Problem Statement

We address a core database problem, but for large problem sizes:

- Process a join $R \bowtie_\theta S$ (arbitrary join predicate).
- $R$ and $S$ are large (many gigabytes, even terabytes).

**Traditional approach:**

- Use a big machine and/or suffer the severe disk I/O bottleneck of block nested loops join.
- Can do distributed evaluation only for certain $\theta$ or certain data distributions (or suffer high network I/O cost).

**Today:**

- Assume a cluster of commodity machines only.
- Leverage modern high-speed networks (10 Gb/s and beyond).
Modern Networks: High Speed?

It is actually very hard to saturate modern (e.g., 10 Gb/s) networks.

- **High CPU demand**
  - Rule of thumb: 1 GHz CPU per 1 Gb/s network throughput (!)

- **Memory bus contention**
  - Data typically has to cross the memory bus **three times**
    \[ \rightarrow \approx 3 \text{ GB/s bus capacity needed for 10 Gb/s network} \]
RDMA-capable network cards (RNICs) can saturate the link using:

- **direct data placement** (avoid unnecessary bus transfers),
- **OS bypassing** (avoid context switches), and
- **TCP offloading** (avoid CPU load).

Data is read/written on both ends using intra-host **DMA**.

**Asynchronous** transfer after **work request** issued by CPU.
Cyclo-Join Idea

1. Distribute
2. Join locally
3. Rotate

RDMA: Join and rotate

Input S

Input R

Host H₁

Host H₂

Host H₃

Host H₄

Host H₅
Cyclo-join has similarities to block nested loops join.

- Cut input data into blocks $R_i$ and $S_j$.
- Join all combinations $R_i \Join S_j$ in memory.

As such, cyclo-join

- can be paired with any in-memory join algorithm,
- can be used to distribute the processing of any join predicate.

Cyclo-join fits into a “cloud-style” environment:

- additional nodes can be hooked in as needed,
- arbitrary assignment host $\leftrightarrow$ task,
- cyclo-join consumes and produces distributed tables $\rightarrow$ n-way joins.
We implemented a prototype of cyclo-join:

- **four processing nodes**
  - Intel Xeon quad-core 2.33 GHz
  - 6 GB RAM per node; memory bandwidth: 3.4 GB/s (measured)

- **10 Gb/s Ethernet**
  - Chelsio T3 RDMA-enabled network cards
  - Nortel 10 Gb/s Ethernet switch

- in-memory **hash join**
  - hash phase *physically re-organizes data* (on each node)
    - better **cache efficiency** during join phase
  - I/O complexity: $\mathcal{O}(|R| + |S|)$
**Experiment 1:** Distribute evaluation of a join where $|R| = |S| = 1.8$ GB.

- **Main benefit:** reduced **hash buildup time**.

![Graph showing wall-clock time for different numbers of hosts with varying sizes of $S \bowtie R$][1]

---

[1]: # hosts / sizes of $S \bowtie R$ [GB]

- 1 host: 1.8 GB
- 2 hosts: 1.8 GB
- 3 hosts: 1.8 GB
- 4 hosts: 1.8 GB
**Experiment 2:** Scale up and join larger \( S \) (hash buildup ignored here).

<table>
<thead>
<tr>
<th># hosts / sizes of ( S \times R ) [GB]</th>
<th>1 host</th>
<th>2 hosts</th>
<th>3 hosts</th>
<th>4 hosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.8 ( \times ) 1.8</td>
<td>1.35</td>
<td>2.08</td>
<td>0.58</td>
<td>0.26</td>
</tr>
<tr>
<td>3.6 ( \times ) 1.8</td>
<td></td>
<td></td>
<td>2.83</td>
<td>3.54</td>
</tr>
<tr>
<td>5.4 ( \times ) 1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.2 ( \times ) 1.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System **scales** like a machine with large RAM would.

CPU\s have to **wait for network transfers** ("synchronization").
Need to wait for network: Does that mean RDMA doesn’t work at all?

- The culprit is the **local memory bus**!
- If RDMA hadn’t saved us some bus transfers, this would be **worse**.
I demonstrated *cyclo-join*:

- **ring topology** to process **large joins**,  
- use **distributed memory** to process **arbitrary joins**,  
- hardware acceleration via **RDMA** is crucial:  
  - reduce CPU load and memory bus contention.

*Cyclo-join* is part of the *Data Cyclotron* project:

- support for **more local join algorithms**,  
- process **full queries** in a **merry-go-round setup**.