Track Join
Distributed Joins with Minimal Network Traffic

Orestis Polychroniou
Rajkumar Sen
Kenneth A. Ross

Oracle Labs
Columbia University
Local Joins

- Algorithms
  - Hash Join
  - Sort Merge Join
  - Index Join
  - Nested Loop Join

- Spilling to disk
  - Bounded by disk bandwidth

- When RAM resident
  - Scale by number of cores
  - Bounded by RAM bandwidth
RAM > Network

- RAM bandwidth?
  - An example
    - 2-channel 1333 MHz RAM = ~18 GB/s
    - Add 4-channel RAM = ~30 GB/s
    - Add 4 CPUs = ~120 GB/s
  - Partition = ~1/3 of bandwidth
    - Partition = copy
      [Satish et.al. SIGMOD ’10, Wassenberg et.al. EuroPar ’11]

- Network bandwidth?
  - Measure (partition) all-to-all
    - 10 Gbit Ethernet < 1 GB/s
    - QDR InfiniBand 4X < 3 GB/s
Broadcast Join

- Network cost
  - Transfer $\min(|R|, |S|) \times 3$
  - Schedule transfers optimally
Hash Join

- Network cost
  - Transfer \((|R| + |S|) \times \frac{3}{4}\)
  - Distribution of (almost) equal partitions
Hash Join

- Pros & cons
  - Broadcast join can be expensive
    - Useful only if |R| << |S|
  - Good for load balancing
    - Hashing randomizes the keys
  - Bad in locality awareness
    - (Again) Hashing randomizes the keys
  - Real datasets have locality
    - Deliberate clustering (optimization)
    - Time-based locality due to appends

Before partition

Node: 1 2 3

key: X
val: Y

After partition

key: X
val: Y

key: X
val: Z
Track Join (2-phase)

1) Partition unique keys

2) Distribute unique keys

- Tracking
  - Hash distribute join keys
  - Eliminate duplicates
Track Join (2-phase)

- Selective broadcast (last step)
  - For a single join key
Track Join (2-phase)

- 2-phase track join
  - Move $R$ tuples to $S$ tuple locations
    - $S$ payloads stay in place: never move over the network
  - Cost: tracking + $\min(|R|, |S|) \times$ repeats
    - $\min(|R|, |S|)$ decided by tuple width ($=\text{payload width}$)

- 3-phase track join
  - Decides tuple “direction” dynamically
    - Which table to move & which to keep in-place
  - **Augment** tracking with counts
    - **Counts** per unique key
Track Join (3-phase)

1) Count-Aggregate
   unique keys

2) Partition unique
   keys & counts

3) Distribute unique
   keys & counts

- Tracking
  - Count-aggregate keys
  - Hash distribute keys & counts
Schedules / Algorithm

- Hash Join (cost = 10)

- 3-phase Track Join (cost = 8)

- 2-phase Track Join (cost = 12)

- 4-phase Track Join (cost = 6)
Track Join (4-phase)

- Compute **optimal** Cartesian product join schedule
  - Track using keys & counts
    - As in 3-phase track join
  - Optimize $R$ to $S$ broadcast, and $S$ to $R$
    - Compute $R$ to $S$ broadcast, and $S$ to $R$
    - Allow migration of $S$ tuples for $R$ to $S$, and $R$ tuples for $S$ to $R$
    - Provably optimal in linear time
    - Pick best (optimized) direction for migrate & broadcast
- Execute the optimal schedule
  - **First** migrate tuples from one table
  - **Then** broadcast tuples from the other table
Schedule Optimization

- Broadcast (cost = 0 + 33)

- Migrate 9? No (cost = 13 + 16 > 28)

- Migrate 4? Yes (cost = 4 + 24 < 33)

- Migrate 6? Yes (cost = 10 + 14 < 28)
Network Cost Approximation

- When to use instead of hash join?
  - Using standard statistics
    - # tuples
    - # distinct keys
  - Distinguish classes of correlation (= similar cartesian products)
    - Use correlated sampling [Yu et.al. SIGMOD ’13]
  - Use track join
    - 2-phase if at least one table has unique keys
    - 4-phase if many key repeats or locality is expected
  - Use hash join
    - If payloads are small (e.g. key & record id only) and no locality exists
Track/Hash/Semi Joins

❖ Track join is a form of semi-join
  ❖ Tracking generates schedules for valid Cartesian products only
    ❖ Non-approximate like Bloom filter based semi-join (Bloom join)
    ❖ Cost (of tracking) = distribute unique join keys (& counts)
  ❖ Still may use semi-join on top of track join
    ❖ Bloom filtering < tracking

