is to treat the cluster as a shared-memory system (shown in Figure 4(c)) with two types of communication patterns: (1) message passing using RDMA-based SEND/RECEIVE verbs, and (2) remote direct memory access through one-sided RDMA READ/WRITE verbs.



(a) SN (IPoEth) (b) SN (IPoIB)

(c) SM (RDMA) (d) NAM (RDMA)

Figure 4: In-Memory Distributed Architectures

However, as previously mentioned, there is no mechanism for cache-coherence. Moreover, machines need to carefully declare the shareable memory regions a priori and connect via queue pairs. The latter, if not used carefully, can also have a negative effect on the performance [29]. In addition, a memory access via RDMA is very different than that of a shared-memory system. While a local memory access only keeps one copy of the data around (i.e., conceptually it moves the data from main memory to the cache of a CPU), a remote memory access creates a fully independent copy. This has a range of implications including garbage collection, cache/buffer management, and consistency protocols.

Thus, in order to achieve the appearance of a shared-memory system, the software stack has to hide the differences and provide a distributed shared-memory space. There have been recent attempts to create a distributed shared-memory architecture over RDMA [19]. However, we believe that a single abstraction for local and remote memory is the wrong approach. Since DBMSs prefer to have full control over memory management (e.g., virtual memory can interfere with a DBMS), we believe the same is true for shared-memory over RDMA. While we had the ambitions to validate this assumption through our experiments, we only found one commercial offering for IBM mainframes [4]. Instead, for our OLTP comparison, we implemented a simplified version of this architecture by essentially using a SN architecture and replacing socket communication with two-sided RDMA verbs (send and receive). We omit this architecture entirely from our OLAP comparison since two-sided RDMA verbs would have added additional synchronization overhead (i.e., an RDMA RECEIVE must be issued strictly before the RDMA SEND arrives at the RNIC).

3.1.4 Network-Attached Memory

Based on the previous considerations, we envision a new type of architecture, referred to as network-attached memory (NAM) and shown in Figure 4(d). In a NAM architecture, compute and storage are logically decoupled. The storage servers provide a shared distributed memory pool, which can be accessed from any compute node. However, the storage nodes are not aware of any DBMS specific operations (e.g., joins or consistency protocols). These are implemented by the compute nodes.

This logical separation helps to control the complexity

and makes the system aware of the different types of main memory. Moreover, storage nodes can take care of issues like garbage collection, data-reorganization, and metadata management (e.g., remote-memory address to data page mapping). Note, that it is possible to physically co-locate storage nodes and compute nodes on the same machine to further improve performance. However, in contrast to previous architectures, the system gains more control over what data is copied and how copies are synchronized.

The NAM architecture also has several other advantages compared to the previously mentioned architectures. Most importantly, storage nodes can be scaled independently of compute nodes. Furthermore, the NAM architecture can efficiently handle data imbalance since any node can access any remote partition without the need to re-distribute the data. Although the separation of compute and storage is not new, existing systems either use an extended key/value like interface for the storage nodes [13, 35, 38] or are focused on the cloud [14, 2], instead of being built from scratch to leverage high-performance networks. Instead, we argue that the storage servers in the NAM architecture should expose an interface that supports fine-grained byte-level memory access that preserves features of the underlying hardware. For example, in Section 4, we show how fine-grained addressability allows us to efficiently implement concurrency control. In the future, we plan to take advantage of the fact that messages of connected queues are ordered.

3.2 Challenges and Opportunities

Unfortunately, moving from a shared-nothing or shared-memory system to a NAM architecture requires a redesign of the entire distributed DBMS architecture from storage management to query processing and transaction management up to query compilation and metadata management.

Query Processing & Transactions: Distributed query processing is typically implemented using a data-parallel execution scheme that leverages repartitioning operators that shuffle data over the network. However, repartitioning operators do not typically consider efficiently leveraging the CPU caches of individual machines in the cluster. Thus, we believe that there is a need for parallel cache-aware algorithms for query operators over RDMA.

Similarly, new query optimization techniques for distributed in-memory DBMSs will be required for high-bandwidth networks. As previously mentioned, existing distributed DBMSs assume that the network is the dominant bottleneck. Therefore existing cost-models for distributed query optimization often only consider network cost [43]. With fast networks and more balanced system, the optimizer needs to consider more factors since bottlenecks can shift from one component (e.g., CPU) to another (e.g., memory-bandwidth) [16].

Additionally, we believe that a NAM architecture requires new load-balancing schemes that implement ideas suggested for work-stealing on single-node machines [34]. For example, query operators could access a central data structure (i.e., a work queue) via one-sided RDMA verbs, which contains pointers to small portions of data to be processed by a given query. When a node is idle, it could pull data from the work queue. In this scenario, distributed load balancing schemes can be efficiently implemented in a decentralized manner. Compared to existing distributed load balancing schemes, this alleviates single bottlenecks and would allow greater scalability while also avoiding stragglers.

Storage Management: Since the latency of one-sided RDMA verbs (i.e., read and write) to access remote data partitions is still much higher than for local memory accesses, we need to optimize the storage layer of a distributed DBMS to minimize this latency.

One idea in this direction is to develop *complex* storage access operations that combine different storage primitives in order to effectively minimize the number of network round-trips between compute and storage nodes. This approach is in contrast to existing storage managers which offer only *simple* read/write operations. For example, in Section 4, we present a complex storage operation for a distributed SI protocol that combines the locking and validation of the 2PC commit phase using a single RDMA atomic operation. However, for such complex operations, the memory layout must be carefully developed. Our current prototype therefore combines the lock information and the value into a single memory location.

Modern RNICs, such as the Connect X4 Pro, provide a programmable device (e.g., an FPGA) on the RNIC. Thus, another idea to reduce storage access latencies is to implement complex storage operations that cannot easily be mapped to existing RDMA verbs in hardware. For example, writing data directly into a remote hash table of a storage node could be implemented completely on the RNICs in a single round-trip without involving the CPUs of the storage nodes, allowing for new distributed join operations.

