A Spinning Join That Does Not Get Dizzy

Philip W. Frey ⊞ Romulo Goncalves ⊟ Martin Kersten ⊟ Jens Teubner ⊞

⊞ Systems Group
Department of Computer Science
ETH Zurich
`firstname.lastname@inf.ethz.ch`

⊟ Database Architectures and
Information Access Group
CWI Amsterdam
`goncalve|mk@cwi.nl`

***Abstract*— As network infrastructures with 10 Gb/s bandwidth and beyond have become pervasive and as cost advantages of large commodity-machine clusters continue to increase, research and industry strive to exploit the available processing performance for large-scale database processing tasks.**

In this work we look at the use of *high-speed networks* for *distributed join processing*. We propose *Data Roundabout* as a lightweight transport layer that uses *Remote Direct Memory Access (RDMA)* to gain access to the throughput opportunities in modern networks. The essence of *Data Roundabout* is a ring-shaped network in which each host stores one portion of a large database instance. We leverage the available bandwidth to (continuously) pump data through the high-speed network.

Based on *Data Roundabout*, we demonstrate *cyclo-join*, which exploits the cycling flow of data to execute distributed joins. The study uses different join algorithms (*hash join* and *sort-merge join*) to expose the pitfalls and the advantages of each algorithm in the data cycling arena. The experiments show the potential of a large distributed main-memory cache glued together with *RDMA* into a novel distributed database architecture.

I. Introduction

Modern data warehouse installations maintain gigantic amounts of business, user, or customer information. It is natural that users expect to be able to query these data flexibly and efficiently, preferably with interactive query processing times.

Processing power—in terms of raw compute power, but also in terms of the necessary main memory resources—is rarely the problem in satisfying these demands. Data centers with real hardware or cloud infrastructures with virtual machines provide an abundance of compute power, at ever-decreasing costs. On the down side, these processing capabilities are only available as distributed network resources and *communication overhead* often limits the amount of processing power that can be used in practice.

Paradoxically, the communication infrastructure is *not* the bottleneck either: today's *high-speed networks*—InfiniBand or 10 Gb/s Ethernet to name just two—provide ample of bandwidth, but remain limited by processing speeds on the hosts.

The dilemma is a consequence of the *mismatch* between the communication schemes of existing distributed systems and the data access patterns that would be required to achieve high network throughput. As we elaborated in [11], high-speed transport mechanisms like *Remote Direct Memory Access (RDMA)* can only operate efficiently if the application adheres to particular communication patterns.

With the *Data Roundabout*, we propose a transport mechanism for distributed databases. *Data Roundabout* ensures a communication pattern that fits the RDMA communication model, while at the same time providing a transport layer that is well suited to implement critical database functionality.

We demonstrate the effectiveness of *Data Roundabout* with *cyclo-join*, a *distributed join algorithm* that leverages RDMA and known CPU-efficient join evaluation techniques to compute very large database joins entirely in (distributed) main memory. This way, *cyclo-join* can reach join throughput rates that are far beyond what conventional networks or commodity hard disks could provide.

In this report we focus on two aspects of our design:

Data Roundabout and RDMA. The performance advantages of RDMA can only be leveraged with a careful algorithmic design that respects the characteristics of hardware-accelerated network communication. We discuss the relevant aspects of RDMA and their implications on the design of our transport layer *Data Roundabout*.

Cyclo-Join Mechanism. Cyclo-join orchestrates the execution of any traditional (single-host) join algorithm in a distributed setup, relying on a fast interconnect to transfer data. In an in-depth evaluation performed on RDMA hardware, we analyze and illustrate the implications of *cyclo-join* depending on the problem type and compare it with local join algorithms.

The broader context of our work is the *Data Cyclotron* project, a joint effort between CWI Amsterdam and ETH Zurich to explore non-traditional architectures to cope with the ever-increasing requirements from large-scale business intelligence and eScience applications. Given the wide availability of RDMA, our approach is to re-think distributed database processing and consider the network as our friend, not as an enemy to be evaded at all cost [16].

This report proceeds as follows. The upcoming Section II provides the necessary context and reviews work on distributed join processing. Hardware-accelerated network processing (RDMA) and its application to the *Data Roundabout* approach are on our agenda for Section III. The architecture and algorithms used for distributed join processing on our *cyclo-join* mechanism are presented in Section IV. We evaluate *cyclo-join* in-depth in Section V, before we look at further related work (Section VI) and summarize our own (Section VII).

II. State of the Art and the *Data Roundabout*

The technology behind most distributed database systems today dates back to early prototypes such as the SDD-1 [23], Distributed INGRES [9], or System R* [20] systems. The underlying assumptions and the approaches taken are largely a consequence of the network environments at that time. A short recap of history is provided.

A. Traditional Distributed Query Processing

Most importantly, in the early days, network communication was fairly slow (3 Mb/s were considered a "high-speed" network) and thus treated as a major cost factor in distributed query processing, if not the only one considered at all [3]. In a distributed setting, the primary goal of join processing techniques, such as the use of semi-joins [4], the shipping of pre-filtered tables [21], or System R*'s "fetch matches" technique [21], was to avoid network communication, often at the expense of additional CPU work.

Another consequence is the basic architecture that has become pervasive in distributed query processing. All data is *partitioned* over available network hosts (often only few of them) and remains there mostly *static*. *Queries*, by contrast, are *shipped* between hosts during query processing, usually along with state information or intermediate query results. This processing model is a good fit for classical workloads, where most queries are known in advance (and data can be partitioned accordingly) and involve only few, simple join predicates.

Only few systems break out of this basic architecture, most notably the scalable distributed data structures proposed by Litwin et al. [19], which adapt to the arrival of data.

B. Distributed Query Processing Today

Today, roughly three decades later, the hardware landscape and application demands have changed significantly. Commodity networks provide extremely high throughput and low latency and, thanks to hardware acceleration, incur only negligible communication cost. Real-time data mining or business intelligence applications have shifted the challenges in distributed large-volume data processing toward complex queries [8] and reflect an increasing importance of ad-hoc queries.

Another shift is driven by economic forces. In the spirit of *cloud computing*, large installations of commodity off-the-shelf systems are becoming preferred over few high-performance machines. Cost effectiveness, fault tolerance, and scalability are achieved by adding and removing machines "as you go". Cloud-style operational models defeat the dedication of machines for keeping specific data or performing specific tasks. Instead, they demand trivial replacement, addition, or removal of network hosts as well as a low overall system complexity.

