Cyclo-Join: A Join that Spins without Getting 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

By leveraging modern networking hardware (*RDMA-enabled network cards*), we can shift priorities in distributed database processing significantly. Complex and sophisticated mechanisms to avoid network traffic can be replaced by a scheme that takes advantage of the bandwidth and low latency offered by such interconnects.

We illustrate this phenomenon with *cyclo-join*, an efficient join algorithm based on continuously pumping data through a ring-structured network. Our approach is capable of exploiting the resources of all CPUs and distributed main-memory available in the network for processing queries of arbitrary shape and datasets of arbitrary size.

1. INTRODUCTION

In this paper, we present a mechanism to perform joins of arbitrary size in main-memory rather than on disk. For that purpose, we distribute the data among a set of hosts connected over a high-performance network ($\geq 10\,\text{Gb/s}$). The hosts are organized in a logical ring around which the data can circulate. We leverage *Remote Direct Memory Access (RDMA)* to ensure fast network transfers at low cost.

Consider you are given two many-gigabyte database tables R and S (*e.g.*, coming from a large TPC-H instance) and you have to compute the relational join $R \bowtie S$ between the two (*e.g.*, as part of evaluating TPC-H query Q3). Of course you get plenty of resources for that: the machines in your cluster contain tens of CPU cores in total, with a combined main-memory that can easily hold the entire data set. The challenge is how to orchestrate the machines in order to solve $R \bowtie S$ in an efficient manner, without creating a bottleneck in any node and without hampering the capabilities needed for concurrent or future queries.

In this work, we demonstrate how modern networking facilities in compute clusters can be beneficial for database processing. In summary, we focus on two contributions:

Cyclo-Join. We demonstrate *cyclo-join*, a distributed join processing technique (Sections 2 and 3). We deviate from the main route in distributed join processing by (continuously) pumping data through the network. We assume that the network is your friend, rather than the enemy to be evaded at all cost.

RDMA. We give a brief overview of RDMA, with a focus on its potential use in a database context (Section 4). Though RDMA can promise significant network performance advantages, the technology calls for a strong algorithm engineering attitude from the system designer.

We implemented *cyclo-join* on real RDMA hardware. We provide an in-depth study of its characteristics in Section 5. It demonstrates the potential of RDMA and its particular usefulness in a technique like *cyclo-join.* We look into related work in Section 6 before we summarize in Section 7.

2. PROCESSING LARGE JOINS IN DISTRIBUTED MAIN-MEMORY

Our challenge is to process joins $R \bowtie S$ whose input relations R and S are both too large to fit into the main-memory of a single machine, but small enough to be held in distributed main-memory.

2.1 Small Joins on a Single Machine

As a point of reference of what an ideal distributed solution could achieve, let us first assume that all input data fits into the memory of a single host. Leaving pre-processing costs (hashing/sorting) aside, the time to perform a well-tuned hash or merge join $R \bowtie S$ can then be as small as

$$(|R| + |S|) \cdot \text{in-memory join throughput}$$

($|R|$ and $|S|$ denote the sizes of R and S, respectively). In practice, the in-memory join throughput often gets very close to the physical bandwidth of the underlying main-memory interconnects.

2.2 Large Joins on a Single Machine

Once the input relations become too large to fit, the single-host algorithm has to resort to secondary storage, typically a hard disk, as temporary buffer. Chances are that the best way of doing is the use of a *block nested loops join (BNLJ)*. It brings chunks of each relation into main-memory where in-memory join techniques are applied:

```
1 save $S$ to disk ;
2 foreach block $R_i$ of $R$ do
3     foreach block $S_j$ of $S$ do                (BNLJ)
4         compute $R_i \bowtie S_j$ in memory ;
```

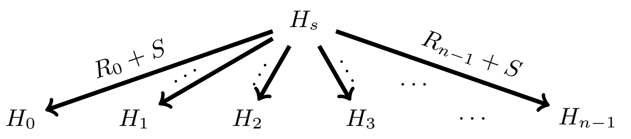

Figure 1: Distributed block nested loops join. The data source H_s has to send the respective R-block R_i to host H_i, plus the full relation S.

The available amount of main-memory determines the partitioning of both input relations into blocks. If we have a buffer space of size M_R to hold the blocks of R, then there will be $n = \lceil |R| / M_R \rceil$ iterations of the outer loop.