Finally, we believe that novel techniques must be developed that allow efficient prefetching using RDMA. The idea is that the storage manager issues RDMA requests (e.g., RDMA READs) for memory regions that are likely to be accessed next and the RNIC processes them asynchronously in the background. In this scenario, the RDMA storage manager would need to first poll the completion queue when a request for a remote memory address arrives to check if the remote memory has already been prefetched. While this is straightforward for sequentially scanning a table partition, index structures require a more careful design since they often rely on random access.

Centralized Master: Typically, a distributed DBMS has one central master node responsible for tasks such as metadata management and query deployment. In a traditional architecture, this central node can become a bottleneck under heavy loads and is a single point of failure. However, in a NAM architecture, any node can read and update the metadata or deploy queries, since all nodes can access central data structures using RDMA.

4. THE CASE FOR OLTP

The traditional wisdom is that distributed transactions, particularly when using two-phase commit (2PC), do not scale [58, 30, 56, 17, 44, 52]. In this section, we show that this is the case on a shared-nothing architecture over slow networks and then present a novel protocol for the NAM architecture that can take full advantage of the network and, theoretically, removes the scalability limit.

4.1 Why 2PC Does Not Scale

In this section, we discuss factors that hinder the scalability of distributed transactions over slow networks. Many modern DBMSs employ Snapshot Isolation (SI) to implement concurrency control and isolation because it promises superior performance compared to lock-based alternatives.

The discussion in this section is based on a 2PC protocol for generalized SI [36, 22]. However, the findings can also be generalized to more traditional 2PC protocols [41].

4.1.1 Dissecting 2PC

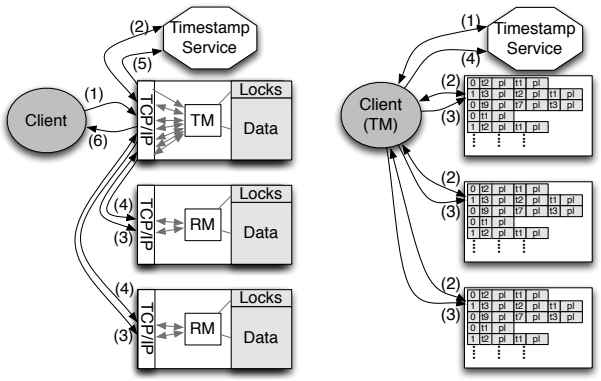
Figure 5(a) shows a simplified traditional 2PC protocol with generalized SI guarantees [36, 22], assuming a shared-nothing architecture and no read-phase (see [13, 15, 48]).That is, we assume that the client (e.g., application server) has read all necessary records to issue the full transaction using a potentially older *read timestamp* (RID), which guarantees a consistent view of the data.After the client finishes reading the records, it sends the commit request to the *transaction manager* (TM) [one-way message 1]. Note that there can be more than one TM to distribute the load across nodes.

As a next step, the TM requests a *commit timestamp* (CID) [round-trip message 2]. In this paper, we assume that an external service provides globally ordered timestamps, as suggested in [13] or [15]. Since the implementation of the timestamp service is orthogonal, we assume that it is not a bottleneck when using approaches like Spanner [15] or epoch-based SI [61].

After receiving the CID, the TM sends prepare messages to the *resource managers* (RMs) of the other nodes involved in the transaction [round-trip message 3]. Each RM (1) checks if any records in its partition have been modified since being read by the transaction and (2) locks each tuple to prevent updates by other transactions after the validation [33], which normally requires checking if any of the records of the write-sets has a higher CID than the RID. The TM can then send commit messages to all involved RMs if the prepare phase was successful [round-trip message 4], which installs the new version (value and CID) and releases the locks. In order to make the new value readable by other transactions, the TM needs to wait until the second phase of 2PC completes [message 4], and then inform the timestamp service that a new version was installed [one-way message 5]. For the remainder, we assume that the timestamp service implements a logic similar to [13] or Oracle RAC [48] in order to ensure the SI properties. That is, if a client requests an RID, the timestamp service returns the largest committed timestamp. Finally, the TM notifies the client about the outcome of the transaction [one-way message 6].

Overall the protocol requires nine one-way message delays if sent in the previously outlined sequential order. However, some messages can be sent in parallel; in particular, the CID [message 2] can be requested in parallel to preparing the RM [message 3], since the CID is not required until the second phase of 2PC [message 4]. This simplification is possible because we assume blind writes are not allowed, such that a transaction must read all data items (and their corresponding RIDs) in its working set before attempting to commit. Similarly, the client can be informed [message 6] in parallel with the second phase of 2PC [message 4]. This reduces the number of message delays to four until the client can be informed about the outcome (one-way message 1, round-trip 3, one-way message 5), and to at least six until the transaction becomes visible (one-way message 1, round-trips 3 and 4, one-way message 6). Compared to a centralized DBMS, the six message delays required for 2PC substantially increases the execution time for a transaction.

Unlike the described 2PC protocol, a traditional 2PC protocol [41] does not use a timestamp service but still requires



(a) Traditional SI (b) RSI Protocol

Figure 5: Distributed 2PC Commit Protocols for SI

a total delay of six messages (including client notification). Thus, our analysis is not specific to SI and can be generalized to other 2PC protocols.

4.1.2 Increased Contention Likelihood

The increased transaction latencies due to message delays increase the chance of contention and aborts. As outlined in Section 2, the average latency for small one-way messages over Ethernet is roughly 35μs, whereas the actual work of a transaction ranges from 10-60μs if no disk or network is involved [30, 23].[2] That is, for short-running transactions, the dominant factor for latency is the network, and 2PC amplifies this bottleneck.

In order to model the contention rate effect, we assume an M/M/1 queue X to estimate the number of waiting (i.e., conflicting) transactions for a given record r with some arrival rate λ. With this model, a 6× increase in transaction processing time, referred to as service time t, yields a service capacity decrease of $\mu = 1/6t$ and an increased conflict likelihood of $P(X >= 0) = 1 - P(X = 0) = 1 - (1 - \lambda/\mu) = 6\lambda t$. However, a transaction rarely consists of a single record. With n records, the likelihood of a conflict increases to $1 - \prod_n P(X = 0) = 1 - (1 - 6\lambda t)^n$, if we employ the simplifying assumption that the access rate to all records is similar and independent. Thus, the intuition that the likelihood of conflicts with 2PC increases is true.

