


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

Almost unnoticed by the database community, advances in network technology have turned a fundamental assumption in distributed data processing—network communication is slow and expensive—upside-down. With the help of *Remote Direct Memory Access (RDMA)*, modern networks can transfer data at rates beyond 10 Gb/s, yet cause only negligible overhead in terms of CPU power.

We study the opportunities offered by these hardware trends from the perspective of the core database algorithms using a simple topological network structure. As starting point we propose the *Data Roundabout*, a ring-shaped network consisting of several machines. Each of them stores a portion of a complete data set. Rather than trying to avoid network communication at all cost, we leverage the available bandwidth and (continuously) pump data through a high-speed network.

In this work we report on *cyclo-join* which exploits the cycling flow of data in the *Data Roundabout* 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 this data cycling arena. The experiments show the potential of a large distributed main-memory cache glued together with *RDMA* into a novel DBMS architecture.

1. INTRODUCTION

With great distributed compute power at everyone’s fingertips—either in terms of real hardware or provided by a cloud infrastructure—user expectations have grown high: users expect even complex ad-hoc queries to be answered in interactive time, automatically distributed to exploit available resources as necessary.

In this work, we look at a particular part of the challenge to meet these expectations. We present *cyclo-join*, a mechanism that uses networked resources to perform *distributed joins*. In particular, we use *Remote Direct Memory Access (RDMA)* with its high-speed characteristics to pro-

cess joins entirely in distributed main memory, in order to reach throughput rates that are far beyond what commodity disks or conventional networks can provide.

The purpose of this report is two-fold:

Cyclo-Join Mechanism. With *cyclo-join*, we describe a slim layer that 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 actual hardware, we analyze and illustrate the implications of *cyclo-join* depending on the problem type and compare it with local join algorithms.

High-Speed Networks and RDMA. Available since long in InfiniBand, and now also for Ethernet systems, RDMA promises significant network performance advantages (≥ 10 Gb/s) and bears interesting potential for distributed data processing. Unleashing this potential, however, requires careful algorithmic (re-)design and a basic understanding of how RDMA operates internally. Our work provides such understanding and discusses trade-offs incurred with RDMA.

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 [15].

This report proceeds as follows. The upcoming Section 2 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 3. The architecture and algorithms used for distributed join processing on the *cyclo-join* are presented in Section 4. The evaluation of *cyclo-join* for uniform and non-uniform workloads is presented in Section 5 and the additional related work is presented in Section 6. Section 7 summarizes our work.

2. 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 [22], Distributed INGRES [8], or System R* [19] systems. The underlying assumptions and the approaches taken to address them are largely a consequence of the network environments at that time. A short recap of history is provided.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

2.1 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 [20], or System R’s “fetch matches” technique [20], was to avoid network communication, often at the expense of additional CPU work.

Another consequence is the generic 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. A notable exception are the scalable distributed data structures [18], which adapt to the arrival of data. 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.

2.2 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 towards complex queries [7] and reflect an increasing importance of ad-hoc queries. Particularly the former class of queries often depends on functionality beyond the classical foreign-key joins, including band or similarity joins.

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.

2.3 The Data Roundabout Approach

As a starting point to explore novel architectures for their potential to meet these 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. Each node communicates only with its immediate neighbors via a 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 slow, distributed disk space.

A fundamental difference to classical distributed systems is that we keep queries and their state static and move base data over the network instead. In fact, we keep (the hot set of the) data *circulating* in the ring *continuously*. Queries

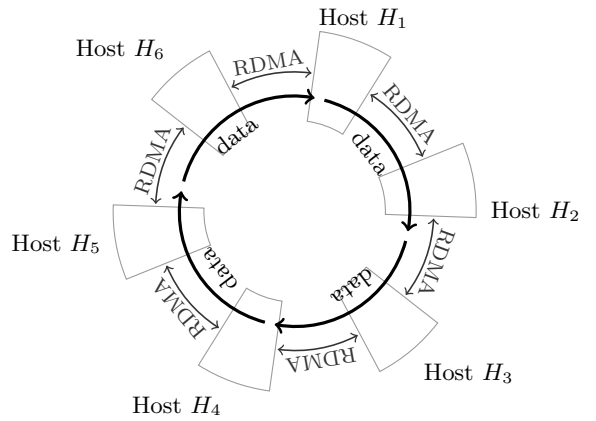


Figure 1: The *Data Roundabout*. Hosts H_i are organized as ring, connected by high-speed RDMA links.