Note that R and S could be the outcome of entire subplan evaluations. In that case, a pipelined execution model would allow us to consume all input data only once. Here we assume that we can request pieces of R and S to be brought into memory once (we account the cost for that to the cost of evaluating R and S, as typical in a pipelined setup).

Therefore, we are forced to buffer relation S on disk and have to bear the cost for the disk write. For every iteration of the outer loop (*i.e.*, n times), we read S back to execute the inner loop on the current R_i. This results in an overall I/O cost of

$$(1 + n) \cdot |S| \cdot \text{disk throughput} \ .$$

Since n is proportional to $|R|$, the disk I/O cost to evaluate $R \bowtie S$ is roughly proportional to $|R| \cdot |S|$. Furthermore, the disk I/O throughput is significantly lower than the memory bus bandwidth. Therefore, being forced to use the disk as intermediate buffer is undesirable.

2.3 Large Joins on Multiple Machines

One way of reducing disk I/O is by parallelizing the outer loop of Algorithm BNLJ across multiple machines in conjunction with a trade-in of network I/O. With n hosts available, we need to run only one outer loop iteration on each host and thus leverage the total main-memory available. To this end, we have to provide each of the n hosts with its respective piece R_i of the input relation R and with the *entire* relation S:[1]

```
1 foreach network host $H_j$ do
2     send block $R_j \in R$ to $H_j$ ;
3 foreach block $S_i$ of S do
4     foreach network host $H_j$ do
5         send block $S_i$ to $H_j$ ;
```

At the expense of network I/O this avoids the need to buffer relation S on disk. Network I/O these days is significantly cheaper than disk I/O. While a typical value for disk bandwidth is ≈ 100 MB/s (or a multiple in RAID configurations), modern interconnects can provide up to several 10 Gb/s. The total I/O cost that the sender H_s has to bear is

$$(|R| + n \cdot |S|) \cdot \text{network throughput} \ .$$

The necessary network transfers are illustrated in Figure 1. Each host receives its share of R plus the full content of S.

[1]We assume that this is done from some host H_s that acts as the data source. The algorithm could trivially be adapted to fetch the source data from one of the processing hosts H_i.

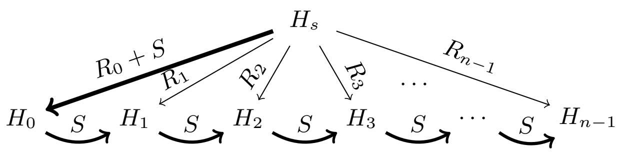

Figure 2: Chaining all processing hosts decreases the network bottleneck at the data source host and takes advantage of the available inter-host bandwidth.

Unfortunately, transmitting the inner join relation S multiple times can cause a serious bottleneck at host H_s.

Note that, since we compute the join in a distributed way, the join result $R \bowtie S$ now ends up as a fragmented relation, distributed over all nodes.

2.4 A Smarter Way to Parallelize

We can decrease the bottleneck at the data source host by taking advantage of the network bandwidth available *between* the processing hosts H_i. We can do so by *chaining* all H_i as illustrated in Figure 2. In this configuration, H_s sends the join relation S only to the *first* processing host H_0. There, we not only evaluate the local fragment of the join, but also *forward* it to the next processing host H_1 using the network link $H_0 \rightarrow H_1$. That is, each node H_i ($i < n - 1$) now executes

```
1 receive $R_i$ from $H_s$ ;
2 foreach block $S_j$ received from $H_s$ or $H_{i-1}$ do
3     compute $R_i \bowtie S_j$ in memory ;
4     forward $S_j$ to host $H_{i+1}$ ;
```

(H_{n-1} simply drops all pieces S_j after processing). The total network I/O load on H_s (which still remains the bottleneck in terms of network I/O volume) is now reduced to

$$(|R| + |S|) \cdot \text{network throughput} \ .$$

In the upcoming section, we discuss how *cyclo-join* pushes our parallelization effort even further to support entire query plans in a distributed fashion. In Section 4, we then investigate how modern networking technology (RDMA) can significantly reduce the remaining network I/O cost.

3. *CYCLO-JOIN*

The ideas of the previous section have minimized the amount of network I/O necessary to process input data that originates from a single host (the data source H_s). In practice, this data may rather be available as distributed tables already, such as those that come from an earlier evaluation of a distributed join. If we are to compute $(R \bowtie S) \bowtie T$, for instance, the result of the inner join $R \bowtie S$ is already distributed at the time we start processing the join with T.