However, we did not consider the read-phase, and it is easy to show that the relative difference is less severe as more records are read (it adds a fixed cost to both).In addition, a redesign of the commit protocol to use RDMA verbs can significantly decrease the conflict likelihood, since the latency is much lower for small messages (see Figure 2(b)). Other recent work has shown that most of these conflicts can even be avoided by leveraging the properties of commutative updates [8]. In fact, newer consistency protocols that take advantage of non-blocking commutative updates can provide high availability without centralized coordination [32]. We therefore believe that the argument against distributed transactions on the grounds of increased conflict likelihood is no longer valid.

4.1.3 CPU Overhead

In addition to the higher likelihood of conflicts, distributed transactions also require additional network messages that

[2]For instance, [27] reported 64μs for a single partition transaction on an ancient 2008 Xeon processor.

increase with the number of server nodes. For example, the distributed protocol shown in Figure 5(a) with one TM server and n involved RMs ($n = 2$ in the figure) requires $2 + 4 \cdot n$ send messages and $3 + 4 \cdot n$ receive messages, with a total of $5 + 8 \cdot n$ messages if sends and receives are equally expensive.

Let us assume that a transaction always has to access all n nodes. If each node has c cores able to execute $cycles_c$ per second and a message costs $cycles_m$, then an optimistic upper bound on the number of transactions per second is $trx_u = (c \cdot cycles_c \cdot (n+1))/(5+8 \cdot n) \cdot cycles_m$. On a modern cluster of three nodes, each with 2.2GHz 8-core CPUs, and assuming $3,750$ cycles per message (see Figure 3), this leads to $\approx 647,000$ trx/seconds. More interestingly, though, if we increase the cluster to four nodes with the same hardware configuration, the maximum throughput goes down to 634,000. These back-of-the-envelope calculations suggest that message overhead consumes almost all the added CPU power, making the system inherently unscalable if the workload cannot be partitioned. Without fundamentally redesigning the protocols and data structures, the CPU overhead will remain as the primary bottleneck. For instance, Figure 2 and Figure 3 show that IPoIB on our FDR 4× network increases bandwidth and reduces latency compared to IPoEth but does nothing to reduce CPU overhead.

4.1.4 Discussion

The traditional wisdom that distributed transactions, especially 2PC, do not scale on slow networks is true.First, distributed transactions increase the contention rate. Second, the protocol itself (not considering the message overhead) is rather simple and has no significant impact on the performance, since 2PC simply checks if a message arrived and what it contained. The increased CPU-load and network bandwidth for handling the messages remain the dominant factors. Assuming three servers connected by a 10Gb Ethernet network, an average record size of 1KB, and transactions updating three records on average, at least 3KB have to be read and written per transaction. In this scenario, the total throughput is limited to $\approx 218,500$ transactions per second.

As a result, complicated partitioning schemes have been proposed to avoid distributed transactions as much as possible [17, 56, 62]. However, these approaches impose a new set of challenges for the developer and do not work for some workloads (e.g., social graphs are notoriously difficult to partition).

4.2 RSI: An RDMA-based SI Protocol

Fast high-bandwidth networks such as InfiniBand are able to resolve the two most important limiting factors: CPU overhead and network bandwidth. However, as our experiments show, the scalability is severely limited without changing the techniques themselves. Therefore, we need to redesign distributed DBMSs for RDMA-based architectures.

In this section, we present a novel RDMA-based SI protocol, *RSI*, that is designed for the NAM architecture. We have also implemented the traditional SI protocol discussed before using two-sided RDMA verbs instead of TCP/IP sockets as a simplified shared-memory architecture. Both implementations are included in our experimental evaluation in Section 4.3.

At its core, RSI moves the transaction processing logic to the client (i.e., compute nodes) and makes each of the servers (i.e., storage nodes) "dumb" as their main purpose is to share their memory with the clients. Moreover, clients implement the transaction processing logic through one-sided RDMA operations (i.e., the client is the transaction manager) allowing any compute node to act as a client that can access data on any storage node (i.e., a server). This design is similar to [13], but optimized for direct memory access rather than cloud services. Moving the logic to the client has several advantages. Most importantly, scale-out becomes much easier since all CPU-intensive operations are done by the clients, which are easy to add. The throughput of the system is only limited by the number of RDMA requests that the server's RNICs (and InfiniBand switches) can handle. Since several RNICs can be added to one machine, the architecture is highly scalable (see also Section 4.3). In addition, (1) load-balancing is easier since transactions can be executed on any node independent of any data-locality, and (2) latencies are reduced as clients can fetch data directly from the servers without involving the TM.

Look 1 Bit	CID_N 63 Bits	$Record_N$ m Bits	CID_{N-1} 64 Bits	$Record_{N-1}$ m Bits	...
0	20003	("A1","B1")			
0	23401	("C1","D2")	22112	("C1","D1")	
1	24401	("E2","F2")	22112	("E1","F1")	

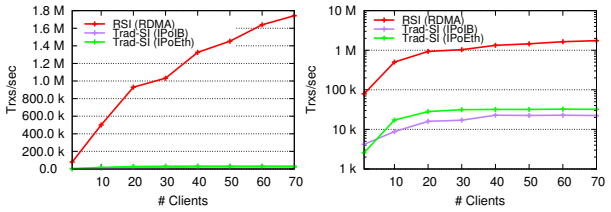
Table 1: Potential Data Structure for RSI

As before, we assume that reads already have happened and that the transaction has an assigned read timestamp, RID. First, the client (acting as the TM) contacts the timestamp service to receive a new commit timestamp CID. In our implementation, we pre-assign timestamps to clients using a bit vector with 60k bits. The first bit in the vector belongs to client 1 and represents timestamp 1, up to client n representing timestamp n. Afterwards, position $n + 1$ again belongs to client 1 and so on. Whenever a timestamp is used by a client, it "switches" the bit from 0 to 1. With this scheme, the highest committed timestamp can be determined by finding the highest consecutive bit in the vector. If all bits are set by a client, we allow clients to "wrap" and start from the beginning. Note, that wrapping requires some additional bookkeeping to avoid that bits are overwritten.

