Bulletin of the Technical Committee on

Data Engineering

March 2017 Vol. 40 No. 1

IEEE Computer Society

Letters

Letter from the Editor-in-Chief . . . . . *David Lomet* 1
Letter from the Special Issue Editor . . . . . *Tim Kraska* 2

Special Issue on Distributed Data Management with RDMA

RDMA Reads: To Use or Not to Use? . . . . . *Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro* 3
Designing Databases for Future High-Performance Networks . . *Claude Barthels, Gustavo Alonso, Torsten Hoefler* 15
Rethinking Distributed Query Execution on High-Speed Networks . . . . . *Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, Tobias Ziegler* 28
Crail: A High-Performance I/O Architecture for Distributed Data Processing . . . . . *Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, Ioannis Koltsidas* 40
Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached . . . . . *Xiaoyi Lu, Dipti Shankar, Dhabaleswar K. (DK) Panda* 53
Beyond Simple Request Processing with RAMCloud . . . . . *Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, Ryan Stutsman* 66

Conference and Journal Notices

ICDE 2017 Conference 74
TCDE Membership Form . . . . . back cover

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

TCDE Chair Election

The Technical Committee on Data Engineering held an election last fall for chair of the TC. The voting deadline was December 22 of last year. The candidates were Xiaofang Zhou and Erich Neuhold. My thanks both candidates for being willing to run. Being chair of the Technical Committee is largely invisible, but it is an important responsibility for the success of the data engineering community.

The winner, with 69% of the vote is the current chair, Xiaofang Zhou, who has now won his second term. Congratulations to Xiaofang for his electoral victory. Xiaofang knows what the job entails, is experienced in doing it, and does it well. I very much appreciate Xiaofang's efforts and his continued involvement, both at the TCDE and at the Computer Society more widely.

Computer Society News

As I reported in the December issue, I successfully ran for Computer Society First Vice President. In addition, I have been appointed Treasurer of the Computer Society by CS President Jean-Luc Gaudiot. As Treasurer, I have both an opportunity and a responsibility to look carefully at the Computer Society finances. In brief, the Computer Society faces financial difficulties. It has been running a deficit for several years, and that has now caused our fund balance to go negative. The Society has been working hard to correct this situation, but with a negative fund balance, now needs to redouble its efforts. "May you [all of us] live in interesting times."

The Current Issue

My own research aims to achieve great data management performance on modern hardware platforms. These platforms differ substantially from "classical" hardware in memory hierarchy, multicore, and solid state disks. And the hardware platform evolution continues. This is important as the ongoing hardware raw speed increases we have known in the past have slowed substantially. It is this context that makes the topic of the current issue of particular interest.

One of the more recent hardware features to emerge is remote direct memory access (RDMA). RDMA is a networking technology that permits one computer to read and write directly into the memory of another computer, enabling high-throughput, low-latency networking. This provides the opportunity for software architectures that take a different view toward distributed data centric systems, re-architecting them in ways that were previously unimaginable.

Tim Kraska, the current issue editor, has seized the opportunity resulting from the emergence of RDMA. The current issue showcases some of the advanced work being done in data management that exploits RDMA. This should be higly relevant to both the research and industrial communities as we all struggle to improve data management throughput, latency, scalability, and availability. I want to thank Tim for organizing this very timely and important issue, which I personally intend to study very carefully to come to grips, in depth, with RDMA technology and its potential.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

High-performance Remote Direct Memory Access (RDMA) capable networks such as Infiniband FDR/EDR are fundamentally changing the design of distributed data centric systems. Until recently, such systems were built on the assumption that the network is the main bottleneck. Thus, the systems aim to reduce communication between nodes using techniques such as locality-aware partitioning schemes, semi-joins, and complicated preprocessing steps. Yet, with next generation technologies, the network is no longer the dominant bottleneck.

Even today, bandwidth from InfiniBand FDR 4× is roughly that of one memory channel. DDR3 memory bandwidth ranges from 6.25GB/s (DDR3-800) to 16.6GB/s (DDR3-2133) per channel, whereas InfiniBand has a bandwidth of 1.7GB/s (FDR 1×) to 37.5GB/s (EDR 12×) per NIC port (Figure 1). Future InfiniBand standards (e.g., HDR, NDR) promise to dramatically exceed memory channel bandwidth. Furthermore, while modern CPUs often take advantage of several memory channels (usually 4 per socket), dual-port NICs are becoming the standard and in contrast to memory channels, these NICs support full duplex rather than half-duplex. Thus, today a single dual-port InfiniBand EDR 4× NIC roughly matches the bandwidth of the standard 4 memory channels of a single CPU socket with a read/write workload.

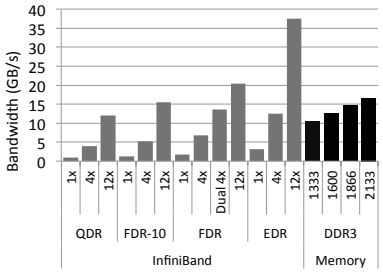


Figure 1: Memory vs Network Bandwidth

RDMA advances have also improved network latency significantly. Our InfiniBand FDR 4x experiments show that it takes only ≈ 1 micro second to transfer 1KB of data using RDMA, compared ≈ 0.08 micro seconds required by the CPU to read the same amount of data from main memory (RAM). With only 256KB, there is virtually no difference between the access times since the bandwidth starts to dominate the transfer time. Nonetheless, one cannot assume that a high-performance network changes a cluster into a NUMA architecture because: (1) RDMA-based memory access patterns differ from local memory access patterns; (2) random access latency for remote requests remains significantly higher; and (3) hardware-embedded coherence that ensures data consistency in NUMA does not exist for RDMA. How RDMA and next generation networks influence data management systems is an open question and the topic of this special issue.

The first paper, **RDMA Reads: To Use or Not to Use?** by Dragojević et al discusses a fundamental question of using the next generation networks: is RDMA or RPC the right building block for remote reads. RDMA reads perform better than RPCs in many cases, but RPCs can perform additional operations before replying to the client. **Designing Databases for Future High-Performance Networks** by Barthels et al makes a case that the recent advances in network technologies enable a re-evaluation and re-design of several system design concepts and database algorithms. Furthermore, the authors argue that databases and networks should be co-designed in the future and require an extended network instruction set architecture (NISA) to better support distributed database designs. **Rethinking Distributed Query Execution on High-Speed Networks** by Salama et al reconsiders traditional query execution models for next generation network technology and presents an end-to-end solution based on the network-attached-memory (NAM) architecture. **Crail: A High-Performance I/O Architecture for Distributed Data Processing** by Stuedi et al presents an I/O architecture redesigned for highspeed networks and RDMA, especially targeting temporary data that requires great performance for operations such as data shuffling or broadcasting. Their solutions allow, for example, the Apache data processing ecosystem to fully exploit new network technology. The last two papers, **Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached** by Lu et al and **Beyond Simple Request Processing with RAMCloud** by Kulkarni et al are concerned with leveraging RDMA for key/value stores with very impressive latency and performance results.

I would like to thank the authors for their insightful contributions to this special issue. Happy reading!

Tim Kraska
Brown University

RDMA Reads: To Use or Not to Use?

Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro
Microsoft Research

Abstract

Fast networks with RDMA are becoming common in data centers and, together with large amounts of main memory in today's machines, they allow unprecedented performance for traditional data center applications such as key-value stores, graph stores, and relational databases. Several systems that use RDMA have appeared recently in the literature, all showing orders of magnitude improvements in latency and throughput compared to TCP. These systems differ in several aspects. Of particular interest is their choice to use or not to use RDMA reads. RDMA reads are appealing as they perform better than RPCs in many cases, but RPCs are more flexible as the CPU can perform additional operations before replying to the client. For example, systems that use RPCs often use indirection per object which makes it easier to de-fragment memory, evict cold objects to a storage tier, support variable sized objects, and perform similar tasks. Indirection per object can be implemented in systems that use RDMA too, but it doubles the cost of object reads, so it is avoided. The designers of RDMA-based systems are left in a dilemma: should they use RDMA reads or not? We help answer this question by examining the trade-off between performance and flexibility that the choice implies.

1 Introduction

Fast networks with RDMA [27, 28] are becoming common in data centers as their price has become similar to the price of Ethernet [12]. At the same time, DRAM has become cheap enough to make it cost-efficient to put 256 GB or more of DRAM in commodity data center servers. Memory sizes are going to increase further as the price of DRAM keeps decreasing and new, higher density, memory technologies such as 3D XPoint [30], PCM [7], and memristors [20] are developed. Main memory is also becoming non-volatile. These new memory technologies are all non-volatile. Non-volatile DIMMs [25, 35] and approaches based on cheap distributed UPS [3] make even DRAM non-volatile. These hardware trends allow building scale-out systems that store data in memory and access it over fast networks to allow unprecedented performance for data center applications.

Scaling out systems like relational or graph databases is challenging with traditional Ethernet networks because networking overheads are so high that scale-out solutions often need to run on tens of machines to match the performance of a single-machine alternative [23]. This makes scale-out with Ethernet unattractive. Partitioning reduces communication, but many workloads, e.g. graphs cannot be ideally partitioned, so they have to access data on remote machines. Batching amortizes communication costs, but it also increases latency, which is not a good trade-off for latency sensitive applications. RDMA networks help reduce the overhead of scaling out by providing micro-second latency access to the memory of remote machines. This is particularly

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

beneficial for applications that have irregular access patterns and that are latency sensitive, like key-value stores, graph stores, and relational databases. RDMA networks also provide high bandwidth and high message rates, enabling systems that have high throughput and low latency at the same time. High message rates are particularly important as modern applications store small objects [1] so network bandwidth is rarely the bottleneck.

Several systems that take advantage of the new hardware have recently appeared in the literature, all showing orders of magnitude improvement in latency and throughput compared to traditional TCP-based systems [3, 7, 12, 15, 21]. They commonly provide a general programming abstraction with distributed transactions to simplify applications built on them. Transactions manipulate objects in memory atomically, allowing applications to ignore concurrency and failures in a distributed system.

Despite having similar application interfaces, RDMA-based systems differ in several aspects, notably in their choice to use or not to use RDMA reads. RDMA reads allow a machine to fetch data from memory of remote machines without involving the remote CPU. This can have significant throughput and latency advantages compared to RPC—2x to 4x in our measurements. For this reason some systems advocate the use of RDMA reads [3, 12, 21, 22]. On the other hand, RPCs are more flexible as the CPU can perform additional operations before replying to the RPC. This is commonly used in systems that only use RPCs to implement a level of indirection per object [10, 7, 8]. This level of indirection makes it easy to transparently perform system-level tasks like memory de-fragmentation, cold object eviction, resizing objects, and similar. Systems that use RDMA reads could implement indirection per object, but this would mean that two RDMA reads need to be issued to read an object—one for the indirection pointer and the other for the object. For this reason they do not implement per-object indirection. Note that this loss of flexibility mainly impacts system-level tasks. Using RDMA reads does not prevent use of RPCs so applications can use them in the same way as on RPC-only systems.

Designers of RDMA-based systems are left in a dilemma: should they use RDMA reads or not? Is the performance benefit worth the loss of flexibility? Our experience suggests that RDMA reads are a good choice in many cases. Applications like relational and graph databases keep enough information about data layout to support de-fragmentation, eviction, and similar tasks and many applications can be modified to do the same. But RDMA reads are not always the right choice. It is sometimes worth sacrificing some performance for flexibility.

We believe that in the future we will not have to make this choice. We expect to see systems with carefully designed software and hardware components that have both the flexibility of RPC and the performance of RDMA reads and writes. How to build these systems is still an open question.

In this paper we examine the implications of using RDMA reads by providing background on RDMA networks (Section 2), discussing the trade-off between performance and flexibility (Sections 3 and 4), contrasting the design of RAMCloud [10, 13, 15, 19], an RPC-only system, and FaRM [3, 5], a system that uses RDMA reads, to illustrate the differences (Sections 5 and 6), and describing challenges that lay ahead (Section 7).

2 RDMA

RDMA networks implement transport protocols directly in hardware and expose them in user mode. The latest NICs have 100 Gbps of bandwidth, 1-3 micro-second latency, can process 200 million messages per second and can issue . a network operation using less than 1500 cycles in our measurements. This is orders of magnitude better than TCP in all respects. RDMA networks implement reliable connected (RC) and unreliable datagram (UD) transports. The former is reliable, in-order transport that supports large transfers, similarly to TCP. The latter is unreliable, unordered transport, that supports datagrams up to 4 kB, similarly to UDP.

In addition to send/receive, RC transport supports RDMA reads and writes. RDMA requests are sent to the remote NIC which performs them without involving the CPU. To enable RDMA access to a block of memory, it has to be registered with the NIC. Registration pins the memory, assigns it an identifier, and maps it in the NIC. Machines that are sent the identifier can access the block of memory using RDMA any number of times without further involvement from the server's CPU. Memory can be de-registered to prevent further RDMA accesses.

Several problems need to be solved to achieve maximum performance with RDMA reads [3]. Most NICs have little memory, so they keep state in system memory and use NIC memory as a cache. This allows supporting larger clusters with little memory, but results in performance degradation when too much state is used at the same time, as the state needs to be fetched from system memory over PCI bus. We have encountered issues with caching of mapping tables for registered memory blocks and connection state. When memory blocks are allocated using a standard allocator, they get mapped using 4 kB pages, so mapping just a few MBs fills the cache. We have measured 4x performance impact with just 64 MB of registered memory. To solve this problem, memory needs to be registered using bigger pages. For instance, FaRM [3] uses a special allocator that allocates 2 GB naturally aligned pages. This enables the NIC to map each 2 GB page as a single entry, allowing FaRM to register hundreds of GBs of memory without impacting performance. Similarly, using too many connections to fit in the cache degrades performance—we measured up to 4x performance degradation. To alleviate this problem, connections can be shared between threads, but this introduces synchronization overheads—we measured 2-3x increase in CPU cycles to issue RDMA read when sharing connections.

The most widely deployed types of RDMA networks today are Infiniband [27] and RoCE [28]. Infiniband offers slightly better performance than RoCE, but it is also more expensive. Infiniband supports link-level flow control, but does not provide other forms of congestion control. This makes it suitable to clusters of up to a few hundred machines connected with a single switch. Infiniband does not offer good performance to applications that use traditional TCP/IP stack, meaning that most legacy applications perform poorly. In contrast, RoCE runs on lossless Ethernet with priority flow control. This provides good support for applications that use TCP/IP. RoCE supports congestion control for connected transport [15, 24]. This means that RoCE scales better than Infiniband and can support data center scale networks. For these reasons, RoCE is deployed in data centers where it is used for traditional data center workloads using TCP/IP or RDMA stacks. In this paper we are mainly considering RoCE as it is more widely available.

3 Performance

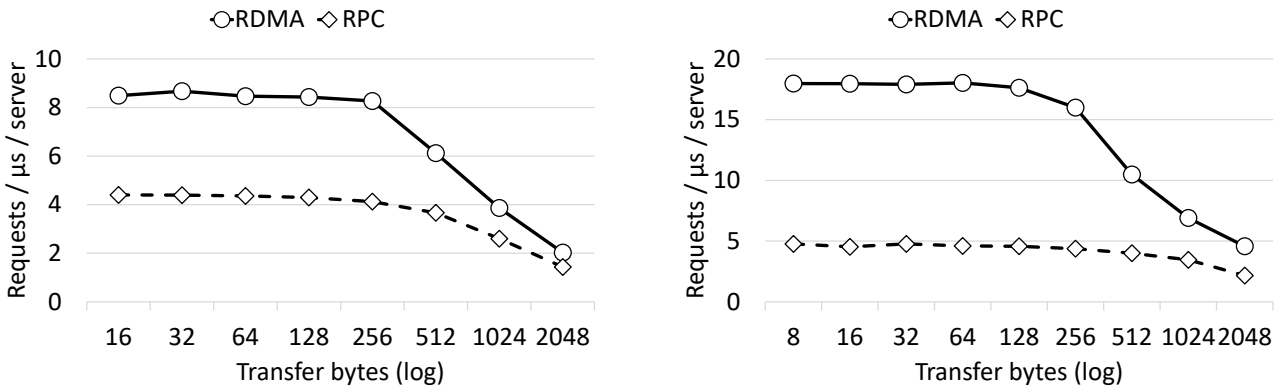
3.1 Connected transport

RDMA reads are more CPU efficient than RPCs for simple operations, like reading an object from memory of a remote machine, because they do not use any CPU cycles on the machine where the object resides. These simple operations are predominant in many applications, like graph databases, key-value stores, and OLTP workloads, which makes RDMA appealing. CPU efficiency translates to performance gains on modern hardware, as most systems are CPU bound—storage and network are out of the performance path, so CPU is the main bottleneck. With NICs that can process 200 million messages per second, even the latest Intel's server CPUs[1] provide a budget of just above 1000 cycles per network operation, which is not enough even to issue all the operations the NIC can process. Systems are expected to remain CPU bound in the future too, as advances in network technology are projected to outpace advances in CPUs [29]. RDMA reads also improve latency, especially on a busy system, as they do not involve the CPU on the target machine, and hence do not have to wait for the CPU to become available.

In addition, RDMA reads exchange fewer network messages than RPC implemented on RC transport. This means that RDMA reads perform better than RPC even when CPU is not the bottleneck, such as with the previous generation of network hardware, which is predominantly in use today.

Figure 1 compares performance of RDMA reads and RC RPCs for small reads. We ran an experiment where each machine in a cluster reads data from memory of all other machines. Each read is to a randomly selected machine and a random piece of memory in it. Reads are performed either directly with RDMA or using RPC implemented with RDMA writes [3]. Figure 1a shows per-machine throughput on a cluster of 20 machines each

[1] At the time of writing Intel's Xeon E7-8890 v4 with 24 cores at 2.2 GHz.



(a) 1 NIC (network-bound)

(b) 2 NICs (CPU-bound)

Figure 1: Per-machine RDMA and connected RPC read performance.

with a single 40 Gbps Mellanox ConnectX-3 NIC. RDMA reads outperform RPC by 2x because the bottleneck in this setup is the NIC's message processing rate. Figure 1b shows what happens when CPU is the bottleneck. It shows per-machine throughput on the same benchmark running on a 90-machine cluster with two 56 Gbps Mellanox ConnectX-3 NICs in each machine. With the higher network message rates provided by two NICs, using RDMA reads provides even more benefit as it outperforms RPC by 4x. This is because the system is CPU bound. Using RPC makes it almost CPU bound on this simple benchmark even when using a single NIC, so providing more network capacity does not help improve RPC throughput significantly.

3.2 Unreliable transport

Using unreliable datagrams for RPC has some performance benefits over using connected transport on current hardware [7]. The main advantage is that UD does not use connections and is, therefore, not impacted by connection caching issues and the overheads of connection sharing described in Section 2. In addition, UD RPC can exploit application-level semantics to send fewer messages than RC RPC. For instance, an RPC response can be treated as the ack to the request meaning that a separate ack message does not need to be sent. RC RPC cannot do this as these application-level semantics are not available to the NIC which is responsible for sending acks. UD also allows more batching when issuing sends and receives to the NIC than RC. UD can send any group of requests as a batch whereas RC can batch only requests on the same connection, which amortizes costs of PCI transactions more efficiently.

The disadvantage of using UD is that, unlike RC, it does not implement congestion control, reliable delivery, and packet fragmentation and re-assembly in hardware, so all of this has to be done in software, which introduces CPU overheads. Of these, congestion control is the most challenging to support. Several promising approaches for congestion control in RoCE exist [15, 24], but they do not work with UD. Without congestion control, UD cannot be used in data centers. Packet re-transmissions, fragmentation and re-assembly can be implemented in software, but this would introduce overheads e.g. because they might require additional message copies.

Work on FaSST [7] has shown that UD RPC that does not implement any of the above features can even outperform RDMA reads in some specific scenarios. To achieve maximum performance, FaSST RPC exposes a very basic abstraction— unreliable send/receive of messages that can fit in a network packet (2 kB for RoCE). If messages are lost, FaSST crashes the machine that observes the loss. FaSST also does not implement any congestion control. This design was motivated by the setup the authors used—a single-switch cluster with an expensive Infiniband network and simple benchmarks that only send small messages. This is very different from large RoCE deployments. For instance, the authors have observed almost no message loss in their experiments, which is why they decided to handle message loss by crashing the machine that detects it instead of

re-transmitting messages.

There are two problems with UD RPC performance argument. First, using it in realistic deployment scenarios would require implementing the missing features, which is not trivial and would introduce overhead. Until these are implemented, it is hard to make any conclusions about performance. In contrast, RC RPCs and RDMA reads can readily be used in any setting. Second, the core of the performance argument relies on the specifics of the hardware available today and new hardware is likely to make it invalid. Having more memory on NICs or better handling of connection cache misses would reduce or even eliminate the need to share connections across threads which would reduce overhead [33]. It was already shown that a NIC integrated with the CPU can support several thousands of connections with no overhead [2].

4 Flexibility

RPCs are more flexible than RDMA reads because the CPU can perform additional operations before sending the response. Systems that only use RPCs use this to implement a level of indirection per object (typically as a hashtable), which allows them to move objects transparently to the application to e.g. change their size, de-fragment memory, or evict them to cold tier. It is possible to implement indirection per object with RDMA reads too. A system could store a pointer to an object in a known location and update that pointer whenever the object moves. Object reads would require two RDMA reads—one to read the pointer and another to read the object. This would make the system more flexible, but would also double the cost of reads, eliminating much of the performance benefit of using RDMA reads. As Figure 1b shows, such a scheme would still be faster than using RC RPCs in CPU-bound workloads, but systems typically avoid it to achieve maximum performance.

To support efficient reading of objects over RDMA, each machine in the cluster needs to know exactly where each object lives in the memory of its host without additional communication steps. To achieve this, machines cache addressing meta-data on the first access and update it only when it changes. Storing per-object addressing meta-data for all objects on all machines would introduce prohibitive communication and memory overheads.[2] To reduce these overheads addressing is done at a coarser granularity. This is different from reading with RPC where the reader only needs to know which machine holds the object so it can send an RPC to it. The machine storing the object is the only one that needs to know the exact location of the object in memory, making indirection per object local and efficient.

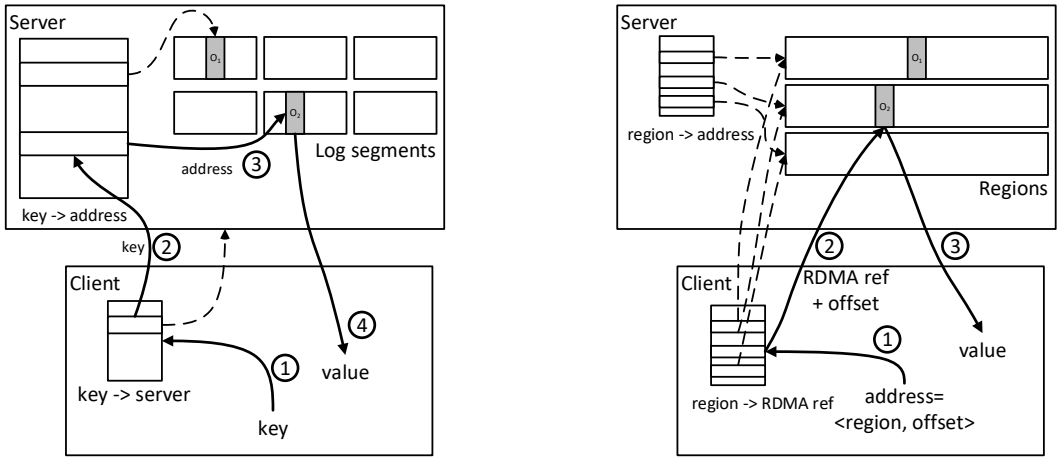
Using RDMA reads also makes it harder to track accesses to objects from remote machines. The NIC does not maintain detailed access information and the CPU on the host machine does not observe incoming read requests. This makes it much harder to identify cold or hot objects and to keep performance metrics, as this has to be done in a distributed fashion.

It is important to note that these flexibility limitations mainly impact system-level tasks. Nothing prevents the application from using RPCs when beneficial e.g. to exploit locality by shipping an operation to the machine that holds the data the operation accesses.

5 RAMCloud

Several RDMA systems, such as RAMCloud [10, 13, 15, 19], HERD [8], and FaSST [7], are RPC-only. We describe RAMCloud here in more detail to illustrate the design of RPC-only systems.

[2] In a workload with average object size of 128 bytes and addressing meta-data of 8 bytes per object, clusters with more than 16 machines would store more addressing meta-data than actual data when meta-data is fully replicated.



(a) RAMCloud

(b) FaRM

Figure 2: RAMCloud and FaRM memory layout. Dashed lines are pointers. Solid lines with numbers next to them are steps to read an object. In RAMCloud, the client maps the object key to the server using a local table in step 1, it sends the message to the server, which uses a local hashtable to find the address of the object in step 2, the server fetches the object from local memory in step 3, and sends the value back to the client in step 4. The server's CPU is involved in steps 2, 3, and 4. In FaRM, the client maps the region part of the address to the RDMA reference of the region using a local region mapping table in step 1, adds the offset to the the reference and issues an RDMA read in step 2, and the server's NIC returns the value in step 3. The server's CPU is not involved in any step.

5.1 Overview

RAMCloud was the first storage system to keep all data in main memory. It exposes a key-value store with distributed transactions to the users. RAMCloud is optimized for storing objects of several hundred bytes. It keeps only the primary copy of data in main memory. Backups are written to secondary storage. It relies on a small amount of non-volatile memory on backups to batch writes to disk to make logging efficient. RAMCloud achieves latency of 4.7 micro-seconds for reads and 13.4 micro-seconds for small object updates [10]. Backups of data are spread across the cluster to make failure recovery fast. RAMCloud recovers 35 GB of data from a failed server in 1.6 seconds [15].

5.2 Memory management

RAMCloud organizes its memory as a log both in memory and on disk (Figure 2a). Log is split into 8 MB segments. RAMCloud stores each object's address only on the primary, in a hashtable that maps the identifier of the table the object belongs to and its key into object's address. When an object is inserted or updated, a new value of the object is appended to the log and its address is updated in the hashtable.