3.1 Idea

To account for such a fully distributed evaluation, we slightly change the logical topology of our network to look as shown in Figure 3. In *cyclo-join*, we organize all processing hosts to form a *ring*. We assume that both input relations are distributed arbitrarily (but reasonably even) across all hosts, say host H_i holds pieces R_i and S_i of R and S. There is no longer an explicit data source host H_s (though, in prac-

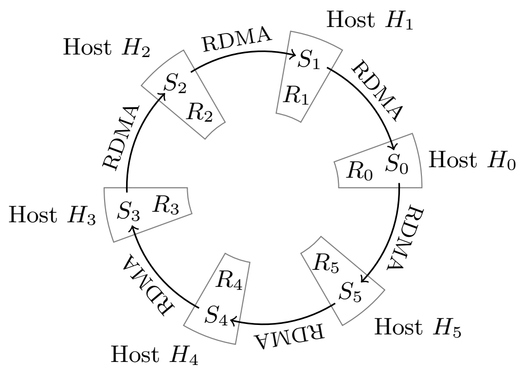

Figure 3: *Cyclo-join*: Network hosts are organized in a logical ring. Relation S circulates in the ring.

tice, one might want to introduce such a host to seed data into the ring).

If we are to evaluate $R \bowtie S$ now, we can compute some sub-results right away, namely those that result from joining pieces that are co-located on some host (using the in-memory join). That is, we compute all $R_i \bowtie S_i$.

Much like in Section 2.4, each node H_i then *forwards* one of its pieces, say S_i, to its next neighbor $H_{(i+1) \bmod n}$, illustrated with arrows $\curvearrowright$ in Figure 3. We join locally again (now $R_i \bowtie S_{(i-1) \bmod n}$), forward, and repeat. For every H_i we get

```
1 foreach block $S_j$ received from $H_{(i-1) \bmod n}$ do
2     compute $R_i \bowtie S_j$ in memory ;
3     forward $S_j$ to host $H_{(i+1) \bmod n}$ ;
```

Unlike in Section 2.4, no sub-relation is dropped: every S_j *continuously circulates* in the ring.

After n iterations, all S_j have performed one full revolution in the *cyclo-join* ring and each host saw the full input relation S once. Hence, every node H_i computed the sub-result $R_i \bowtie S$ locally and the full join result $R \bowtie S$ is available in a distributed form, much like in Sections 2.3 and 2.4.

The time it takes to obtain a full ring revolution depends on two independent factors: the processing time of an individual in-memory join as well as the network transfer speed in the *cyclo-join* ring. In Section 5, we are going to assess how both factors interact.

3.2 *Cyclo-Join* Characteristics

Cyclo-join essentially provides the necessary infrastructure to leverage existing in-memory techniques to the processing of large data sets in a distributed environment.

Leveraging Main-Memory Resources. The main effect of *cyclo-join* is the efficient use of available main-memory resources in a multi-host setup. In many cases, this is going to make the join processing viable at all, when no single host would be available to perform the full join locally.

Applicability. A virtue of *cyclo-join* is that it does not depend on any particular pattern that supported join types would have to satisfy. As such, *cyclo-join* can also be applied to join problems that are not amenable to any of the existing (often hash-based) optimization strategies.

In-Memory Join Processing. Likewise, *cyclo-join* is oblivious of the algorithm that is used to implement the in-memory join. A consequence is that the use of *cyclo-join* will not always yield the same benefit. The resulting CPU load distribution, for instance, will benefit those in-memory join implementations best that would show poor scaling otherwise (such as nested loops joins).

3.3 Implementation Details

Cyclo-join itself is amenable to straightforward optimizations. Most importantly, observe that lines **2** and **3** of the *cyclo-join* pseudo-code above can be executed fully independently (likewise, lines **3** and **4** in the pseudo-code shown in Section 2.4). In Section 4, we are going to exploit this opportunity to perform both tasks *asynchronously* and in *parallel*.

At the receiver end, our implementation uses a *double buffering* scheme. In effect, by overlapping network transfers and join processing, we can hide most of the latency incurred with network I/O.