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 shrunk, depending on CPU and/or main memory demand. Furthermore, we do not partition data based on a priori workload knowledge, which lets us seamlessly handle ad-hoc queries.

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

3. REMOTE DIRECT MEMORY ACCESS

Network communication is known to be compute- and memory I/O-intensive [9]. High-volume data transfers thus depend on dedicated hardware assistance, which modern network cards provide in terms of *Remote Direct Memory Access (RDMA)*.¹

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 and provides a motivation for the design of *cyclo-join*.

3.1 RDMA Basics

The basic idea of RDMA is illustrated in Figure 2. (i) The network card of the sending host fetches the data directly out of local main-memory using intra-host DMA. (ii) It then transmits the data across the network to the remote host where (iii) a receiving RNIC places the data straight in its destination memory location. On both hosts, the CPUs only need to perform control functionality, if at all.

RDMA offers the following benefits relevant for this work: *Asynchronous I/O*. In contrast to the synchronous *socket* interface offered by TCP, RDMA uses an asynchronous communication model between the application and the hard-

¹Network interface cards (NICs) that provide this functionality are also called *RDMA-enabled NICs* or *RNICs*.

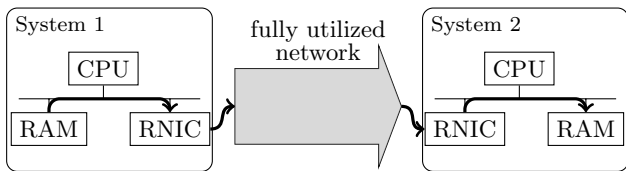


Figure 2: Network transfer using RDMA. RNICs handle data transfer autonomously; data has to cross each memory bus only once.

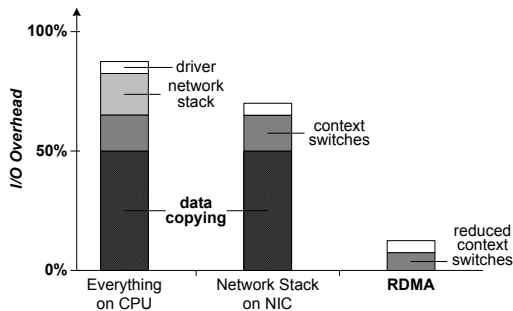


Figure 3: Only RDMA is able to significantly reduce the local communication overhead induced at high-speed data transfers.

ware. The available operations are described by the application in terms of *work requests*. Work requests are posted to *queues* on the network adapter where they are processed in hardware. This enables the overlap of communication with data processing such that the network delay can be mostly hidden.

Zero-Copy and Direct Data Placement. Traditional network interfaces (e.g., sockets), require intermediate copy steps by the operating system when moving data between the network and the application memory of a host (in order to guarantee proper isolation between different applications). This consumes a substantial amount of CPU power. A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/s network throughput [9].

As shown in Figure 3 (left-most chart), data copying is the dominant share of the CPU load required to process network data. Thus, simply offloading the network stack processing to the network card does not change the picture much (middle chart in Figure 3).

With RDMA on the other hand, the data is moved directly between the network and the application memory without intermediate copying—this is known as *zero-copy*. The key concept that enables zero-copy is RDMA’s *direct data placement*, a mechanism whereby data is enriched with local placement information such that the RNICs are able to directly access the data in local main-memory using their DMA engines.

As the data transfer is handled entirely by the network cards, RDMA offers high-speed communication between two hosts with minimal involvement of either CPU. The whole network stack processing is also performed by the network cards, resulting in a further CPU load reduction and also in fewer context switches, thereby causing less cache pollution (right-most chart in Figure 3).

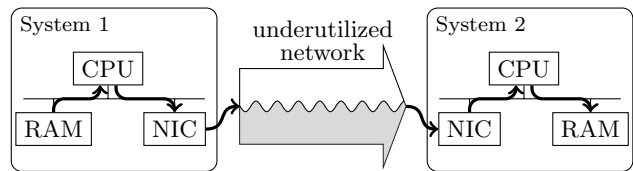


Figure 4: Kernel TCP/IP. Data copying goes through CPU; several memory bus crossings.