RAMCloud de-fragments the logs to achieve high memory utilization. It uses two-level de-fragmentation to ensure both high memory space utilization and low disk bandwidth utilization. In-memory de-fragmentation is done at the primary, without communicating with backups or accessing disk. The primary picks a segment and copies live objects from it into a number of 64 kB seglets. This saves memory while keeping the contents of the logical segment unchanged and thus does not require communication with backups. Using small fixed-size seglets instead of allocating a block of memory that can fit all the objects keeps fragmentation low. Full de-fragmentation is done by scanning the segments in memory, copying live objects from them into new segments,

and sending the new segments to backups to be written to disk. RAMCloud runs full de-fragmentation only if running out of space on disk or if there are too many object tombstones in the log. RAMCloud achieves 80-90% memory utilization with little performance impact.

5.3 Flexibility

Having indirection per object through the hashtable gives RAMCloud flexibility. It allows RAMCloud to transparently move objects to de-fragment its logs. RAMCloud can also change size of an object if an application requires it. Evicting objects from memory into a cold tier would also be straightforward—it is enough to simply move the object and update its location in the hashtable.

6 FaRM

A number of RDMA systems, such as Pilaf [12], FaRM [3, 5], DrTM [21], and NAM-DB [22], read objects using RDMA reads. We describe FaRM here in more detail to illustrate the trade-offs involved.

6.1 Overview

FaRM is an in-memory, distributed computing platform that exposes the aggregate memory of a cluster of machines as a single shared address space. Applications manipulate objects stored in the address space by invoking FaRM APIs to transactionally allocate, free, read and write objects. Transactions are fully general and strictly serializable. Similarly to RAMCloud, FaRM is optimized for small objects. Servers in a FaRM cluster both store data and execute application code. FaRM supports atomic reads of remote objects. It inserts cache-line versions into stored objects and uses them to ensure object reads are atomic. In addition to supporting direct reads of remote objects, FaRM allows for function shipping by using efficient RDMA-based RPCs. It also supports allocating new objects close to existing objects to enable locality-aware optimizations.

FaRM protocols were carefully designed and optimized to achieve high throughput and low latency. For instance, FaRM achieves throughput of 140 million transactions per second with median latency of 60 micro-seconds in the TATP benchmark [16] on a 90 machine cluster. It also does 270 million tpmC on a TPC-C [34] like benchmark. Latency of a single small key-value store lookup is 4.3 micro-seconds. FaRM recovers from failures quickly. It takes just 50 ms to get back to maximum throughput after a single-machine failure.

6.2 Memory management

FaRM's memory management scheme is designed to support efficient RDMA reads and low-overhead object allocation. It minimizes the amount of addressing meta-data that needs to be stored across the cluster and eliminates communication during common-case object allocation.

FaRM organizes its address space as 2 GB memory regions (Figure 2b), which are the unit of addressing, replication, and recovery. Regions are replicated in memory, typically on three machines, but replication degree can be configured per region. Region placement is done by a centralized region allocator which assigns region replicas to machines. It places each replica in a different failure domain, and balances the number of regions per machine while attempting to respect placement hints the application can provide. The region allocation information is replicated across the machines in the cluster during allocation and is reconstructed during region allocator recovery. Each region is assigned a 33-bit region identifier in sequence. Addresses of FaRM objects consist of the region identifier and a 31-bit offset inside the region. This addressing scheme is designed to keep addressing meta-data small and to avoid frequent updates. Addressing information is kept per region, so each machine can cache it for all regions in the system with small space overhead. Region placement changes only when machines fail or join the system, so this cache does not need to be updated frequently.

Regions are split into 1 MB slabs. Object are allocated from slabs. Each slab contains objects of the same size. FaRM supports 256 different size classes between 64 bytes and 1 MB. Slabs are allocated on demand when threads run out of slabs for allocating objects of a particular size. Each active slab is owned by a single thread to avoid thread synchronization during allocation. Slab meta-data is replicated to machines storing region replicas during slab allocation and deallocation. This meta-data is used during recovery.

Object allocations are performed on the primary without any communication with the backups. An objects is marked allocated by setting the allocated bit in its header. Setting the bit does not require any additional messages—it gets set on backups during the commit of the allocating transaction. The allocated bits are used to reconstruct object allocator meta-data on the new primary after failures.

6.3 Flexibility

FaRM's object allocation scheme achieves the goals it was designed for, but it is not as flexible as a typical RPC-only system—it maintains indirection per region instead of per object. This makes it hard for the system to support variable sized objects, memory de-fragmentation, cold object eviction, and similar.

To perform these tasks, FaRM relies on the application and its knowledge of the data layout. The application can move an object as long as it knows where all the pointers to the object being moved are. To do so, it executes a transaction to atomically allocate new space for the object, copy the contents into it, update all pointers to the object, and free the old object. By using a transaction, the application deals with the issues of consistency and fault tolerance during the move—other transactions either see the old pointers and the old version of the object or the new ones. This is suitable for applications such as relational and graph databases that keep track of data layout and pointers between objects, which makes it easy to move objects. Applications that allow arbitrary pointers between objects, on the other hand, cannot easily move objects.

Next we discuss how FaRM and applications using it deal with some of the flexibility limitations.

Object size. When reading an object, the application needs to supply the object's address and size. FaRM cannot infer the size from the address, as the slab-level meta-data required to do this is kept only on the machines storing object replicas, not on all machines in the cluster. In most cases this is not a significant challenge. Applications that impose schema on stored objects, such as databases or language runtimes, know the size of the objects they are reading. Applications that do not typically store a 1-byte FaRM size class with the pointers to the object and use it when performing the read. Some of our applications adopt this approach and it serves them well. By storing the size class in the pointer we can avoid increasing the pointer size and still support 2^{56} bytes in the address space.

Variable sized objects. FaRM's memory management approach prevents it from resizing objects in place. To resize an object, the application needs to move it into a larger (or smaller) newly allocated space. Alternatively, variable sized objects can be represented as a list of chunks. This works particularly well when the object consists of a fixed-sized and a variable-sized part. In those cases it is common that the majority of operations only access the fixed-sized part, so the list-of-chunks approach improves performance. For this reason, some of applications use it even when they have enough information to move objects.

Memory de-fragmentation. FaRM's slabs can get fragmented when an application allocates a large number of objects of a certain size and then frees most of them. This reduces memory efficiency because most slabs will be mostly empty. A typical solution to this problem is to de-fragment slabs by moving live objects to as few slabs as possible.

De-fragmentation requires knowing which objects to move and how to move them. The former is known by FaRM and the latter by the application, making it impossible for either to perform de-fragmentation on its own.

	Reliable connections	Unreliable datagrams
Scalability	Limited	Good
Congestion control	Supported	No
Reliable	Yes	No
In-order delivery	Yes	No
Message size	2 GB	2 kB
RDMA reads and writes	Yes	No

Table 1: RDMA transport capabilities

To solve this, we are planning to extend FaRM to invoke application-defined callbacks with addresses of objects that should be moved for de-fragmentation and have the application move the objects. To obtain the space for the new object during de-fragmentation, the application can simply invoke FaRM's allocator. The allocator always prefers mostly full slabs for new allocations to reduce fragmentation.

Dealing with de-fragmentation at the application level in this way instead of performing it transparently by the system can be beneficial as application-level knowledge can be used to optimize the process. For instance, the application can choose to postpone de-fragmentation to avoid increasing load in busy periods, or to completely ignore the callback if it will soon need to allocate objects of the same size.

The application needs more information about object pointers to perform de-fragmentation than for resizing—the callback is invoked just with the address and size of the object to be moved, without providing a path to the object as is the case when resizing. This means that there needs to be a simple way to find the pointers to the object given just its address. For instance, to support de-fragmentation, the application might need to use a doubly-linked list where it would otherwise use a singly-linked list.

Cold objects. Despite the decreasing price of main memory, it still remains much more expensive than SSDs and disks. To reduce costs, some applications store only frequently accessed objects in memory and keep the rest on disk. A system that uses RDMA reads needs to solve two challenges to support this kind of tiering: identifying cold objects and evicting them to the storage tier. Identifying cold objects is hard as the CPU on the host machine does not observe remote accesses and the NIC does not keep detailed access stats. Alternatively, the application can decide which objects to move to the cold tier and when using application-level semantics. For instance, the application might periodically evict objects older than some threshold. Objects can be moved in a similar fashion to de-fragmentation—pointers to the object are either replaced by cold-tier pointers or invalidated when the objects is evicted.

7 Future challenges

7.1 Networking

Table 1 summarizes the advantages and drawbacks of using RDMA RC and UD transports, as discussed in Section 3. It shows that neither transport provides all desired features. UD provides good scalability, but it leaves dealing with congestion control, re-transmissions, and packetization and re-assembly to software. It also does not support RDMA reads and writes. On the other hand, RC has scalability issues because of limitations of available hardware. Building a network transport that provides all desired features remains a challenge.

To build it, we need to first rethink what the important features are. Not all features from Table 1 are equally important and a simpler transport can be more efficient. For instance, most distributed systems robustly deal with reordered messages, so providing in-order delivery guarantees in the network is not useful. Re-transmission can

be more efficient when message semantics are known, so it makes sense to either implement it in software or expose the semantics to hardware. For instance, idempotent operations, like a read-only RPC, can be safely executed multiple times, so there is no need to reliably respond to it, as the operation can simply be re-executed if the response is lost. It might also be viable to build hardware support for the common case, and leave corner cases to software. For instance, optimizing hardware for RDMA reads that fit in a network packet or even not supporting bigger RDMA reads at all would be acceptable to many applications.

NIC hardware can be improved to reduce the CPU cost of issuing RDMA operations regardless of the abstraction it implements. The predominant overheads when interacting with the NIC are PCI transactions used to issue and complete operations. These overheads can be reduced by batching—issuing or completing multiple operations with a single PCI transaction [6]. There are opportunities to allow more batching than today, for instance by allowing applications to issue batches containing operations on different connections, which is not supported today. Alternatively, instead of waiting for PCI messages from the CPU, the NIC could periodically poll request queues in memory. This would remove the need for the CPU to initiate PCI transactions and CPU overheads of initiating them, but it would also increase latency, so it might not always be a good trade-off.

7.2 Hardware acceleration

Widespread hardware acceleration is going to change the way we think about and build systems in the future. With the end of Moore's law nearing, big companies are heavily investing in hardware acceleration. For instance, Microsoft is deploying FPGA Catapult boards [18, 3], Google is deploying custom TPU ASICs to speed up machine learning [31], Intel has announced work on integration of Xeon processors with coherent FPGAs in a single package [26], and Mellanox has a product brief for a ConnectX-3 Pro card with FPGA [32]. This creates an opportunity to build hardware that makes distributed systems better.

The main challenge with hardware specialization is in selecting the right primitives to move to hardware, because of cost and complexity of development, even when using FPGAs. For this reason, primitives like encryption and compression are the best targets—they are relatively simple and they would benefit many applications. Time synchronization, which would make time-based protocols better, or remote read tracking, which could help with detecting cold or hot objects, are also good candidates, as they would be useful in many cases. Implementing parts of a general, widely-used distributed computation platform in hardware would also benefit many applications. We have experimented with hardware implementation of FaRM atomic reads and have seen relatively small but consistent performance improvements. Other operations used by such a platform like pointer chain following, hashtable lookups, and even parts of transaction commit protocol are good candidates for acceleration in hardware.

With the right hardware primitives system builders would not have to choose between performance and flexibility, like they do today with RDMA reads. A system that stores data in a hashtable and implements hashtable lookups in the NIC would give us the best of both worlds—performance of hardware lookups and flexibility of RPC-only systems. Some work on offloading hashtable functionality to FPGA has already been done and the results are promising [8, 9]. The main remaining questions are what the right software-hardware design is, how close to today's NIC's request rates can performance of a hardware hashtable get and how much flexibility will have to be traded off to support efficient hardware implementation.

8 Conclusion

So, should you use RDMA reads or not? As always, the answer is: it depends on the workload. RDMA reads can deliver benefits both in terms of latency and throughput, but they are less flexible than RPCs. We discussed ways to overcome these flexibility limitations in applications with well defined schemas and data layout like relational and graph databases. RDMA reads are a good choice for those applications. On the other hand, using

RDMA reads complicates the system and puts additional requirements on the application, at the extreme wasting space and burning through some of the performance benefit it brings. In the future we might have it all—a fast hashtable implemented in hardware can give us performance of hardware and flexibility of a key-value store, but this will require overcoming a number of challenges.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS 2012.*

[2] V. Basant, P. Rahoul. Oracle's Sonoma processor: Advanced low-cost ' processor for enterprise workloads. *HOTCHIPS 2015*.

[3] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, D. Burger. A Cloud-Scale Acceleration Architecture. *MICRO 2016*.

[4] A. Dragojević, D. Narayanan, M. Castro, O Hodson. FaRM: Fast Remote Memory. *NSDI 2014.*

[5] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. *SOSP 2015.*

[6] M. Flajslik, M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. *USENIX ATC 2013.*

[7] S. Hudgens. Overview of Phase-Change Chalcogenide Nonvolatile Memory Technology. *MSR bulletin, 2004, vol 29, number 11.*

[8] Z. István, D. Sidler, G. Alonso. Building a distributed key-value store with FPGA-based microservers. *FPL 2015*.

[9] Z. István, D. Sidler, G. Alonso, M. Blott, K. Vissers. A Flexible Hash Table Design For 10Gbps Key-value Stores on FPGAs. *FPL 2013*.

[10] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, Diego and S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, S. Yang. The RAMCloud Storage System. *TOCS, September 2015, vol 33, number 3*.

[11] A. Kalia, M. Kaminsky, D. G. Andersen. FaSST: Fast, Scalable, and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. *OSDI 2016*.

[12] A. Kalia, M. Kaminsky, D. G. Andersen. Using RDMA Efficiently for Key-Value Services. *SIGCOMM 2014*

[13] C. Lee, S.J. Park, A. Kejriwal, S. Matsushita, J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. *SOSP 2015*.

[14] C. Mitchell, G. Yifeng, L. Jinyang. Using one-sided RDMA reads to build a fast, CPU-efcient key-value store. *USENIX ATC 2013*.

[15] R. Mittal, T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *SIGCOMM 2015*.

[16] S. Neuvonen, A. Wolski, M. Manner, V. Raatikka. Telecom Application Transaction Processing benchmark. `http://tatpbenchmark.sourceforge.net`.

[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, M. Rosenblum. Fast Crash Recovery in RAMCloud. *SOSP 2011*.

[18] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, S. Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ISCA 2014*.

[19] S. M. Rumble, A. Kejriwal, J. Ousterhout. Log-structured Memory for DRAM-based Storage. *FAST 2011*.

[20] D. Strukov, G. S. Snider, D. R. Stewart, S. R. Williams. The missing memristor found. *Nature, 2008, vol 453, number 7191*.

[21] X. Wei, J. Shi, Y. Chen, R. Chen, H. Chen. Fast In-memory Transaction Processing using RDMA and HTM. *SOSP 2015*.

[22] E. Zamanian, C. Binnig, T. Kraska, T. Harris. The End of a Myth: Distributed Transactions Can Scale. *VLDB 2017*.

[23] X. Zhu, W. Han and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. *USENIX ATC 2015*.

[24] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, M. Zhang. Congestion Control for Large-Scale RDMA Deployment. *SIGCOMM 2015*.

[25] Agiga Tech. `http://www.agigatech.com`.

[26] D. Bryant. Disrupting the Data Center to Create the Digital Services Economy. `http://itpeernetwork.intel.com/disrupting-the-data-center-to-create-the-digital-services-economy/` *Intel IT Peer Network*, June 18, 2014.

[27] Infiniband Trade Association. `http://www.infinibandta.org`.

[28] Infiniband Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE) *2010*.

[29] Infiniband Trade Association. InifiniBand Roadmap. `http://www.infinibandta.org/content/pages.php?pg=technology_overview`.

[30] Intel. 3D XPoint Unveiled—The Next Breakthrough in Memory Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html`.

[31] N. Jouppi. Google supercharges machine learning tasks with TPU custom chip. `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html` *Google Cloud Platform Blog*, May 18, 2016

[32] Mellanox Technologies. Programmable ConnectX-3 Pro Adapter Card. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_Programmable_ConnectX-3_Pro_Card_EN.pdf`

[33] Mellanox Technologies. Connect-IB: Architecture for Scalable High Performance Computing. `http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf`

[34] Transaction Processing Performance Council (TPC). TPC BENCHMARK C: Standard Specification. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`.

[35] Viking Technology. `http://www.vikingtechnology.com`.

Designing Databases for Future High-Performance Networks

Claude Barthels, Gustavo Alonso, Torsten Hoefler
Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

Abstract

High-throughput, low-latency networks are becoming a key element in database appliances and data processing systems to reduce the overhead of data movement. In this article, we focus on Remote Direct Memory Access (RDMA), a feature increasingly available in modern networks enabling the network card to directly write to and read from main memory. RDMA has started to attract attention as a technical solution to quite a few performance bottlenecks in distributed data management but there is still much work to be done to make it an effective technology suitable for database engines. In this article, we identify several advantages and drawbacks of RDMA and related technologies, and propose new communication primitives that would bridge the gap between the operations provided by high-speed networks and the needs of data processing systems.

1 Introduction

Distributed query and transaction processing has been an active field of research ever since the volume of the data to be processed outgrew the storage and processing capacity of a single machine. Two platforms of choice for data processing are database appliances, i.e., rack-scale clusters composed of several machines connected through a low-latency network, and scale-out infrastructure platforms for batch processing, e.g., data analysis applications such as Map-Reduce.

A fundamental design rule on how software for these systems should be implemented is the assumption that the network is relatively slow compared to local in-memory processing. Therefore, the execution time of a query or a transaction is assumed to be dominated by network transfer times and the costs of synchronization. However, data processing systems are starting to be equipped with high-bandwidth, low-latency interconnects that can transmit vast amounts of data between the compute nodes and provide single-digit microsecond latencies. In light of this new generation of network technologies, such a rule is being re-evaluated, leading to new types of database algorithms [6, 7, 29] and fundamental system design changes [8, 23, 30].

Modern high-throughput, low-latency networks originate from high-performance computing (HPC) systems. Similar to database systems, the performance of scientific applications depends on the ability of the system to move large amounts of data between compute nodes. Several key features offered by these networks are (i) user-level networking, (ii) an asynchronous network interface that allows the algorithm to interleave computation and communication, and (iii) the ability of the network card to directly access regions of main memory without going through the processor, i.e., remote direct memory access (RDMA). To leverage the advantages of these networks,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

new software interfaces and programming language extensions had to be designed [17, 25]. These interfaces and languages not only contain traditional message-passing functionality, but also offer support for accessing remote memory directly without any involvement of a remote process. The latter can be used to reduce the amount of inter-process synchronization, as processes are not notified about remote memory accesses through the network card. Having less synchronization between processes is crucial in order to scale applications to thousands of processor cores. The downside is that using one-sided memory operations requires careful management of the memory regions accessible over the network.

The impact of high-performance interconnects on relational database systems has been recently studied [6, 7, 8, 23, 29, 30]. While it has been shown that distributed algorithms can achieve good performance and scale to a large number of cores, several drawbacks of the technology have also been revealed. In this article, we provide a comprehensive background on remote direct memory access (RDMA) as well as several related concepts, such as remote memory access (RMA) and partitioned global address space (PGAS). We investigate the role of these technologies in the context of distributed join algorithms, data replication, and distributed transaction processing. Looking ahead, we highlight several important directions for future research. We argue that databases and networks need to be co-designed. The network instruction set architecture (NISA) [12, 18] provided by high-end networks needs to contain operations with richer semantics than simple read and write instructions. Examples of such operations are conditional reads, remote memory allocation, and data transformation mechanisms. Future high-speed networks need to offer building blocks useful to a variety of data processing applications for them to be truly useful in data management. In this article, we describe how this can be done.

2 Background and Definitions

In this section, we explain how the concepts of Remote Direct Memory Access (RDMA), Remote Memory Access (RMA), and Partitioned Global Address Space (PGAS) relate to each other. Furthermore, we include an overview of several low-latency, high-bandwidth network technologies implementing these mechanisms.

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a hardware mechanism through which the network card can directly access all or parts of the main memory of a remote node without involving the processor. Bypassing the CPU and the operating system makes it possible to interleave computation and communication, thereby avoiding copying data across different buffers within the network stack and user space, which significantly lowers the costs of large data transfers and reduces the end-to-end communication latency.

In many implementations, buffers need to be registered with the network card before they are accessible over the interconnect. During the registration process, the memory is pinned such that it cannot be swapped out, and the necessary address translation information is installed on the card, operations that can have a significant overhead [14]. Although this registration process is needed for many high-speed networks, it is worth noting that some network implementations also support registration-free memory access [10, 27].

RDMA as a hardware mechanism does not specify the semantics of a data transfer. Most modern networks provide support for one-sided and two-sided memory accesses. Two-sided operations represent traditional message-passing semantics in which the source process (i.e., the sender of a message) and the destination process (i.e., the receiver of a message) are actively involved in the communication and need to be synchronized; i.e., for every send operation there must exist exactly one corresponding receive operation. One-sided operations on the other hand, represent memory access semantics in which only the source process (i.e., the initiator of a request) is involved in the remote memory access. In order to efficiently use remote one-sided memory operations, multiple programming models have been developed, the most popular of which are the Remote Memory Access (RMA) and the Partitioned Global Address Space (PGAS) concepts.

2.2 Remote Memory Access

Remote Memory Access (RMA) is a shared memory programming abstraction. RMA provides access to remote memory regions through explicit one-sided read and write operations. These operations move data from one buffer to another, i.e., a read operation fetches data from a remote machine and transfers it to a local buffer, while the write operation transmits the data in the opposite direction. Data located on a remote machine can therefore not be loaded immediately into a register, but needs to be first read into a local main memory buffer. Using the RMA memory abstractions is similar to programming non-cache-coherent machines in which data has to be explicitly loaded into the cache-coherency domain before it can be used and changes to the data have to be explicitly flushed back to the source in order for the modifications to be visible on the remote machine.

The processes on the target machine are generally not notified about an RMA access, although many interfaces offer read and write calls with remote process notifications. Apart form read and write operations, some RMA implementations provide support for additional functionality, most notably remote atomic operations. Examples of such atomic operations are remote fetch-and-add and compare-and-swap instructions.

RMA has been designed to be a thin and portable layer compatible with many lower-level data movement interfaces. RMA has been adopted by many libraries such as ibVerbs [17] and MPI-3 [25] as their one-sided communication and remote memory access abstraction.

RDMA-capable networks implement the functionality necessary for efficient low-latency, high-bandwidth one-sided memory accesses. It is worth pointing out that RMA programming abstractions can also be used over networks which do not support RDMA, for example by implementing the required operations in software [26].

2.3 Partitioned Global Address Space

Partitioned Global Address Space (PGAS) is a programming language concept for writing parallel applications for large distributed memory machines. PGAS assumes a single global memory address space that is partitioned among all the processes. The programming model distinguishes between local and remote memory. This can be specified by the developer through the use of special keywords or annotations [9]. PGAS is therefore usually found in the form of a programming language extension and is one of the main concepts behind several languages, such as Co-Array Fortran or Unified Parallel C.

Local variables can only be accessed by the local processes, while shared variables can be written or read over the network. In most PGAS languages, both types of variables can be accessed in the same way. It is the responsibility of the compiler to add the necessary code to implement a remote variable access. This means that from a programming perspective, a remote variable can directly be a assigned to a local variable or a register and does not need to be explicitly loaded into main memory first as is the case with RMA.

When programming with a PGAS language, the developer needs to be aware of implicit data movement when accessing shared variable data, and careful non-uniform memory access (NUMA) optimizations are required for applications to achieve high performance.

2.4 Low-latency, High-bandwidth Networks

Many high-performance networks offer RDMA functionality. Examples of such networks are InfiniBand [19] and Cray Aries [2]. Both networks offer a bandwidth of 100 Gb/s or more and a latency in the single-digit microsecond range. However, RDMA is not exclusively available on networks originally designed for supercomputers: RDMA over Converged Ethernet (RoCE) [20] hardware adds RDMA capabilities to a conventional Ethernet network.

3 RMA & RDMA for Data Processing Systems

Recent work on distributed data processing systems and database algorithms has investigated the role of RDMA and high-speed networks [6, 7, 8, 23, 29, 30]. The low latency offered by these interconnects plays a vital role in transaction processing systems where fast response times are required. Analytical workloads often work on large volumes of data which need to be transmitted between the compute nodes and thus benefit from high-bandwidth links. In this section, we investigate the advantages and disadvantages of RDMA in the context of a relational database by analyzing three use-cases: (i) join algorithms, (ii) data replication, and (iii) distributed coordination.

3.1 Distributed Join Algorithms

Many analytical queries involve join operations in order to combine data from multiple tables. This operation usually involves transmitting significant amounts of data between the compute nodes, and thus new network technologies can improve the performance of the overall system.

Recent work on join algorithms for multi-core servers has produced hardware-conscious join algorithms that exhibit good performance on multicore CPUs [1, 3, 4, 5]. These implementations make use of Single-Instruction-Multiple-Data (SIMD) extensions and are NUMA-aware. In order to extend this work beyond the scope of a single cache-coherent machine, we have augmented the partitioning phase of the radix hash join and the sorting phase of the sort-merge join to redistribute the data over an RDMA-enabled network while it is being processed at the same time, thus interleaving computation and communication [6, 7].

Radix hash join: The radix hash join is a partitioned hash join, which means that the input is first divided into small partitions before hash tables are built over the data of each partition of the inner relation and probed with the data from the corresponding partition of the outer relation. In order to create a large number of small partitions and to avoid excessive TLB misses and cache trashing, a multi-pass partitioning scheme [24] is used.

Each process starts by creating a histogram over the input. The histogram tracks how many elements will be assigned to each partition. These histograms are exchanged between all the processes and are combined into a global histogram to which every process has access. From this information, the join operator can determine an assignment of partitions to processes. This assignment can be arbitrary or such that it minimizes the amount of network traffic [28]. During the first partitioning pass, we separate the data as follows: (i) tuples belonging to local partitions (i.e., partitions which are assigned to the same node) are written into a local buffer, and (ii) tuples which need to be transmitted over the network are partitioned into RDMA buffers. Careful memory management is critical. First, we want to avoid RDMA memory registration costs during the join operation. Therefore, we allocate and register the partitioning buffers at start-up time. Second, we want to interleave the partitioning operation and the network communication. Therefore, the RDMA buffers are of fixed size (several kilobytes) and are usually not large enough to hold the entire input. When one of these buffers is full, a network transmission request is immediately created. Since the network processes these requests asynchronously, we have to allocate at least two RDMA buffers for each partition and process. This double-buffering approach allows the algorithm to continue processing while data is being transmitted over the network. The algorithm needs (i) to ensure that data belonging to the same partition ends up in consecutive regions of memory, as this simplifies further processing and improves data locality, and (ii) to use one-sided operations in order to avoid unnecessary synchronization between the processes. To achieve these goals, the information from the histogram is used. From the global histogram, each process knows the exact amount of incoming data and hence can compute the required RDMA buffer size. The processes need to have exclusive access to sections of this memory region. This can be done through a prefix sum computation over all the process-level histograms. Once the partitions are created, hash tables can be constructed and probed in parallel to find all matching tuples.