This simple scheme allows clients to use timestamps with no synchronization bottleneck but implicitly assumes that all clients make progress at roughly the same rate. If this assumption does not hold (e.g., because of stragglers or long running transactions), additional techniques are required to skip bits, which go beyond the scope of this paper.

Next, the client has to execute the first phase of 2PC and check if the version has not changed since it was read (i.e., validation phase of 2PC). As before, this operation requires a lock on the record to prevent other transactions from changing the value after the validation and before the transaction is fully committed.

In a traditional design, the server would be responsible of locking and validating the version. In order to make this operation more efficient and "CPU-less", we propose a new storage layout to allow direct validation and locking with a single RDMA-operation shown in Table 1. The key idea is to store up to n versions of a fixed-size record of m-bits length in a fixed-size *slotted* memory record, called a **record block**, and have a global dictionary (e.g., using a DHT) to exactly determine the memory location of any record within the cluster. We will explain the global dictionary and how we handle inserts in the next subsections and assume for



(a) Linear-Scale (b) Log-Scale

Figure 6: RSI vs 2PC (Throughput)

the moment, that after the read phase all memory locations are already known. How many slots (i.e., versions) a record block should hold depends on the update and read patterns as it can heavily influence the performance. For the moment, assume that every record has $n = max(16KB \ / \ record\text{-}size, 2)$ slots for different record versions and that every read retrieves all n slots. From Figure 2(b) we know that transferring 1KB to roughly 16KB makes no difference in the latency therefore making n any smaller has essentially no benefit. Still, for simplicity, our current implementation uses $n = 1$ and aborts all transactions which require an older snapshot.

The structure of a slot in memory is organized as follows: the first bit is used as a lock (0=no-lock, 1=locked) while the next 63 bits contain the latest commit-id (CID) of the most recent committed record, followed by the payload of the record, followed by the second latest CID and payload and so on, up to n records. Using this data structure, the TM (i.e., the client) is directly able to validate and lock a record for a write using a *compare-and-swap* operation on the first 64 bits [round-trip message 2]. For example, assume that the client has used the RID 20003 to read the record at memory address 1F (e.g., the first row in Table 1) and wants to install a new version with CID 30000. A simple RDMA *compare-and-swap* operation on the first 64 Bits of the record at address $1F$ with test-value 20003, setting it to $1 << 63|20003$), would only acquire the lock if the record has not changed since it was read by the transaction, and fails otherwise. Thus, the operation validates and prepares the resource for the new update in a single round-trip. The TM uses the same technique to prepare all involved records (with SI inserts always succeeding).

If the *compare-and-swap* succeeds for all intended updates of the transaction, the transaction is guaranteed to be successful and the TM can install a new version. The TM therefore checks if the record block has a free slot, and, if yes, inserts its new version at the head of the block and shifts the other versions to the left. Afterwards, the TM writes the entire record block with a signaled WRITE to the memory location of the server [message 3].

Finally, when all the writes have been successful, the TM informs the timestamp service about the outcome [message 3] as in the traditional protocol. This message can be sent unsignaled. Overall, our RDMA-enable SI protocol and storage layout requires 3 round-trip messages and one unsignaled message, and does not involve the CPU in the normal operational case. As our experiments in the next section will show, this design enables new dimensions of scalability.

4.3 Experimental Evaluation

To evaluate these algorithms, we implemented the traditional SI protocol (Figure 5(a)) on the shared-nothing architecture with IPoETH (Figure 4(a)) and IPoIB (Figure 4(b)). We also implemented a simplified variant of the shared-memory architecture (Figure 4(c)) by replacing TCP/IP sockets with two-sided RDMA verbs (requiring significant modifications to memory management). We slightly adjusted the traditional SI implementation by using a local timestamp server instead of a remote service, thereby giving the traditional implementation an advantage. Finally, our RSI protocol utilizes the NAM architecture (Figure 4(d)) and used an external timestamp service.

We evaluated all protocols on the cluster described in Section 2.2. We use four machines to execute the clients, three as the NAM storage-servers, and one as the TM in the traditional case or the timestamp server in the RSI case. We measured both protocols with a simple and extremely write-heavy workload, similar to the checkout transaction of the TPC-W benchmark. Every transaction reads three products, creates one order and three orderline records, and updates the corresponding product stocks. Our dataset consisted of one million products, where each record is roughly 1KB, and all data was evenly distributed across the nodes. Clients waited until a transaction was completed before issuing the next transaction.

Figure 6 shows the scalability of the traditional SI-protocol and our new RSI protocol with a variable number of client threads. The traditional SI-protocol over IPoIB has the worst scalability, with $\approx 22,000$ transactions per second, whereas IPoEth achieves $\approx 32,000$ transactions per second. The IPoIB implementation performs worse because of the less efficient TCP/IP implementation for IPoIB, which plays an important role for small messages. In contrast, our RSI protocol achieved ≈ 1.8 million *distributed* transactions per second. The shared-memory architecture using two-sided RDMA verbs (not shown) achieved a throughput of 1.1 million transactions per second, or only 66% of our RSI protocol. However, we also noticed that the two-sided RDMA verb implementation not only stops scaling after 40 clients, but that the throughput also decreases to only $\approx 320,000$ transaction per second with 70 clients, while our RSI implementation scales almost linearly up to 60 clients. One reason for the decrease in performance is that the TMs become a major bottleneck. However, our RSI implementation no longer scaled linearly after 60 clients, since we only had one dual-port FDR 4× RNIC per machine, with a bandwidth of 13.8GB/s. With the three 1KB records per transactions, we can achieve a theoretical maximum throughput of $\approx 2.4M$ transactions per second (every transaction reads/writes at least 3KB). For greater than 60 clients, the network is saturated.

We therefore speculate that distributed transactions no longer have to be a scalability limit when the network bandwidth matches the memory bandwidth. Furthermore, complex partitioning schemes might become obsolete in many scenarios, although they can still reduce latency and help to manage frequently accessed items.

5. THE CASE FOR OLAP

In order to motivate the redesign of distributed DBMSs for OLAP workloads, we first discuss why existing distributed algorithms, which were designed for a shared-nothing architecture over slow networks, are not optimal for high-performance networks the RDMA capabilities. Then, we present novel RDMA-optimized operators for the NAM ar-

chitecture, which require fundamental redesigns of core components (e.g., memory management, query optimization), as discussed in Section 3.2. This paper focuses on distributed joins and aggregates, which are the predominant operators in most OLAP workload.