A second effect is less obvious: RDMA also significantly reduces the memory bus load as the data is directly transferred to/from its location in main-memory using intra-host DMA. Therefore, the data crosses the memory bus only once per transfer. The kernel TCP/IP stack on the other hand requires several such crossings (see Figure 4). This may lead to noticeable *contention* on the memory bus under high network I/O. Adding additional CPU cores to the system is thus *not* a replacement for RDMA (we further substantiate this claim in Appendix B).

For the motivation of this work it is important to realize that RDMA allows us to transfer large amounts of data at a high speed without causing any significant overhead. We thus have almost all of the CPU cycles available for the join processing.

3.2 Applying RDMA

As we showed in earlier work [10], not every application can take full advantage of RDMA. Rather, applications have to respect the characteristics of RDMA to take full advantage of the hardware-accelerated 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. This enables the network interface card to access the application memory through its DMA engine without any involvement of the operating system. The registration process is rather CPU intensive [10] 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 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.

3.3 Data Roundabout Design on RDMA

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

Considering the aforementioned requirements, we designed

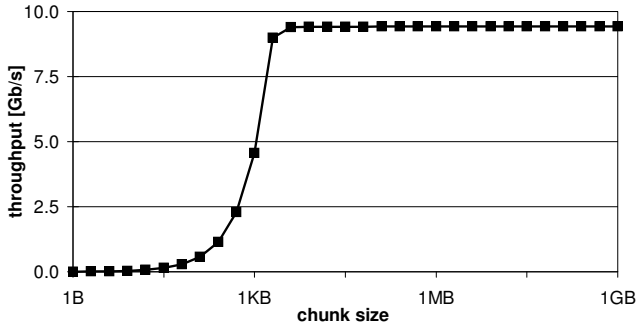
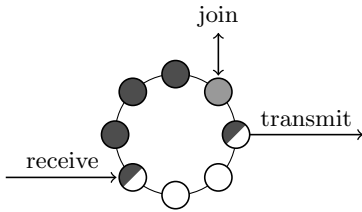


Figure 5: RDMA requires a minimum chunk size to saturate the link.

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 thread*, a *receive thread* and a *transmit thread*.

The *join thread* is responsible for computing the joins and operates on one ring buffer element at a time. When it is finished processing the current buffer, it asks for that buffer to be forwarded by the *transmit thread* and continues with the next buffer while the *transmit thread* is forwarding the processed data. If the next buffer has already been filled by the *receive thread*, the *join thread* 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 thread* is not hindered by the concurrent data transfers.

4. 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*.

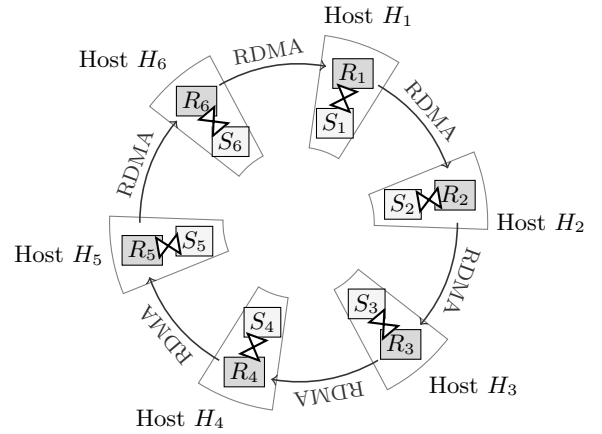


Figure 6: *Cyclo-join*: Network hosts are organized in a logical ring. Relation R circulates in the ring while S remains stationary.

4.1 Problem Scenario

Our focus is on the evaluation of a binary join $R \bowtie_p S$, where both input relations 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 or similarity joins.

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 using two runs of *cyclo-join*.

4.2 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 3.3, *transmit thread* and *receive thread* asynchronously move pieces of R in and out, attempting to keep the *join thread* 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.