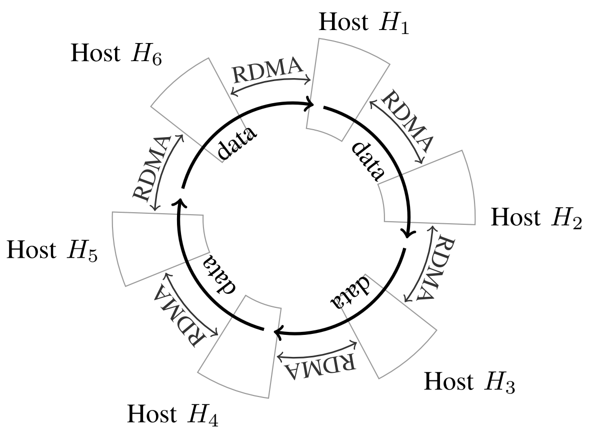


Fig. 1. The *Data Roundabout*. Hosts H_i are organized as ring, connected by high-speed RDMA links.

C. *The* Data Roundabout *Approach*

As a starting point to explore novel architectures for their potential to meet the above requirements, we propose the *Data Roundabout* which consists of a (potentially large) number of commodity systems. All participating nodes are connected to form a *logical storage ring* structure (currently implemented using a star-shaped physical network).

Each node communicates only with its immediate neighbors via a bidirectional high-speed RDMA connection, as illustrated in Figure 1 for a *Data Roundabout* of size six (*i.e.*, one that consists of six hosts). We assume the combined main memory of all participating hosts to be large enough to hold the *hot set* of the database in a distributed fashion; other data may be kept in slower, distributed disk space.

A fundamental difference to the classical distributed database design is that we keep queries and their state *static* and *move base data* over the network instead. We keep (the hot set of the) data *continuously circulating* in the ring. Queries remain local to one or more nodes and pick necessary pieces of data as they flow by in the ring.

The *Data Roundabout* satisfies the requirements sketched before: the ring is built from commodity systems and its design and data flow pattern are deliberately kept simple, in order to ease maintenance and scalability. Hence, a *Data Roundabout* system can trivially be extended or shrunken, depending on CPU and/or main memory demand. Any failing node can easily be replaced by another machine (or its role can be taken over by some other node in the ring). Furthermore, we do not partition the data based on any a priori workload knowledge which lets us naturally handle ad-hoc queries.

Taking full advantage of modern networking hardware and the idea of rotating data, however, requires certain care in the design of a distributed algorithm. In this work we focus on RDMA transport facilities (see upcoming section) and on the processing of joins in *cyclo-join*.

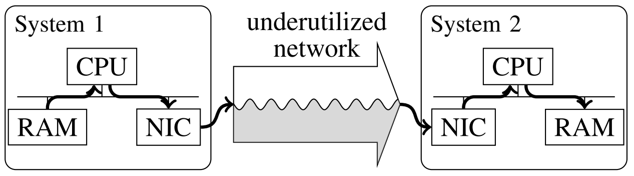


Fig. 2. Kernel TCP/IP. Data copying goes through CPU; several memory bus crossings.

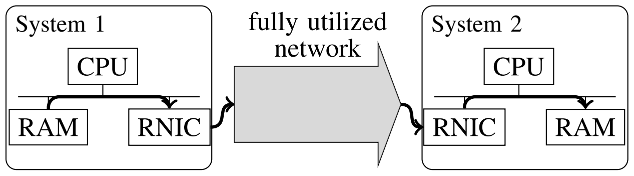


Fig. 4. Network transfer using RDMA. RNICs handle data transfer autonomously; data has to cross each memory bus only once.

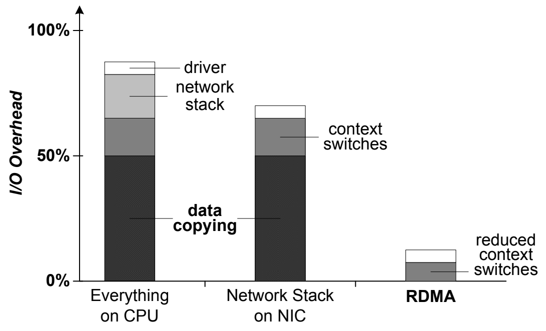


Fig. 3. Only RDMA is able to significantly reduce the local communication overhead induced at high-speed data transfers.

III. Remote Direct Memory Access

Network communication is known to be compute- and memory I/O-intensive [10]. High-volume data transfers thus depend on dedicated hardware assistance, which modern network cards provide in terms of *Remote Direct Memory Access (RDMA)*. Network interface cards (NICs) that provide RDMA functionality are also called *RDMA-enabled NICs* or *RNICs*.

To make efficient use of RDMA, however, applications have to respect some of the characteristics of the hardware-accelerated transport mechanism. This section summarizes the most relevant characteristics, built on our earlier RDMA evaluation [11]. They motivate the design of *Data Roundabout*, which is discussed toward the end of this section.

A. Traditional TCP Communication

In traditional TCP network communication, typically implemented using the Berkeley `socket` API, the operating system kernel is in charge of processing the network stack, which includes executing the respective protocols and moving the payload data. Strongly simplified, this leads to a data flow as shown in Figure 2 for a data transfer from System 1 to System 2. Data is fetched from the main memory of the source host, processed by the source host CPU, then fed into the sending network interface card, which sends the data over the network. At the receiver side, data is again processed by the CPU, then placed in main memory.

The intensive involvement of the kernel on both sides causes a substantial amount of CPU load. A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/s network throughput [10].

Interestingly, the major CPU cost factor in traditional TCP handling is *not* network stack processing. In Figure 3, we separated the different CPU cost factors that arise in TCP processing (see leftmost bar). As can be seen, protocol processing (cf. "network stack") plays only a minor role in the high CPU load. Thus, off-loading the protocol processing to a *TCP offload engine* on a modern NIC usually yields only little advantage over a fully software-based setup (as can be seen in the middle of Figure 3).

Instead, CPU cost is dominated by *moving the payload data*. As indicated in Figure 2, the data to be transferred crosses the system's memory bus at least two times on its way from the host memory to the network card (the same happens symmetrically on the receiver side[1]). In reality, the enforcement of process isolation mechanisms by the operating system kernel usually even requires three or more crossings of the memory bus.