Our current in-memory join implementation is based on a rather straightforward variant of *hash join*. During the execution of *cyclo-join*, we take advantage of the fact that we can re-use temporary data structures over a full revolution in the ring, even though technically there are n independent join invocations on each node. In the experimental part of this work (Section 5), we characterize our in-memory join implementation in more depth and illustrate its interaction with the processing of *cyclo-join*. Before that, we look into a considerable cost factor that we inherited from Section 2, network processing.

4. RDMA: HARDWARE-ACCELERATED NETWORK PROCESSING

In order to achieve significant performance advantages, *cyclo-join* requires a high-throughput, low-latency underlying transport mechanism.

With the advent of Infiniband and 10Gb Ethernet, physical networks would now support transport speeds that are almost as fast as local main-memory access. But it is known for years that the traditional TCP/IP stack induces a significant load on the local CPU and on the memory bus [4]. According to Mackert and Lohman [9], more than 90% of the CPU cost to evaluate a distributed join in the R* system were spent in the network stack. This observation has inspired a line of work in earlier systems that aimed at reducing I/O cost by reducing the transfer volume [2, 9, 14].

Remote Direct Memory Access. Here we address the problem from a different end. Modern, *RDMA-enabled* network interface cards (so-called *RNICs*) offer support from the hardware side. They can handle high-speed network I/O ($\geq 10\,\text{Gb/s}$) between two hosts with minimal involvement of either CPU. A key concept behind RDMA is *direct data placement* which is a mechanism whereby data is enriched with local placement information such that the RNIC is able to directly access the data in main-memory using DMA. The RNIC has a TCP offload engine built in such that it can perform the network stack processing autonomously.

Figure 4 illustrates a typical RDMA data path: thanks to the placement information, the RNIC of the sending host can fetch the data directly out of local main-memory using DMA. It then transmits the data across the network to

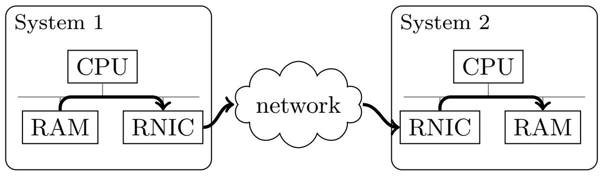

Figure 4: Network transfer using RDMA. RNICs handle data transfer autonomously; data has to cross each memory bus only once.

the remote host where 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 Benefits. The most apparent benefit of using RDMA is the CPU load reduction thanks to the aforementioned direct data placement (avoid intermediate data copies) and OS bypassing techniques (reduced context switch rate). A rule of thumb in network processing states that about 1 GHz in CPU performance is necessary for every 1 Gb/s network throughput [5]. Experiments on our test platform confirmed this rule: even under full CPU load, our 2.33 GHz quad-core system was barely able to saturate the 10 Gb/s link.

A second effect is less obvious: RDMA also significantly reduces the memory bus load as the data is directly DMAed to/from its location in main-memory. Therefore, the data crosses the memory bus only once per transfer. The kernel TCP/IP stack on the other hand requires several such crosses. This may lead to noticeable *contention* on the memory bus under high network I/O. Thus, adding additional CPU cores to the system is *not* a replacement for RDMA.

Applying RDMA. By design, the RDMA interface is quite different from a classical Socket interface. A key difference, which we exploit in our *cyclo-join* implementation, is the asynchronous execution of the data transfer operations which allows overlapping of communication and computation. Taking full advantage of RDMA is not trivial as it has hidden costs [6] with regard to its explicit buffer management. Due to these costs, not every application can benefit from RDMA. However, the *cyclo-join* is an application that clearly can. Figure 5 depicts the raw data throughput per host achieved by the transport layer of our *cyclo-join* implementation as a function of the transfer buffer size. RDMA performs best when large data sets are transferred. With small transfer units as they occur, *e.g.*, in a tuple-by-tuple transmission, only a small fraction of the available bandwidth can be used. We can saturate our network with transfer units ≥ 8 MB.

Availability. RDMA is available for quite some time now through Infiniband [7]. Since recently, RDMA can also be used over Ethernet [11] and therefore no specialized network infrastructure is needed anymore to realize the setup that we describe here (besides the RNICs at a mere $ 800 each).