Sort-merge join: The sort-merge join aims at interleaving sorting and data transfer. Similar to the radix hash join, data is first partitioned into ranges of equal size and histograms indicating how many elements fall into each

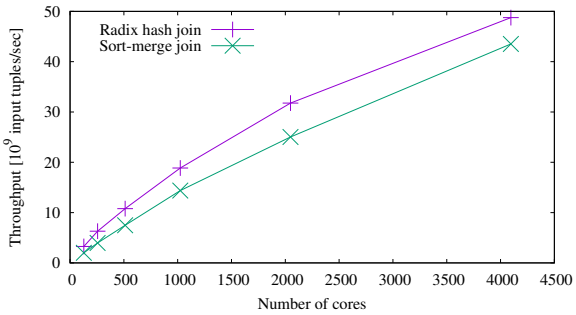


Figure 1: Distributed join algorithms can scale to hundreds of machines with thousands of CPU cores.

range are created. The process level histograms are combined into a global histogram. The range partitioning is performed locally by each process. The number of partitions should be equal to the number of cores in order to exploit the full parallelism offered by the machine. The partitioning step ensures that the data of the same partition can be transmitted to the same target node. In order to sort the data, each process takes a subset of the elements of each partition (several kilobytes) and sorts these elements. For sorting, we use an efficient out-of-place in-cache sorting routine. The sorted output is stored in RDMA buffers and is immediately transmitted to the destination. The process continues sorting the next elements without waiting for the completion notification of the network transfer. Using this strategy, sorting and network communication are interleaved. On the target node, the sorted runs end up consecutively in main memory. To avoid write-write conflicts, the information from the histograms is used to calculate sections of exclusive access for each process. After the data has arrived on the target node, the individual sorted runs are merged using a multi-level merge tree. After the merge operation, the data has been range-partitioned across the compute nodes, and, within each partition, all elements are sorted. A final pass over the sorted data finds the matching tuples.

Evaluation: We performed extensive evaluations of both algorithms on a rack-scale cluster and on two high-end supercomputers [6, 7]. The high-performance computing (HPC) machines provide us with a highly-tuned distributed setup with a high number of CPU cores, which allowed us to study the behaviour of joins in future systems. Given the increasing demand for more compute capacity, i.e., multiple servers per rack, multiple sockets per machine, multiple cores per socket, and multiple execution contexts per core, we estimate that future rack-scale systems (e.g., database appliances) will offer thousands of processor cores with several terabytes of main memory, connected by high-speed interconnects. In Figure 1, we observe that both algorithms can scale to several thousands of CPU cores. Both hash- and sort-based algorithms achieve a high throughput at large scale.

From this result, we conclude that the asynchronous interface of modern networks can be used to implement efficient hashing and sorting routines. Join processing and communication can be interleaved to a large extent. In that context, RDMA can be used to hide parts of the remote memory access latency. However, our results also indicate that a substantial part of the available capacity is not used. In order to further improve performance, the network interface needs be extended such that information about the communication pattern can be pushed down to the network card. Such information could be used by the hardware to improve decisions when to execute the asynchronous data transfers. Both algorithms employ an all-to-all communication pattern, which could be further optimized by introducing a light-weight network scheduling policy on the network card.

During the hashing and sorting operation, each process is working on its own set of input data. Using one-sided RMA operations reduces the amount of synchronization in these phases, as the target process does not need to be actively involved in the communication in order for the transfer to complete. However, the benefits of RMA do not come for free, as they require upfront investment in the form of a histogram computation phase. The global histogram data provides the necessary information to determine the size of the RDMA buffers and allows the processes to compute offsets into these buffers for exclusive access. Although computing and exchanging

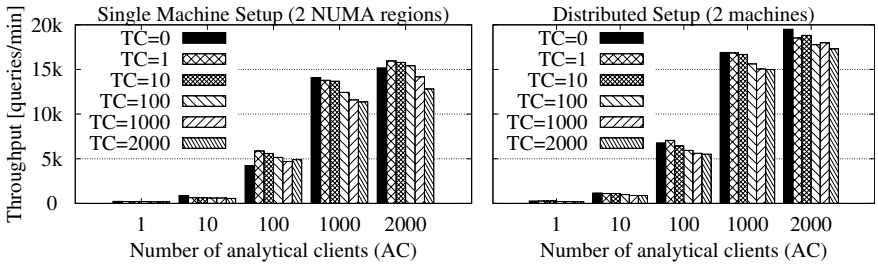


Figure 2: BatchDB isolates workloads generated by analytical clients (AC) and transactional clients (TC) through data replication. Cross-machine replication reduces interference and improves overall throughput.

these histograms can be done with great efficiency once the input is materialized, this process is difficult in a query pipeline in which tuples are streamed through non-blocking operators. Therefore, future RMA systems should extend their functionality beyond simple read and write operations. For example, an append operation, which would sequentially populate a buffer with the content of different operations, would reduce the complexity of the histogram computation phase. In addition, a remote memory allocation mechanism would eliminate the requirement to compute the size of each RDMA buffer. Once an RDMA buffer is full, a remote process could allocate new memory regions as needed.

3.2 Data Replication and Transformation

BatchDB [23] is a database engine designed to reduce the interference caused by hybrid workloads running in the same system while maintaining strict guarantees with respect to performance, data freshness, consistency, and elasticity. To that end, BatchDB uses multiple workload-specific replicas. These copies of the data can either be co-located on the same machine or distributed across multiple compute nodes. Transactions operate exclusively on a write-optimized version of the data, i.e., the primary copy. Updates to this component are propagated to the satellite replicas. The replicas are responsible for converting the data into their respective format, applying the updates, and signaling a management component that the updates have been applied. It is worth noting that this design is not limited to traditional database workloads but can be extended to new kinds of data processing, e.g., machine learning, by adding additional specialized replicas. In order to meet strict data freshness requirements, low-latency communication is essential. Since the updates need to be propagated to all replicas, having sufficient bandwidth on the machine hosting the primary copy is important.

In Figure 2, we observe that BatchDB maintains the performance of analytical queries in the presence of transactional workloads, independent of whether the replicas are located on the same machine with two NUMA regions or on two different machines (one replica per machine) where the updates are propagated through an InfiniBand network. At the same time, the transactional throughput is maintained as more analytical queries are executed [23]. We observe that the RDMA transfer does not negatively effect the performance of the system. In fact, the distributed setup has a higher throughput as the system benefits from better workload isolation.

BatchDB uses a shared scan in its analytical component. The scan consists of an update and read pointer. The update pointer is always ahead of the read pointer, which ensures that updates are first applied to the data before it is read. Incoming updates and queries are queued while the scan is running. A new batch of updates and queries is dequeued at the beginning of each scan cycle. Because the thread scanning the data is not involved in the data transfer and the update propagation latency is lower than the scan cycle, the performance of the analytical component can be maintained.

To keep the bandwidth requirements at a minimum, the transactional component only forwards the attributes of the tuples that have changed. A reliable broadcast, a mechanism currently not offered by all networks and interfaces, could speed up the update propagation process when multiple replicas are attached.

In BatchDB, both components use a row-store format. However, we expect that this will not to be the case for future workload-specific replicas. To ensure that the system can be extended, it is the responsibility of each replica to convert the change set, which is sent out in row-store format, to its internal representation. Given that this transformation might involve a significant amount of processing, it could impact the performance of the satellite component. To that end, we propose that future networking technology enables the destination node to push down simple rewrite rules to the network card. The networking hardware should be able to change the data layout while writing the incoming updates to main memory.

Low-latency data and state replication mechanisms have many applications. In BatchDB, replication is used to achieve performance isolation of OLAP and OLTP workloads, but such a mechanism could also be used to increase fault-tolerance and prevent data loss. Transforming data while it is transmitted is a general mechanism which is useful to any system that requires different data formats during its processing.

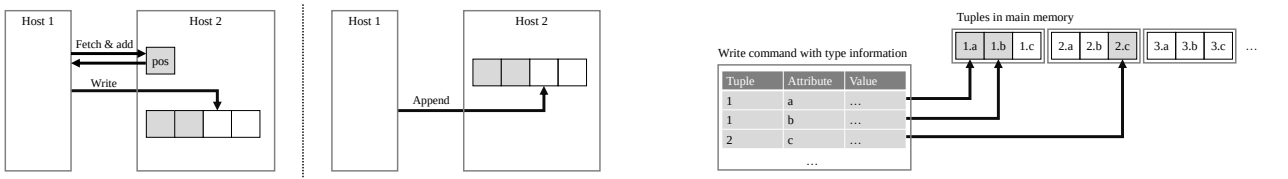
3.3 Distributed Coordination

Distributed transaction processing requires cross-machine coordination and synchronization. Two fundamental coordination mechanisms used in relational databases are (i) multiple granularity locking and (ii) multi version concurrency control. Many RDMA implementations offer remote atomic compare-and-swap and fetch-and-add operations. These operations can be used to implement a simple mutual exclusion semaphore. However, the locking system of a database does not only rely on mutexes but uses a more sophisticated protocol.

In multiple granularity locking, locks are acquired on objects in a hierarchical way, i.e., before a lock can be taken on an object, all its parent nodes need to be locked. In addition to shared and exclusive locks, these locking protocols use intention shared and intention exclusive locks, which indicate that a transaction intends to acquire shared, respectively exclusive, locks on one or more child nodes. Each lock consists of a request queue and a granted group. The queue contains all pending requests which have not been granted, and the granted group is the set of granted requests. The head element of the queue can be added to the granted group if it is compatible with all the other lock types in the group. Locks should always be taken on the finest level of granularity possible in order to favor parallelism and to enable as many concurrent data accesses as possible [15].

To use this locking scheme in a distributed system, one requires (i) an efficient way to add a new request to a remote request queue and (ii) a method to check if the head of a queue is compatible with all the elements in the granted group. The former can be implemented using a buffer into which the content (e.g., lock type, requesting process id) of a new request will be written. A process finds a new slot in the buffer by issuing a remote fetch-and-add operation to a tail counter. Once the operation has succeeded, the content is written to a queue. For the latter, we need to avoid scanning through all the granted requests. Instead of materializing the granted group in main memory, each lock contains a summary of the group state in the form of counters, one counter per lock type. A process reads these counters to determine if there is at least one incompatible request in the granted group. These counters are atomically updated whenever a requests enters or leaves the granted group. In order to avoid polling on the head element of the queue and the lock counters, the check is performed only when a new request is inserted into an empty queue and whenever a process releases a lock. The process releasing a lock determines if the head element is compatible and notifies the corresponding process.

The drawback of the above approach is that multiple round trip times are required to acquire even uncontended locks. Unless lock requests are issued ahead of time, the transaction or query is blocked and cannot continue processing. Although the remote memory access latency is constantly improving (micro-second latency), it is still significantly higher than the time required to access local memory (nano-second latency). In the case of locking, slow memory access times lead to reduced application-level performance. This performance problem is amplified through the hierarchy of the locks. To acquire a low-level lock, a process needs to be granted a series of locks (i.e., the path from the root to the lock). Combined with the fact that multiple round-trips are needed to acquire each lock, we expect the network latency and the message rate of the network card to have a significant impact on performance.



(a) Remote append operations would enable efficient concurrent accesses to the same buffer.

(b) Write operations with type information would enable more sophisticated memory accesses.

Figure 3: RDMA operations with high-level semantics.

Having more advanced RMA operations at our disposal would simplify the implementation sketched above and eliminate several round trips. As shown in Figure 3a, an append functionality would be useful to add new requests to a remote queue. If simple comparisons could be pushed to the remote network card, the checks if two requests are compatible could be executed directly on the node where the lock resides.

We expect locks at the top of the hierarchy to be more contended than low-level locks. Having a small number of highly contended locks has been a significant problem in many high-performance computing (HPC) applications. Schmid et al. [31] propose a combination of several data structures to implement scalable RMA locks distinguishing between readers with shared access and writers needing exclusive access.

As an alternative to locking, snapshot isolation is used in many database engines. One advantage of this approach is that queries, i.e., read-only transactions, do not need to be synchronized with other concurrent transactions. Locks are only needed to resolve write conflicts at commit time. This advantage does not come for free but requires careful memory management and version control. Following pointers to fetch a specific version of a record can cause a significant amount of random remote memory accesses, which would lead to poor performance. Instead, the memory needs to be laid out such that a remote process can compute the memory location of a specific version of a record and retrieve it with a single request. Each query and transaction needs to receive the current version number. When committing data, the snapshot number needs to be updated. Using a global counter for managing the snapshot versions will be an inherent bottleneck of the system, as it represents a point of synchronization. Timestamp vectors composed out of multiple snapshot numbers, one for each thread executing the transaction or query, can be used to create a scalable timestamp oracle [32].

4 Future Research Directions

In this section, we look at the future development of these technologies and propose a set of network primitives tailored to the needs of a data processing system. We argue that data processing systems and networks need to be co-designed to facilitate data movement between compute nodes, while at the same time keeping the network interface generic enough to support a variety of communication-intensive applications.

4.1 Fine-Grained Network Scheduling

When using RDMA, the operating system is not in control of the exact time when a data transfer is executed. The application is often aware of the communication pattern, e.g., an approximate amount of data that needs to be transmitted, and the nodes involved in the data exchange. In the case of join algorithms, the number of exchanged tuples is determined through the histograms and the algorithm is aware that, in our experiment setup, all nodes are involved in the communication. The sort-merge join uses a course-grained communication schedule by first partitioning the data into ranges and then letting each process i start sorting at range $i + 1$, resulting in a pairwise all-to-all communication pattern. This communication pattern reduces contention on both the sender and receiver. However, with smaller ranges and a larger number of nodes, pairwise communication

without explicit synchronization becomes difficult to maintain. Furthermore, not every algorithm can schedule its communication in a course-grained way. Maintaining synchronization for several thousand CPU cores with thousands of small buffers requires having a light-weight scheduling mechanism.

Implementing such a scheduling mechanism at the application level is difficult, as the data transfer is executed asynchronously by the network card. In order to enable the network to make fine-grained decisions when to execute a data transfer, the application needs to push down information about the nodes involved in the communication, the amount of incoming/outgoing data, and the granularity of the transmission, e.g., the average buffer size used during the communication. Such an interface could be enhanced further with information about the priority of the transmission and the required quality of service in terms of latency and bandwidth.

4.2 Advanced One-Sided Operations

Modern network implementations offer one-sided read and write operations as well as a hand-full of additional operations (e.g., atomic operations). To facilitate the development of future systems, this set of operations needs to be extended. The primitives offered by the network should be designed to meet the needs of a variety of data processing applications. Such a Network Instruction Set Architecture (NISA) [12, 18] should contain operations which manipulate, transform, and filter data while it is moving through the network.

For example, a conditional read operation (i.e., a read with conditional predicate) could be used to filter data at the source and avoid transmitting unnecessary data entries. With the current state of the art, the entire content of a remote buffer is fetched over the network before the processor can filter the data and copy only the relevant entries. A conditional read operation on the other hand would drop elements as the data is flowing through the remote network card. This would eliminate the need for a second pass over the data, lowering the overall CPU costs of such a transfer-filter combination. A database would be able to push a selection operator directly into the network. Furthermore, such operations would be crucial for systems using snapshot isolation, as the data could also be filtered based on the snapshot number, making it straightforward to read consistent versions.

Many databases store their data in compressed format. On-the-fly compression and de-compression could be done by the network card as data is read from or written back to remote memory, thus eliminating the CPU overhead of compression, avoiding unnecessary copy operations, and reducing the overall storage requirements. Such a functionality is not only important when accessing main memory, but also in systems where data is directly accessed from persistent storage via one-sided operations, e.g., RDMA over fabrics.

As the data is passing through the network card, simple aggregation operations and histogram computations can be performed on the fly. Recent work on database hardware accelerators has shown that database tables can be analyzed as they are retrieved from persistent storage and histograms on the data can be computed at virtually no extra performance cost [21]. Furthermore, network cards could be used to offload many data redistribution operations, e.g., the radix partitioning of the hash join can be implemented in hardware [22].

Synchronizing multiple processes intending to add data to the content of a common buffer is a difficult task often involving multiple network operations or extensive processing. As explained in Section 3.3, adding a new lock request to a list requires acquiring a free slot, followed by a write operation which places the data in that memory location. Both join algorithms use a global histogram computation which allows the processes to compute the location of memory sections into which each process can write exclusively. Having a modified write operation that does not place data at a specific memory address, but rather appends it next to the content of a previous operation would improve performance and reduce application complexity.

One-sided read and write instructions are unaware of the data type they are operating on. However, many analytical queries are only interested in specific attributes of a tuple. Having data stored in column-major format is useful, as the operator only needs to access the specific memory regions where the desired attributes are stored. In a row-major format, data belonging to the same tuple is stored consecutively in memory. Although the majority of networks offer gather-scatter elements, in large databases, it is not feasible to create one gather-scatter element for each individual tuple. Specifying a data layout and the set of attributes that need to be read

would enable the network card to determine a generic gather-scatter access pattern. Only accessing the required attributes and transforming the data as it moves through the network corresponds to a remote projection.

It is important to note that pushing down type information is a more powerful mechanism than a simple access pattern in which the card alternates between reading b and skipping s bytes. For example, different attributes for different tuples could be consolidated into the same operation. As described in Section 3.2, many data replication mechanisms forward attribute-level change sets in order to not waste valuable bandwidth. With precise type information, the network card could directly update the corresponding attributes for each tuple individually as illustrated in Figure 3b.

4.3 Distributed Query Pipelines

In Section 3.1, we analyzed how RDMA can be useful in the context of a single database operator. The input data is fully materialized at the start of the experiment. Although a database has a cost-based optimizer which keeps statistics on the intermediate result sizes, it is difficult to precisely determine the data size produced by different operators in a complex pipeline. This is especially the case for non-blocking operator pipelines in which the intermediate data is often never fully materialized.

As pointed out in Section 4.2, more sophisticated one-sided operation would increase the flexibility of operators when dealing with dynamic data sizes unknown ahead of time, e.g., an append operation would avoid the histogram computation of the join algorithms. However, this does not eliminate the fact that, in many implementations, RDMA buffers need to be allocated and registered by the host on which the memory resides. Using such buffers as output buffers of remote up-stream operators is challenging, as the previous operator cannot allocate new memory in case the output is larger than expected. Therefore, current data processing systems either need to have a memory management system on every compute node which signals its remote counterpart to allocate more memory, or ensure that the RDMA buffers are of sufficient size. To overcome this limitation, future interconnects need to be able to allocate memory on remote nodes.

Operators inside a pipeline need to be synchronized at certain points in time. In particular, down-stream operators need to be made aware when new input data is available for processing. Many high-speed networks offer signaled write operations which notify the remote host of the RDMA transfer. If the data needs to be accessed through a read operation, the common wisdom is to use two-sided operations where the up-stream operator sends a message to the next operator in the query pipeline. Such a signaling mechanism leaves the previous operator in control when to notify the down-stream operator of the existence of new data. However, different types of operators have different data input access patterns. For example, a blocking pipeline operator has to wait for all the data to be ready before being able to continue processing, while a streaming operator can proceed when some amount of input data is available. One possible way to address this challenge is to introduce a delayed read operation which is executed only when certain conditions on the remote side are fulfilled, e.g., when at least a specified amount of data is ready for processing.

4.4 Future Run-Time Systems

The Message-Passing Interface (MPI) [25] is the de-facto standard interface for writing parallel computations for high-performance computing (HPC) applications [16]. Although the interface has been designed for large scale-out architectures, it can be used on a variety of different compute platforms, from laptops to high-end supercomputers. Since its release in 1994, the interface has been extended to support not only message passing primitives, but also provides support for one-sided operations.

Traditionally, databases and data processing systems employ low-level hardware features to optimize the processing, e.g., SIMD vector instructions. When it comes to the network interface, many systems bind to hardware-specific interfaces to achieve peak performance, making the application code not portable. To overcome this problem, we observed that recent work in the area of databases started using standard communication

libraries such as MPI instead of hand-tuned code [6, 11]. Given that MPI is an interface description, an MPI application can be linked against many different library implementations, each tailored to a specific network.

MPI in particular has been designed in the context of scientific HPC applications. Such applications are characterized by a high degree of parallelism (i.e., several thousand processes) and a finite lifespan (typically several minutes to several hours). This is in contrast to a database system, which is often designed with limited parallelism and an infinite up-time in mind. As such, the MPI interface is missing a couple of features crucial for database systems. For example, the interface does not allow the application to specify quality of service requirements. It can therefore not prioritize latency sensitive communication although many networks offer such mechanisms. Furthermore, fault-tolerance is important for mission critical systems such as databases, but the current version of MPI (i.e., MPI-3) does not provide adequate functionality to detect and cope with failures.

The advantage of an interface like MPI is that it provides a set of expressive high-level operations to the application programmer. Using operations that have a rich semantic meaning enables the library developer to reason about the intentions of the application and be able to choose the right set of network primitives and hardware-specific features in order to efficiently implement the network operations.

5 Conclusions

In this article, we have provided an overview of high-performance networks and discussed the implications of these technologies on modern databases and data processing systems by looking at three use cases: distributed join algorithms, data replication, and distributed coordination. From the analysis, we conclude that recent advances in network technologies enable a re-evaluation and re-design of several system design concepts and database algorithms. For example, data processing systems have to interleave the computation and network communication, have to consider the data layout, and have to rely on careful memory management for the network to be efficiently used.

Despite their current potential, future networks need to include new functionality better suited to data-intensive applications. These features include: the ability to push down application knowledge to the network to enable smart scheduling; and more sophisticated one-sided operations that extend as well as complement the existing read and write operations. Dynamic memory management mechanisms such as a remote allocation operation would be useful for systems with complex processing pipelines in which the size of intermediate results is not known ahead of time. More research is needed to identify all features and to define suitable interfaces.

Finally, we expect that future high-level communication libraries will include additional functionality required by mission-critical infrastructure such as databases.

References

[1] M.-C. Albutiu, A. Kemper, T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*. 5(10):1064–1075, 2012

[2] B. Alverson, E. Froese, L. Kaplan , D. Roweth. Cray XC Series Network. *Cray Inc. Whitepaper*, 2012

[3] C. Balkesen, J. Teubner, G. Alonso, M. T. Özsu. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE TKDE*, 27(7):1754–1766, 2015

[4] C. Balkesen, J. Teubner, G. Alonso, M. T. Özsu. Main-memory Hash Joins on Multi-core CPUs: Tuning to the underlying hardware. *ICDE*, 362–373, 2013

[5] C. Balkesen, G. Alonso, J. Teubner, M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1): 85–96, 2013

[6] C. Barthels, I. Müller, T. Schneider, G. Alonso, T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 10(5):517–528, 2017

[7] C. Barthels, S. Loesing, G. Alonso, D. Kossmann. Rack-Scale In-Memory Join Processing using RDMA. *SIGMOD*, 1463–1475, 2015

[8] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian. The End of Slow Networks: It's Time for a Redesign. *PVLDB*, 9(7):528–539, 2016

[9] B. Carlson, T. El-Ghazawi, R. Numrich, K. Yelick. Programming in the PGAS Model. *SC*, 2003

[10] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. D. Steinmacher-Burow, J. J. Parker. The IBM Blue Gene/Q Interconnection Network and Message Unit. *SC*, 26:1–10, 2011

[11] A. Costea, A. Ionescu, B. Raducanu, M. Switakowski, C. Bârca, J. Sompolski, A. Luszczak,M. Szafranski, G. de Nijs, P. A. Boncz. VectorH: Taking SQL-on-Hadoop to the Next Level. *SIGMOD*, 1105–1117, 2016

[12] S. Di Girolamo, P. Jolivet, K. D. Underwood, T. Hoefler. Exploiting Offload Enabled Network Interfaces. *IEEE MICRO*, 36(4):6–17 ,2016

[13] P. W. Frey, R. Goncalves, M. L. Kersten, J. Teubner. A Spinning Join That Does Not Get Dizzy. *ICDCS*, 283–292, 2010

[14] P. W. Frey, G. Alonso. Minimizing the Hidden Cost of RDMA. *ICDCS*, 553–560, 2009

[15] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. *Morgan Kaufmann*, 1993

[16] W. Gropp, T. Hoefler, R. Thakur, E. Lusk. Using Advanced MPI: Modern Features of the Message-Passing Interface. *MIT Press*, 2014

[17] J. Hilland, P. Culley, J. Pinkerton, R. Recio. RDMA Protocol Verbs Specification. *IETF*, 2003

[18] T. Hoefler. Active RDMA - new tricks for an old dog. *Salishan Meeting*, 2016

[19] InfiniBand Trade Association. InfiniBand Architecture Specification Volume 2 (1.3.1). *IBTA*, 2016

[20] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 (1.2.1) - Annex A17 RoCEv2. *IBTA*, 2014

[21] Z. István, L. Woods, G. Alonso. Histograms as a Side Effect of Data Movement for Big Data. *SIGMOD*, 1567–1578, 2014

[22] K. Kara, J. Giceva, G. Alonso. FPGA-Based Data Partitioning. *SIGMOD*, 2017

[23] D. Makreshanski, J. Giceva, C. Barthels, G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads. *SIGMOD*, 2017

[24] S. Manegold, P. A. Boncz, M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE TKDE*, 14(4):709–730, 2002

[25] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard - Version 3.0. *MPI Forum*, 2012

[26] F. D. Neeser, B. Metzler, P. W. Frey. SoftRDMA: Implementing iWARP over TCP Kernel Sockets. IBM Journal of Research and Development, 54(1):5, 2010

[27] F. Petrini, W. Feng, A. Hoisie, S. Coll, E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002

[28] O. Polychroniou, R. Sen, K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. *SIGMOD*, 1483–1494, 2014

[29] W. Rödiger, S. Idicula, A. Kemper, T. Neumann. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks. *ICDE*, 1194–1205, 2016

[30] W. Rödiger, T. Mühlbauer, A. Kemper, T. Neumann. High-Speed Query Processing over High-Speed Networks. *PVLDB*, 9(4):228–239, 2015

[31] P. Schmid, M. Besta, T. Hoefler. High-Performance Distributed RMA Locks. *HPDC*, 19–30, 2016

[32] E. Zamanian, C. Binnig, T. Kraska, T. Harris. The End of a Myth: Distributed Transactions Can Scale. *CoRR*, 2016

Rethinking Distributed Query Execution on High-Speed Networks

Abdallah Salama[†*], Carsten Binnig[†], Tim Kraska [†], Ansgar Scherp[*], Tobias Ziegler[†]
[†]Brown University, Providence, RI, USA
[*]Kiel University, Germany

Abstract

In modern high-speed RDMA-capable networks, the bandwidth to transfer data across machines is getting close to the bandwidth of the local memory bus. Recent work has started to investigate how to redesign individual distributed query operators to best leverage RDMA. However, all these novel RDMA-based query operators are still designed for a classical shared-nothing architecture that relies on a shuffle-based execution model to redistribute the data.