4.3 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 then combined to produce $R_j \bowtie S_i$ (as such, *cyclo-join* operates similar to a block nested loops join [11]). *Cyclo-join* can thus play together with arbitrary implementations of \bowtie and support arbitrary join predicates.

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 ported the MonetDB implementations of *partitioned hash join* and *sort-merge join* to our *cyclo-join* setup. 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 an implementation of *nested loops join* (not further discussed in this report).

4.3.1 Hash-Based Equi-Join

Our implementation of *partitioned hash join* is derived from the *radix join* algorithm [21] as found in the most recent distribution of the MonetDB system². 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 on $s_{i,k}$) particularly cache-efficient, since all hash probes can be handled fully by the L2 cache.

As detailed in [21], establishing the necessary partitioning in a single pass would often lead to a high number of conflicts in the TLB and/or data caches. Hence, we employ a multi-pass partitioning phase, where each pass produces a *refinement* of its predecessor in a cache-efficient manner. For details refer to [21].

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.

²Available since release Nov_2009

4.3.2 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 .

4.4 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 reuse 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 A.

5. EXPERIMENTAL EVALUATION

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

5.1 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.

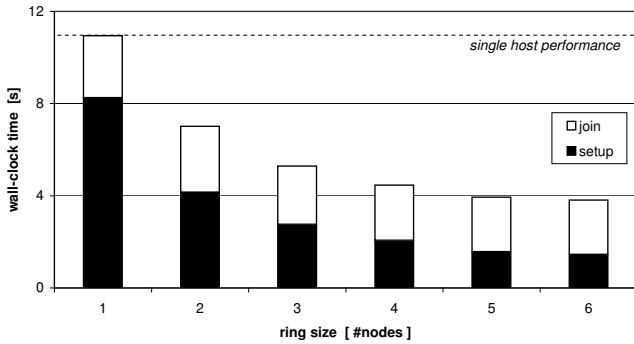


Figure 7: Joining a fixed data set on an increasing number of nodes.

5.2 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. A particular observation is the non-existence of synchronization time, *i.e.*, the *Data Roundabout* overhead imposed by the RDMA.

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. As we show in Appendix A this effect can be observed in our implementation of sort-merge join.

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

³In an earlier report on *cyclo-join* [11] 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 buses even when using RDMA, such that *join threads* frequently had to *wait* for the arrival of new data.

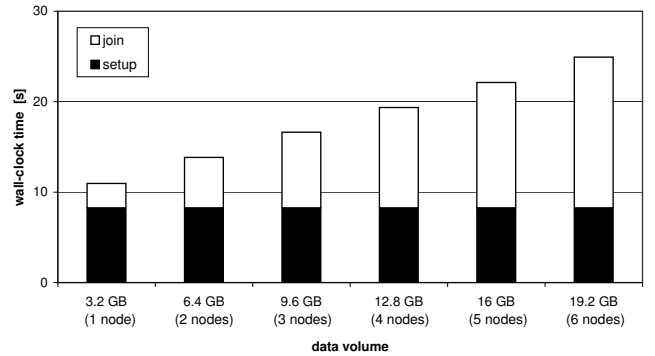


Figure 8: Each node adds 3.2 GB to the data set.

setup cost by a factor of six (16.2s for single-host execution vs. 2.7s 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 benefit the join phase in this configuration are 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:

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

Highly skewed data invalidates the assumption of rare hash collisions. In Section 5.4, we illustrate how this affects the performance of the join phase in the *cyclo-join* setting.

5.3 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).

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

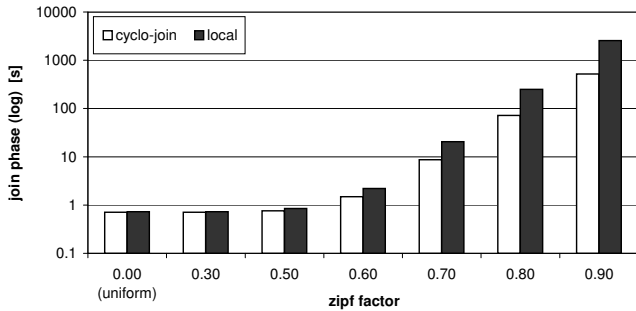


Figure 9: Join phase on skewed data.

limited (even the modern Intel i7/Nehalem CPUs are limited to 64 GB of physical memory [14]), *cyclo-join* can be trivially scaled up to large configurations. *Cyclo-join* makes *distributed memory* available to process joins of arbitrary size.