5.1 Existing Distributed OLAP Operators

The most network-intensive OLAP operation is the distributed join [53]. Most distributed join algorithms have three components that can be combined in different ways: (1) a local join algorithm, (2) a partitioning scheme, and (3) an optional reduction technique. For example, either a hash or sort-merge join could be used as the local join algorithm, whereas partitioning schemes range from static to dynamic hash partitioning [18]. Also, several techniques to reduce the partitioning cost have been proposed, the most prominent being the semi-join reduction using a Bloom filter [51].

The following section explains the most common partitioning technique for distributed join algorithms over shared-nothing architectures: the grace hash join (GHJ). Later, we expand the distributed join algorithm with an additional semi-join reduction using Bloom filters to further reduce communication. For both, we develop a simple cost model and argue why these algorithms are, in most cases, no longer optimal for distributed DBMSs over RDMA-capable networks. Throughout the rest of this section, we assume that no skew exists in the data (i.e., all nodes hold roughly the same amount of data before and after partitioning).

5.1.1 An Optimized Grace Hash Join

The GHJ executes a distributed join in two phases. In the first phase (*partitioning phase*), the GHJ scans the input relations and hash-partitions them on their join key such that the resulting sub-relations can be joined in the second phase locally per node (*local join phase*). The cost of the GHJ T_{GHJ} is therefore given by the sum of the runtime of the partitioning phase T_{part} and the local join phase T_{join}.

We do not consider any static pre-partitioning, so the cost for repartitioning can be split into the cost of partitioning the two join relations R and S. The cost of repartitioning R can now further be split into the cost of (1) reading the data on the sender, (2) transferring the data over the network, and (3) materializing the data on the receiver. Assuming that the cost of sending R over the network is $T_{net}(R) = w_r \cdot |R| \cdot c_{net}$ and scanning R in-memory is $T_{mem}(R) = w_r \cdot |R| \cdot c_{mem}$, with $|R|$ being the number of tuples, w_R being the width of a tuple $r \in R$ in bytes, and c_{net} (c_{mem}) the cost of accessing a byte over the network (memory), the repartitioning cost of R can be expressed as:

$$\begin{aligned} T_{part}(R) = & \underbrace{T_{mem}(R)}_{\text{Reading (sender)}} + \underbrace{T_{net}(R)}_{\text{Shuffling (net)}} + \underbrace{T_{mem}(R)}_{\text{Writing (receiver)}} \\ & = w_r \cdot |R| \cdot c_{mem} + w_r \cdot |R| \cdot c_{net} + w_r \cdot |R| \cdot c_{mem} \\ & = 2 \cdot w_r (\cdot c_{mem} \cdot |R| + c_{net} \cdot |S|) \end{aligned}$$

The partition cost for S is similar. Note that we disregard CPU costs because we assume that the limiting factors are memory and network accesses, which is reasonable for a simple hash-based partitioning scheme.

For the local join algorithm of the GHJ, we use the fastest local in-memory join algorithm, the parallel radix join [9], which includes two phases. In the first phase, the algorithm scans each input relation, partitioning them locally into cache-sized blocks using multiple passes over the data. As shown in [9], most relations can be efficiently partitioned in a single pass with software managed buffers. After partitioning the data, the algorithm then scans the relations again to join the cache-sized blocks. Existing work [46, 9] has shown that both phases of the radix join are bound by the memory bandwidth. Thus, we can estimate the total cost for the local radix join as:

$$\begin{aligned} T_{join}(R,S) = & \underbrace{(T_{mem}(R) + T_{mem}(S))}_{\text{Radix Phase 1}} + \underbrace{(T_{mem}(R) + T_{mem}(S))}_{\text{Radix Phase 2}} \\ & = 2 \cdot c_{mem} \cdot (w_r \cdot |R| + w_s \cdot |S|) \end{aligned}$$

The total runtime of the GHJ T_{GHJ} is therefore:

$$\begin{aligned} T_{GHJ} &= T_{part}(R) + T_{part}(S) + T_{join}(R,S) \\ &= (w_r|R| + w_s|S|) \cdot (4 \cdot c_{mem} + c_{net}) \end{aligned}$$

5.1.2 Adding Semi-Reduction using Bloom Filters

As shown in the final cost equation from the previous section, the GHJ requires roughly 4× more memory accesses than network transfers. However, in distributed DBMSs, the network cost typically comprises up to 90% of the total runtime of a join [53]. Thus, state-of-the-art join algorithms (e.g., track join [47], Neo-Join [53]) attempt to reduce the amount of data sent over the network through expensive computations (e.g., Neo-Join uses a linear solver) or multiple communication round-trips to perform complex data partitioning.

Here, we focus on the most traditional approach: a semi-join reduction using a Bloom filter. The core idea of the semi-join reduction is to send only tuples in the input relations R and S that have a join partner in the other relation. Therefore, the algorithm first creates Bloom filters b_R and b_S over the join keys of R and S, respectively. Then, b_R and b_S are copied across all nodes that hold a partition of S and R, and each node uses its Bloom filter to remove tuples that are guaranteed to have no join partner (i.e., if the Bloom filter matches a join key, it must be sent).