As can be seen in Figure 3, data movement causes roughly 50 % of the total CPU cost. In addition, the resulting memory bus traffic can cause significant *bus contention* (assuming three bus crossings, 10 Gb/s full-duplex communication leads to $\approx$ 7.5 GB/s of bus traffic!).

B. RDMA Benefits

The high resource consumption of traditional TCP processing has inspired the design of RDMA, which uses dedicated hardware assistance to speed up network processing. RDMA is based on three main concepts:

Zero-Copy and Data Placement. Memory bus bandwidth and CPU resource consumption can be reduced by reducing the number of intermediate data copies. The concept of *zero copy* has become pervasive in modern networking: rather than bothering the CPU with memory transfers, the network card features its own *DMA controller* that can directly transmit/receive data from/into the main memory of the host.

Although this concept works well for kernel-space functionality, *zero copy* alone cannot fully avoid intra-host copies. For proper process isolation, application data must still be moved from/to user-space memory buffers by the kernel before/after data transmission/reception.

RDMA avoids this necessity with its *direct data placement* functionality. By *tagging* all transfer units with placement information, RDMA can directly operate on user-space memory and, *e.g.*, identify the right target memory buffer autonomously upon package reception. This enables true *zero copy* processing, where no data has to be touched by either host CPU.

[1]In classical network transport protocols, the receiving side faces an even larger processing cost than the sender since the data reception is triggered by (asynchronous) interrupts whereas on the sending side it is (synchronously) driven by the kernel.

Observe in Figure 4 that this not only reduces the load on the host CPU. In practice often more significant is the resulting *bus traffic reduction*, which we found to be a potential bottleneck in high-speed distributed database processing in our earlier work [12].

Asynchronous I/O. In contrast to the synchronous Berkeley `socket` API, RDMA uses a fully asynchronous communication model. All operations are described by the application in terms of *work requests*. They are posted to *queues* on the network adapter from where they are picked up and processed by the hardware (*i.e.*, the RNIC). This allows for overlapping communication with data processing such that the network delay can be mostly hidden.

TCP Offloading. Direct data placement and the asynchronous programming model together enable full data transfer offloading. Protocol processing in hardware not only removes the network stack overhead from the CPU, but also reduces the context switch rate and thus results in a lower CPU cache pollution. This results in efficient network bandwidth utilization at negligible interference with the CPU (see also right-most part in Figure 3).

C. Applying RDMA

Not every application can take full advantage of RDMA [11]. Rather, applications have to respect the characteristics of RDMA to take full advantage of the hardware-accelerated zero copy transport.

First, all buffers (for receiving and for sending data) have to be *sized* and *registered* with the network card *before* starting an RDMA-based data exchange. Only this allows the network interface card to access the application memory through its DMA engine without any involvement of the operating system (and thus enables *direct data placement*). The registration process is rather CPU intensive [11] as it involves several address translations and because the memory must be pinned and protected from being swapped out to disk. Whenever speed is a major concern, the cost of registration renders on-demand allocation and registration of memory buffers infeasible.

Second, each data transfer is initiated by posting work requests to the RNIC, a control task that still has to be performed by the CPU. To keep the resulting CPU overhead limited, it is desirable to transfer the data in *large chunks* which requires fewer work requests to be posted. Also the RNIC itself is able to handle large data transfers more efficiently than small ones. Figure 5 shows raw network throughput achievable with RDMA over 10 Gbps Ethernet when using transfer units of different sizes. RDMA is only able to saturate the link once transfer units reach a size that is $\gtrsim 4$ kB. In practice, we found that additional application overhead slightly shifts this figure, such that we can expect maximum network throughput for units of size 1 MB and larger.

D. Data Roundabout *Design on RDMA*

The *Data Roundabout* strives for a decentralized mode of operation. Each node thus only communicates with its two direct neighbors and all data is forwarded in the same direction, say clockwise.

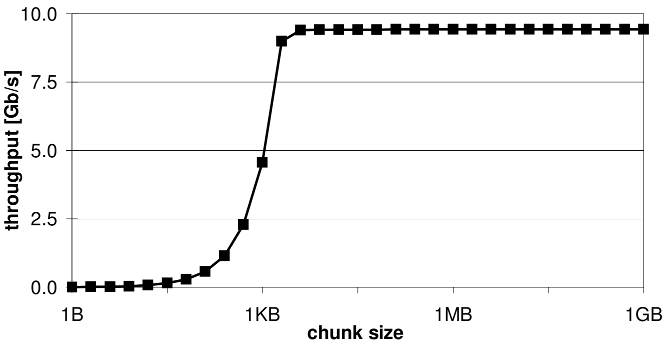
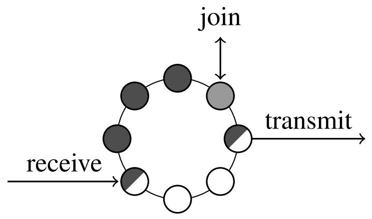


Fig. 5. RDMA requires a minimum chunk size to saturate the link.

Considering the aforementioned requirements, we have designed each node in *Data Roundabout* to be equipped with a statically allocated *ring of memory buffers* to hold the data rotating in the roundabout:



We size and register all of the ring buffer elements in the beginning and reuse them at join execution time to minimize the memory registration cost. As RDMA works best on large buffers, we always transfer a whole ring buffer element and not a single tuple, for instance.

The data propagation within the hosts has been designed in an asynchronous way involving the following three entities: a *join entity*, a *receiver* and a *transmitter*.

The *join entity* is responsible for computing the joins and operates on one ring buffer element at a time. When it has finished processing the current buffer, it asks for that buffer to be forwarded by the *transmitter* and continues with the next buffer while the *transmitter* is forwarding the processed data. If the next buffer has already been filled by the *receiver*, the *join entity* can start processing it immediately.

Overlapping communication and computation is a key part of the *Data Roundabout* architecture because it hides the data propagation delay of the network. Furthermore, since the CPU and memory bus overhead caused by RDMA communication is low, the *join entity* is not hindered by the concurrent data transfers.

IV. *Cyclo-Join*: Distributed Join Processing on the *Data Roundabout*

To effectively exploit the throughput opportunities offered by the *Data Roundabout* architecture, algorithms on top of the transport layer have to adhere to a rather stringent data flow pattern. *Cyclo-join* is a distributed join strategy that provides this data flow pattern and enables us to compute arbitrary database joins in distributed main memory over input data of arbitrary size.