In this paper, we revisit query execution for distributed database systems on fast networks in a more holistic manner by reconsidering all aspects from the overall database architecture, over the partitioning scheme to the execution model. Our experiments show that in the best case our prototype database system called I-Store, which is designed for fast networks from scratch, provides $3\times$ *speed-up over a shuffle-based execution model that was optimized for RDMA.*

1 Introduction

Motivation: Distributed query processing is at the core of many data-intensive applications. In order to support scalability, existing distributed database systems have been designed along the principle to avoid network communication at all cost since the network was seen as the main bottleneck in these systems. However, recently modern high-speed RDMA-capable networks such as InfiniBand FDR/EDR have been become cost competitive. To that end, these networks are no longer used only for high-performance computing clusters, but also become more prevalent in normal enterprise clusters and data centers as well.

With modern high-speed RDMA-capable networks, the bandwidth to transfer data across machines is getting close to the bandwidth of the local memory bus [3]. Recent work has therefore started to investigate how high-speed networks can be efficiently leveraged for query processing in modern distributed main-memory databases [13, 12, 2, 1]. The main focus of this line of work so far was centered around the question of how to redesign individual distributed query operators such as the join to leverage RDMA to make best use of fast networks.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

However, none of the before-mentioned papers has revisited distributed query processing on fast networks in a more holistic manner. Instead, all the novel RDMA-based distributed algorithms are still designed for a classical shared-nothing architecture. In this classical database architecture, data is partitioned across the nodes, whereas each node has direct access only to its local partition. For processing a distributed join operator, data is first shuffled to other nodes based on the join keys before a local join algorithm (e.g., a radix join) is applied to the re-shuffled input. A main benefit of this processing model on slow networks is that it allows the distributed join operators to reduce the amount of data that is transferred over the network; e.g., by applying a semi-join reduction as a part of the data shuffling process. While reducing the bandwidth consumption is a valid reason for slow networks, it should not be the main design consideration when designing distributed query processing algorithms on fast networks. To that end, instead of relying on a shared-nothing architecture and a shuffle-based processing model that was designed to reduce the bandwidth consumption in slow networks, efficient distributed query execution for fast networks should rather focus on other aspects.

First, the main difference of a remote memory accesses in modern RDMA-capable networks compared to a local memory access is not the bandwidth, but the access latency, which is still an order of magnitude higher in the remote case. Thus, relying on an execution model that needs to shuffle the input data for every join operator in a query plan and therefore needs to pay the high latencies of remote data transfers multiple times per query is not optimal. Second, for modern scalable distributed database systems there are other important properties such as elasticity as well as load-balancing that are not naturally supported in a distributed database system that is built on a shared-nothing architecture and uses a shuffle-based execution model.

Contributions: The main contribution of this paper is that it is a first attempt to revisit query execution for distributed database systems on fast networks from the ground up.

First, instead of building on a classical shared-nothing architecture we build our distributed query engine *I-Store* on a more recent architecture for fast networks called the Network-Attached-Memory (NAM) architecture [3]. The main idea of the NAM architecture is that it logically decouples compute and memory nodes and uses RDMA for communication between all nodes as shown in Figure 1. Memory servers provide a shared distributed memory pool that holds all the data, which can be accessed via one-sided RDMA operations from compute servers that execute the queries. In this architecture, compute servers can be scaled independently from memory servers. Furthermore, a compute server can access the data that resides in any of the memory server independent of its location. To that end, elasticity as well as load-balancing can naturally be supported in this architecture.

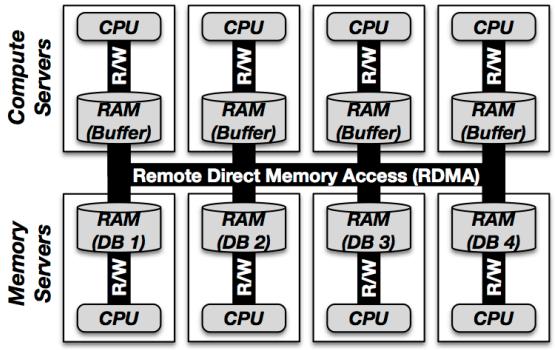


Figure 1: The NAM Architecture

Second, we propose a novel partitioning scheme for the NAM architecture. The main difference to existing partitioning schemes is that tables are split into partitions using multiple partitioning functions (i.e., one per potential join key). Furthermore, even more importantly, partitioning is not used to co-locate data (e.g., for joins) as in a classical shared-nothing architecture. Instead, the partitioning information is used to allow compute servers to efficiently stream over the data in the memory servers at runtime when executing a query.

Third, on top of this partitioning scheme we present an execution model called distributed pipelined query execution for analytical queries on fast networks. The main difference to the existing shuffle-based execution schemes is two-fold: (1) First, in a classical shuffle-based execution scheme intermediate data would need to be re-distributed before each join operator can be executed. In our scheme, we avoid shuffling of intermediates completely. The idea of our model is that we stream data from memory servers to compute servers at runtime while each compute server involved in the query executes a fully pipelined plan without any shuffling operators. In order to achieve this, we partially replicate data when streaming partitions to different compute servers and

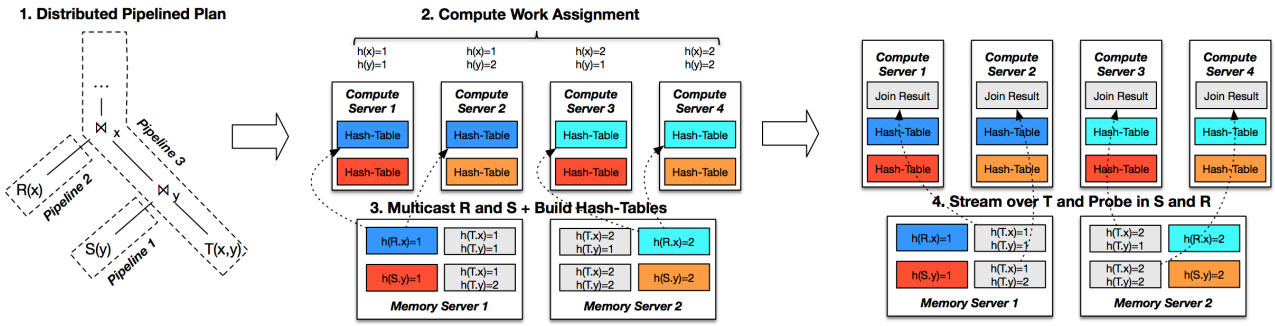


Figure 2: Overview of Distributed Query Execution in *I-Store*

thus trade bandwidth for minimizing the overall network latency of query execution; the main bottleneck in fast networks. (2) Second, our scheme can evenly distribute work across different compute servers and thus efficiently support elasticity and load-balancing.

Fourth, we have implemented all the before-mentioned ideas in a system called *I-Store*. At the moment, in *I-Store* we only support SPJA queries with equi-joins in *I-Store*, an important class of queries often found in OLAP workloads.

Outline: In Section 3, we first give an overview of the main ideas of our query execution scheme as well as how it interacts with the partitioning scheme. Afterwards, in Sections 3 and 4 we discuss the details of partitioning and execution scheme. In our evaluation in Section 5, we then analyze the different design decisions of our execution scheme using microbenchmarks and compare the runtime of our distributed pipelined execution scheme to a shuffle-based execution scheme for queries with a varying number of joins based on the SSB benchmark. Finally, we discuss related work and conclude in Sections 6 and 7.

2 Overview

The distributed query execution model in *I-Store* is based on the notion of a pipelined execution model as it is typically used in single-node parallel databases today, however, adapted for the NAM architecture as outlined before. The main idea is streaming data to multiple compute servers whereas each server executes a query plan in a pipelined manner without any need to shuffle intermediate data between the compute nodes. In order to achieve this, we partially replicate data when streaming data to different compute servers. In consequence, we initially trade some network bandwidth for minimizing the overall network latency; the main bottleneck in fast networks. Moreover, based on the partial replication scheme we can evenly distribute the work to all compute nodes involved in executing a query and thus achieve load balancing as well as elasticity.

Figure 2 gives an overview of all steps required for executing a SQL query that joins three tables using our distributed pipeline model. All steps when executing a distributed pipelined plan are supported by our multi-way partitioning scheme that splits tables by all their join keys using one fixed partitioning function per join key. In the example, we use a hash-based partitioning function (denoted as h): while table R and S are split by one join attribute (either x or y), table T is split by both join keys using two different partitioning functions that are iteratively applied to the table. The result of applying this partitioning scheme to the tables in our example can be seen in Figure 2 (center). Furthermore, partitions are split into smaller fixed sized blocks that enable an efficient load-balancing but can still be efficiently transferred via RDMA.

In the following, we discuss the query execution of the particular SQL query in Figure 2 on top of this partitioning scheme.

In *step 1* (Figure 2, left hand side), the given SQL query is compiled into a so called distributed pipelined plan. The execution of a distributed pipelined plan in parallel using multiple compute nodes is similar to the

parallel execution of a pipelined plan in a single-node using multiple threads. For the query in our example, a single-node pipelined plan would be executed as follows: First, the pipelines over tables R and S would be executed in parallel to build global hash tables that can be seen by all parallel threads. Once the hash tables are built, the last pipeline would be executed by streaming in parallel over table T and probing into the previously built global hash tables for R and S. While the general idea of a distributed pipelined plan is similar, there is an important difference: the pipelines to build and probe do not use a global hash table that can be accessed by any of the compute servers. Instead, in our distributed pipelined model, we partially replicate the data to all compute servers in a streaming manner such that joins can be executed fully locally on each compute server.

After compiling a SQL query into a pipelined plan, in *step 2* (Figure 2, center) we then evenly assign the work to all compute servers that participate in the query execution without yet copying the data over the network. Assigning work to compute servers is only a logical operation and does not involve any data movement yet. In Figure 2, for example, we assign the blocks of different partitions to the four different compute servers required to execute all joins fully locally; e.g., the blocks of the partitions denoted by $h(x) = 1$ and $h(y) = 1$ are assigned to compute server 1 while the blocks of the partition denoted by $h(x) = 1$ and $h(y) = 2$ are assigned to compute server 2 etc. If data is skewed, blocks that belong to the same partition can be assigned to different compute servers. Assigning work evenly to the available compute servers while minimizing replication is a part of our query optimizer to achieve load-balancing.

Based on the assignment, the data of R and S required to build local hash tables is replicated in a streaming manner to the compute servers as shown in *step 3* (Figure 2, center). For example, the partition of R denoted by $h(R.x) = 1$ is replicated to compute server 1 and 2 since both compute servers require this partition to execute the probing pipeline in the subsequent phase when streaming over the assigned T partitions. Furthermore, local hash tables are built while the data is streaming into the compute servers leveraging the fixed-sized blocks as transfer unit to overlap network communication and computation to further reduce network latencies. The result of this phase is shown in Figure 2 (center). It is important to note that in *I-Store* compute servers apply filters when building the hash tables; i.e., we do not push filters into memory servers to avoid that they become a bottleneck. We are also currently working on ideas for distributed indexing where compute servers cache the upper levels of indexes to read only the relevant blocks.

Once the hash tables are built in our example query, in the last *step 4* (Figure 2, right hand side) each compute server streams over the data of the T partitions that it was assigned to in step 2. In our example, compute server 1 fetches the data of the partition of table T denoted by $h(T.x) = 1$ and $h(T.y) = 1$. It is important to note that the compute servers again stream over the data of the partitions of T using the fixed-size blocks and start the probing into the previously created hash tables once the first block is transferred. Moreover, also note that for executing the probing pipeline in a compute server, data can also be replicated in case that T does not contain all join keys.

Finally, if an aggregation is in the query plan and the group-by key is not a join key, then one compute node collects the partial result of all the other compute nodes and applies a post-aggregation over the union of the individual results. This clearly contradicts our goal to not use a shuffle operator in a distributed pipelined plan. However, aggregation results are typically small and do not increase the overall query latency when transferred over the network. Moreover, if scalability of the aggregation is an issue we can also shuffle the data to multiple compute servers based on the group-by keys; i.e., we do not use aggregation trees since they again would increase the network latency.

3 Multiway-Partitioning and Storage Layout

As mentioned before, our partitioning scheme splits the tables of a given database based on all the possible join keys. As a first step in our partitioning scheme, we therefore define one partitioning function for each join key. This function is then used to split a table which contains that join key into n parts. For example, in Figure 2 we

use the function $h(x) = (x\%2) + 1$ to partition all tables that contain the join key x into two parts (which are tables R and T in the example). If a table contains more than one join key, we iteratively apply all the according partitioning functions to this table (e.g., table T contains x and y).

At the moment, we only support hash-based partitioning functions. However, in general we could also use other partitioning functions such as a range-based function as long as the output is deterministic (i.e., round robin is not supported). Moreover, the number of partitions a function creates should best reflect the maximum number of compute servers that are going to be used for query processing. That way, we can efficiently support elasticity by assigning different partitions to distinct compute servers.

Another important aspect, as mentioned before, is that partitions are split into smaller blocks of a fixed size that enable load-balancing but can still be efficiently transferred via RDMA. In our current system, we currently use a fixed size of 2KB for splitting partitions into blocks since this size allows us to leverage the full network-bandwidth in our own cluster with InfiniBand FDR $4\times$ as shown in a previous paper already [3]. The blocks that result after partitioning a table are distributed to the different memory servers. For distributing the blocks to memory servers, co-location does not play any role. Instead, it is only important to equally distribute all blocks of a table to different nodes to avoid network skew when transferring the table data to the compute servers.

Finally, for storing all blocks in the memory servers we use a distributed linked-list structure where each block logically stores a remote pointer to the next block as shown in Figure 3. A remote pointer consists of the node identifier and an offset in the remote memory region of the memory node the next block resides in. The main purpose for using a linked list is that for metadata handling we only need to know the head of the list instead of storing the addresses of all blocks of a table. However, we still want to enable that blocks can be efficiently retrieved by to compute servers using RDMA. In *I-Store*, the preferred access method of a compute server to retrieve blocks from memory servers is to use one-sided RDMA reads in order to not push computation to the memory servers.

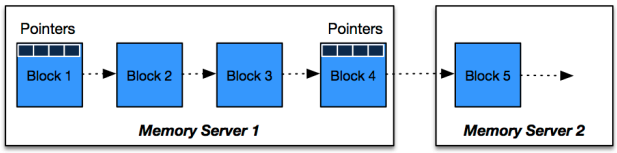


Figure 3: Storage Layout in Memory Servers

In order to avoid remote pointer chasing, which would increase the latency since we need to read one block before we can read the next block, we physically store pointers together in groups. For example, in Figure 3 every fourth block stores pointers for the next four blocks. This storage layout allows us, for example, to efficiently implement remote pre-fetching when streaming over the data of a table in a compute server; i.e., after fetching the first block into a compute server we can fetch the next 4 blocks in the background while streaming over the first block to execute a query pipeline.

4 Distributed Pipelined Query Execution

Based on our partitioning scheme, we can execute queries using our distributed pipelined scheme as outlined already in Section 3 . The two main tasks of this scheme are (1) compute a work assignment for each compute server and (2) based on the assignment execute the pipelines on each compute server by streaming data from memory severs to compute servers. In the following, we discuss the details for both tasks.

4.1 Computing the Work Assignment

The goal of computing the work assignment is that every compute server has to execute approximately the same amount of work to achieve a load-balanced execution. Otherwise, individual compute nodes might run longer and thus have a negative impact on the overall query runtime. In order to enable a balanced work distribution, we implement the following assignment procedure for a given partitioning scheme as shown in Figure 4 (left hand side). The example partitioning is based again on three tables R, S, and T with join attribute x and y as

before. Different from the partitioning used in the first example, the tables R and S are partitioned by a hash partitioning function h which splits the tables into 4 parts on either x or y. Moreover, table S is thus partitioned into 16 parts using the same partitioning functions, however, iteratively applied first partitioning function to x and then splitting the output using the partitioning function on y.

Based on the partitioned data and a given pipelined plan, we compute the assignment as follows:

Assume that we want to execute the same query as shown already in Figure 2 (left hand side) over the partitioning given by Figure 4 (left hand side). First, for every possible combination of partitions that need to be shipped to a compute node together to enable a fully local pipelined execution, we estimate the execution cost. For example, the partition of R with $(h(x) = 1$ (blue partition 1), the partition , $h(y) = 1)$ of S (red partition 1) as well as the partition $(h(x) = 1, h(y) = 1)$ of T (grey partition 1.1) must be shipped to the same compute server the join query can be executed fully locally on that compute node. At the moment, we leverage a very simple cost function to compute the execution cost for a set of partitions, which only reflects the total size of the partitions that need to be transferred together to a compute node. In future, we plan to include not only the data transfer cost but also the cost of executing the pipelined query plan over the transferred partitions.

Second, based on the computed costs we then assign the partitions to a given number of compute nodes such that every compute node gets approximately the same amount of work. In case all partitions have the same size (i.e., if data is distributed in a uniform way), the assignment becomes trivial as shown in Figure 4 (right hand side). However, computing the assignment under data skew where individual partitions have different sizes becomes more complex. Actually, computing the work assignment under data skew into a given number of compute servers essentially is a k-partitioning problem where k is the number of compute servers and the costs per partition combination represent the numbers in the set that needs to be divided equally. Since this problem is NP-hard, we plan to leverage approximate solutions based on heuristics (e.g., the largest differencing method) to solve the assignment problem. Furthermore, since partitions are further split into smaller fixed size blocks, we also plan to use these blocks to readjust the assignment to achieve a fairer distribution under heavy skew after applying the heuristics.

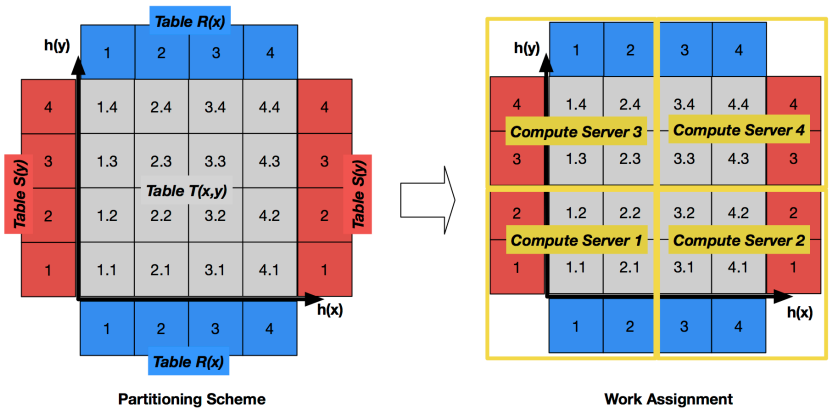


Figure 4: Computing the Work Assignment

4.2 Streaming Data to Compute Servers

Once the work assignment is computed, each compute server executes the pipelines in a query plan over the partitions it was assigned to. An example of a pipelined plan was already shown in Figure 2 (left hand side). For executing the pipelines in the plans, we transfer the partitions to the compute servers in a streaming manner such that pipelines can be executed in compute servers while data is streaming in from memory servers.

The methods we use for streaming partitions to compute servers can be differentiated into two cases: (1) Partitions are streamed to multiple compute servers, (2) Partitions are streamed to one compute server. For example, to execute the pipelines 1 and 2 in Figure 2 (center), each partition of R and S is streamed to two of the four compute servers. However, for the partitions of T, no replication is required to execute pipeline 3. In order to implement these two cases in a streaming manner, we using different techniques as explained next.

Streaming to one Compute Server: Streaming data from memory servers to one compute servers (i.e.,

without replication) is implemented in *I-Store* using one-sided RDMA reads that are issued by the compute server. For example in Figure 2 (right hand side) the compute server 1 issues RDMA reads to retrieve the data for partition 1.1. The main reason is that the CPU of the memory server is not involved in the data transfer and thus does not become a bottleneck when we scale-out the compute servers to support elasticity. The advantage of using one-sided operations to scale-out computation in the NAM architecture has been shown in a previous paper already [3]. Streaming is implemented on a block basis; i.e., each block is retrieved by a compute server using a separate RDMA read.

Furthermore, we additionally implement pre-fetching to further reduce the latency of retrieving each block as shown in Figure 5. We therefore reserve a read buffer for n blocks and leverage the queue-based programming model of RDMA where a read request can be separated from the notification of its completion. The main idea of pre-fetching is that a compute server requests the NIC to read the next n blocks without waiting for the completion event (CE) for the RDMA reads. Instead, the compute server only polls for completion event (CE) in the completion queue of the NIC when it needs to process the next block for executing a query pipeline (e.g., to apply a filter predicate on all tuples in the block and add them to a hash table). While the compute server is executing the query pipeline for the current block, the NIC can process further outstanding read requests in the background. Moreover, once a block is processed by the query pipeline executed by a compute server, the compute server can request the NIC to read the next block into the free slot in the read buffer again without waiting for its completion. For more details about the queue-based programming model supported by RDMA, we refer to [3].

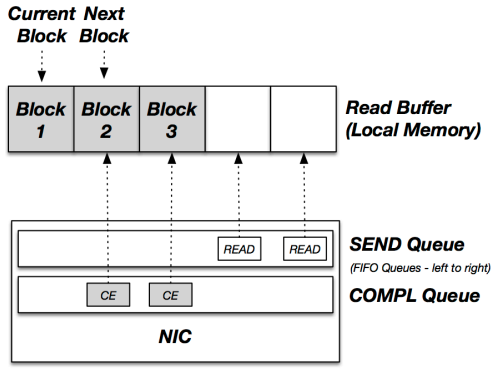


Figure 5: Prefetching using RDMA Reads

Streaming to Multiple Compute Servers: Streaming data to multiple compute server can be implemented using the same technique as discussed before by multiple compute servers. For example, in Figure 2 (center) the compute server 1 and 3 need to read the the same partitions of table R that is denoted by $h(R.x) = 1$ (red). To that end, compute server 1 and 3 can use the same technique as discussed before to stream the partition independently to both compute servers using RDMA reads. However, when both compute servers read the same partition from memory server 1, the available bandwidth on the memory server drops by half for each compute server as shown in Figure 6 (left hand side). The reason is that the same blocks of the partition $h(R.x) = 1$ (red) needs to be transferred from memory server 1 to both compute servers. This gets even worse, if we further scale-out the number of compute servers and thus need to replicate the same partition to more compute servers.

To that end, we use RDMA multicast operations in *I-Store* to stream partitions if they are required by multiple compute servers. The advantage when using multicast operation is that the memory server only needs to transfer the same blocks once to the switch which then replicates the blocks to all compute servers in the multicast-group. The only disadvantage is that multicast operations in RDMA are two-sided (i.e., the memory server's CPU is actively involved in the data transfer). However, when using multicast operations, each block of a partition only needs to be send once independent of

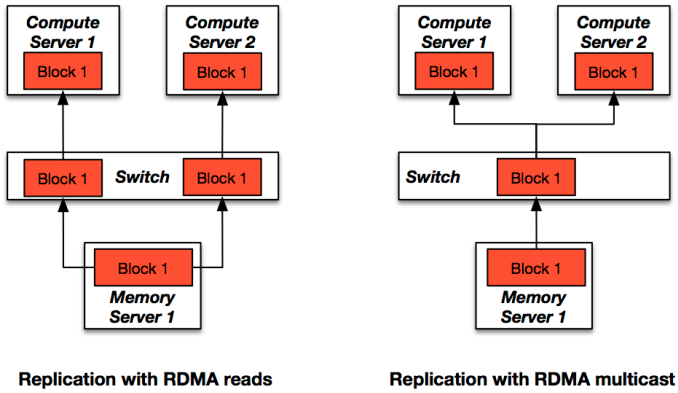


Figure 6: RDMA Read vs. Multicast

the number of compute servers which receive the block as shown in Figure 6 (right hand side). Thus, using two-sided multicast operations does not limit the scalability of our processing model. Finally, it is important to note that for multicast operations, we can implement a similar prefetching technique as described before for RDMA reads. The only difference is that both sides (compute and memory servers) must be involved in the prefetching implementation; i.e., a memory server needs to register multiple RDMA send operations at its NIC while the compute server needs to register multiple RDMA receive operations at its NIC.

5 Experimental Evaluation

In this section, we present the results of our initial evaluation of our distributed query execution engine implemented in *I-Store*. For the evaluation we executed different experiments: (1) In a first experiment, we show the end-to-end performance when running analytical queries. The main goal of this experiment is to compare our distributed pipelined model to a classical shuffle-based model. (2) In a second experiment, we discuss the results of different microbenchmarks that we executed to validate our design decisions (e.g., using RDMA multicast operations vs. reads).

For executing all the experiments, we used a cluster with 8 machines connected to a single InfiniBand FDR 4X switch using a Mellanox Connect-IB card. Each machine has two Intel Xeon E5-2660 v2 processors (each with 10 cores) and 256GB RAM. The machines run Ubuntu 14.01 Server Edition (kernel 3.13.0-35-generic) as their operating system and use the Mellanox OFED 2.3.1 driver for the network.