5.4 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 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.

6. A LOOK INTO THE 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, 20, 23]. 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 [12] 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)* [13]. 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 3.2, *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 [17] 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.

7. 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 [12, 15].

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 Round-*

about 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).

8. REFERENCES

- [1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, pages 922–933, 2009.
- [2] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proceedings of the ACM SIGMOD*, pages 199–210, 1995.
- [3] Philip Bernstein, Nathan Goodman, Eugene Wong, Christopher Reeve, and James Rothnie. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, pages 602–625, 1981.
- [4] Philip Bernstein and Dah ming Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, pages 25–40, 1981.
- [5] Thomas Bowen, Gita Gopal, Gary Herman, Takako Hickey, Kuo Lee, William Mansfield, John Raitz, and Abel Weinrib. The Datacycle Architecture. *Communications of the ACM*, pages 71–81, 1992.
- [6] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, pages 1313–1324, 2008.
- [7] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. 1000 Tables Inside the From. *Proceedings of the VLDB Endowment (PVLDB)*, pages 1450–1461, 2009.
- [8] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In *Proceedings of the ACM SIGMOD*, pages 169–180, 1978.
- [9] Annie Foong, Thomas Huff, Herbert Hum, Jaidev Patwardhan, and Greg Regnier. TCP Performance Re-Visited, Analysis of Systems and Software. In *Proceedings of the IEEE ISPASS*, pages 70–79, 2003.
- [10] Philip Frey and Gustavo Alonso. Minimizing the Hidden Cost of RDMA. In *Proceedings of the ICDCS*, pages 553–560, 2009.
- [11] Philip Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. Spinning Relations: High-Speed Networks for Distributed Join Processing on New Hardware. In *Proceedings of the DaMon*, pages 27–33, 2009.
- [12] Romulo Goncalves and Martin Kersten. The Data Cyclotron Query Processing Scheme. (under submission).
- [13] Intel Corp. *Accelerating High-Speed Networking with Intel I/O Acceleration Technology*, 2006.
- [14] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, 2009.
- [15] Martin Kersten. The Database Architecture Jigsaw Puzzle. In *Proceeding of the IEEE ICDE*, pages 3–4, 2008.
- [16] Changkyu Kim, Eric Sedlar, and Jatin Chhugani. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment*, pages 1378–1389, 2009.
- [17] Hsiang Kung and Charles Leiserson. Systolic Arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282, 1978.
- [18] Witold Litwin, Marie Neimat, and Donovan Schneider. LH* - A Scalable, Distributed Data Structure. *Proceedings of the ACM TODS*, pages 480–525, 1996.
- [19] Guy Lohman, C. Mohan, Laura Haas, Dean Daniels, Bruce Lindsay, Patricia Selinger, and Paul Wilms. Query Processing in R*. *Query Processing in Database Systems*, pages 31–47, 1985.
- [20] Lothar Mackert and Guy Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of the VLDB Endowment*, pages 149–159, 1986.
- [21] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *Proceedings of the IEEE TKDE*, pages 709–730, 2002.
- [22] James Rothnie, Philip Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry Landers, Christopher Reeve, David Shipman, and Eugene Wong. Introduction to a System for Distributed Databases (SDD-1). *Proceedings of the ACM TODS*, pages 1–17, 1980.
- [23] Patrick Valduriez and Georges Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *Proceedings of the ACM TODS*, pages 133–161, 1984.

APPENDIX

A. SORT-MERGE JOIN

In this section, we present the experimental results obtained by applying the *sort-merge join* rather than the partitioned hash join. Figures 10 and 11 correspond to Figures 7 and 8, respectively.

A.1 Setup Cost vs. Join Cost

The runtime characteristics of sort-merge join resemble the behavior of the partitioned hash join as shown earlier. Sorting, however, incurs a significantly higher cost than the generation of hash tables, which is why we see considerably higher 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 16.2s to 6.4s seconds, a more than two-fold advantage.⁴

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. High setup costs will better amortize if *cyclo-join* operates on larger rings.