In this section, we describe and motivate the inner workings of *cyclo-join*.

A. Problem Scenario

Our focus is on the evaluation of a binary join $R \bowtie_p S$, where both input relations (tables) are assumed to be large (too large to fit into the local memory of a single machine, in particular). R and S together fit conveniently into the *distributed* memory, the ring storage, of a *Data Roundabout* network, however.

We do *not* pose restrictions on the join predicate p. Although our experiments focus on equi-joins—thus demonstrate how *cyclo-join* can be combined with efficient in-memory algorithms such as hash or sort-merge joins—*cyclo-join* is not bound to equality predicates. Modern application classes could use this flexibility of *cyclo-join* to accelerate, for instance, band joins [7] or similarity joins (a critical operation in data cleaning and integration).

We assume that, prior to join processing, both input tables are distributed over all network hosts H_i. We do not care how the data is distributed, but we assume that the distribution of at least S is reasonably even. This assumption is readily provided, for instance, in recent database prototypes for cloud infrastructures such as HadoopDB [1].

The join $R \bowtie S$ is computed in a fully distributed fashion and its result is again available as a distributed table. As such, the join output could naturally be used as input to subsequent processing in a larger query plan. The ternary join $(R \bowtie S) \bowtie T$ could, for example, be evaluated by using two runs of *cyclo-join*.

B. Cyclo-Join *Operation*

The idea of *cyclo-join* is illustrated in Figure 6: one of the two relations, say S, is kept *stationary* during processing (partitioned into sub-relations S_i) while the fragments of the other relation, say R, are *rotating* in the *Data Roundabout*.

All ring members (Hosts H_i) join each fragment R_j flowing by against their local piece of S (S_i) *locally* using a commodity in-memory join algorithm. The result of each $R_j \bowtie S_i$ is a part of the overall join result. *Cyclo-join* accumulates all $R_j \bowtie S_i$ at H_i. After one revolution of R, all hosts H_i have seen the full relation R and have thus computed the partial join results $R \bowtie S_i$. Since the S_i are a partitioning of S, the full join result $R \bowtie S$ is now available as a distributed table spread across all H_i (ready, *e.g.*, for further processing, as mentioned above).

The task of the *Data Roundabout* transport layer is to efficiently move the rotating relation around the network. As illustrated earlier in Section III-D, the *transmitter* and *receiver* asynchronously move pieces of R in and out, attempting to keep the *join entity* busy at all times. Depending on the shape of the input data, this may be easier to achieve if the smaller of the two input relations is chosen as the one that is kept rotating.

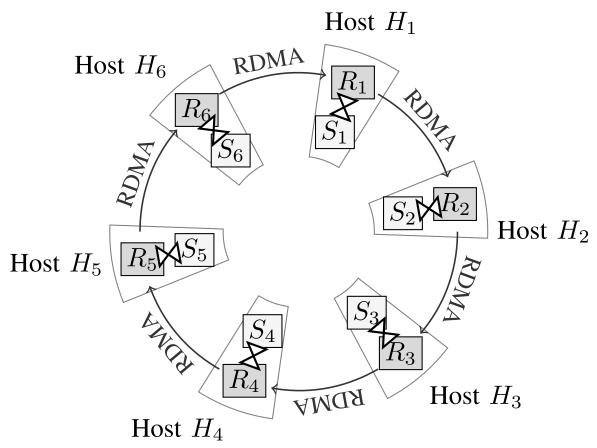


Fig. 6. *Cyclo-join*: Network hosts are organized in a logical ring. Relation R circulates in the ring while S remains stationary.

C. A Selection of Join Algorithms

Within a full revolution of input relation R, all possible combinations of fragments R_j and S_i of R and S, respectively, are co-located on some host once and are then combined to produce $R_j \bowtie S_i$. *Cyclo-join* can thus play together with arbitrary implementations of $\bowtie$ and support arbitrary join predicates. In particular, the join algorithm does not need to be aware of the distributed nature of the setup—we can thus use any algorithm which was designed for local execution.

Since we strive for fully distributed in-memory processing, we focus on join algorithms that are known to perform well in main memory-based setups. We have thus ported the MonetDB[2] implementations of the *partitioned hash join* and *sort-merge join* to our *cyclo-join*. The former inherently provides support only for equi-joins, while our implementation of the latter can also handle band joins. For all other join predicates, our system falls back to the universal but slower *nested loops join* (not further discussed in this report).

1) Hash-Based Equi-Join: Our implementation of the *partitioned hash join* is derived from the *radix join* algorithm [22] as found in the most recent distribution of the MonetDB system. The implementation is carefully tuned to exploit the cache characteristics of modern CPU hardware, including the size of the on-chip L2 cache and the size of an L2 cache line.

Radix join operates in two phases. During a *setup phase* we partition all input data and create hash tables on the partitions of the stationary in-memory join argument S_i. A subsequent *join phase* then scans partitions of R_j and probes into hash tables of the S_i partitions.

We obtain a partitioning of the two input fragments R_j and S_i into sub-fragments $r_{j,k}$ and $s_{i,k}$ using the same hash function on their join keys. The goal is to achieve a partitioning where each piece $s_{i,k}$ of the stationary fragment S_i and an associated hash table fit into the L2 CPU cache. Such a partitioning makes the subsequent join phase (that uses a standard hash join to scan $r_{j,k}$ and probe into a hash table

[2]http://monetdb.cwi.nl

on $s_{i,k}$) particularly cache-efficient, since all hash probes can be handled fully by the L2 cache. For details refer to [22].

The join phase of our *partitioned hash join* can straightforwardly exploit the parallelism provided by modern multi-core systems by computing the disjoint $r_{j,k} \bowtie s_{i,k}$ and $r_{j,l} \bowtie s_{i,l}$ on separate CPU cores. Our implementation uses all four cores on our quad-core systems to run the join phase in parallel.

2) Sort-Merge Join: Sort-merge join operates in two phases as well. The *setup phase* here involves sorting both input fragments by their join keys. During the *join phase*, the sorted fragments are scanned in parallel and "merged" by aligning matches or skipping forward on misses. Though sorting incurs an additional cost over the simpler partitioning in *partitioned hash join*, the join phase of *sort-merge join* favors an even more cache-efficient (strictly sequential) access pattern and can be implemented to readily support band joins or inequality predicates.