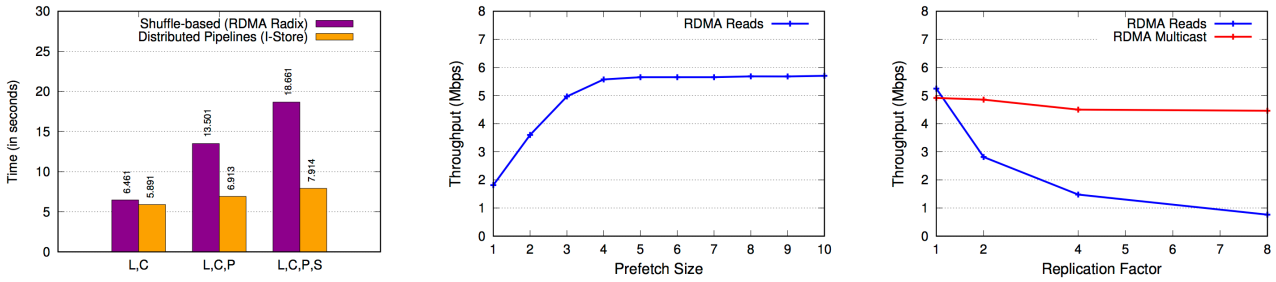
5.1 Exp. 1: System Evaluation

As data set, we used the schema and data generator of the Star Schema Benchmark (SSB) [10] and created a database of $SF = 100$. Afterwards, we partitioned the database using our partitioning scheme as follows: The individual dimension tables (`Customer`, `Parts`, `Supplier`) were hash-partitioned into 8 parts using their primary keys (which is the join key). We did not include the `Date` table into our queries since it is small and thus could easily be replicated at runtime entirely to all compute nodes. The fact table was partitioned into into $8^3 = 256$ partitions using the join keys of the three before-mentioned dimension tables.

As a workload, we executed queries with a different number of joins. We started with a query that joins only two tables; the fact table and the `Customer` (C) table. Afterward, we subsequently added one more of the dimension tables (i.e., `Parts` (P), `Supplier` (S)) to the query. For executing the queries in *I-Store*, we used 8 memory and 8 compute servers (i.e., one memory and compute server were co-located on one physical node) and used 10 threads per compute server for query execution.

In order to compare our approach to a shuffle-based execution model that needs to re-distribute the inputs for every join operator, we used the code that is available for the RDMA radix join as presented in [2]. The RDMA radix join shuffles the data in the first partitioning phase over the network. Since the code is designed to join only two tables, we simulated queries with more than two tables by running the algorithm multiple times on different input tables without including to the cost of materializing the join output. For executing our workload using the RDMA radix join, we partitioned all tables into 8 partitions using a hash-based scheme and distributed the partitions to all 8 nodes in our cluster. Furthermore, we configured the join algorithm to use 10 threads per node as for our join. For loading the data, we modified the code of the RDMA radix join to use the key distribution of the before-mentioned partitioned SSB tables and the tuple-width as specified by the schema of the SSB benchmark [10].

The result can be seen in Figure 7a. We see that both approaches have a similar runtime for a binary join query. The reason is that both approaches need to transfer approximately the same amount of data over the network. However, since we stream data to compute nodes and overlap the pipeline execution with the data transfer, we can achieve a slightly better performance. Furthermore, for queries which join 3 or 4 tables our



(a) Query Execution (b) Effect of Prefetching (c) Reads vs. Multicast

Figure 7: Experimental Evaluation of *I-Store*

approach shows a clear benefit: While the RDMA radix join needs to shuffle data for every join, we can fully stream the pipelines without any further shuffling which results in a performance gain of $3\times$ in the best case.

In this experiment, only one of the joined tables in a query was large while all other tables where relatively small. We believe that this is a common in most data warehousing scenarios with fact and dimension tables. In future, we plan to also look into large-to-large table joins.

5.2 Exp. 2: Microbenchmarks

In this experiment we validated the efficiency of our decisions for streaming data from memory servers to compute servers in *I-Store* to enable a pipelined query execution in the compute servers.

Exp. 2a - Streaming to one Compute Server: First, we analyzed the approach when streaming data to one compute server (i.e., blocks do not need to be replicated) and compared a streaming approach using a varying number of prefetched blocks starting with only 1 block (no pre-fetching) up to 10 blocks. As block size we used $2KB$ and the tuple width was set to $8B$ (i.e., one large integer). Moreover, on top of the incoming data stream we executed a simple scan pipeline that reads each incoming block and computes a sum operator over the $8B$ integer values for all tuples in the block. The results can be seen in Figure 7b where the y-axis shows the number of scanned blocks per second (million blocks per second, Mbps) for different pre-fetch sizes. We can see that compared to no pre-fetching (size=1) we can gain a speed-up of up to factor $3\times$ using pre-fetching (size=10). Moreover, pre-fetching more than 10 blocks also does not seem to further increase the throughput in that case. In fact, even when pre-fetching only 5 blocks we already reach almost the maximum throughput in this experiment.

Exp. 2b - Streaming to multiple Compute Servers: Second, we also analyzed the effect of using RDMA reads vs. RDMA multicasts to stream blocks to multiple servers that require replication. We varied the replication factor from 1 to 8 to show the effect of streaming blocks from 1 memory server to a maximum number of 8 compute servers. For simulating query pipelines, we used the same setup as before: As block size we used $2KB$ and the tuple width was $8B$ (i.e., one large integer). Moreover, on top of the incoming data stream we executed the same simple scan pipeline as before. The results can be seen in Figure 7c: the x-axis shows the number of receiving compute servers and the y-axis shows the average number of scanned blocks (million blocks per second, Mbps) per compute server. We can see that when using RDMA reads the throughput in a compute server drops linearly from $5m$ to $0.8m$ when increasing the number of compute servers involved. However, when using RDMA multicasts the throughput remains constant at around $4m$ which is a little lower in case no replication is required but significantly higher when replication is required (from 2 to 8 compute nodes in this experiment).

6 Related Work

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 [13, 11, 9, 5, 6]. Some recent work has investigated building RDMA-aware DBMSs [14, 15] on top of RDMA-enabled key/value stores [7], but again query processing in these systems comes as an afterthought instead of first-class design considerations.

Furthermore, the proposed ideas for RDMA build upon the huge amount of work on distributed processing (see [8] for an overview). Similar to our work, other work [12, 2, 1] also discussed distributed query processing for RDMA but focused only on the redesign of individual algorithms without rethinking the overall query processing architecture. For example, [12, 2] implements RDMA-based versions of two existing join algorithms (an RDMA radix join as well as an RDMA sort merge join). Furthermore, [12] adapts the shuffle operator of a classical shared-nothing DBMS but includes some interesting ideas of how to handle data skew and achieve load-balancing using this operator. Finally, SpinningJoins [4] also make use of RDMA. However, 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).

7 Conclusions

In this paper we revisited query execution for distributed database systems on fast networks. While recent work has already started to investigate how to redesign individual distributed query operators to best leverage RDMA, in this paper we reconsidered all aspects from the overall database architecture, over the partitioning scheme, to the query execution model. Our initial experiments showed that in the best case our prototype system provides $3\times$ speed-up over a shuffle-based execution model that was optimized for RDMA. In future, we plan to investigate aspects such as elasticity as well as efficient load-balancing in more depth since these properties are natural to our execution model.

References

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

[2] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1463–1475, 2015.

[3] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[4] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 283–292, 2010.

[5] N. S. Islam, M. Wasi-ur-Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of HDFS over infiniband. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 35, 2012.

[6] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance RDMA capable interconnects. In *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*, pages 743–752, 2011.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306, 2014.

[8] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[9] 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. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 641–650, 2013.

[10] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*, pages 237–252, 2009.

[11] A. Pruscino. Oracle RAC: architecture and performance. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 635, 2003.

[12] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205, 2016.

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

[14] 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 Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 182–190, 2014.

[15] C. Tinnefeld, D. Kossmann, M. Grund, J. Boese, F. Renkes, V. Sikka, and H. Plattner. Elastic online analytical processing on ramcloud. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 454–464, 2013.

Crail: A High-Performance I/O Architecture for Distributed Data Processing

Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle,
Radu Stoica, Bernard Metzler, Nikolas Ioannou, Ioannis Koltsidas
IBM Research Zurich

Abstract

Effectively leveraging fast networking and storage hardware for distributed data processing remains challenging. Often the hardware integration takes place too low in the stack, and as a result performance advantages are overshadowed by higher layer software overheads. Moreover, new opportunities for fundamental architectural changes within the data processing layer are not being explored. Crail is a user-level I/O architecture for the Apache data processing ecosystem, designed from the ground up for high-performance networking and storage hardware. With Crail, hardware performance advantages become visible at the application level and translate into workload runtime improvements. In this paper, we discuss the basic concepts of Crail and show how Crail impacts workloads in Spark, like sorting or SQL.

1 Introduction

In recent years, we have seen major performance improvements in both networking and storage hardware. Network interconnects and fabrics have evolved from 1-10Gbps bandwidth ten years ago to 100Gbps today. Storage hardware has undergone similar improvements while moving from spinning disks to non-volatile memories such as flash or Phase Change Memory (PCM). These hardware changes have also brought up the need for new I/O interfaces. Traditional operating system abstractions like sockets for networking or the block device for storage were shown to be insufficient to make the rich semantics and high performance of the hardware available to applications. Instead, new interfaces such as RDMA[1] or NVMe have emerged. These interfaces enable user-level hardware access and asynchronous I/O and allow applications to take full advantage of the high-performance hardware. Table 1 compares the performance of some of the new I/O technologies with the performance of legacy software/hardware interfaces. As can be observed, the performance improvements are in the range of 100-1000x.

Meanwhile, systems for large scale data processing are confronted with increasing demands for better performance while applications are becoming more complex and data sizes keep growing. Here, the new I/O technologies present opportunities for data processing systems to further reduce the response times of analytics queries on large data sets. In particular, performance critical I/O operations, such as data shuffling or sharing of data between jobs or tasks, should benefit from faster I/O. Today, examples of applications that are capable

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]Commodity RDMA network interconnects have been around since early 2000 and have their roots in user-level I/O from the 90s. Only recently they have gained acceptance in the broader data center community.

	RTT (us)	Throughput (MBit/s)
1GbE / sockets	82	942
100GbE / verbs	2.4	92,560

(a)

	Latency (us)		Throughput (MB/s)	
	read	write	read	write
Disk / Block device	5,978	5,442	136.8	135.3
NVMe Flash / SPDK	64.2	9.18	2,657	1,078

(b)

Table 1: I/O performance legacy vs. state-of-the-art for (a) networking (b) storage.

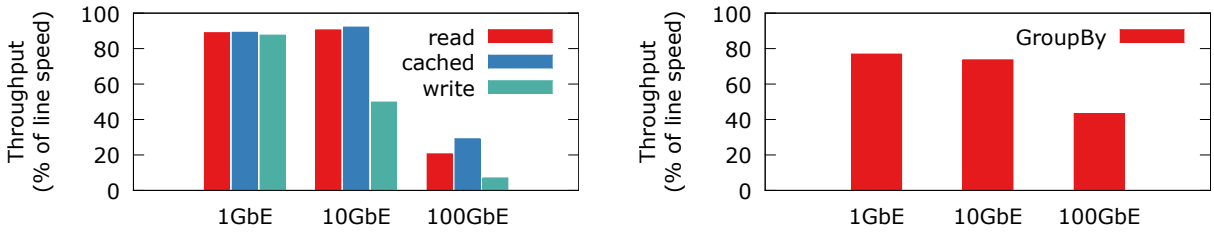


Figure 1: HDFS remote read performance (left) and Spark groupBy() performance (right). HDFS deployment is on NVMe SSDs, Spark deployment is completely in memory.

of leveraging fast I/O hardware can be found in supercomputing, but also more recently in several academic efforts [11, 3, 12]. These applications, however, are written from scratch to serve one particular workload. Unfortunately, properly leveraging high-performance networking and storage hardware in general purpose data processing frameworks, such as Spark, remains challenging for a number of reasons:

Deep Layering: Often hardware integration takes place too low in the stack, and as a result performance advantages are overshadowed by higher layer software overheads. This problem gets amplified with today's prominent frameworks such as Spark, HDFS, Flink, etc., due to the heavy layering these systems employ. Besides involving the host operating system for each I/O operation, data also needs to cross the JVM boundaries and the specific I/O subsystems of the frameworks. The long code path from the application down to the hardware leads to unnecessary data copies, context switches, cache pollution, etc., and prevents applications from taking full advantage of the hardware capabilities. For instance, writing out an object to the network during a Spark shuffle operation requires no less than 5 memory copies. The effect of those overheads at the application level is illustrated in Figure 1, where we show examples of the I/O performance in HDFS and Spark for different network technologies. As can be observed, while both systems make good use of the 1 and 10Gbps network, the network is underutilized in the 100Gbps configuration as the software becomes the bottleneck. A more detailed analysis of software related I/O bottlenecks in data processing workloads can be found in [26].

Data locality: The improved performance of modern networking and storage hardware opens the door to rethinking the interplay of I/O and compute in a distributed data processing system. For instance, low latency remote data access enables schedulers to relax the requirements on data locality and in turn make better use of compute resources. At the extreme, storage resources can be completely disaggregated which is more cost effective and simplifies maintenance. The challenge is in the first place to spot what opportunities the hardware provides, and then to exploit the opportunities in a way that is non-intrusive and still compatible with the general architecture of a given data processing framework.

Legacy interfaces: With modern networking and storage hardware, I/O operations are becoming more complex. Not only are there more options to store data (disk, flash, memory, disaggregated storage, etc.) but also

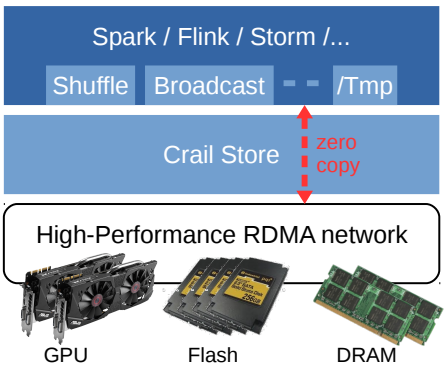


Figure 2: Crail I/O architecture overview.

it is getting increasingly important to use the storage resources efficiently. For instance, some newer technologies such as PCM permit data to be accessed at a byte granularity. Mediating storage access through a block device interface is a bad fit in such a case. Moreover, as the traditional compute layer is extended with GPUs and FPGAs, new interfaces to integrate accelerator memories into the distributed storage hierarchy are needed. Integrating such a diverse set of hardware technologies in the most efficient way, while still keeping the data processing system general enough, is challenging.

2 Crail I/O

In this paper, we present Crail, an open source user-level I/O architecture for distributed data processing. Crail is designed from the ground up for high-performance storage and networking hardware and provides a comprehensive solution to the above challenges. The design of Crail was driven by five main goals:

1. Bare-metal performance: enable I/O operations in distributed systems to take full advantage of modern networking and storage hardware.
2. Practical: integrate with popular data processing systems (e.g., Spark, Flink, etc.) in a form that is non-intrusive and easy to consume.
3. Extensible: allow future hardware technologies to be integrated at a later point in time in a modular fashion.
4. Storage tiering: provide efficient storage tiering that builds upon the new data localities of high-performance clusters.
5. Disaggregation: provide explicit support for resource disaggregation.

Crail is specifically targeting I/O operations on temporary data that require the highest possible performance. Examples of such operations in data processing systems are data shuffling, broadcasting of data within jobs, or general data sharing across jobs. Crail is not designed to provide the high availability and fault tolerance required by durable storage systems.

2.1 Overview

The backbone of the Crail I/O architecture is the Crail store, a high-performance multi-tiered data store for temporary data in analytics workloads. Data processing frameworks and applications may directly interact with

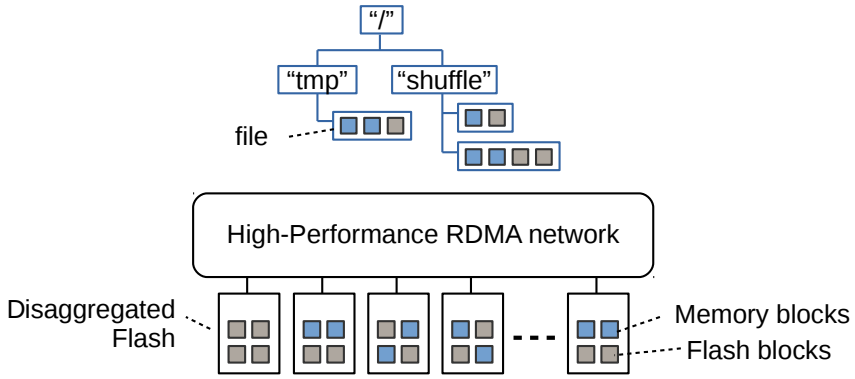


Figure 3: Crail file system namespace

Crail store for fast storage of working-set data, but more commonly the interaction takes place through one of the Crail modules (see Figure 1).

Crail modules implement high-level I/O operations, typically tailored to a particular data processing framework. For instance, the Crail Spark module implements various I/O intensive Spark operations such as shuffle, broadcast, etc. The Crail HDFS module provides a standard HDFS interface that can be used by different frameworks or applications to store performance-critical temporary data. Both modules can be used transparently with no need to recompile either the application or the data processing framework.

Crail modules are thin layers on top of Crail store. Consequently, the modules inherit the full benefits of Crail in terms of user-level I/O, performance and storage tiering. Due to the modules being so lightweight, implementing new modules for a particular data processing framework or a specific I/O operation requires only a moderate amount of work.

2.2 Crail Store

For the rest of the paper, if the context permits and to simplify the reading, we refer to Crail store simply as Crail. Crail implements a file system namespace across a cluster of RDMA interconnected storage resources such as DRAM or flash. Storage resources may be co-located with the compute nodes of the cluster, or disagreggated inside the data center, or a mix of both. Files in the Crail namespace consist of arrays of blocks distributed across storage resources in the cluster as shown in Figure 3. Crail groups storage resources into different tiers. A storage tier is defined by the storage media used to store the data, and the network protocol used to access the data over the network. In the current implementation, Crail offers three storage tiers:

1. **DRAM**: The DRAM tier uses one-sided RDMA operations (read/write) to access remote DRAM. With one-sided operations, the storage nodes remain completely passive and, thus, are not consuming any CPU cycles for I/O. Using one-sided operations is also effective for reading or writing subranges of storage blocks, as only the relevant data is shipped over the network rather than shipping the entire block. The DRAM tier provides the lowest latency and highest throughput among all the storage tiers. It is implemented using DiSNI [2], an open-source user-level networking and storage stack for the JVM [25].

2. **Shared Volume:** The shared volume tier supports flash access through block storage protocols (e.g., SCSI RDMA protocol, iSCSI, etc.). This tier is best used to integrate disaggregated flash.

3. **NVMeF:** The NVMeF storage tier exports NVMe flash accessed over RDMA fabrics. This storage tier provides ultra-low latency access to both direct attached and disaggregated flash.

[2] http://github.com/zrlio/disni

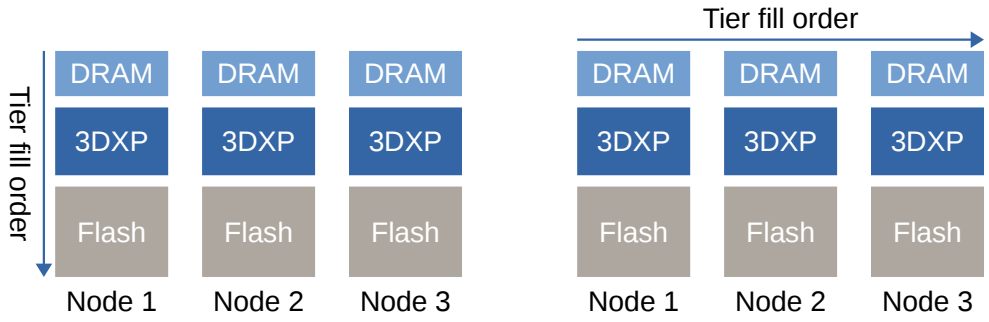


Figure 4: Vertical tiering (left) vs horizontal tiering (right).

Crail provides an extendable storage interface where new storage tiers can be plugged in transparently without requiring changes to the file system core. Apart from supporting traditional storage tiers, we are working on integrating accelerator memory into the Crail storage hierarchy. This will allow, for instance, GPU memory to become accessible via Crail from anywhere in the cluster.

2.3 Crail Store Interface

Crail exposes a hierarchical file system API comprising functions to create, remove, read and write files and directories. Besides those standard operations, much of the Crail API is designed to empower the higher level Crail modules with the right semantics to best leverage the networking and storage hardware for their data processing operations (e.g., shuffle, broadcast, etc.). We discuss a few specific examples below:

Asynchronous I/O: All file read/write APIs in Crail are fully asynchronous which facilitates interleaving of computation and I/O during data processing. Also, the asynchronous APIs perfectly match with the asynchronous hardware interfaces in RDMA or NVMeF, therefore allowing for a very efficient implementation.

Storage tiering: Crail provides fine-grained control over how storage blocks are allocated during file writing. By specifying location affinities, applications can indicate a preference as to which physical storage node or set of nodes should be used to hold the data of a file. Similarly, by specifying storage affinities, applications can indicate which storage tier should preferably be used for a particular file. In the absence of a dedicated storage affinity request, Crail uses horizontal tiering where higher performing storage resources are filled up across the cluster prior to lower performing tiers being used. Horizontal tiering stands in contrast to traditional vertical tiering where local resources are always preferred over remote ones. With low latency RDMA networks, remote data access has become very effective, and as a result, horizontal tiering often leads to a better usage of the storage resources. Figure 4 illustrates both approaches in a configuration with DRAM, 3DXP and flash.

File types: Crail supports three different file types: regular data files, directories and multifiles. Regular data files are append-and-overwrite with only a single-writer permitted per file at a given time. Append-and-overwrite means that – aside from appending data to the file – overwriting existing content of a file is also permitted. Directories in Crail are just regular files containing fixed length directory records. The advantage is that directory enumeration becomes just a standard file read operation which makes enumeration fast and scalable with regard to the number of directory entries. Multifiles are files that can be written concurrently. Internally, a multifile very much resembles a flat directory. Multiple concurrent substreams on a multifile are backed with separate files inside the directory. The *multistream* API allows applications to read the collective data as if it were a single file. Later in the paper, we show how multistreams are used during mapreduce shuffle operations.

I/O buffers: To allow zero-copy data placement from the network interface to the application buffer (as implemented in RDMA), the corresponding memory has to be pinned and registered with the network interface. Crail exports functions to allocate dedicated I/O buffers from a reusable pool – memory that is pinned and

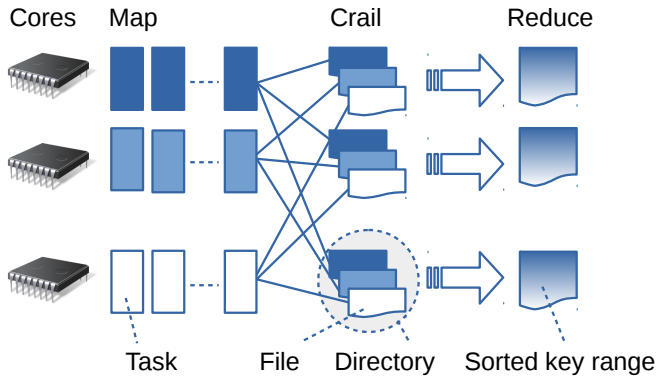


Figure 5: The Crail shuffle module.

registered with the hardware to enable bypassing of both the OS and the JVM during network transfers.

2.4 Crail Modules

Crail offers two independent modules that interface directly with the data processing layer: (a) a Spark module that contains Spark specific implementations for shuffle and broadcast operations, and (b) an HDFS adaptor that allows many of the prominent data processing frameworks to use Crail seamlessly for storing performance critical temporary data. We discuss the different components briefly as follows.

Shuffle: The shuffle engine maps key ranges to directories in Crail. Each map task, while partitioning the data, appends key/value pairs to individual files in the corresponding directories. Tasks running on the same core within the cluster append to the same files, which reduces storage fragmentation (see Figure 5). Individual shuffle files are served using horizontal tiering. In most cases that means the files fill up the memory tier as long as there is some DRAM available in the cluster, after which they extend to the flash tier. The shuffle engine ensures such a policy using the Crail location affinity API. Note that the shuffle engine is completely zero-copy, as it transfers data directly from the I/O memory of the mappers to the I/O memory of the reducers.

Broadcast: The Crail-based broadcast plugin for Spark stores broadcast variables in Crail files. In contrast to the shuffle engine, broadcast is implemented without location affinity, which makes sure the underlying blocks of the Crail files are distributed across the cluster, leading to a better load balancing when reading broadcast variables.

HDFS adaptor: The HDFS adaptor for Crail implements a straightforward mapping between the synchronous HDFS function calls and the corresponding asynchronous operations in Crail. Functionality in Crail that is not exposed in the HDFS API, such as storage affinities, can be configured through admin tools on a per-directory basis.

3 Evaluation

A detailed evaluation of the individual components in the Crail I/O architecture (Crail store, individual modules) is beyond the scope of this paper. Instead, we demonstrate the benefits of Crail on two specific workloads, sorting and SQL, both evaluated using Spark. We used a cluster with the following specification: 2xPower8 10-core @2.9Ghz, 512 GB DDR4 DRAM, 4x Huawei ES3600P V3 1.2TB NVMe SSD, with 100Gbps Ethernet Mellanox ConnectX-4 EN (RoCE). Nodes in the cluster run Ubuntu 16.04 with Linux kernel version 4.4.0-31.

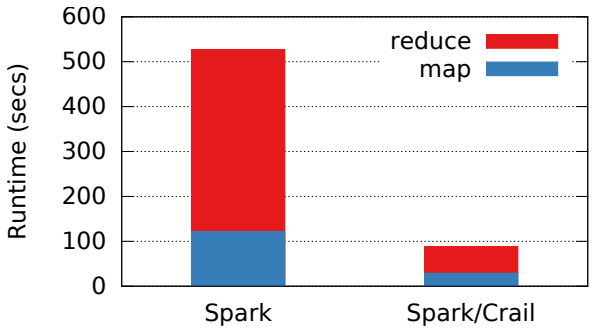


Figure 6: Performance comparison for Spark sorting, vanilla vs. Crail

3.1 Sorting

Sorting large data sets efficiently on a cluster is particularly interesting from a network perspective, as most of the input data will have to cross the network at least once. Here, we explore the sorting performance of Spark/Crail on a 128-node slice of the cluster. We compare vanilla Spark with Spark/Crail on a 12.8 TB dataset (Spark version 2.0.2). In both cases, plain HDFS on NVMe SSDs is used for input and output, and the shuffle engine (including networking, serialization and sorting) is exchanged in the Spark/Crail configuration.