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.* [16] recently studied the trade-off between hash and sort-merge joins recently 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.

A.2 “Synchronization” Cost

Contrast to our observations in Section 5.2 when running the partitioned hash join, Figure 11 also shows that the join phase has now 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.

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

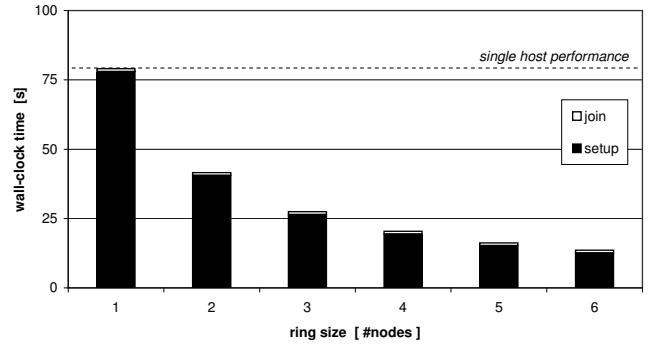


Figure 10: Sort-Merge Join: Joining a fixed data set on an increasing number of nodes.

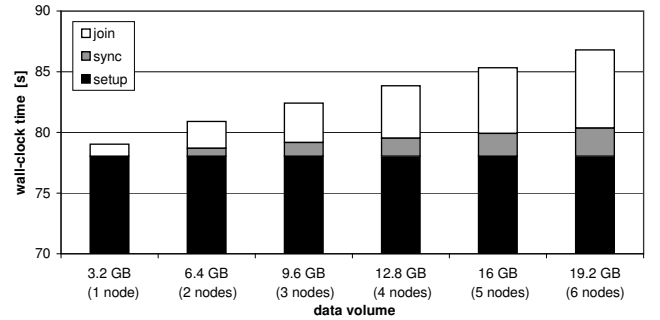


Figure 11: Sort-Merge Join: Each node adds 3.2 GB to the data set.

B. RDMA VERSUS TCP SOCKETS

We presented the *cyclo-join/Data Roundabout* pair running on top of a hardware-accelerated RDMA transport layer. Here we analyze what would happen if we used a traditional, TCP sockets-based transport in our system.

To this end, we generated data instances of sizes $|R| = |S| = 160$ million tuples (corresponding to a data volume of $2 \cdot 6.7$ 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 transmit and receive threads 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, in order to have the remaining cores available for network communication. When using 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

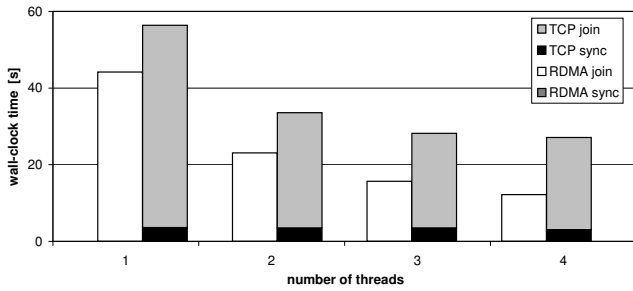


Figure 12: Hash join on RDMA versus TCP with varying number of join threads.

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

Table 1: CPU load during the join phase of the hash join. 100 % refers to all four cores being completely busy.

the immediate buffer copies. It 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 a lower cache pollution.

In the case of RDMA, the two transport threads only post the work requests to the adapter and wait for new data to arrive. Due to the *direct data placement* of RDMA, the no-

tification to the forwarding threads is delayed. The receive thread, the more expensive end of the communication, is only notified once the whole data chunk has been placed in main memory. The transmit thread is never notified of the transmit completion since the *Data Roundabout* performs the flow control at application level.

With TCP sockets, on the other hand, the receive thread must copy the data from the socket buffer into the application memory which costs CPU cycles. Therefore, the receive thread is notified constantly and not only once the whole chunk is locally available. Also, the incoming data might not fit completely into the socket buffer, causing the transport to stall until the receive thread has copied the data from the socket buffer into the application memory area.

The largest performance difference between RDMA and TCP results when using all four cores for the join processing. Join threads and communication threads 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 1) 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.

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.