Much like in MonetDB, our implementation relies on an efficient implementation of `qsort` in the C library, and we leverage available parallelism by sorting both input fragments (R_i and S_i) in parallel. We note that our implementation bears some potential for improvement here, such as the use of a SIMD-optimized sorting algorithm [6]. The join phase also runs multi-threaded: We split the R_j into a number of non-overlapping sub-partitions ($r_{j,k}$) equal to the number of cores in the system. Individual threads then join the stationary S_i with one piece of R_j.

D. *Interacting with* Cyclo-Join

Our descriptions of *partitioned hash join* and *sort-merge join* assumed that only a single join $R_j \bowtie S_i$ had to be evaluated. Such an evaluation involves the execution of both join phases, hashing/sorting and joining.

In practice, our join implementations sees the same input data over and over again. It thus makes sense to invoke the setup phase of either join implementation only once, then re-use its output during the full execution of *cyclo-join*. The effort spent in the setup phase is then amortized over multiple executions of the join phase. We can do so by sending access structures (such as hash tables) or re-organized data (sorted or partitioned) over the *Data Roundabout* transport layer.

This is an instance where we can exploit the bandwidth provided by our RDMA transport mechanism. Rather than investing CPU cycles to reduce network traffic—the common strategy in existing systems—we spend some network capacity to save CPU work. Sometimes, *cyclo-join* may thus suggest a different balance between the efforts spent on pre-processing and join computation. We will assess and discuss such trade-offs in more detail based on experimental evidence in Section V-E.

V. Experimental Evaluation

This section shows *cyclo-join* in action and provides an analysis of its characteristic features.

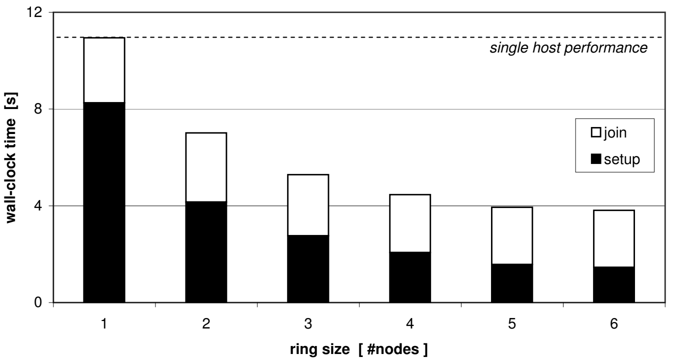


Fig. 7. Joining a fixed data set on an increasing number of nodes.

A. *Test Environment*

Our experiments use *Data Roundabout* instances of up to six network hosts (IBM HS21 BladeServers), which is the maximum number of RDMA-equipped machines we currently have available. Each of them has a quad core Intel Xeon CPU running at 2.33 GHz, 32 KB L1 data cache and 32 KB L1 instruction cache, 4 MB unified L2 cache and 6 GB of main memory. The BladeServers are running Fedora Core 9 with a vanilla 2.6.27 Linux kernel.

RDMA hardware support is provided by Chelsio T3 RNICs (S320EM-BCH) which offer full TCP/IP offloading (TOE) and iWARP RDMA support. The RNICs are interconnected through a Nortel 10 Gb Ethernet Switch Module.

B. *Local Join versus* Cyclo-Join

First, we investigate how well *cyclo-join* is able to leverage the available resources. We generated input relations R and S that are just about large enough to fit into the main memory of a single machine (140 million rows per table with 12 bytes per tuple; this resulted in a total data volume of 2×1.6 GB). The 4-byte join key was populated with uniformly distributed integer numbers.

Figure 7 shows the execution times we observed when computing the join $R \bowtie S$ on a single host and when the evaluation was distributed over up to six network hosts using *cyclo-join*. In all cases we used the partitioned hash join to perform local joins. In the distributed case, we spread all data evenly across all network hosts before join processing.

The most apparent observation is that the distribution of the join considerably reduced the total join processing time, which we separated into the time spent in the *setup phase* (shown in dark gray) and in the *join phase* (white bars) of the partitioned hash join. No execution time was lost otherwise; in particular we could not see any delays due to network processing.

Data Roundabout Overhead. A design goal of *Data Roundabout* was to leverage RDMA such that network communication can be fully overlapped with data processing. Our measurements confirmed that, indeed, *Data Roundabout* was able to fully hide network cost and perform all communication asynchronously to the actual join processing.[3]

Network processing will only cause an effect on the observable execution speed if the in-memory join thread can finish its task significantly faster than RDMA can bring in new data. This effect can be observed in our implementation of sort-merge join, which we are going to look at in Section V-E.

Setup Cost. The separation into the two processing phases shows where the runtime improvement comes from. Distributing the generation of a hash table over the stationary relation S cut down the time spent in the setup phase according to the number of participating nodes. Distribution over six *Data Roundabout* hosts, for instance, reduced the setup cost by a factor of six (16.2 s for single-host execution vs. 2.7 s on six hosts).

Join Cost. The total amount of time spent in the *join phase*, by contrast, is *not* affected by the distribution through *cyclo-join*, a behavior that might seem surprising at first. The reason why *cyclo-join* does not accelerate the join phase in this configuration is the particular characteristics of a hash join. During the join phase, the local hash joins scan their current piece of the outer join relation (*i.e.*, R_j) and perform a hash lookup for each tuple in R_j. Assuming a reasonably "friendly" configuration (a proper hash function and rare hash collisions), the cost of a hash lookup is *independent* of the size of the (local part of the) *inner* join relation S_i.

During a full run of *cyclo-join*, each participating host will scan all pieces R_j of R—hence, the entire relation R—exactly once. The total cost of the join phase is thus *independent* of the number of network hosts:

$$\text{cost}\,(S_i \bowtie R) \cong |R| \cdot \text{cost per hash lookup} \ . \qquad (\star)$$

Highly skewed data invalidates the assumption of rare hash collisions. In Section V-D, we illustrate how this affects the performance of the join phase in the *cyclo-join* setting.

C. Large In-Memory Join

The primary purpose of *cyclo-join* is to distribute the processing of large join instances that could not be evaluated on a single host. We verified this capability by scaling up the problem size, while simultaneously distributing the problem over more network hosts (we keep the per-host data volume constant). Figure 8 illustrates the resulting processing times for data volumes up to 19.2 GB.