The cost of creating b_R includes both a scan over the data $T_{mem}(R)$ and transmission over the network $T_{net}(b_R)$:

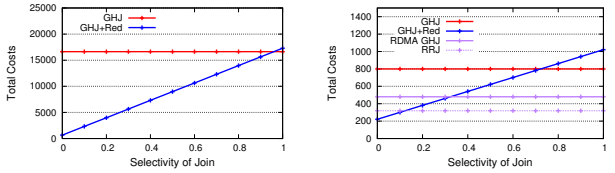
$$T_{bloom}(R) = \underbrace{T_{mem}(R)}_{\text{Create Reducer}} + \underbrace{T_{net}(b_R)}_{\text{Ship Reducer}}$$

However, the size of the Bloom filter b_r is normally very small, so that $T_{bloom}(R)$ can be disregarded. Assuming that $sel_S(b_R)$ is the selectivity of the Bloom filter b_R over relation S (including the error rate of the Bloom filter), the total cost for a GHJ with a semi-join reduction using Bloom filters is:

$$\begin{aligned} T_{ghj+bloom} = & \underbrace{T_{bloom}(R) + T_{bloom}(S)}_{\text{Create Bloom-Filter}} + \\ & \underbrace{T_{part}(sel_R(b_S) \cdot R) + T_{part}(sel_S(b_R) \cdot S)}_{\text{Reduced Partitioning Cost}} + \\ & \underbrace{T_{join}(sel_R(b_S) \cdot R, sel_R(b_R) \cdot S)}_{\text{Reduced Join Cost}} \end{aligned}$$

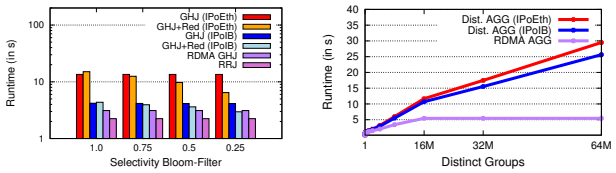
This equation models the cost of creating the Bloom filter plus the reduced partitioning and join costs. Assuming that the selectivity between both relations is the same, $sel = sel_R(b_S) = sel_S(b_R)$ leads to this simplified total cost:

$$\begin{aligned} T_{join+bloom} = & (w_r|R| + w_s|S|) \cdot \\ & (c_{mem} + 4 \cdot sel \cdot c_{mem} + sel \cdot c_{net}) \end{aligned}$$



(a) IPoEth (b) IPoIB and RDMA

Figure 7: Join Cost Analysis



(a) Join (b) Aggregation

Figure 8: Traditional vs RDMA-optimized

5.1.3 *Discussion*

Figure 7 plots all the previously mentioned costs of traditional distributed joins for various join selectivities. For the network cost c_{net} per byte, we used the idealized latency per byte from Section 2 for messages of size 2KB. For the Bloom filters, we assume a 10% error of false positives (i.e., 50% selectivity still selects 60% of the data). We use $|R| = |S| = 1M$ as table sizes and $w_r = w_s = 8$ as tuple width. For memory, we assume a cost of $c_{mem} = 10^{-9}$s for accessing a single byte. However, the relative relationships of the different constants c_{cpu}, c_{mem}, and c_{net} are more important than the absolute cost of accessing a single byte from memory.

For an IPoEth network, the results demonstrate that a semi-join reduction (GHJ+Red) almost always pays off (Figure 7(a)). However, the tradeoffs change and thus, the optimization, for existing distributed join algorithms (Figure 7(b)).For example, the network cost is no longer the dominant factor.Only if the Bloom filter selectivity is below $sel < 0.8$ (0.7 in Figure 7(b) due to the 10% Bloom filter error rate), a semi-join reduction pays off due to reduction in join and shipping cost. Yet, both GHJ and GHJ+Red for IPoIB still do not take full advantage of the network capabilities. In the next section, we outline a new join algorithm that directly fully leverages InfiniBand using RDMA.

We now describe two new join algorithms that leverage the RDMA-based NAM architecture presented in Section 3. First, we redesign the GHJ to use one-sided RDMA verbs to write directly into remote memory of storage nodes for partitioning. We call this join the RDMA GHJ. The main goal of the partitioning phase of the RDMA GHJ for the NAM architecture is to enable data parallel execution of the join phase by the compute nodes.

The input tables for the partitioning phase are pre-fetched from the storage nodes to the compute nodes. Moreover, for writing the output partitions back to the storage nodes, the RDMA GHJ leverages selective signaling to overlap computation and communication. Thus, only the CPU of the sender is active during the partitioning phase, and the cost of partitioning reduces to $T_{part} = T_{mem}(R) + T_{mem}(S)$ because the remote data transfer for writing is executed in the background by the RNICs when using selective signaling. Finally, the join phase also uses pre-fetching of the partitioned tables. This leads to reduced overall join costs which renders a semi-join reduction even less beneficial when compared to the classical GHJ as shown in Figure 7(b).

While this optimization may sound trivial, however, it requires a significant redesign of the join algorithm's buffer management to work efficiently on the NAM architecture. Each server needs to reserve a buffer for every output partition on the storage servers to ensure that data is not overwritten during the shuffling phase. Moreover, the partitioning phase must be designed such that the compute nodes which execute the partitioning phase can be scaled-out independently from the storage nodes. Describing these techniques in more detail goes beyond the scope of this paper.

However, we can go a step further than just optimizing the partitioning phase of the GHJ to leverage RDMA. The previously described partitioning phase of the radix join used to optimize block sizes for cache-locality is very similar to the partitioning phase of the GHJ. Therefore, instead of trying to adjust distributed join algorithms like GHJ, we propose extending the in-memory radix join [9] to leverage RDMA directly. We refer to this new algorithm as *RRJ* (RDMA Radix Join). A similar algorithm was recently presented in [11]. However, unlike our algorithm, their join has been optimized for a shared-nothing architecture while our RRJ algorithm is optimized for the NAM architecture, enabling an efficient scale-out by adding additional compute servers.

5.2 RDMA Join Algorithms

Our new RRJ algorithm uses remote software managed buffers for the partition phase. Software managed buffers for the single-node radix join are presented in [9] to achieve a high fan-out of the radix-partitioning phase and avoid multiple passes. RRJ adopts this idea to work optimally in the NAM architecture with RDMA by applying the following changes: (1) buffers are copied in the background to storage nodes using selective signaled WRITEs; and (2) buffer sizes are optimized to leverage the full bandwidth of RDMA. Our micro-benchmarks in Section 2.2 show that $2KB$ messages saturate the InfiniBand bandwidth. Moreover, the fan-out of the remote radix-partitioning phase is selected such that all buffers fit into the L3 cache of the CPU.

Note that the resulting RRJ algorithm is not simply a straightforward extension of the radix join. For example, our current implementation uses manually allocated RDMA-enabled memory on the buffer and storage nodes.In a redesigned distributed DBMS, a major challenge is to manage global memory allocation efficiently without imposing a performance penalty on the critical path of distributed algorithms.