Anatomy of Spark Sorting: A Spark sorting job consists of two phases. The first phase is a mapping or classification phase - where individual workers read their part of the key-value (KV) input data and classify the KV pairs based on their keys. This phase involves only very little networking as most of the workers run locally on the nodes that host the HDFS data blocks. During the second so called reduce phase, each worker collects all KV pairs for a particular key range from all the workers, de-serializes the data and sorts the resulting objects. This pipeline runs on all cores in multiple waves of tasks on all the compute nodes in the cluster.

The main difference between the Crail shuffler and the Spark built-in shuffler lies in the way data from the network is processed in a reduce task. The Spark shuffler is based on TCP sockets; thus, many CPU instructions are necessary to bring the data from the networking interface to the buffer inside Spark. In contrast, the Crail shuffler shares shuffle data through the Crail file system, and therefore data is transferred directly via DMA from the network interface to the Spark shuffle buffer within the JVM.

Performance: Figure 6 shows the overall performance of Spark/Crail vs vanilla Spark. With a cluster size of 128 nodes, each node sorts 100 GB of data - if the data distribution is uniform. As can be seen, using the Crail shuffler, the total job runtime is reduced by a factor of 6. Most of the gains come from the reduce side, which is where the networking takes place. The Crail shuffler – due to its zero-copy data ingestion – avoids data copies, system calls, kernel context switches and generally reduces the CPU involvement during the network phase. As a result, data can be fetched and processed much faster using the Crail shuffler than with the built-in shuffler. In Figure 7 we show the data rate at which the different reduce tasks fetch data from the network. Each point in the figure corresponds to one reduce task. In our configuration, we run 3 Spark executors per node and 5 Spark cores per executor. Thus, 1920 reduce tasks are running concurrently (out of 6400 reduce tasks in total), generating a cluster-wide all-to-all traffic of about 70Gbps per node during that phase. In contrast, the peak network usage we observed for vanilla Spark during the run was around 10Gbps. Also note that Spark/Crail is sorting 12.8 TB of data in 98 seconds, which calculates to a sorting rate of 3.13 GB/min/core. This is only about 28% slower than the current 2016 winner of the sorting benchmark – a sorting benchmark running native code optimized specifically for sorting[3].

[3]http://www.crail.io/blog/2017/01/sorting.html

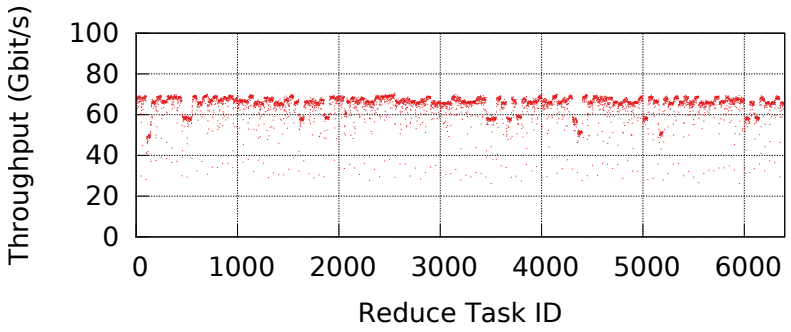


Figure 7: Network usage in a Spark/Crail sorting benchmark.

3.2 SQL

SQL is one of the most important and popular workloads supported by Spark. In a recent development cycle (between `1.6` and `2.0`), Spark/SQL has gained significant amounts of development and optimization efforts, most notably in the unification of DataSet and DataFrame APIs, and the whole-stage code generation logic. Put together, Spark/SQL has evolved into a fast and powerful SQL system. In the following we show how Crail can lift the performance of a Spark/SQL join operation through efficient network and storage I/O. The experiments are executed on a 9-node (1 driver + 8 workers) slice of the cluster.

Anatomy of a Spark/SQL Join: A join operation in Spark can be thought of as a cartesian product of two RDDs. The operation re-arranges the RDD partitions and executes a predicate on the desired columns. Here, we focus on EquiJoin where the predicate for matching rows is based on the equality of two values (referred to as a key). An EquiJoin between two tables in Spark is implemented as a map reduce job.

During the map phase, the tables are read from distributed storage and materialized as rows. The rows are then shuffled in the cluster based on a hash computed from the key column. This ensures that (a) the work will be equally distributed between worker nodes; and (b) all rows within a certain key range will be sent to the same worker node. Spark schedules a stage (comprising a number of tasks) for each of the two input tables. The key I/O operations that dictate the performance of the map phase are the reading of the input data and the writing of the shuffle data.

In the reduce phase, each worker collects two sets of shuffled rows from two input files over the network, sorts the rows in both sets based on the keys, and then scans both sets for matches. These steps are analogous to a sort-merge join operation on a local dataset. Finally, the matching rows are again written out to the distributed storage. The three key I/O operations in the reduce phase are fetching row data from the network (network bounded), sorting (CPU bounded), and joining and writing-out of rows (mixed CPU-I/O).

Performance: Spark/SQL supports multiple data sources and formats (e.g., HDFS, Hive, Parquet, JSON, etc). Our input tables consist of two parquet datasets each with a `<int,long,float,double,String>` schema. Each table contains 128 million rows, with randomly populated column entries. The `<String>` column is a 1024 character long random string. Hence, on average, each row contains a little more than 1kB of data. The join is made on the `<int>` column.

We explore two configurations: in the first configuration, vanilla Spark is used and the parquet files are stored on HDFS (`v2.6.5`). In the second configuration, we store the parquet files on Crail (using the HDFS adaptor) and also enable the Crail shuffler for Spark. In both configurations, the input tables are evenly distributed over the 8 machines with a total size of 256GB (128GBx2). The parquet files are generated using the open-sourced

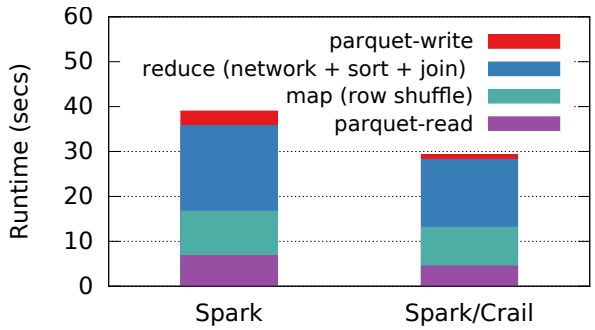


Figure 8: Performance of a SQL join in Sprak vs Spark/Crail.

`ParquetGenerator`[4] tool.

Figure 8 illustrates the performance of the join operation in both configurations. As can be observed, the use of the Crail architecture results in around 24.8% performance gains for simple SQL join workloads. These gains come from all four phases where I/O operations are involved, including parquet reading/writing and data shuffling. For instance, it takes 1.9 μs to fetch a single row from HDFS during parquet reading, but only 1.1 μs to fetch a row from Crail (using the HDFS adaptor). The net gains are about 33.8% (6.8 vs 4.5 seconds) for reading the parquet data. After reading, data is classified and written out as shuffle data. Here as well, the use of the Crail shuffler results in about 13.1% (9.9 vs 8.6 seconds) gains. The gains are limited as for both vanilla Spark and Crail, shuffle-writeouts are local operations. The shuffle map phase is followed by the reduce phase where the shuffle data is read over the network. Here, Crail delivers significant gains in raw networking performance. The shuffle data (32 (16x2) GB/node) is read in about 3.2 seconds, which calculates to a bandwidth utilization of a little more than 80Gbps. For vanilla Spark, the peak bandwidth utilization does not go above the 50Gbps mark. These gains from the raw network performance result in a 20.9% (19.1 vs 15.1 seconds) runtime improvement in the reduce phase. However, in comparison to sorting (Section 3.1), gains from the network do not proportionally translate into performance improvements of the reduce phase. This is because fetching data from the network is followed by de-serialization, sorting, and join operations, whose collective time eclipses the raw networking time. We are exploring various serializer designs with Crail to improve this situation.

As the final step, the joined rows are written out (roughly 2 GB/node), and here as well, Crail's simple write path delivers significant gains over HDFS (3.2 vs 1.1 seconds). Overall, the use of the Crail architecture results in 24.8% performance gains.

3.3 Flash Disaggregation

Traditionally, data processing platforms such as Spark or Hadoop have been designed around the concept of data locality where computation is scheduled to execute on the node that stores the data to avoid expensive network operations. Recently, the idea of storage disaggregation has become more popular [9, 16]. Disaggregation decouples storage and compute, which simplifies the deployment and management of the hardware resources, and also drives down cost. The hope is that due to the improved network speed of high-performance interconnects the overhead of crossing the network during I/O can be reduced to a minimum. However, as we have seen in Figure 1, the actual overheads in Spark and Hadoop network operations are caused mainly by the software stack despite the fast networking hardware. Consequently, running Spark or Hadoop in a disaggregated environment is costly from a performance point of view. With Crail, the software overheads during I/O are reduced which opens the door for storage disaggregation. This is illustrated in Figure 9 where we compare the performance of the distributed sorting workload in two configurations. In the Crail configuration, data is read, shuffled and

[4]https://github.com/zrlio/parquet-generator

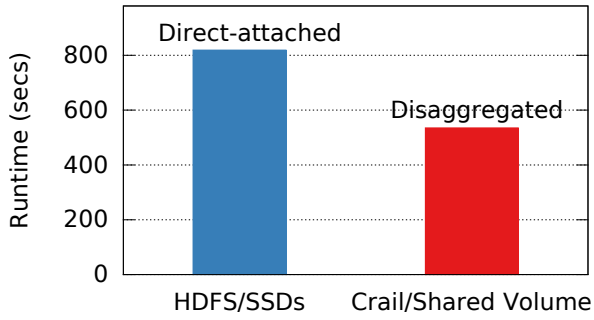


Figure 9: Using Crail for flash disaggregation in Spark.

written out using Crail's disaggregated flash tier. We use Crail's shared volume storage plugin in combination with an industry-standard flash enclosure connected via SRP (SCSi over RDMA) over a 56Gbps InfiniBand link. The second configuration is a vanilla Spark deployment, non-disaggregated, with both HDFS and Spark shuffler configured to use local memory and SSDs. We made sure that both configurations have the same aggregated flash bandwidth (10 GB/s) and about the same total flash capacity. As can be observed, the disaggregated Crail configuration performs better than the non-disaggregated Spark configuration, despite all the I/O operations being remote. This shows that with Crail, remote I/O operations can be accelerated to a level that outperforms local I/O operations in Spark, and as a result makes disaggregation a viable solution.

4 Related Work

Lately, various efforts have focused on making better use of fast network or storage hardware for data processing. Most of those works fall into one of two categories. The first category consists of systems developed from scratch for the new hardware. Examples of this type are flash optimized key/value stores [6, 18], RDMA-based key/value stores [12, 8, 22], RDMA-based distributed memory systems [3, 21], RDMA-based distributed transaction processing systems [8, 15], distributed join engines leveraging RDMA [1, 24], or even completely new database architectures designed for high-speed networks [4]. All these efforts manage to achieve very good performance due to the hardware software co-design, but they are often not integrated into a publicly available data processing platform and/or only target very specific applications. In contrast, Crail is an I/O architecture that can be applied to different open source data processing frameworks supporting a wide range of applications and workloads.

The second category of work aims to integrate fast I/O hardware into existing systems and frameworks. For instance, in one body of work the network stack of the HDFS file system or the Spark shuffle manager have been re-written for high-speed networks [11, 3, 12]. Similarly, efforts have been put in place to better integrate fast networks into Tachyon [19], a popular distributed caching system. However, all these works are retrofitting an InfiniBand-based messaging service in an existing message-based networking code, resulting in only marginal improvements in performance. Crail, on the other hand, implements a tight semantic integration of fast I/O hardware and high-level data processing operations such as shuffle, broadcast etc., and as a result manages to improve workload performance substantially.

With the availability of high-performance networks, resource disaggregation has become a viable option for data processing systems [10]. A recent workload-driven assessment has concluded that current high-performance network technologies are now sufficient to maintain application performance in a disaggregated datacenter – assuming the necessary end-host optimizations (e.g., RDMA) are being leveraged [9]. Specifically for storage, ReFlex – a recently proposed system for flash disaggregation – enables applications to maintain their performance while accessing remote flash devices [17]. Erfan Zamanian et. al further propose NAM-DB, a novel

distributed database architecture that uses RDMA and disaggregated memory for implementing snapshot isolation and scalable transactions. Crail's disaggregated architecture builds on similar concerns of providing fast access to remote resources. However, in contrast to those existing works, Crail provides a comprehensive solution to accessing memory and storage across a cluster, both local and disggregated. For instance, Crail naturally exploits any mix of local and remote resources at a fine granular manner through horizontal tiering.

Finally, there has been a body of work on improving the I/O efficiency within a single machine, most recently [23, 2, 5, 13]. These efforts are orthogonal to Crail as Crail permits different user-level I/O techniques to be integrated into its distributed I/O architecture through the storage tier interface.

5 Conclusion

We presented Crail, an I/O architecture tailored to best integrate fast networking and storage hardware into distributed data processing platforms. Crail is based on a distributed data store acting as a fast I/O bus for platform specific I/O modules. Crail not only takes full advantage of the capabilities of modern hardware in terms of zero-copy or asynchronous I/O, but it also changes the way local and remote storage resources are used in a high-speed network deployment. Crail is a recently started open source project (`www.crail.io`) and there are many opportunities to contribute.

References

[1] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1463–1475, 2015.

[2] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, 2014.

[3] Adithya Bhat, Nusrat Islam, Xiaoyi Lu, Wasi Rahman, Dipti Shankar, and Dhabaleswar Panda. A plugin-based approach to exploit rdma benefits for apache and enterprise hdfs. In *The Sixth workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, 2015.

[4] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[5] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, 2012.

[6] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, 2010.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[8] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, 2015.

[9] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, 2016.

[10] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

[11] Nusrat Islam, Xiaoyi Lu, and Dhabaleswar Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *24th International Conference on Supercomputing (ICS '16)*, 2016.

[12] Nusrat Islam, Wasi Rahman, and Dhabaleswar Panda. In-memory i/o and replication for hdfs with memcached: Early experiences. In *IEEE International Conference on Big Data*, pages 213–218, 2014.

[13] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[14] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, 2014.

[15] Anuj Kalia, Michael Kaminsky, and David 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)*, pages 185–201, GA, 2016.

[16] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash local flash. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[18] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 207–219, 2015.

[19] Mellanox. RDMA Improves Tachyon Remote Read Bandwidth and CPU Utilization by up to 50%, https://community.mellanox.com/docs/DOC-2128.

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

[21] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, 2015.

[22] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[23] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, 2014.

[24] Wolf Rodiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205, 2016.

[25] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. jverbs: Ultra-low latency for data center applications. In *The Proceedings of the Fourth ACM Symposium on Cloud Computing*, SoCC '13, pages 10:1–10:14, 2013.

[26] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir]relevance of network performance for data processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016.

Scalable and Distributed Key-Value Store-based Data Management Using RDMA-Memcached

Xiaoyi Lu, Dipti Shankar, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{luxi, shankard, panda}@cse.ohio-state.edu

Abstract

Over the recent years, distributed key-value stores have been extensively used for designing scalable industry data management solutions. Being an essential component, the functions and performance of a distributed key-value store play a vital role in achieving high-speed data management and processing. Thus, many researchers and engineers have proposed various designs to continuously improve them. This paper first presents a survey of recent related works in the field of distributed key-value store and summarizes the research hotspots. Along these emerging research dimensions, we present a design overview of our proposed RDMA-Memcached library, which aims for achieving high throughput and low latency by taking full advantage of native InfiniBand features including RDMA and different transports (RC, UD, Hybrid), and high-performance storage such as SSDs. Though Memcached was originally used for caching online query results, we have implemented further extensions for Memcached to enrich its functions, APIs, and mechanisms, which extend its capabilities to more use cases. With our proposed designs, RDMA-Memcached can be used as a scalable data caching layer or a Burst Buffer component for both online data processing and offline data analytics. We also propose a set of benchmarks which are publicly available to help researchers to evaluate these advanced designs and extensions.

1 Introduction

In the Web 2.0/3.0 era, many social web services exist that deal with people's personal information and relationships (e.g. Facebook, Twitter), whereas a number of services also exist that store other information, such as goods (e.g. Amazon), photos (e.g. Flickr), travel (e.g., Yahoo! Travel), etc. These web-services essentially handle either traditional SQL-based or cloud NoSQL-based On-Line Data Processing (OLDP) workloads. Typically, most of the data that is accessed by these web-service applications are stored in databases and accessing these databases are often expensive in terms of latency and throughput. Since millions of users access these web-services, the servers must be equipped to deal with a large number of simultaneous access requests efficiently and fairly. Today, with cloud computing emerging as a dominant computing platform for providing scalable on-line services, there has been a tremendous increase in the number of systems developed and deployed for cloud data serving, thus making the performance and scalability of cloud OLDP applications extremely vital. Studies [1, 2] have shown that leveraging a distributed and scalable key-value store is invaluable to application servers in these scenarios.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Due to this diversified nature of social networking websites and other web services, massive amounts of data is constantly generated and is stored and processed on data centers in an offline fashion. These offline data analytics tasks have necessitated the design of Big Data middleware (e.g., Hadoop, Spark, etc.) that is capable of delivering high-performance. Traditionally, disk-centric distributed file systems such as the Hadoop Distributed File System (HDFS) have been employed as the storage layer for offline Big Data analytics. But researchers found memory-centric distributed file systems such as Alluxio (formerly Tachyon) [3] can provide higher throughput and scalability for these workloads. Inspired by the 'in-memory I/O' concept, recent research works [4, 5, 6] have started to explore the use of key-value store based burst-buffer systems to boost I/O performance of Big Data applications.

Because of these usage requirements on distributed key-value stores, the classical designs and implementations such as Memcached [7] have become an essential component of today's scalable industry data management solutions. Being an elementary component, the functions and performance of a distributed key-value store play a crucial role in achieving high-speed data management and processing. Thus, many novel designs and enhancements have been proposed to continuously improve them. Recent studies, such as [8, 3, 12, 12], have shed light on the fact that the novel designs of Memcached for clusters with Remote Direct Memory Access (RDMA) capable interconnects have demonstrated that the latency of Memcached operations can be significantly reduced. Research works, including [13, 14, 15, 16] propose several hybrid, high-performance key-value stores, which can deliver higher overall performance for data-intensive workloads, by taking advantage of high-speed flash-based storage such as SATA SSDs, PCIe-/NVMe-based SSDs.

Thus, it is essential to systematically study and understand the performance characteristics, advantages, and trade-offs of key-value stores that exist on modern multi-core, networking, and storage architectures. In the light of these facts, we need to consider the following broad questions and challenges for achieving scalable key-value store-based data management and processing: **1)** What kind of advanced designs have researchers proposed in the literature and community for improving the functions and performance of distributed key-value stores? **2)** How can existing key-value stores such as Memcached be re-designed from the ground up to utilize RDMA and different transports available on high-performance networks? What kind of new extensions can be proposed for fully exploiting the benefits of high-performance storage, such as SSDs? **3)** How to enrich the functions, APIs, and mechanisms of current-generation key-value stores, to extend its capabilities to more use cases? **4)** How to design a set of benchmarks to help researchers and engineers understand these new features for key-value stores that capture the requirements of different key-value store application scenarios? **5)** Can we show performance benefits for some concrete use cases by using these advanced designs for key-value stores?

To address the above challenges, this paper is organized as follows. In Section 2, we first present a survey of highly cited works in this field, and summarize the research hotspots, to help understand the challenges of obtaining high-performance for distributed key-value stores. In Section 3, we present a design overview of our proposed RDMA-Memcached library [17], which aims for achieving high throughput and low latency by fully taking advantage of native InfiniBand features, such as RDMA and different transports (RC, UD, Hybrid). In Section 4, we have undertaken further extensions (i.e., hybrid memory, non-blocking APIs, and burst buffer design with efficient fault-tolerance support) into Memcached to extend its capabilities to more use cases. We also propose a set of benchmarks to help researchers to evaluate these advanced designs and extensions in Section 5. With our proposed designs, RDMA-Memcached can be used as a scalable data caching layer for online data processing workloads or a Burst Buffer component for offline data analytics. We show some use cases and benefits in Section 6. Section 7 concludes the paper.

2 A Survey on Related Work

Through thorough studies of highly cited works in the field of high-performance key-value store design, five major research hotspots have been identified, including network I/O, data management, fault tolerance, hardware

acceleration, and usage scenario. A brief overview of these five dimensions along with works that have focused on each of these individual aspects is presented below.

Network I/O: Key-value store operations are network-intensive by nature. With the emergence of high performance interconnects (e.g., InfiniBand, RDMA over Converged Ethernet/RoCE), several research works have focused on the network perspective. These works are represented in the first five rows in Table 1. Research works including HERD [8], Pilaf [12], FaRM [3], and HydraDB [19] propose different ways to design a low-latency communication engine that leverages zero-copy and CPU-bypass features of RDMA, while exploiting multi-core architectures. Similarly, MICA [12] designs a key-value store that maps client requests directly to specific CPU cores at the server NIC level, using Intel DPDK along with the UDP protocol.

Key-Value Data Management: Most key-value stores contain two important data structures: (1) A hash table or tree to lookup the key-value pairs stored, and, (2) a list or log that stores the actual key-value pairs. These data structures are co-designed with the underlying hardware, network protocol, and available storage for the best performance. Some works such as MICA [12], FaRM [3] take a holistic approach to design the key-value data structures with their communication engines. The data management can be divided into the following categories:

- Hashing: The hashing scheme employed is vital for inserting and looking up data quickly and consistently in a multi-client and multi-threaded server environment that leverages the large number of CPU cores available on the nodes in today's data centers and scientific compute clusters. One such notable research work, MemC3 [20], proposed a new hashing scheme for read-heavy workloads. Several general studies on hashing like optimistic hashing, cuckoo hashing have been experimented with and used to implement key-value stores.

- Cache Management: An efficient cache must be able to store and retrieve data quickly, while conserving the available memory. Most volatile in-memory key-value stores have intelligent eviction techniques that aim to minimize the miss ratio and memory usage, and in turn improve the average response time. While Memcached uses LRU-based eviction, Twitter memcached [21] proposed configurable eviction policies. Research works such as LAMA [22] propose more adaptive techniques for automatically repartitioning available memory at the Memcached servers by leveraging locality inherent in the Memcached requests to reduce the miss ratio.

- Storage Architecture: While the works mentioned above (first ten rows of Table 1) are meant for volatile in-memory (i.e., DRAM) stores, several works such as NVMM [23] in Table 1 have focused on leveraging non-volatile byte-addressable main memory. Along similar lines, fatcache [16] exploits high-speed SSDs to extend available storage capacities of the server.

Fault-Tolerance and Data Availability: Though traditionally in-memory key-value stores were volatile in nature, with their increased use, fault-tolerance and data availability are desirable features. In-memory replication is a popular technique for enabling fault-tolerance that is implemented in FaRM [3] and HydraDB [19]. On the other hand, research works such as Cocytus [24] are leveraging an alternative and more memory-efficient fault technique, namely Erasure Coding (EC), to design an efficient and resilient version of Memcached.

Hardware Acceleration: Hardware-based acceleration that leverages InfiniBand HCAs, GPGPUs, Intel KNL, etc., plays a prominent role in speeding up applications on modern HPC clusters; they are paving their way into data center environments. While key-value store operations are not primarily compute-intensive, offloading indexing and communication to the underlying accelerators can help define better overlap of concurrent data operations to increase overall throughput and client's performance. Mega-KV [25] is a fast in-memory key-value store that maintains a hash table on the GPGPU and utilizes its high memory bandwidth and latency hiding capability to achieve high throughput. On the other hand, FPGA-Memcached [27] is a Memcached implementation that employs a hybrid CPU+FPGA architecture. Similarly, MemcachedGPU [26] presents a

GPU-accelerated networking framework to enhance the throughput of Memcached to exploit massively parallel GPUs for batched request processing at the servers.

Usage Scenario: Key-value based solutions are vital for accelerating data-intensive applications. The two major categories of Big Data workloads that currently leverage key-value stores include: (1) Online workloads, such as database query caching, and, (2) Offline workloads, such as I/O-intensive Hadoop or Spark workloads over parallel file systems, graph processing, etc. FaRM [3] can be used as a query cache and has also been used to implement offline graph processing stores. Similarly, HydraDB [19] can be used to accelerate cloud-based NoSQL key-value store workloads and offline Hadoop/Spark workloads such as Sort.

Our proposed work, RDMA-Memcached [17, 8, 13, 28], attempts address various factors in each of the five identified hotspots. To improve network I/O performance, RDMA-Memcached leverages RDMA semantics on modern high-performance interconnects. For improving data management, it leverages a log-structured SSD-assisted extension to increase hit rate without compromising performance, while also proposing non-blocking API extensions. RDMA-Memcached can support both replication and online erasure coding schemes for achieving fault tolerance support [35]. Through careful designs, RDMA-Memcached can be used to accelerate several online and offline Big Data workloads.