Distribution of the hash generation phase now leads to a size-independent setup cost. This is because we distributed join processing such that the per-host data volume remains constant. The time spent in the join phase now scales linearly with the size of the input data (or, more precisely, the size of the rotating relation R). This confirms our assessment of the join phase cost for the hash join, as given by Equation ($\star$).

[3]In an earlier report on *cyclo-join* [12] we had used a different hash join implementation with a higher memory bandwidth demand. This had led to a situation where our system was contended on the local memory bus even when using RDMA, such that *join threads* frequently had to *wait* for the arrival of new data.

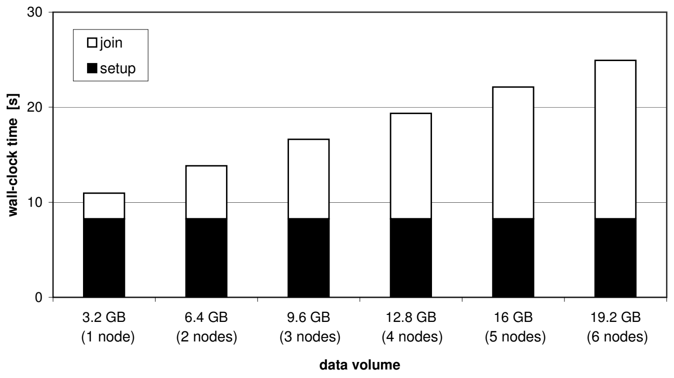


Fig. 8. Each node adds 3.2 GB to the data set.

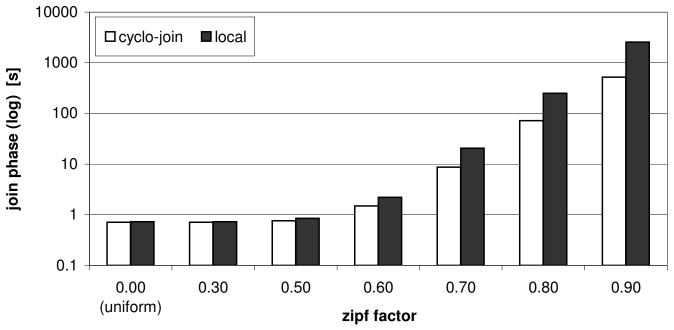


Fig. 9. Join phase on skewed data.

The important outcome of the experiment is that with *cyclo-join* we were able to process large problem sizes purely in distributed memory. A single machine with a large enough memory might have achieved comparable throughput in its join phase (though with higher setup costs). However, while the amount of memory addressable by a single host is severely limited (even the modern Intel i7/Nehalem CPUs are limited to 64 GB of physical memory [15, Table 2-2]), *cyclo-join* can be trivially scaled up to much larger configurations. *Cyclo-join* makes *distributed memory* available to process joins of arbitrary size.

D. Skewed Input

The previous experiments were all based on hash-friendly uniform key distributions. Real-world use cases rarely follow perfect uniformity, but exhibit various flavors of skew. We explore the effect of skewed input data on the *cyclo-join* mechanism by generating input tables according to a Zipf distribution with varying Zipf factors z.

For various z values we generated input data of size $|R| = |S| = 412$ MB (36 million 12-byte tuples). For each generated instance we ran the join $R \bowtie S$ once on a single host and once on a *cyclo-join* ring that consists of six hosts. Figure 9 reports the execution times that we measured for the *join phase* of our partitioned hash join. We omitted the setup phase in this graph since it is unaffected by the data skew.

For Zipf factors of $z = 0.6$ and greater, the exponential increase of the number of *duplicates* in the data sets begins

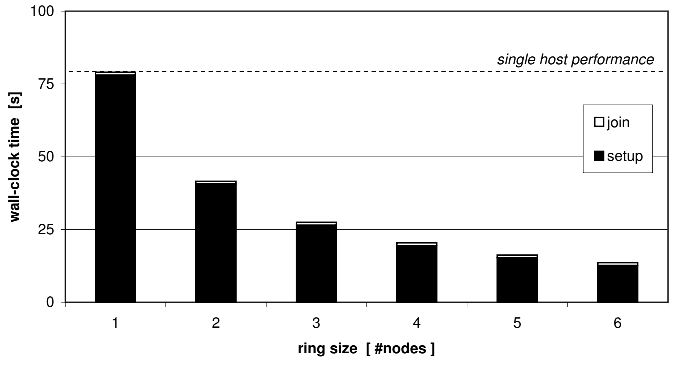


Fig. 10. Sort-Merge Join: Joining a fixed data set on an increasing number of nodes.

to have a noticeable effect on the execution time of our in-memory hash join. This is not a surprise: the increasing number of *hash collisions* lets hash join slowly degrade toward a nested loops-style evaluation.

The distributed join (white bars) can handle the increasing skew appreciably better. While, in line with our previous experiments, the processing of uniformly distributed data cannot benefit from a *cyclo-join*-based execution, Figure 9 shows a five-fold advantage of *cyclo-join* for input data with a skew of $z = 0.9$.

The benefit comes from two sources. First, the ring buffer mechanism of *Data Roundabout* balances differences in the execution speeds of the participating hosts. Thus, a host that is stuck in a chunk of data with a high number of duplicates will not immediately slow down the remainder of the ring. A follower in the *Data Roundabout* will only have to start waiting once it has fully consumed all data in its ring buffer.

Secondly, distribution will lead to a better use of CPU caches. *Cyclo-join* will chop all input data (in particular the inner join relation S) into pieces. Thus, even in the presence of skew, individual partitions within our hash join are less likely to exceed the size of our CPU caches and the join phase can perform more work from within caches.

E. Sort-Merge Join and Setup Cost vs. Join Cost

The runtime characteristics of *sort-merge join* resemble the behavior of the partitioned hash join shown above. Sorting, however, incurs a significantly higher cost than the generation of hash tables, which can cause considerable *setup costs*. As can be seen in Figure 10, this leads to significantly longer execution times for small *Data Roundabout* configurations.

The high setup cost slightly pays off during the join phase (shown again as white bars; we will discuss the light-gray "sync" part in a moment). Merging two sorted tables yields a cache-friendly, strictly sequential data access pattern. In the case of our largest join configuration (19.2 GB distributed over 6 hosts), this cut down the time spent in the join phase from 16.2 s to 6.4 s seconds, a more than two-fold advantage.[4]

[4]Even with the new "sync" time considered (2.3 s), the advantage is still a factor of 1.8.