Assuming that the network cost is similar to the memory cost and that one partitioning pass is sufficient when using software managed buffers, the RRJ algorithm has a total expected cost of:

$$T_{RRJ} = 2 \cdot c_{mem} \cdot (w_r \cdot |R| + w_s \cdot |S|)$$

The results of the cost analysis of both algorithms, the RDMA GHJ and the RRJ, is shown in Figure 7(b) and demonstrates that the popular semi-join reduction for distributed joins only pays off in corner cases (i.e., for very, very low join selectivities).

5.3 RDMA Aggregation Algorithms

Since the primary concern for distributed aggregation in a shared-nothing architecture over slow networks is to avoid

network communication [43], traditional approaches typically use a hierarchical scheme. In a first phase, all nodes individually execute an aggregation over their local data partition. In a second phase, the intermediate aggregates are merged using a global union, and a post-aggregation is executed over that union. However, this scheme suffers from two problems: (1) data-skew can cause individual nodes in the first phase to take much longer to finish than other nodes; and (2) a large number of distinct group-by keys leads to a high execution cost for second phase.

In order to tackle these issues, we present a novel RDMA-optimized aggregation operator, which implements a distributed version of a modern in-memory aggregation operator [50, 34] for our NAM architecture. In a first phase, this operator uses cache-sized hash tables to pre-aggregate data that is local to a core (thread). Moreover, if the hash tables are full it flushes them to overflow partitions. In our RDMA-variant of this operator we directly copy the data in the background to remote partitions while the pre-aggregation is still active. In a second phase, individual partitions are then post-aggregated in parallel to compute the final aggregate. Since this operator uses fine-grained parallelism in the first phase and there are more partitions than worker threads in the second phase, it is more robust towards data-skew and a varying number of distinct group-by keys.

5.4 Experimental Evaluation

We implemented all the discussed distributed join and aggregation variants and executed them using four servers (10 threads per node). Each node in hosted compute and a storage node using the previously described configuration.

For the join workload, we used a variant of [9] adopted for the distributed setting: for each node we generated a partition that has the size $|R| = |S| = 128\ Million$ and a tuple width $w_r = w_s = 8B$. We generated different datasets such that the selectivity of the Bloom filter covers 0.25, 0.5, 0.75, and 1.0 to show the effect of reduced network costs.

Figure 8(a) shows the total runtime of the GHJ and GHJ+Red over Ethernet (IPoEth) and IP over InfiniBand (IPoIB) as well as our two RDMA variants, RDMA GHJ and RRJ, over InfiniBand (RDMA) when using 8 threads per node. As shown, the new RRJ algorithm significantly outperforms the other state-of-the-art join algorithms for different semi-join selectivities. These results are in line with our cost analysis, though the results vary slightly as caching and CPU effects play a more crucial role for the RDMA variants.

In a second experiment, we analyze the performance of our RDMA Aggregation (RDMA AGG) and compare it to a classical hierarchical distributed aggregation (Dist. AGG). For the classical aggregation, we used the algorithm as described in [50, 34] as local aggregation operations. For the workload, we used one table with the size $|R| = 128\ Million$ per partition. Each tuple of R has two attributes (one group-by key and one aggregation attribute) of 4B each resulting in a tuple width of $w_r = 8B$. Moreover, we generated data sets with a different number of distinct values for the group-by keys ranging from 1 to $64M$ using a uniform distribution.

Figure 8(b) shows the results. For the traditional hierarchical aggregation (Dist. AGG), the runtime increases with the number of distinct group-by keys due to the cost of the global union and post-aggregation (i.e., the post-aggregation has to be executed over a global union that produces an output with a size of $\#nodes \cdot \#groupkeys$). While showing a similar performance for a small number of distinct group-by keys (i.e., 0.17ms), our RDMA Aggregation (RDMA AGG) is more robust for a larger number of distinct group-by keys and shows major performance gains in that case.

Our experiments shown in Figure 8(a) and Figure 8(b) demonstrate that a redesign of distributed DBMS operators for the NAM architecture provides major benefits not only in terms of performance but also other aspects (e.g., robustness). Unlike traditional distributed operators for the shared-nothing and shared-memory architecture, our operators are optimized for the NAM architecture, thus enabling an efficient scale-out by adding additional compute nodes. Moreover, the NAM architecture also enables more efficient schemes to handle data-skew using fine-grained parallelism and work-stealing algorithms.

6. RELATED WORK

A major focus in the HPC community has been the development of techniques that take advantage of modern hardware, particularly network technologies like InfiniBand [39, 26, 28]. While the vast majority of this work is limited to specific applications, the results and gained experiences are highly relevant for developing general-purpose DBMSs for high-performance networks.

In this paper, we made the case that networks with RDMA capabilities should directly influence the architecture and algorithms of distributed DBMSs. Many projects in both academia and industry have attempted to add RDMA as an afterthought to an existing DBMS [54, 3]. For example, Oracle RAC [3] has RDMA support, including the use of RDMA atomic primitives. However, RAC does not directly take advantage of the network for transaction processing and is essentially a workaround for a legacy system.

Some recent work has investigated building RDMA-aware DBMSs [60, 59] on top of RDMA-enabled key/value stores [29, 42], but transactions and query processing are an afterthought instead of first-class design considerations. Other systems that separate storage from compute nodes [13, 35, 38, 14, 2] also treat RDMA as an afterthought. IBM pureScale [10], for instance, uses RDMA to provide active-active scaleout for DB2 but relies on a centralized manager to coordinate distributed transactions. Conversely, our NAM architecture natively leverages RDMA primitives to build a shared distributed memory pool without a centralized coordinator.

The proposed ideas for RDMA build upon the huge amount of work on distributed transaction protocols (e.g., [61, 22, 12, 37]) and join processing (see [31] for an overview). Other work [11, 55] also presents distributed join algorithms for RDMA but focuses only on the redesign of the traditional shared-nothing architecture using two-sided verbs. SpinningJoins [25] also make use of RDMA, but this work assumes severely limited network bandwidth (only 1.25GB/s) and therefore streams one relation across all the nodes (similar to a block-nested loop join).