Research Work	Network I/O	Key-Value Data Management	Fault-Tolerance	Hardware Acceleration	Usage Scenario
MICA [12]	Intel DPDK + UDP	Keyhash-based partition + Lossy concurrent hash+ Append-only log	-NA-	Zero-Copy NIC-to-NIC	Online
HERD [8]	RDMA/RoCE	Based on MICA's hashing design	-NA-	IB/40GigE HCA	Online
FaRM [3]	RDMA/RoCE	Hopscotch hashing	Replication	IB/40GigE HCA	Online/ Offline
Pilaf [12]	RDMA/RoCE	2-4 cuckoo hashing + Self-verifying data structs + async logging	-NA-	IB/40GigE HCA	Online
HydraDB [19]	RDMA/RoCE	NUMA-aware + compact hash + remote pointer sharing	Replication	IB/40GigE HCA	Online/ Offline
MegaKV [25]	Based on MICA	Based on MICA's caching design	-NA-	GPU-offload for indexing	Online
Memcached-GPU [26]	TCP/UDP	Set-associative hash + Local LRU per hash set	-NA-	GPU Network offload mgmt.	Online
MemC3 [20]	TCP/UDP	Optimistic cuckoo hash + Concurrent reads w/o locks + CLOCK cache management	-NA-	-NA-	Online
LAMA [22]	TCP/UDP	Locality-aware memory repartitioning based on footprints	-NA-	-NA-	Online
FPGA-Memcached [27]	TCP/UDP	Based on Default Memcached	-NA-	FPGA acceleration	Online
NVMM [23]	TCP/UDP	Memcached w/ non-volatile main memory support	Durable writes w/ NVMM	-NA-	Online
Cocytus [24]	TCP/UDP	Based on Default Memcached	Hybrid EC and replication	-NA-	Online
FatCache [16]	TCP/UDP	Default Memcached w/ SSD eviction	-NA-	-NA-	Online
RDMA-Memcached [17]	RDMA/RoCE [8, 28]	Hybrid Memory extension w/ SSD [13]	Replication/ EC [35]	IB/40GigE HCA	Online [37]/ Offline [6]

Table 1: Research Hotspots Addressed in Recent Distributed Key-Value Store Solutions

3 Design Overview of RDMA-Memcached

Figure 1 depicts the design overview of our proposed RDMA-Memcached. To enable RDMA-based communication for Memcached, we have to redesign the communication substrate in Memcached server and Libmemcached client components. In the server side, we also propose a hybrid memory scheme with SSD-based slab management to enlarge the caching capacity. For the client side, to achieve better overlapping among communication, computation, and SSD accessing, we propose a new set of non-blocking APIs for Libmemcached. We also extend the current fault-tolerance mechanism in Memcached from replication-only to 'replication & erasure coding.' Advanced designs with accelerators (such as Nvidia GPGPU, Intel Xeon Phi) will be available soon. Our designs can support all the existing Memcached applications with no changes needed. For offline data analytics, our proposed non-blocking APIs and Burst-Buffer mode will deliver the best performance.

3.1 RDMA-based Communication Substrate

The current-generation Memcached library is designed with the traditional BSD Sockets interface. While the Sockets interface provides a great degree of portability, the byte-stream model within Sockets interface mismatches with Memcached's memory object model. In addition, Sockets-based implementations internally need to copy messages, resulting in further loss of performance. High-performance interconnects and their software APIs, such as OpenFabrics Verbs [29], provide RDMA capability with memory-based semantics that fits very well with the Memcached model. Scientific and parallel computing domains use the Message Passing Interface (MPI) as the underlying basis for most applications. MPI libraries, such as MVAPICH2 [30], can achieve very low one-way latencies in the range of 1-2 us. Whereas, even the best implementation of Sockets on InfiniBand is still much slower.

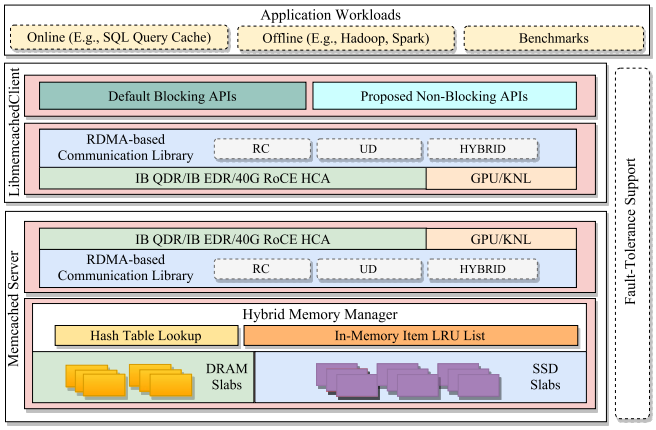


Figure 1: Design Overview of RDMA-Memcached

Inspired by these observations and experiences, we propose an RDMA-based communication substrate [8] over native Verbs APIs for Memcached. Our communication substrate exposes easy-to-use Active Message based APIs that can be used by Memcached, without re-implementing performance critical logic (like buffer management, flow control, etc.). Our design is optimized for both small and large messages. If the amount of data being transferred fits into one network buffer (e.g., $\leq$8 KB), it is packaged within one transaction and the communication engine will do one eager send for it. With this, we can save handshaking overhead, which is needed for RDMA operations. For large messages, we use RDMA Read based approach to transfer data with high-performance and overlapping. These designs can work directly with both InfiniBand and RoCE networks. We have done extensive performance comparisons of our Memcached design with unmodified Memcached using Sockets over RDMA, 10 Gigabit Ethernet network with hardware-accelerated TCP/IP, and IP-over-IB protocols on different generation InfiniBand HCAs. Our performance evaluation reveals that the latencies of RDMA-Memcached are better than those schemes by up to a factor of 20. Further, the throughput of small Get operations can be improved by a factor of six when compared to Sockets over 10 Gigabit Ethernet network. A similar factor of six improvement in throughput is observed over Sockets Direct Protocol.

3.2 Multi-Transport (RC/UD/Hybrid) Support

The basic design with RDMA mentioned above has shown that the use of RDMA can significantly improve the performance of Memcached. This design is based on InfiniBand as connection-oriented Reliable Connection (RC) transport. However, exclusive use of RC transport hinders scalability due to high memory consumption. The reason is that a typical RC connection with standard configuration parameters requires around 68 KB [31] of memory and RC requires a distinct QP per communicating peer. Since a Memcached server may need to serve thousands of clients simultaneously, this may cause really high memory consumption which limits the scalability of RDMA-based designs for Memcached. On the other hand, the Unreliable Datagram (UD) transport of InfiniBand addresses this scalability issue because UD is a connectionless transport, and a single UD QP can communicate with any number of other UD QPs. This can significantly reduce the memory consumption for a large number of connections. However, UD does not offer RDMA, reliability, and message ordering. Moreover, messages larger than a Maximum Transmission Unit (MTU) size (4 KB in current Mellanox hardware) have to be segmented and sent in MTU size chunks.

Due to these, we find none of these transports can achieve both high-performance and scalability as they are currently designed. Pure RC-based Memcached designs may deliver good latency and throughput, but may not have the best scalability. Pure UD-based Memcached designs may deliver good scalability, but may not deliver the same performance as Memcached with RC. Thus, we introduce a hybrid transport model [9] which leverages the best features of RC and UD to deliver both high-performance and scalability for Memcached. The idea behind this is we can limit the maximum number of RC connections to a specified threshold. Further connections can be made over UD transport. This ensures that the memory requirement does not increase above a particular limit. Another feature is to dynamically change the transport mode from UD-to-RC or RC-to-UD, based on the communication frequency between the server and a specific client, and we can also impose a limit on the maximum number of RC connections. These hybrid modes can be selected depending on platform characteristics. For UD transport, we proposed novel designs to handle the reliability, flow control, and large message transfers. With all of these, our performance evaluations reveal that the hybrid transport delivers performance comparable to that of RC while maintaining a steady memory footprint as that of UD. The scalability analysis with 4,096 client connections reveals that the hybrid transport achieves good memory scalability.

4 Advanced Extensions for RDMA-Memcached

4.1 Hybrid Memory Extensions

Internet-scale distributed key-value store-based caches have become a fundamental building block of several web architectures in the recent years, for scaling throughput and reducing overall application server access latency. However, the issues of cost and power requirements limit the possibilities for increasing the DRAM capacities available at these caching servers. With the advent of high-performance storage (e.g. SATA SSD, PCIe-/NVMe-SSD) that offer superior random I/O performance in comparison to conventional disk storage, several efforts have been directed towards employing 'RAM+SSD' hybrid storage architectures for extending in-memory key-value stores, with the goal of achieving high data retention while continuing to support millions of operations per second. Such solutions to increasing caching layer capacities without sacrificing performance have been explored by Twitter [16], Facebook [15], and many others.

We proposed a hybrid memory scheme for RDMA-enhanced Memcached [17, 13], which has an SSD-assisted design that extends the in-memory slabs with SSD-based slabs to enlarge the data capacity available to the Memcached servers. It employs the existing in-memory lookup table to store the key i.e., the index of the data object, and a log-structured SSD storage to store the value of evicted data objects. For Memcached Set requests, a newly inserted object is cached in the DRAM or in-memory slabs, and the index points to the in-memory location. When the in-memory slabs are full, the cached objects are compacted and evicted to the

SSD from the LRU list on a per-slab basis using batched writes; the corresponding indices in the lookup table are updated to point to the appropriate file offset. Similarly, Memcached Get requests are served either from the in-memory slabs or the SSD using random reads. Garbage collection features are provided via lazy reclamation, and it keeps the data in the SSD up-to-date. With higher data retention enabled through the use of high-speed SSDs, the Memcached caching layer can provide better hit rate, and in turn increased performance.

4.2 Non-Blocking API Extensions

Blocking key-value request semantics are inherent to most online Big Data applications; data needs to be available to the client upon request. Additionally, these tend to be read-mostly workloads. Recently, several Offline Big Data analytical workloads have started to explore the benefits of leveraging high-performance key-value stores. Prominent examples include a Memcached-based Burst Buffer layer for the Lustre parallel file systems [6, 4], a Memcached-based Burst-buffer for HDFS [5], and intermediate data caching with Memcached in Hadoop MapReduce [32]. These write-heavy workloads incorporate different I/O and communication characteristics compared to the traditional online workloads. On the other hand, while the SSD-assisted hybrid memory design for RDMA-based Memcached can guarantee a very high success rate due to its property of high data retention, studies have shown that these designs have two major bottlenecks: (1) The current hybrid design with RDMA-based Memcached provides support for blocking Memcached Set/Get APIs only, i.e., memcached_set and memcached_get, which mandates that the client waits until the operations complete at the Memcached server, and, (2) the server-side SSD I/O overhead is in the critical path of the Memcached Set/Get request.

To alleviate these bottlenecks, new non-blocking extensions to the existing Libmemcached APIs have been proposed for RDMA-based Hybrid Memcached. The proposed APIs, i.e., `memcached_iset/ iget` and `memcached_bset/bget` [17, 28], introduce mechanisms to fetch and store data into Memcached in a non-blocking manner. The proposed extensions allow the user to separate the request issue and completion phases. This enables the application to proceed with other tasks, such as computations or communication with other servers while waiting for the completion of the issued requests. The RDMA-based Libmemcached client library benefits from the inherent one-sided characteristics of the underlying RDMA communication engine and ensures the completion of every Memcached Set/Get request. In this way, we can hide significant overheads due to blocking waits for server response at the client side and SSD I/O operations at the server side. In addition to bridging the performance gap between the hybrid and pure in-memory RDMA designs, the server throughput of the hybrid Memcached design can be increased by about overlapping concurrent Set/Get requests.

4.3 Burst-Buffer Design Extensions

Key-Value Store-based Burst-Buffer: Distributed key-value store-based caching solutions are being increasingly used to accelerate Big Data middleware such as Hadoop and Spark on modern HPC clusters. However, the limitation of local storage space in these environments, due to the employment of the Beowulf architecture, has placed an unprecedented demand on the performance of the underlying shared parallel storage (e.g., Lustre, GPFS). Memory-centric distributed file systems such as Alluxio [3] have enabled users to unify heterogeneous storage across nodes for high throughput and scalability. While this approach is being explored to develop a two-tier approach for Big Data on HPC clusters [33], it was not designed specifically for traditional HPC infrastructure, as it still depends on data locality for best performance. Such an I/O contention to shared storage can be alleviated using a burst-buffer approach. Research works [6, 4] are actively exploring employing generic key-value stores such as Memcached to build burst-buffer systems.

We propose a hybrid and resilient key-value store-based Burst-Buffer system (i.e., Boldio [6]) over Lustre for accelerating I/O-phase that can leverage RDMA on high-performance interconnects and storage technologies. Boldio provides efficient burst-buffer client/server designs that enable optimal I/O request overlap via non-blocking API semantics and pipelined stages. It employs a client-initiated replication scheme, backed by a

shared-nothing asynchronous persistence protocol, to ensure resilient flushing of the buffered I/O to Lustre. To enable leveraging this light-weight, high-performance, and resilient remote I/O staging layer, we design a file system class abstraction, `BoldioFileSystem`, as an extension of the Hadoop `LocalFileSystem`. Thus, many Hadoop and Spark based Big Data applications can leverage our design transparently.

Resilience Support with Erasure Coding in Key-value Stores: To enable the resilience that is critical to Big Data applications, it is necessary to incorporate efficient fault-tolerance mechanisms into high-performance key-value stores such as RDMA-Memcached that are otherwise volatile in nature. In-memory replication is being used as the primary mechanism to ensure resilient data operations. The replication can be initiated by the client, as done by Libmemcached and its RDMA-based variant, or the Memcached server can be enhanced to replicate data to other servers in the cluster (however, this requires modifying the underlying shared-nothing architecture of Memcached). The replication scheme incurs increased network I/O with high remote memory requirements.

On the other hand, erasure coding [34] is being extensively explored for enabling data resilience, while achieving better storage efficiency. Several works [24, 35] are exploring the possibilities of employing Online Erasure Coding for enabling resilience in high-performance key-value stores. With erasure coding, each data entity is divided into K fragments and encoded to generate M additional fragments. These K + M fragments are then distributed to N unique nodes. Any K fragments can be used to recover the original data object, which means erasure coding can tolerate up to M simultaneous node failures. Since erasure coding requires a storage overhead of N/K where (K<N), as compared to the high overhead of M+1 (i.e., replication factor) that is incurred by replication, it provides an attractive alternative to enable resilience in key-value stores with better memory efficiency. While this provides a memory-efficient resilience, it also introduces a new compute phase in the critical path of each Set/Get request. This necessitates us to design efficient methods to overlap and hide these overheads to enable high performance. The new APIs proposed in Section 4.2 enable us to design a non-blocking request-response engine that can perform efficient Set/Get operations by overlapping the encoding/decoding involved in enabling Erasure Coding-based resilience with the request/response phases, by leveraging RDMA on high performance interconnects.

5 Benchmarks

Since key-value stores like Memcached are mostly used in an integrated manner, it is vital to measure and model performance based on the characteristics and access patterns of these applications. On the other hand, the emergence of high-performance key-value stores that can operate well with 'RAM+SSD' hybrid storage architecture and non-blocking RDMA-Libmemcached API semantics have made it essential for us to design micro-benchmarks that are tailored to evaluate these upcoming novel designs.

5.1 Micro-Benchmarks

Micro-benchmarks suites such as Mutilate [36] are designed with load generation capabilities to model real-work Memcached workloads (e.g., Facebook), with high request rates, good tail-latency measurements, and realistic request stream generation. Our proposed OSU HiBD (OHB) Memcached micro-benchmarks [17] model the basic Set/Get latencies that can be used to measure the performance of RDMA-based Memcached design. This micro-benchmark suite provides benchmarks to measure the request latencies using both blocking and non-blocking RDMA-Libmemcached APIs. These benchmarks are mainly designed to model the performance of RDMA-Memcached on different clusters with high-performance interconnects such as InfiniBand, RoCE, etc.

5.2 Hybrid Memory Benchmarks

Higher data retention is a desirable feature of the Memcached server cluster. The failure to retrieve data from key-value store-based caching layer due to the insufficient space in the caching tier can be caused by the consis-

tency requirements or the caching tier's eviction policy; in this case, data needs to be fetched from the back-end database. However, most benchmarks are used to evaluate performance in a stand-alone manner. To model the effect of the back-end database into the stand-alone micro-benchmarks [37], we introduce hybrid Memcached micro-benchmarks with a pluggable module for analysis and prediction of an estimate of the caching tier miss-penalty. This module is designed to get an estimate on the database access latency (i.e., miss-penalty) that a request must incur if it can not get the data in the caching tier and have to access the back-end database. This is just a one-time process for the database used. The miss-penalty estimated by the prediction module is added to the overall execution time for every query that incurs a caching tier miss. In this manner, the overhead of using the database is minimal, and it needs not be involved while running the actual tests; giving us the opportunity to tune the parameters of the hybrid design for best performance.

5.3 YCSB with Memcached

Cloud serving workloads are key-value store-based workloads that give higher priority to scalability, availability, and performance over consistency of the data. YCSB [39] is a typical benchmark for these cloud data processing applications. These services are often deployed using NoSQL-like distributed key-value stores. YCSB consists of an extensible workload generator that models a variety of data serving systems into five core workloads. The workload executor, i.e., the YCSB clients loads the generated data into the data store, generates an access pattern based on the chosen workload or input parameter selection, and evaluates the underlying store to benchmark the performance and scalability of the cloud serving system. The YCSB distribution has a plugin for Spymemcached Java client [38]. For benchmarking RDMA-based Memcached, we leverage the flexible YCSB framework to develop a new client interface using JNI to design an RDMA-aware YCSB client for benchmarking hybrid RDMA-Memcached design. Memcached-over-MySQL with YCSB is also presented in [37].

5.4 Bursty I/O Benchmarks

The OHB micro-benchmarks presented in Section 5.1 can be extended to represent a block-based pattern to mimic bursty I/O workloads [28], that are inherent in various HPC workloads including (1) Checkpoint-Restart for scientific applications (e.g., MPI), and, (2) Hadoop or Spark over parallel file systems such as Lustre (described in Section 4.3). Data blocks are read and written as smaller chunks that can fit into key-value pairs within Memcached. Bursty I/O micro-benchmarks can model the chunk or key-value pair size, the overall workload size, data access pattern, workload operation mix (read:write operations per client) to study different dimensions of the performance of a key-value store-based burst buffer layer. Specifically, to study the advantages of I/O request overlap, a large iteration of non-blocking Set/Get requests can be issued and monitored in an asynchronous fashion to ensure data requests have successfully completed.

6 Use Cases and Benefits

Leveraging a distributed key-value store based caching layer has proven to be invaluable for scalable data-intensive applications. To illustrate the probable performance benefits, we present two use cases:

6.1 Use Case: Online Caching

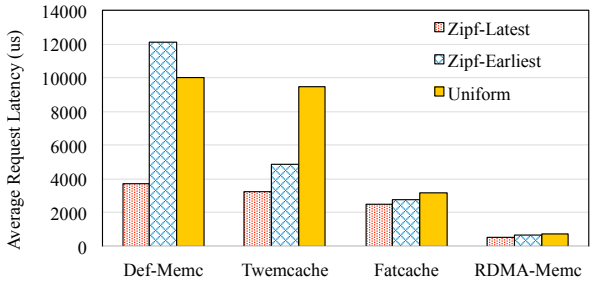
To illustrate the benefits of leveraging RDMA-Memcached to accelerate MySQL-like workloads, we employ the hybrid micro-benchmarks presented in Section 5.2 on an Intel Westmere cluster; each node is equipped with 12 GB RAM, 160 GB HDD, and MT26428 InfiniBand QDR ConnectX interconnects (32 Gbps data rate) with PCI-Ex Gen2 interface running RHEL 6.1. We use 8-16 compute nodes to emulate clients, running for both the socket-based (Libmemcached v1.0.18) and RDMA-enhanced implementation (RDMA-Libmemcached 0.9.3).

We run these experiments with 64 clients, 100% reads, an aggregated workload size of 5 GB, a key-value size of 32 KB, and an estimated average miss penalty of 95 ms.

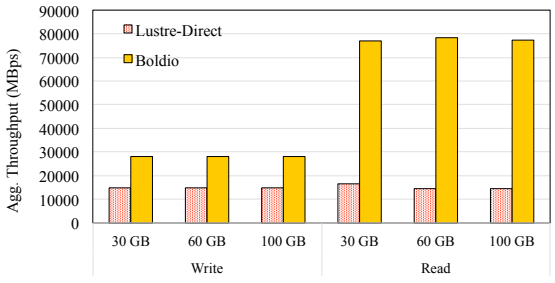
We compare various Memcached implementations including default Memcached [7], twemcache [21], fatcache [16] and RDMA-Memcached [17]. From Figure 2a, for the uniform access pattern, where each key-value pair stored in the Memcached server cluster is equally likely to be requested, we observe that SSD-assisted designs such as fatcache and RDMA-Memcached can give the best performance due to their high data retention. Specifically, RDMA-Memcached can improve performance by up to 94% over in-memory default Memcached designs (including twemcache). We study the performance with two skewed access patterns, namely, Zipf-Latest (Latest-Is-Popular) and Zipf-Earliest (Earliest-Is-Popular), which model key-value pair popularity based on the most recently and least recently updated key-value pairs, respectively. As compared to the other solutions in Figure 2a, we observe that RDMA-Memcached can improve performance by about 68%–93%.

6.2 Use Case: Offline Burst Buffer

To illustrate the performance of RDMA-Memcached for accelerating Big Data I/O workloads, we evaluate the proposed Boldio design, presented in Section 4.3, with Hadoop running directly over Lustre parallel file system. We use the production-scale cluster, namely, SDSC Gordon [40], that is made up of 1,024 dual-socket compute nodes with two eight-core Intel Sandy Bridge processors and 64 GB DRAM, connected via InfiniBand QDR links. This cluster has sixteen 300GB Intel 710 SSDs distributed among the compute nodes. It consists of a Lustre setup with 1 PB capacity. We use 20 nodes on SDSC Gordon for our experiments. For Boldio, we use 16 compute nodes on Cluster A as YARN NodeManagers to launch Hadoop map/reduce tasks and a 4-node Boldio burst-buffer cluster over Lustre (replication = 2). For fair resource distribution, we use a 20-node Hadoop cluster for the Lustre-Direct mode. From Figure 2b, we observe that by leveraging non-blocking RDMA-enabled key-value store semantics over SSD-assisted Memcached server design, Boldio can sustain the 3x and 6.7x improvements in read and write throughputs over default Hadoop running directly over Lustre.



(a) Memcached Set/Get over simulated-MySQL



(b) Hadoop TestDFSIO Throughput with Boldio

Figure 2: Performance Benefits with RDMA-Memcached based Workloads

7 Conclusion

In this paper, we have described our approach to enhance and extend the design of Memcached in following five dimensions: 1) We have described a novel design of Memcached over RDMA capable networks. The RDMA-based design significantly improves the performance of Memcached software. 2) We introduced a hybrid transport model for RDMA-Memcached which takes advantage of the best features of RC and UD to deliver better scalability and performance than that of a single transport. 3) We proposed a hybrid memory extension for RDMA-Memcached, which enlarges aggregated cache capacity with SSD-based slab designs. 4) A new set of non-blocking APIs were proposed to extend the existing Libmemcached APIs for achieving higher communication, processing, and I/O overlapping. 5) To expand the usage scope of Memcached to accelerate I/O

in Big Data workloads such as Hadoop, Spark, etc., we proposed an RDMA-Memcached based Burst-Buffer system, which obtained the performance benefits from all above-mentioned designs as well as fault-tolerance support with replication and erasure coding.

To evaluate these advanced designs and extensions, we also proposed a set of benchmarks. To show the benefits from RDMA-Memcached designs, we have shown the use cases and the associated benefits of using RDMA-Memcached as a scalable data caching layer for online data processing workloads or a Burst Buffer component for offline data analytics. Memcached libraries with some of these designs and benchmarks are publicly available from the High-Performance Big Data (HiBD) [17] project. In the future, we intend to look at extending our designs to run in a heterogeneous environment for further taking advantage of accelerators, such as GPGPU and Intel KNL. We also plan to explore more use cases and applications with our designs.

Acknowledgments. This research was funded in part by National Science Foundation grants #CNS-1419123 and #IIS-1447804. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

[1] Oracle Corp. Designing and Implementing Scalable Applications with Memcached and MySQL, A MySQL White Paper. *Tech. Report*, 2010.

[2] D. Shankar, X. Lu, J. Jose, M. Wasi-ur-Rahman, N. Islam, and D. K. Panda. Can RDMA Benefit Online Data Processing Workloads on Memcached and MySQL? *ISPASS*, 159-160, 2015.

[3] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. *SoCC*, 6:1-15, 2014.

[4] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications. *IEEE BigData*, 2014.

[5] N. Islam, D. Shankar, X. Lu, M. Wasi-ur-Rahman, and D. K. Panda. Accelerating I/O Performance of Big Data Analytics on HPC Clusters through RDMA-based Key-Value Store. *ICPP*, 2015.

[6] D. Shankar, X. Lu, and D. K. Panda. Boldio: A Hybrid and Resilient Burst-Buffer Over Lustre for Accelerating Big Data I/O. *IEEE BigData*, 2016.

[7] Memcached: High-Performance, Distributed Memory Object Caching System. *http://memcached.org/*, 2017.

[8] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur-Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. *ICPP*, 2011.

[9] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur-Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. *CCGrid*, 236-243, 2012.

[10] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory *NSDI*, 401-414, 2014.

[11] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. *NSDI*, 2014.

[12] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. *USENIX ATC*, 2013.

[13] X. Ouyang, N. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda. SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks. *ICPP*, 470-479, 2012.

[14] H. Lim, B. Fan, D.G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. *SOSP*, 1-13, 2011.

[15] Facebook. McDipper: A Key-Value Cache for Flash Storage. *https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920*, 2013.

[16] Twitter. fatcache: Memcached on SSD. *https://github.com/twitter/fatcache*.

[17] OSU NBC Lab. High-Performance Big Data (HiBD). *http://hibd.cse.ohio-state.edu*.

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. *SIGCOMM*, 2014.

[19] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. HydraDB: A Resilient RDMA-driven Key-Value Middleware for In-Memory Cluster Computing. *SC*, 22:1-11, 2015.

[20] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. *NSDI*, 371-384, 2013.

[21] Twitter. twemcache: Twitter Memcached. *https://github.com/twitter/twemcache*.

[22] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. LAMA: Optimized Locality-Aware Memory Allocation for Key-Value Cache. *USENIX ATC*, 57-59, 2015.

[23] Y. Zhang, and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. *MSST*, 1-10, 2015.

[24] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *FAST*, 167-180, 2016.

[25] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *VLDB*, 1226-1237, 2015.

[26] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. *SoCC*, 43-57, 2015.

[27] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached Appliance. *FPGA*, 245-254, 2013.

[28] D. Shankar, X. Lu, N. Islam, M. Wasi-Ur-Rahman, and D. K. Panda. High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits. *IPDPS*, 393-402, 2016.

[29] OpenFabrics Alliance. *https://www.openfabrics.org*.

[30] OSU NBC Lab. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. *http://mvapich.cse.ohio-state.edu/*.