5. *CYCLO-JOIN* IN ACTION

We built a prototype implementation of an RDMA-accelerated *cyclo-join* to verify some of our expectations in Section 3 and assess the potential of *cyclo-join*. We bench-

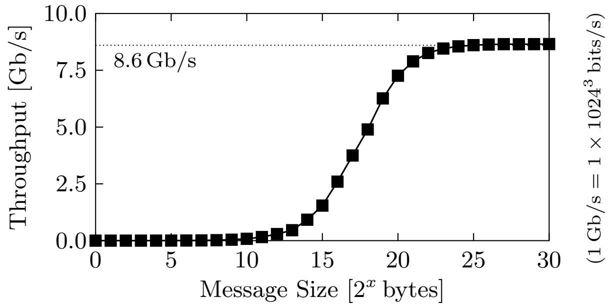

Figure 5: Network throughput achievable with RDMA in a *cyclo-join* ring configuration.

mark our hard- and software infrastructure in the upcoming section, before we actually study *cyclo-join* in Section 5.2.

5.1 Hard- and Software Baseline

We conducted our measurements in an IBM BladeCenter with four HS21 blade servers. Each of them hosts a quad-core Intel Xeon E5345 CPU, clocked at 2.33 GHz, equipped with 32 KB L1 data cache per core and a shared 4 MB L2 data cache for every two cores. Each machine contains at least 6 GB of PC2-5300 SDRAM with a theoretical peak bandwidth of 5 GB/s.[2]

The memory bandwidth that is actually achievable with software is lower. We verified the memory bandwidth of our platform by running Zack Smith's *bandwidth* tool [13] on each of our machines. We observed a maximum read bandwidth of 3.4 GB/s and a write bandwidth of at most 2.5 GB/s. In all measurements that follow, we thus assume a memory bus bandwidth of 3.4 GB/s.

We use Chelsio T3 network cards (model S320EM-BCH) as our RDMA back-end. All machines are connected via a Nortel 10 Gb Ethernet switch.

All join experiments in the following use tables which we populated with random data. The join column is a 4-byte integer containing 32-bit random values. Each tuple also contains an 8-byte random payload. To eliminate a dependency on join hit rates, in all experiments we only *count* the number of matches, but do not actually materialize the join result in main-memory. When reading the studies that follow, keep in mind that result materialization would cause additional memory bus traffic.

In-Memory Join Kernel. As mentioned earlier, we implemented the in-memory join in our system as a hash join, running in two phases:

(1) During the *hash phase* we create a hash table on all entries of the outer join relation R. We also physically reorganize both input relations to cluster all data by hash values.
(2) In the *join phase* we then scan the inner relation S and probe into the hash table for each tuple. Since we had both input tables reorganized before, this effects in a single sequential read of both input tables from main-memory.

In the *cyclo-join* setup we are going to run the hash phase only once, but execute the join phase for every S-block re-

[2]Often reported as 5.3 GB/s when actually 5.3 billion bytes per second are meant.

	hash	join	throughput
1 thread	76 s	5.29 s	0.68 GB/s
2 threads	76 s	2.64 s	1.35 GB/s
3 threads	76 s	1.75 s	2.04 GB/s
4 threads	76 s	1.34 s	2.67 GB/s
MonetDB (1 thread)	41.9 s		

Table 1: In-memory join throughput. 1.8 GB $\bowtie$ 1.8 GB with tuples of 12 bytes each.

configuration	hash	join	sync	total
Ⓐ: 1 host	75.7 s	1.35 s	–	77.0 s
Ⓑ: 2 hosts	33.3 s	1.61 s	0.80 s	35.7 s
Ⓒ: 3 hosts	21.8 s	1.75 s	0.60 s	24.1 s
Ⓓ: 4 hosts	14.2 s	1.74 s	0.42 s	16.4 s

Table 2: Effect of distributing a join 1.8 GB $\bowtie$ 1.8 GB over multiple nodes.

ceived from the network.

To make best use of our available hardware, we run the join phase using multi-threaded code. Each S-block is divided into four equi-sized sub-blocks that are then distributed over the four available cores. Table 1 illustrates how this effort pays off in decreased join execution time. Our multi-threaded implementation achieves a data throughput of 2.67 GB/s, about $3/4$ of the memory bandwidth our machine would be able to provide.

We did not invest efforts in optimizing the hash phase. For reference, we listed the total hash join execution time (including hash-buildup time) achievable with a distribution copy of MonetDB 5. Techniques like those implemented in MonetDB [10] or suggested by others [12] could be used to bring our implementation up to comparable speeds.

