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

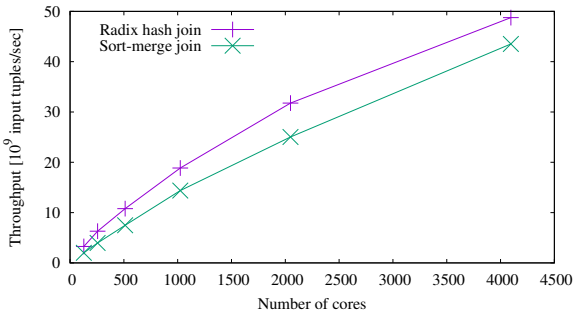


Figure 1: Distributed join algorithms can scale to hundreds of machines with thousands of CPU cores.

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

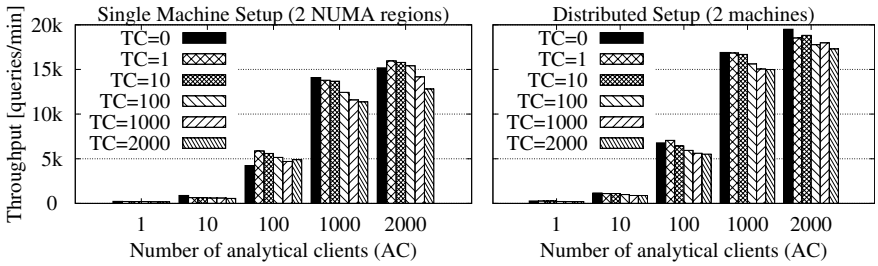


Figure 2: BatchDB isolates workloads generated by analytical clients (AC) and transactional clients (TC) through data replication. Cross-machine replication reduces interference and improves overall throughput.

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.

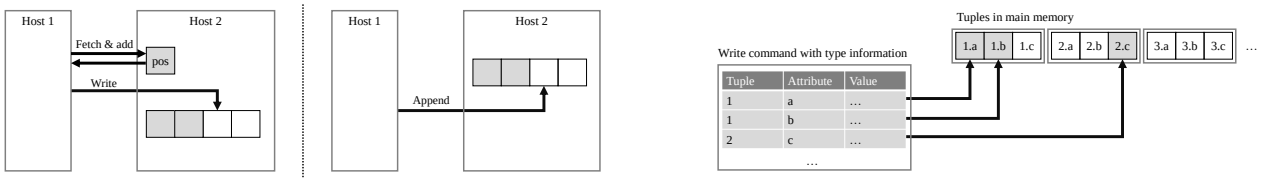
3.3 Distributed Coordination

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