[31] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design Using Unreliable Datagram for Ultra-scale InfiniBand Clusters. *ICS*, 180-189, 2007.

[32] S. Zhang, H. Jizhong, L. Zhiyong, K. Wang and S. Feng. Accelerating MapReduce with Distributed Memory Cache. *ICPADS*, 472-478, 2009.

[33] P. Xuan, W. B. Ligon, P. K. Srimani, R. Ge and F. Luo. Accelerating Big Data Analytics on HPC Clusters using Two-Level Storage. *DISCS*, 2015.

[34] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. *FAST*, 253-265, 2009.

[35] D. Shankar, X. Lu, and D. K. Panda. High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads. *ICDCS*, 2017.

[36] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS*, 53-64, 2012.

[37] D. Shankar, X. Lu, M. Wasi-Ur-Rahman, N. Islam, and D. K. Panda. Benchmarking Key-Value Stores on High-Performance Storage and Interconnects for Web-Scale Workloads. *IEEE BigData*, 2015.

[38] Spymemcached. *http://code.google.com/p/spymemcached/*.

[39] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. *SoCC*, 2010.

[40] SDSC Gordon. *http://www.sdsc.edu/services/hpc/hpc_systems.html*.

Beyond Simple Request Processing with RAMCloud

Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman
University of Utah

Abstract

RAMCloud is a scale-out data center key-value store designed to aggregate the DRAM of thousands of machines into petabytes of storage with low 5 µs access times. RAMCloud was an early system in the space of low-latency RDMA-based storage systems. Today, it stands as one of the most complete examples of a scalable low-latency store; it has features like distributed recovery, low fragmentation through a specialized parallel garbage collector, migration, secondary indexes, and transactions.

This article examines RAMCloud's key networking and dispatch decisions, and it explains how they differ from other systems. For example, unlike many systems, RAMCloud does not use one-sided RDMA operations, and it does not partition data across cores to eliminate dispatch and locking overheads.

We explain how major RAMCloud functionality still exploits modern hardware while retaining a loose coupling both between a server's internal modules and between servers and clients by explaining our recent work on Rocksteady, a live data migration system for RAMCloud. Rocksteady uses scatter/gather DMA and multicore parallelism without changes to RAMCloud's RPC system and with minimal disruption to normal request processing.

1 Introduction

Remote direct memory access (RDMA) has been a recent hot topic in systems and database research. By eliminating the kernel from the network fast path and allowing the database to interact directly with the network card (NIC), researchers have built systems that are orders of magnitude faster than conventional systems both in terms of throughput and latency.

RAMCloud is an example of early research in this space. RAMCloud [15] is a key-value store that keeps all data in DRAM at all times and is designed to scale across thousands of commodity data center nodes. In RAMCloud, each node can service millions of operations per second, but its focus is on low access latency. End-to-end read and durable write operations take just 5 µs and 14.5 µs respectively on our hardware (commodity hardware using DPDK). RAMCloud supports distributed unordered tables, transactions [10], and ordered secondary indexes [9].

Internally, RAMCloud servers leverage modern NIC features for high performance, but, in many ways, its internal networking and dispatch structure is somewhat conventional. Other research systems have more aggressive approaches that use techniques like simpler data models (for example, distributed shared memory), strict workload partitioning across cores, or one-sided RDMA operations to eliminate server CPU involvement

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

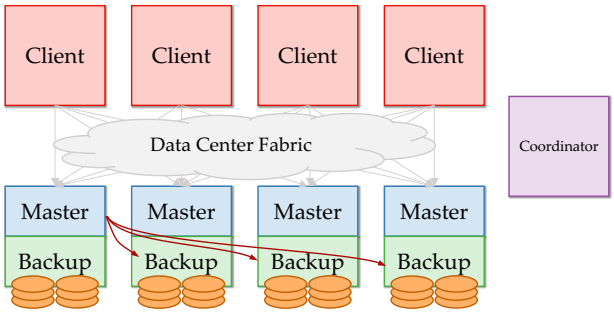


Figure 1: RAMCloud Architecture.

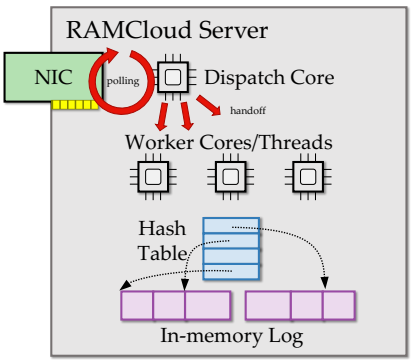


Figure 2: Internal Server RPC Dispatch.

and improve access latency [3, 4, 8, 6, 12]. Today, other systems achieve inter-machine latencies of 2.5 µs and serve hundreds of times more operations per second per machine [11, 16].

Despite this, RAMCloud's internal design has advantages. This article considers RAMCloud's internal server networking, dispatching, and synchronization; extracts lessons from its design choices; and discusses its tradeoffs. RAMCloud's design results in loose coupling between servers and clients and between the internal modules of servers; it also helps support many of the "extra" things that servers in a large cluster must do. To be useful, low-latency scale-out stores must evolve from micro-optimized key-value stores to more robust systems that deal with fault-tolerance, (re-)replication, load balancing, data migration, and memory fragmentation.

Finally, we exemplify RAMCloud's principles by describing the design of a new live migration system in RAMCloud called Rocksteady. Rocksteady uses scatter/gather DMA while building only on simple RPCs, and it runs tightly pipelined and in parallel at both the source and destination while minimizing impact on tail latency for normal request processing.

2 RAMCloud Networking, Transports and Dispatch

In RAMCloud, each node (Figure 1) operates as a *master*, which manages RAMCloud records in its DRAM and services client requests, and a *backup*, which stores redundant copies of records from other masters on local SSDs. Each cluster has one quorum-replicated *coordinator* that manages cluster membership and table-partition-to-master mappings.

RAMCloud only keeps one copy of each record in memory to avoid replication in expensive DRAM; redundant copies are logged to (remote) flash. It provides high availability with a fast distributed recovery that sprays the records of a failed server across the cluster in 1 to 2 seconds [14], restoring access to them. RAMCloud manages in-memory storage using an approach similar to that of log-structured filesystems, which allows it to sustain 80-90% memory utilization with high performance [17].

Request processing in each server is simple (Figure 2); its key design choices are explained here.

Kernel Bypass and Polling. RAMCloud processes all requests with no system calls. It relies on kernel bypass to interact directly with the NIC. Interrupt processing would introduce extra request processing latency and would hurt throughput, so RAMCloud employs a dispatch thread pinned to a single CPU core that polls NIC structures to fetch incoming requests. When a RAMCloud process is completely idle, it still fully consumes a single core. As the number of cores per socket increases, the relative efficiency of polling improves. Aside from keeping data in DRAM, these two optimizations alone account for most of RAMCloud's latency improvements.

Request Dispatching. The remaining cores on the CPU socket are worker cores; they run a single, pinned thread each. The dispatch core takes incoming requests from the NIC and hands each of them off to a worker core. Dispatch comes at a significant cost in RAMCloud; for example, workers complete small read operations in

less than 1 µs, but dispatch costs (handoff and extra cache misses) account for about 40% of request processing latency on the server. Furthermore, data is not partitioned within a machine, and each worker core can access any record; hence, workers need additional synchronization for safety. Dispatching is also the key throughput bottleneck for workloads consisting of small requests and responses. Section 2.1 below discusses how this seemingly costly decision improves efficiency when considering the system as a whole.

NUMA Oblivious. RAMCloud is intentionally non-uniform memory access (NUMA) oblivious. Cross-socket data access and even cross-socket access to reach the NIC is expensive. The expectation is that RAMCloud servers should always be restricted to a single CPU socket. Making RAMCloud NUMA-aware would only provide small gains. For example, with two socket machines, some RAMCloud tables and indexes could see up to a 2× speed up if their data fit within what two sockets could service rather than one. Unfortunately, nearly all tables are likely either to be most efficient on just one socket or to require the resources many sockets [2]. Tables that require the resources of exactly two sockets are likely rare. Servers already need some form of load balancing that works over the network, and intersocket interconnects do not offer significant enough bandwidth improvements (2-3× compared to network links, and they only reach one or three other sockets) to make them worth the added complexity. If machines with many sockets (8 or more, for example) become commonplace, then this decision will need to be revisited.

Remote Procedure Call (RPC). *All* interactions between RAMCloud servers and between clients and servers use RPC. Some systems use one-sided RDMA operations to allow clients to directly access server-side state without involving the server's CPU. RAMCloud explicitly avoids this model, since it tightly infuses knowledge of server structures into client logic, it complicates concurrency control and synchronization, and it prevents the server from reorganizing records and structures. One-sided operations suffer from other problems as well: they are hardware and transport specific, and they do not (yet) scale to large clusters [7].

Zero-copy Networking. Even though clients are not allowed to directly access server state via one-sided RDMA operations, servers do use scatter/gather DMA to transmit records without intervening copies. RAMCloud's RPC layer allows regions of memory to be registered with NICs that support userlevel DMA, including its in-memory log, which holds all records. This can have a significant performance impact on bandwidth-intensive operations like retrieving large values or gathering recovery data.

Transport Agnosticism. RAMCloud supports a wide variety of transport protocols and hardware platforms, and it applies the same consistent RPC interface over all of them. Initially, the lowest latency protocol was built over Infiniband with Mellanox network cards, but, today, a custom, packet-based transport protocol running atop Ethernet via DPDK is the preferred deployment. For transports and hardware that support it, RAMCloud uses zero-copy DMA to transmit records without extra configuration. Some transports supported by RAMCloud, like Infiniband's Reliable Connected mode are fully implemented in hardware on the NIC; the NIC provides fragmentation, retransmission, and more. Overall, though, they provide decreased transparency compared to software transport protocols, and they are harder to debug.

2.1 Interface and Threading Tradeoffs

RAMCloud's choice of strict RPC interfaces and its use of concurrent worker threads over a shared database may seem simplistic compared to other systems that carefully avoid server CPU use and cross-core synchronization. On an operation-by-operation basis these choices are costly, but not when the system is considered as a whole.

Using RPC allows semantically rich operations on the server and helps in synchronizing server background work with normal request processing. For example, a RAMCloud read operation fetches a value based on a string key, which involves two server-side DRAM accesses: one for a hash table bucket, and another to access the value and check its full key. One-sided operations would require two round-trips to fetch the value, destroying their latency advantage. Clever techniques like speculatively fetching extra buckets, caching chunks of remote structures, or inlining values in buckets if they are fixed size can help, but they make assumptions about locality

and index metadata size. Reads are the simplest case, and more complicated operations only make exploiting one-sided operations harder.

Initially, RAMCloud servers were single-threaded. Each server directly polled the NIC for incoming requests and executed each operation in a simple run-to-completion loop. Seemingly, this fit RAMCloud's latency goal; *median* access times for small reads were 400 ns lower than what RAMCloud's dispatch gives today. This model is similar to partitioned systems; each machine would need to run a separate server on each core. Unfortunately, the system was fragile. Fast recovery required fast failure detection, and even small hiccups in request processing latency would result in timeouts. The timeouts would result in failure detection pings, which would get stuck behind the same hiccups. The resulting false recovery would put additional load on the servers, causing cascading failures. Responding to pings is a trivial background task; since then, RAMCloud has evolved to include several heavier, but essential background operations that would only exacerbate the issue.

RAMCloud's primary value comes from its low and *consistent* access latency. Its 99.9^{th} percentile read access time is just 100 µs. Its flexible dispatch is a key reason for that. When a database is strictly partitioned across cores, requests are statistically likely to collide even if they are uniformly distributed across cores [13]. To combat this, partitioned systems must significantly underutilize hardware if access latency distribution is a concern. Partitioning also requires more careful and aggressive load redistribution to combat transient workload changes. There is also evidence that real-world large-scale in-memory stores are DRAM-capacity limited, not throughput limited [1], eliminating the main argument against RAMCloud's flexible dispatch.

RAMCloud servers aren't passive containers; they actively perform tasks in the background. For example, when servers crash in RAMCloud, every server assists in the recovery of the crashed server; similarly, every server must re-replicate some of its backup data to ensure crashes do not eventually lead to data loss. One of the most important background tasks in RAMCloud is log cleaning, a form of generational garbage collection that constantly defragments live data in memory [17]. Cleaning is key to long-term high memory utilization (and RAMCloud's cost effectiveness, since DRAM is its primary cost). Log cleaning runs with little impact on normal case performance, and it automatically scales to more cores when memory utilization is high and performance is limited by garbage collection.

Strictly partitioning the database among cores and eliminating synchronization would make log cleaning impractical. Cleaning would have to be interleaved with normal operations, which would hurt latency (head-of-line blocking) and throughput (interleaving tasks would thrash core caches). It would also make log cleaning less effective. The cleaner tries to globally separate low-churn records from high-churn records to reduce write amplification due to cleaning [1, 17]; partitioning data among cores would restrict its analysis to partition boundaries.

Aside from handoff and synchronization costs, RAMCloud's worker model leaves room for improvement. Partitioned systems effectively rely on clients on one machine to route requests to a specific core on a server to eliminate request handoff costs. Strictly mapping requests to cores can hurt tail latency, but an optimistic approach where clients issue requests directly to server cores and fallback to general dispatch could improve latency and throughput. Ideally, NIC hardware support like Intel's Flow Director and Mellanox's flow steering could help, but the latency and overhead to reconfigure the NIC-level steering tables makes them hard to use to combat transient contention.

A second serious issue is managing client threads. Synchronous requests to a remote server (both client-to-server or server-to-server) block a thread for 5 µs. This leaves too little time to efficiently (OS) context switch to another task and back. As a result, RPC callers must juggle asynchronous RPC requests for efficiency, especially on servers where remote replication delays would otherwise dominate worker CPU time. Better userlevel threading primitives could help and simplify code.

A good model for stable and low-latency request dispatching remains an open research question today.

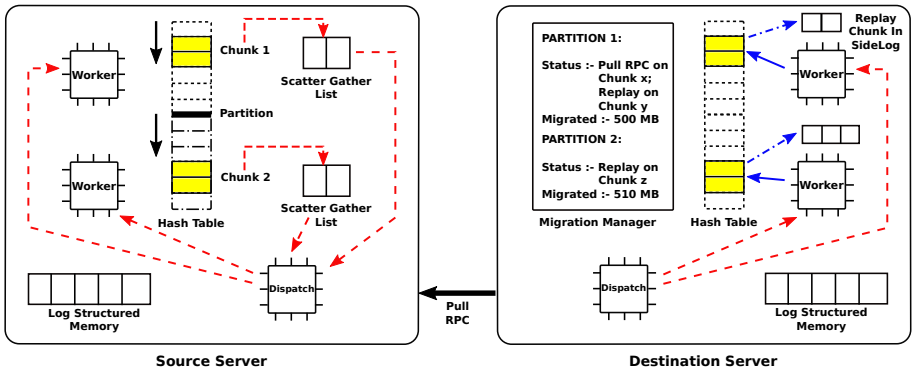


Figure 3: Low-impact Pipelined and Parallel Data Migration.

3 Rocksteady: Fast, Low-impact Data Migration

Data migration demonstrates the effectiveness of RAMCloud's dispatch model. RAMCloud must be able to quickly adapt to changing workloads. This requires scaling up, scaling down, and rebalancing data and load in response to changing workloads and server failures. Ideally, since all data lives in memory and RAMCloud employs fast networking, data migration should be fast. However, moving data rapidly without impacting access latency is challenging. RAMCloud's dispatch model helps with this.

Rocksteady is our ongoing effort to build a fast live-migration mechanism that exploits RAMCloud's unique properties while retaining 99.9th percentile access latencies of 100 µs. It is fully asynchronous at both the migration source and destination; it uses modern scatter/gather DMA for zero-copy data transfer; and it uses deep pipelining and fine-grained adaptive parallelism at both the source and destination to accelerate transfer while instantly yielding to normal-case request processing.

3.1 Rocksteady Design

Rocksteady instantly transfers ownership of all involved records to the destination at migration start, similar to some other approaches [5]. The destination subsequently handles all requests for records; writes can be serviced the moment ownership is transferred over, while reads can be serviced only after the requested records have been migrated over from the source. If demanded records are missing at the destination, they are given migration priority. This approach favors instant load reduction at the source.

All data transfer in Rocksteady is initiated by the destination server; a *pull* RPC request sent to the source returns a specified amount of data which is then replayed to populate the destination's hash table. There are two benefits to employing a destination driven protocol. First, the source server is stateless; a migration manager at the destination keeps track of all progress made by the protocol. Second, the loose coupling between source and the destination helps pipelining and parallelization; the source and the destination can proceed more independently to avoid stalls.

Rocksteady is heavily influenced by RAMCloud's core principles. Figure 3 gives an overview of how the destination pulls data and controls parallelism at both the source and destination. Its key features are explained below in more detail.

Asynchronous and Adaptive. Rocksteady is integrated into RAMCloud's dispatch mechanism. All pull RPC requests sent to the source are scheduled onto workers by the source's dispatch thread. Similarly, all data received at the destination is scheduled for replay by the destination's dispatch thread. Leveraging the dispatch gives four benefits. First, Rocksteady blends in with background system tasks like garbage collection and (re-)replication. Second, Rocksteady can adapt to system load ensuring minimal disruption to normal request processing (reads

and writes) while migrating data as fast as possible; to help with this, pull and replay tasks are fine-grained, so workers can be redirected quickly when needed. Third, since the source and destination are decoupled, workers on the source can always be busy collecting data for migration, while workers on the destination can always make progress by replaying the earlier responses. Finally, Rocksteady makes no assumptions of locality or affinity; a migration related task (pull or replay) can be dispatched to *any* worker, so any idle capacity on either end can be put to use.

Zero-copy Networking with RPCs. *All* data transfer in Rocksteady takes place through RAMCloud's RPC layer, allowing the protocol to be both transport and hardware agnostic. Destination initiated one-sided RDMA reads may seem to promise fast transfers without the source's involvement, but they breakdown because the records under migration are scattered across the source's in-memory log. RDMA reads do support scatter/gather DMA, but reads can only fetch a single contiguous chunk of memory from the remote server (that is, a single RDMA read *scatters* the fetched value locally; it cannot *gather* multiple remote locations with a single request). As a result, an RDMA read from the destination could only return a single data record per remote operation unless the source pre-aggregated all records for migration beforehand, which would undo its benefits. Additionally, one-sided RDMA would require the destination to be aware of the structure and memory addresses of the source's log. This would complicate synchronization, for example, with RAMCloud's log cleaner. Epoch-based protection can help (normal-case RPC operations like read and write synchronize with the local log cleaner this way), but extending epoch protection across machines would couple the source and destination more tightly. Even without one-sided RDMA, Rocksteady does use scatter/gather DMA [15] when supported by the transport and the NIC to transfer data from the source without *any* intervening copies.

Parallel and Pipelined. Rocksteady's *migration manager* runs at the destination's dispatch thread. At migration start, the manager logically divides the source server's key hash space into *partitions*; these partitions effectively divide up the source's hash table, which contains a pointer to each record that needs to be migrated. Partitioning is key to allowing parallel pulls on the source. A different pull request can be issued concurrently for each partition of the hash space, since they operate on entirely different portions of the source's hash table and records. When dispatched at the source, a worker linearly scans through the partition of the hash table to create a scatter/gather list of pointers to the records in memory, avoiding *any* intervening copies.

Records from completed pull requests are replayed in parallel into the destination's hash table on idle worker threads. Pull requests from distinct partitions of the hash table naturally correspond to different partitions of the destination's hash table as well. As a result, parallel replay tasks for different partitions proceed without contention. Depending on the destination server's load, the manager can adaptively scale the number of in-progress pull RPCs, as well as the number of in-progress replay tasks. The source server prioritizes normal operations over pull requests, which allows it to similarly adapt when it detects load locally.

Performing work at the granularity of distinct hash table chunks also lets Rocksteady pipeline work within a hash table partition. Whenever a pull RPC completes, the migration manager first issues a new, asynchronous pull RPC for the next chunk of records on the same partition and immediately starts replay for the returned chunk.

Eliminating Synchronous Re-replication Overhead. During normal operation, all write operations processed by a server are synchronously replicated to three other servers acting as backups. This adds about 10 µs to the operation, though concurrent updates are amortized by batching. Migration creates a challenge; if records are appended to this log and replicated, then migration will be limited by log replication speed.

Rocksteady avoids this overhead by making re-replication lazy. The records under migration are already recorded in the source's recovery log, so immediate re-replication is not strictly necessary for safety. All updates that the destination makes to records that have been migrated to it are logged in the normal way as part of its recovery log, but record data coming from the source is handled differently to eliminate replication from the migration fast path.

To do this, each server's recovery log can fork *side logs*. A side log is just like a normal log, except that it

is not part of the server's recovery log until a special log record is added to the recovery log to merge it in. This makes merging side logs into the main recovery log atomic and cheap. Rocksteady uses them in two ways. First, all records migrated from the source are appended to an in-memory side log; the data in side logs is replicated as migration proceeds, but lazily and at low priority. The side log is only merged into the server's recovery log when all of the data has been fully replicated. The second purpose of the side logs is to avoid contention between parallel replay workers; each worker uses a separate side log, and all of them are merged into the main log at the end of re-replication. During migration, the destination serves requests for records that are only in memory in a side log and have not been replicated or merged into the main log; next, we show how to make this process safe.

Lineage-based Fault Tolerance for Safe Lazy Re-replication. Avoiding synchronous re-replication of migrated data creates a serious challenge for fault tolerance. If the destination crashes in the middle of a migration, then neither the source nor the destination would have all of the records needed to recover correctly; the destination may have serviced writes for some of the records under migration, since ownership is transferred instantly at the start of migration. Rocksteady takes a unique approach to solving this problem that relies on RAMCloud's fast recovery. RAMCloud's distributed recovery works to restore a crashed server's records back into memory in 1 to 2 seconds.

To avoid synchronous re-replication of all of the records as they are transmitted from the source to the destination, the migration manager registers a dependency for the source server on the tail of the destination's recovery log at the cluster coordinator. If either the source or the destination crashes during migration, Rocksteady transfers ownership of the data back to the source. To ensure the source has all of the destination's updates, the coordinator induces a recovery of the source server which logically forces replay of the destination's recovery log tail along with the source's recovery log. This approach keeps things simple by reusing RAMCloud's recovery at the expense of extra effort in the rare case that a machine involved in migration crashes.

Rocksteady shows the benefits of RAMCloud's design. It uses simple (asynchronous) RPC, but it benefits from zero-copy scatter/gather DMA. It uses multicore parallelism, but it avoids latency inducing head of line blocking. As RAMCloud is extended with more and more functionality, its simple networking and dispatch model becomes increasingly important.

4 Conclusion

Userlevel NIC access, RDMA, and fast dispatch have been hot topics driven in large part by the elimination of I/O in favor of DRAM as a storage medium. RAMCloud was an early system in this space, but many models have emerged for exposing large clusters of DRAM-based storage over the network and for building systems on top.

Hardware capabilities significantly influence RAMCloud's networking and dispatch design choices, yet, in many ways, its choices are rather conventional. Sticking to a server-controlled dispatch model and strict RPC interfaces has helped RAMCloud remain modular by insulating servers from one another and insulating clients from server implementation details. These choices have enabled several key features that are likely to prove to be essential in a real, large-scale system while preserving low and predictable access latency.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1566175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Cidon, D. Rushton, and R. Stutsman. Memshare: A Dynamic Multi-tenant Memory Key-value Cache. Arxiv, October, 2016.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *USNIX OSDI '12*, pages 261–264.

[3] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *USENIX NSDI '14*, pages 401–414.

[4] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance . In *ACM SOSP '15*, pages 85–100.

[5] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *ACM SIGMOD '15*, pages 299–313.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC '16*, pages 437–450.

[7] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *USENIX OSDI '16*, pages 185–201.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM '14*, pages 295–306.

[9] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *USENIX ATC '16*, pages 57–70.

[10] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *ACM SOSP '15*, pages 71–86.

[11] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-value Store Server Platform. In *ACM ISCA '15*, pages 476–488.

[12] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USNIX NSDI '14*, pages 429–444.

[13] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.

[14] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *ACM SOSP '11*, pages 29–41.

[15] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, Aug. 2015.

[16] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-Tolerant Software Distributed Shared Memory. In *USENIX ATC '15*, pages 291–305.

[17] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *USENIX FAST '14*.

33rd IEEE International Conference on Data Engineering 2017
April 19-22, 2017, San Diego, CA

http://icde2017.sdsc.edu/

http://twitter.com/icdeconf #icde17

Call for Participation

The annual ICDE conference addresses research issues in designing, building, managing, and evaluating advanced data systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Venue: ICDE 2017 will be held at the Hilton San Diego Resort and Spa

Conference Events:

- Research, industry and application papers
- ICDE and TKDE posters
- Keynote talks by Volker Markl, Laura Haas, and Pavel Pevzner
- Ph.D. Symposium with keynote speaker Magda Balazinska
- Panels on (i) Data Science Education; (ii) Small Data; and (iii) Doing a Good Ph.D.
- Tutorials on (i) Web-scale blocking, iterative, and progressive entity resolution; (ii) Bringing semantics to spatiotemporal data mining; (iii) Community search over big graphs; (iv) The challenges of global-scale data management; (v) Handling uncertainty in geospatial data.
- Topic-based demo sessions on (i) Cloud, stream, query processing, and provenance; (ii) Graph analytics, social networks, and machine learning; and (iii) Applications, data visualization, text analytics, and integration.

General Chairs:

Chaitanya Baru (UC San Diego, USA)

Bhavani Thuraisingham (UT Dallas, USA

PC Chairs:

Yannis Papakonstantinou(UC San Diego)

Yanlei Diao (Ecole Polytechnique, France)

Affiliated Workshops:

- **HDMM 2017:** 2nd International Workshop on Health Data Management and Mining
- **DESWeb 2017:** 8th International Workshop on Data Engineering Meets the Semantic Web
- **RAMMMNets 2017:** Workshop on Real-time Analytics in Multi-latency, Multy-party, Metro-scale Networks
- **Active'17** Workshop on Data Management on Virtualized Active Systems and Emerging Hardware
- **HardDB 2017** Workshop on Big Data Management on Emerging Hardware
- **WDSE:** Women in Data Science & Engineering Workshop

IEEE computer society

IEEE Advancing Technology for Humanity

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398