5.2 *Cyclo-Join*

The main interest of our experiments is to assess the potential of *cyclo-join*, but also its principal or hardware-specific limitations. Toward this end, we dissect our join implementation to see how its components interact with *cyclo-join.*

5.2.1 *Parallelized Hashing*

In Table 2, we show the effect of distributing the evaluation of a join $R \bowtie S$, where each relation is 1.8 GB in size. For each configuration Ⓐ to Ⓓ, we dissect processing times into the time required for the hash table build-up and on join processing (columns 'hash' and 'join'). Column 'sync' reports additional time that our implementation has to spend waiting for data to arrive via RDMA.

Distributed processing has a significant effect on the hash phase of our join algorithm, whose performance improves roughly proportional to the number of nodes in our network. This comes at no surprise: a join evaluation that uses n hosts splits up both input table into n equi-sized chunks, which are then hashed independently. In our setup, hashing dominates the overall cost, which leads to a more than four-fold improvement in total execution time.

As can be seen in column 'join' of Table 2, the join phase of our algorithm, by contrast, is hardly affected by parallelization. In the join phase, we pay a penalty for chopping

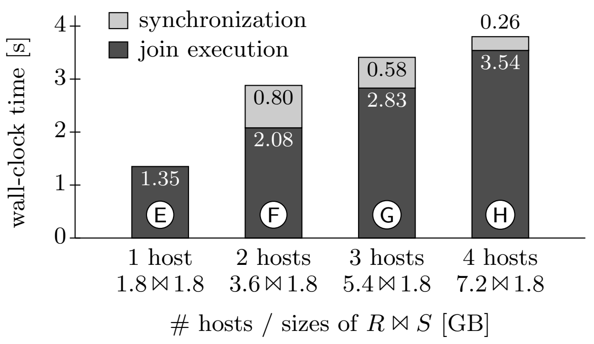

Figure 6: Execution times for increasing data volumes, distributed over an increasing number of nodes.

input data into pieces (R now has to be processed n times in total). Distribution over n nodes just about compensates this penalty and we see no significant net change in the time spent in the join phase.

To understand this effect better, but also to assess the performance limits of *cyclo-join*, let us now "zoom in" into the join phase.

5.2.2 *Parallelizing the Join Phase of Hash Join*

Figure 6 illustrates the outcome of a similar effect. But we now only focus on the costs of the join phase and of network transfers. Configuration Ⓔ is a repetition of configuration Ⓐ. In Ⓕ through Ⓗ we increase the size of input relation R and bring in more nodes to provide the necessary main-memory resources.

Join Execution Time. The bars printed in dark gray in Figure 3 illustrate the total amount of time spent in the in-memory join execution routine. The numbers are consistent with those that we saw earlier (2.67 GB/s, see Table 1). To exemplify, in configuration Ⓕ, each node runs two iterations of a 1.8 GB $\bowtie$ 0.9 GB join, *i.e.*, processes 5.4 GB in total. Divided by the observed join execution time (2.08 s), this implies a data throughput of 2.58 GB/s.

Synchronization Time. On top of the time required for join execution, we need to spend some time waiting for the completion of RDMA transfers, indicated using light-gray in Figure 6. This *synchronization time* considerably affects the overall execution time.

One might be tempted to blame the network for the this delay. If we look at configuration Ⓗ, for instance, 3.54 seconds should be more than enough to bring in the necessary 1.8 GB of data over a 10 Gb/s link. The real culprit is the memory bandwidth of our machines. In configuration Ⓗ, for example, a full *cyclo-join* evaluation requires four rotation steps. During each step, on each host, we need to

read the stationary block R_i	1.80 GB
read the rotating block S_j	0.45 GB
send S_j to the next host via RDMA	0.45 GB
receive S_{j+1} from prev. host via RDMA	0.45 GB
	3.15 GB

Thus, a total of $4 \cdot 3.15\,\text{GB} = 12.6\,\text{GB}$ need to cross the memory bus on each of our machines to compute the join. In our systems, this bus is limited to 3.4 GB/s, effectively limiting the bandwidth that RDMA can use while a local join is being processed.[3] During this time, $3.54\,\mathrm{s} \cdot 3.4\,\mathrm{GB/s} = 12.0\,\mathrm{GB}$ can be brought over the memory bus. Once the join has completed, RDMA processing can proceed at its maximum speed of 1.1 GB/s per connection. 0.26 seconds are exactly the time needed to move the missing data at this speed ($2 \cdot 1.1\,\mathrm{GB/s} \cdot 0.26\,\mathrm{s} = 0.6\,\mathrm{GB}$).