Most related to our transaction protocol is FaRM [20, 19]. However, FaRM uses a more traditional message-based approach and focuses on serializability, whereas we implemented snapshot isolation, which is more common in practice because of its low-overhead consistent reads. More importantly, in this work we made the case that distributed transactions can now scale, whereas FaRM tries to explicitly leverage locality as much as possible.

7. CONCLUSION

We argued that emerging high-performance network technologies necessitate a fundamental rethinking of the way we build distributed DBMSs. Our experiments for OLTP and OLAP workloads indicate the potential of fully leveraging the network. This opens up a wide research area with many interesting research challenges, such as the trade-off between local and remote processing or creating simple abstractions to hide the complexity of RDMA verbs.

8. ACKNOWLEDGMENTS

This research is funded in part by the Intel Science and Technology Center for Big Data, the NSF CAREER Award IIS-1453171, the Air Force YIP AWARD FA9550-15-1-0144, NSF IIS-1514491, and gifts from SAP, Oracle, Google, Mellanox, and Amazon.

9. REFERENCES

[1] www.jedec.org/standards-documents/docs/jesd-79-3d.
[2] http://snowflake.net/product/architecture.
[3] Delivering Application Performance with Oracles InfiniBand Tech. http://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf, 2012.
[4] Shared Memory Communications over RDMA. http://ibm.com/software/network/commserver/SMCR/, 2013.
[5] Intel Data Direct I/O Technology. http://www.intel.com/content/www/us/en/io/direct-data-i-o.html, 2014.
[6] I. T. Association. InfiniBand Roadmap. http://www.infinibandta.org/, 2013.
[7] S. Babu et al. Massively parallel databases and mapreduce systems. *Foundations and Trends in Databases*, 2013.
[8] P. Bailis et al. Eventual consistency today: limitations, extensions, and beyond. *Comm. of ACM*, 2013.
[9] C. Balkesen et al. Multi-core, main-memory joins: Sort vs. hash revisited. In *VLDB*, 2013.
[10] V. Barshai et al. *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*. IBM Redbooks, 2012.
[11] C. Barthels et al. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, 2015.
[12] C. Binnig et al. Distributed snapshot isolation: Global transactions pay globally, local transactions pay locally. *VLDB Journal*, 2014.
[13] M. Brantner et al. Building a database on S3. In *SIGMOD*, 2008.
[14] D. G. Campbell et al. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.
[15] J. C. Corbett et al. Spanner: Googles globally distributed database. *ACM TOCS*, 2013.
[16] A. Crotty et al. An Architecture for Compiling UDF-centric Workflows. In *VLDB*, 2015.
[17] C. Curino et al. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *VLDB*, 2010.
[18] D. J. DeWitt et al. The Gamma Database Machine Project. *IEEE Trans. Knowl. Data Eng.*, 1990.
[19] A. Dragojevic et al. FaRM: Fast Remote Memory. In *NSDI*, 2014.
[20] A. Dragojevic et al. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.
[21] A. J. Elmore et al. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *SIGMOD*, 2015.
[22] S. Elnikety et al. Database replication using generalized snapshot isolation. In *SRDS*, 2005.
[23] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, 2012.
[24] M. Feldman. RoCE: An Ethernet-InfiniBand Love Story. *HPC wire*, 2010.
[25] P. Frey et al. A spinning join that does not get dizzy. In *ICDCS*, 2010.
[26] N. S. Islam et al. High performance RDMA-based design of HDFS over InfiniBand. In *SC*, 2012.
[27] E. P. C. Jones et al. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.
[28] J. Jose et al. Memcached design on high performance RDMA capable interconnects. In *ICPP*, 2011.
[29] A. Kalia et al. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.
[30] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.
[31] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.
[32] T. Kraska et al. MDCC: multi-data center consistency. In *EuroSys*, 2013.
[33] J. Lee et al. SAP HANA distributed in-memory database system: Transaction, session and metadata management. In *ICDE*, 2013.
[34] V. Leis et al. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.
[35] J. J. Levandoski et al. High Performance Transactions in Deuteronomy. In *CIDR*, 2015.
[36] Y. Lin et al. Middleware based data replication providing snapshot isolation. In *SIGMOD*, 2005.
[37] Y. Lin et al. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 2009.
[38] S. Loesing et al. On the Design and Scalability of Distributed Shared-Data Databases. In *SIGMOD*, 2015.
[39] X. Lu et al. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.
[40] P. MacArthur et al. A performance study to guide RDMA programming decisions. In *HPCC*, 2012.
[41] C. Mohan et al. Transaction Management in the R* Distributed Database Management System. In *TODS*, 1986.
[42] J. K. Ousterhout et al. The case for ramcloud. *Commun. ACM*, 2011.
[43] M. T. Ozsu. *Principles of Distributed Database Systems*. Prentice Hall Press, 3rd edition, 2007.
[44] A. Pavlo. *On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems*. PhD thesis, Brown University, 2014.
[45] A. Pavlo et al. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.
[46] O. Polychroniou et al. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, 2014.
[47] O. Polychroniou et al. Track join: distributed joins with minimal network traffic. In *SIGMOD*, 2014.
[48] A. Pruscino. Oracle RAC: Architecture and performance. In *SIGMOD*, 2003.
[49] A. Quamar et al. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, 2013.
[50] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. In *VLDB*, 2013.
[51] S. Ramesh et al. Optimizing Distributed Joins with Bloom Filters. In *ICDCIT*, 2008.
[52] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *VLDB*, 2012.
[53] W. Rödiger et al. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, 2014.
[54] W. Rödiger et al. High-speed query processing over high-speed networks. In *VLDB*, 2015.
[55] W. Rödiger et al. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, 2016.
[56] M. Stonebraker et al. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.
[57] H. Subramoni et al. RDMA over Ethernet - A preliminary study. In *CLUSTER*, 2009.
[58] A. Thomson et al. The case for determinism in database systems. In *VLDB*, 2010.
[59] C. Tinnefeld et al. Elastic online analytical processing on RAMCloud. In *EDBT*, 2013.
[60] C. Tinnefeld et al. Parallel join executions in RAMCloud. In *Workshops ICDE*, 2014.
[61] S. Tu et al. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.
[62] E. Zamanian et al. Locality-aware partitioning in parallel database systems. In *SIGMOD*, 2015.