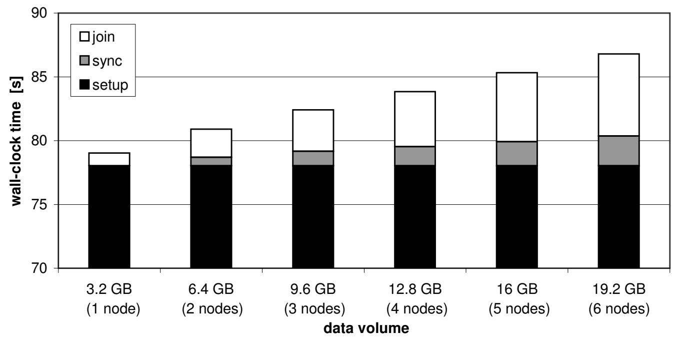


Fig. 11. Sort-Merge Join: Each node adds 3.2 GB to the data set.

In *cyclo-join*, the setup cost is a one-time investment which—in contrast to a single-host execution—the in-memory join steps can benefit from several times. How often a join execution takes advantage of the up-front investment depends on the size of the *Data Roundabout* ring. The larger the rings on which *cyclo-join* operates, the better the high setup costs can be amortized.

Thus, the use of *cyclo-join* may suggest a different balance between the effort spent into a join's setup phase and its resulting performance in the join phase. For the two particular implementations that we have at hand, MonetDB's partitioned hash join and a `qsort`-based sort-merge join, we expect that the latter implementation would overpass the former in *Data Roundabout* configurations of ≈ 30 nodes upward (*i.e.*, for data volumes $\gtrsim 100$ GB).

Kim *et al.* [17] recently studied the trade-off between hash and sort-merge joins with highly tuned implementations of both algorithms and concluded almost comparable performance already on single hosts. Sort-merge join would then likely be the better choice already for *cyclo-join* configurations running on only few nodes.

F. "Synchronization" Cost

While sort-merge join incurs high setup cost, its execution speed in the join phase is faster than that of partitioned hash join. In Figure 11, we see that with sort-merge join, the join phase has become too fast to fully hide the cost of network communication. The time shown in light gray is the time that the *join threads* now spent waiting for new data to arrive via the *Data Roundabout* transport layer (we say they *synchronize* with the *Data Roundabout* layer).

The performance that we observe indicates that we are hitting the limits of the physical 10 Gb/s transport layer. For a full *cyclo-join* run, the entire relation R has to be pumped once through each participating host. For the 6-host configuration in Figure 11, this means that $|R| = 9.6$ GB of data crossed each *Data Roundabout* link in $6.4\,\text{s} + 2.3\,\text{s} = 8.7\,\text{s}$, which corresponds to a network throughput of 1.1 GB/s, very close to the theoretical maximum of 10 Gb/s, *i.e.*, 1.25 GB/s.

G. RDMA vs. Software-Based TCP

In Section III we argued the necessity of RDMA based on CPU and memory bandwidth resource consumption. In this

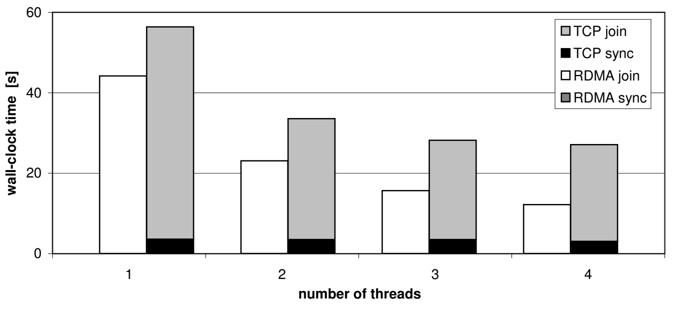


Fig. 12. Hash join on RDMA versus TCP with varying number of join threads.

section, we compare RDMA-based processing to a TCP-based implementation running as part of the operating system kernel.

To this end, we generated data instances of sizes $|R| = |S| = 160$ million tuples (corresponding to a data volume of $2 \times 6.7\,\text{GB}$) and distributed the evaluation of $R \bowtie S$ over a *Data Roundabout* installation of size six (as before). We ran the join with RDMA support, but also with the standard mechanisms provided by the Linux kernel. That is, we changed the transmitter and receiver of *Data Roundabout* to use `send` and `recv` calls instead of their RDMA counterparts.

Since this obviously causes additional load on the available CPU cores, we configured *cyclo-join* to allocate a varying number of cores for join processing, keeping the remaining cores available for TCP handling. When we use only two join threads, for instance, two CPU cores should always remain fully available for the two communication threads (transmit and receive).

Figure 12 shows the execution times we observed for the *join phase* of our partitioned hash join. Since the setup phase is independent of the transport mechanism, we omitted it in the comparison.

The RDMA-based *cyclo-join* outperforms the TCP-based one in all configurations. RDMA is even better in the case where only 1 core is computing the join and three cores should be available for the data propagation. This is due to the fact that RDMA not only saves CPU cycles by avoiding the immediate buffer copies, but also reduces the context switch rate, since the communication with the network is based on queues. This results in less disturbance of data processing operations and therefore in a lower cache pollution.

The largest performance difference between RDMA and TCP results when using all four cores for the join processing. Join and communication entities now all compete for the available CPU cycles, pollute each others caches, and cause a large number of context switches. The benefits of the cache-efficient join algorithm are mostly annihilated.

Adding more CPUs is not an alternative to RDMA. In the case where all cores are processing the join, total CPU utilization reaches only about 86 % (Table I) which indicates that adding further CPUs would not yield an improvement. RDMA, on the other hand, incurs a CPU load which matches the number of cores that are computing the join and is able to fully utilize the available compute resources. The join processing is never interrupted by the network.

	cpu load TCP	cpu load RDMA
1 thread	31 %	25 %
2 threads	59 %	50 %
3 threads	84 %	76 %
4 threads	86 %	100 %

TABLE I
CPU LOAD DURING THE JOIN PHASE OF THE HASH JOIN. 100 % REFERS TO ALL FOUR CORES BEING COMPLETELY BUSY.

We further observe that even though the transport is multi-threaded, the TCP approach (in contrast to RDMA) is not able to fully hide the synchronization time.

VI. A Look into the Research Neighborhood

We kept the design of *cyclo-join* and its transport layer *Data Roundabout* deliberately simple. As such we think many of the ideas presented in this paper would blend well with existing research work and with some of the recent developments in hardware technology.