5.3 Discussion

Our observations have implications on what can be expected from a *cyclo-join* implementation in practice.

Viability. Most importantly, our experiments show that *cyclo-join* is indeed a useful way to leverage the main-memory resources of networked machines to process joins over large data volumes. The join phase in configuration Ⓗ, for instance, performs essentially as good as a single machine with a very large memory would. At the same time, we only need to invest a fourth of the time that a single-host configuration would spend in its hash phase.

Bottlenecks. Distributed databases are classically built on the assumption that the network represents the most critical bottleneck, either due to a slow physical network link or due to the significant CPU overhead for network processing [9]. None of these turns out to slow down our implementation, which rather is only limited by the available main-memory bandwidth. We would like to note that more recent systems may already provide the necessary memory bus speeds to max out the potential of *cyclo-join.*

Necessity of RDMA. Our observations underpin the importance of using RDMA as a transport layer for *cyclo-join.* As we pointed out in Section 4, even with sufficient CPU resources, software-based network processing would cause significantly more memory bus traffic and likely slow down join processing.

6. RELATED WORK

We presented our work as an alternative to traditional distributed join techniques such as *fetch matches* [9], *semi-join*-based strategies [2], or ones that use *Bloom filters* [14]. Many of their underlying ideas, however, could still lead to interesting improvements also in a *cyclo-join* setup.

The use of distributed memories and high-speed networks has been explored for OLTP workloads in [8] and for distributed systems in [1, 3].

Our spinning join setup resembles the *DataCycle* system [3] or the *Broadcast Disks* of [1]. Including their techniques into *cyclo-join* is part of our ongoing research and we expect to see significant performance advantages, but also interesting insights into the potential of a merry-go-round setup from doing so.

7. SUMMARY AND OUTLOOK

In this paper we reported on *cyclo-join*, a novel approach to distributed join processing in modern computing networks. *Cyclo-join* goes particularly well with modern networking technology and can efficiently leverage the potential offered by modern, *RDMA-enabled* hardware.

We kept the algorithmic structure of *cyclo-join* deliberately simple. The essence of *cyclo-join* is a merry-go-round of network hosts through which we continuously pump data using RDMA. One consequence is that *cyclo-join* can support arbitrary input data, arbitrary join conditions, and arbitrary network sizes. In experiments, we found the combination of *cyclo-join* and RDMA to be efficient enough to hit the local *main-memory speed* as the limiting performance factor.

Cyclo-join is part of our ongoing research effort *Data Cyclotron.* In *Data Cyclotron*, we plan to push the idea of a data merry-go-round even further: to support arbitrary query types and to adapt dynamically to changing query or data workloads.

[3]Memory controllers are constructed in a way that prioritizes memory transactions issued by the CPU.

8. REFERENCES

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proc. of the ACM SIGMOD*, San Jose, CA, USA, 1995.

[2] Philip A. Bernstein and Dah-Ming W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1), 1981.

[3] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The Datacycle Architecture. *Commun. ACM*, 35(12), 1992.

[4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27, 1989.

[5] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP Performance Re-Visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, 2003.

[6] P. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *Proc. of the 29th ICDCS*, Montreal, QC, Canada, June 2009.

[7] InfiniBand Trade Association. InfiniBand Architecture Specification. `http://www.infinibandta.org`.

[8] S. Ioannidis, E. Markatos, and J. Sevaslidou. Using Network Memory to Improve the Performance of Transaction-Based Systems. In *Proc. of the 4th ACM LCR*, Pittsburgh, PA, USA, May 1998.

[9] L. Mackert and G. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. of VLDB*, Kyoto, Japan, August 1986.

[10] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions Knowledge and Data Engineering*, 2002.

[11] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement, 2005.

[12] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the 20th Int'l Conference on VLDB*, Santiago de Chile, Chile, September 1994.

[13] Z. Smith. Bandwidth: a Memory Bandwidth Benchmark. `http://home.comcast.net/~fbui/bandwidth.html`.

[14] P. Valduriez and G. Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM TODS*, 9(1), March 1984.