The availability of a fast transport mechanism eliminates much of the urgency to reduce network transfer volumes as it was the primary goal of earlier work [4], [21], [24]. Yet, network traffic might still become a concern, for instance in scenarios with highly concurrent or memory-intensive workloads, and much of the existing work could become relevant to address such scenarios.

Our spinning join setup resembles the *DataCycle* system [5] or the *Broadcast Disks* of [2], systems that put significant effort into properly scheduling data on the transport stream. Integrating the ideas of this work into our system is part of our ongoing work [13] and inspired a number of design decisions in our evolving system prototype.

More recent work in the research neighborhood are new systems designed for cloud environments. While systems built on MapReduce-style architectures (such as the recently proposed HadoopDB [1]) can achieve excellent scale-out for certain types of queries, they still lack a convincing means to perform arbitrary joins *across* the pre-assigned data partitions. *Cyclo-join* could fill this gap and enable the vision of distributed true-SQL system.

On the technology side, *cyclo-join* could be a very interesting application for Intel's emerging *I/O Acceleration Technology (I/OAT)* [14]. With help of the *Direct Cache Access (DCA)* feature of I/OAT, capable network controllers can place data directly into CPU caches. As we showed in Section III-C, *Data Roundabout* works well already with RDMA transfer units under a megabyte, small enough to be loaded straight into caches. This might not only help to cut down transport latencies, but also yield an even further reduction of main memory bus contention.

Finally, we would like to relate our work to the *systolic systems* developed in the 1980s. Systolic systems are composed of a network of processors with a simple rhythmical (hence the term "systolic") data flow in-between. Although Kung and Leiserson [18] had small-scale, on-chip processing units in mind when they presented the first "systolic algorithms," some of the observations made at the time may still be applicable to a *cyclo-join* ring.

VII. Summary

Modern advances in networking technology may shift the priorities in distributed data processing. We demonstrated how the bandwidth offered by modern networks (10 Gb/s and beyond) can be exploited with help of *Remote Direct Memory Access (RDMA)*, a network transfer protocol with widely available hardware support. To this end we developed *cyclo-join*, a mechanism that can distribute the evaluation of relational database joins. *Cyclo-join* is built on top of the ring-shaped *Data Roundabout* transport layer, which has promising characteristics also in other settings [13], [16].

With *cyclo-join*, large database joins can be processed as *in-memory joins* by taking advantage of the distributed main memory in a cluster system. The system becomes CPU-limited instead of bound by disk or network I/O. Other than in a centralized system, the capacity of a *Data Roundabout* storage ring can be scaled up trivially, making it possible to process input data of arbitrary size. In line with the idea of cloud computing, such scaling may even be performed at runtime and as application workloads demand.

The effect of distributing CPU load depends on the particular join problem and on the algorithm chosen to perform intra-host joins. We showed that critical and CPU-intensive sub-tasks, such as hash generation or joins over skewed data, can benefit best from the *cyclo-join* mechanism.

Our current research effort goes into the integration of *cyclo-join* into the prototype *Data Cyclotron* system. This involves the establishment of a complete SQL-enabled system and a complete *cost model* for *cyclo-join*.

Acknowledgment

Jens Teubner is supported by a Swiss National Science Foundation *Ambizione* grant (no. PZ00P2_126405).

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 922–933, 2009.

[2] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," in *Proc. of the ACM SIGMOD Conference on Management of Data*, 1995, pp. 199–210.

[3] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 4, pp. 602–625, 1981.

[4] P. Bernstein and D. ming Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, vol. 28, no. 1, pp. 25–40, 1981.

[5] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib, "The Datacycle Architecture," *Communications of the ACM*, vol. 35, no. 12, pp. 71–81, 1992.

[6] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[7] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "An Evaluation of Non-Equijoin Algorithms," in *Proc. of the 17th Int'l Conference on Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991, pp. 443–452.

[8] N. Dieu, A. Dragusanu, F. Fabret, F. Llirbat, and E. Simon, "1000 Tables Inside the From," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2, pp. 1450–1461, 2009.

[9] R. Epstein, M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Data Base System," in *Proc. of the ACM SIGMOD Conference on Management of Data*, 1978, pp. 169–180.

[10] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier, "TCP Performance Re-Visited, Analysis of Systems and Software," in *Proceedings of the IEEE ISPASS*, 2003, pp. 70–79.

[11] P. Frey and G. Alonso, "Minimizing the Hidden Cost of RDMA," in *Proc. of the 29th Int'l Conference on Distributed Computing Systems (ICDCS)*, Montreal, QC, Canada, 2009, pp. 553–560.

[12] P. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning Relations: High-Speed Networks for Distributed Join Processing on New Hardware," in *Proc. of the 5th Workshop on Data Processing on New Hardware (DaMoN)*, Providence, RI, USA, 2009, pp. 27–33.

[13] R. Goncalves and M. Kersten, "The Data Cyclotron Query Processing Scheme," (under submission).

[14] *Accelerating High-Speed Networking with Intel I/O Acceleration Technology*, White Paper, Intel Corp., 2006.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Intel Corp., Sep. 2009.

[16] M. Kersten, "The Database Architecture Jigsaw Puzzle," in *Proc. of the IEEE Int'l Conference on Data Engineering (ICDE)*, 2008, pp. 3–4.

[17] C. Kim, E. Sedlar, and J. Chhugani, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.

[18] H. Kung and C. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse Matrix Proceedings*, 1978, pp. 256–282.

[19] W. Litwin, M. Neimat, and D. Schneider, "LH* - A Scalable, Distributed Data Structure," *ACM Transaction on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[20] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, *Query Processing in R**. Springer, 1985.

[21] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," in *Proc. of the 12th Int'l Conference on Very Large Data Bases (VLDB)*, 1986, pp. 149–159.

[22] S. Manegold, P. Boncz, and M. Kersten, "Optimizing Main-Memory Join on Modern Hardware," *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, vol. 14, no. 4, pp. 709–730, 2002.

[23] J. Rothnie, P. Bernstein, S. Fox, N. Goodman, M. Hammer, T. Landers, C. Reeve, D. Shipman, and E. Wong, "Introduction to a System for Distributed Databases (SDD-1)," *ACM Transaction on Database Systems (TODS)*, vol. 5, no. 1, pp. 1–17, 1980.

[24] P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 1, pp. 133–161, 1984.