❖ However may skip semi-join unlike hash join
  ❖ Tracking < Bloom filtering

❖ Hash join can become tracking-aware
  ❖ Use record ids (rids) to track joining payloads
    ❖ In the best case as good as 2-phase track join
Network Traffic Simulations

- Unique keys join (1 billion vs. 1 billion tuples)
  - R: 20 bytes
  - S: 60 bytes
  - R: 40 bytes
  - S: 60 bytes
  - R: 60 bytes
  - S: 60 bytes

Network Traffic (GB)

- HJ
- 2TJ-R
- 2TJ-S
- 3TJ
- 4TJ

- S Tuples
- R Tuples
- Keys & Nodes
- Keys & Counts
Simulating Locality

- Simulate locality patterns and **degree** of locality
  - Experiment 1: 1 vs. 5 keys per Cartesian product
  - Experiment 2: 5 vs. 5 keys (=25 in result) **intra**-table collocated
  - Experiment 3: 5 vs. 5 keys **intra**-table & **inter**-table collocated

- 5,0,0,0,0,…
- 2,2,1,0,0,…
- 1,1,1,1,0,…
Simulating Locality

Network Traffic (GB)

<table>
<thead>
<tr>
<th></th>
<th>HJ</th>
<th>2TJ-R</th>
<th>2TJ-S</th>
<th>3TJ</th>
<th>4TJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>20</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>25</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>30</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>35</td>
<td>40</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>40</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>45</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>45</td>
</tr>
</tbody>
</table>

Network Traffic (GB)

<table>
<thead>
<tr>
<th></th>
<th>HJ</th>
<th>2TJ-R</th>
<th>2TJ-S</th>
<th>3TJ</th>
<th>4TJ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>20</td>
<td>25</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>25</td>
<td>30</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>30</td>
<td>35</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>35</td>
<td>40</td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>40</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>55</td>
<td>45</td>
<td>50</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>45</td>
</tr>
</tbody>
</table>
Real Workloads

- Real commercial vendor workloads
  - Profiled using commercial DBMS
    - 4 nodes x 2 CPUs (2.9 GHz) x 8 cores
    - QDR InfiniBand 4X
  - Extracted the most expensive queries
    - Extracted the most expensive join from them
    - Executed in the DBMS as a hash join

- Simulating track join
  - Multiple encoding schemes
    - Variable length types
    - Optimal compression schemes
Real Workload 1 Traffic Simulation

- Most expensive query of workload
  - Query joins 7 relations and aggregates
  - Most expensive join takes 23% of time
  - Almost entirely unique keys

- Fixed byte encoding
- Variable byte encoding
- Dictionary compression

Network Traffic (GB)

- S Tuples
- R Tuples
- Keys & Nodes
- Keys & Counts
Real Workload 1 Traffic Simulation

- Most expensive query of workload
  - Exhibited significant locality
  - **Shuffle** the data randomly
  - **No** locality is possible now

- Fixed byte encoding
- Variable byte encoding
- Dictionary compression

Network Traffic (GB)
Real Workload 2 Traffic Simulation

- Most expensive query of workload
  - 2-phase suffices for unique keys
    - 3-phase / 4-phase are redundant
- Workload 2 is different
  - No unique keys
- Very high selectivity
  - R: ~40 million tuples
  - S: ~200 million tuples
  - RS: >1 billion tuples
- Variable byte encoding
  - Base 100 / byte

<table>
<thead>
<tr>
<th>Network Traffic (GB)</th>
<th>Original order</th>
<th>Shuffled order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HJ</td>
<td>HJ</td>
</tr>
<tr>
<td></td>
<td>2TJ-R</td>
<td>2TJ-R</td>
</tr>
<tr>
<td></td>
<td>2TJ-S</td>
<td>2TJ-S</td>
</tr>
<tr>
<td></td>
<td>3TJ</td>
<td>3TJ</td>
</tr>
<tr>
<td></td>
<td>4TJ</td>
<td>4TJ</td>
</tr>
</tbody>
</table>

Legend:
- S Tuples
- R Tuples
- Keys & Nodes
- Keys & Counts
Real Workload Experiments

❖ Implementation
  ❖ **Sort** for in-memory join

  ❖ De-pipelined operators
    ❖ De-couple network & CPU measurement
    ❖ Experiments are invariant of network speed

❖ Run on small private cluster
  ❖ 4 nodes x 2 CPUs (2.66 GHz) x 4 cores
  ❖ Accurately project any network speed

❖ Evaluate real workloads
  ❖ The same cases we simulated
  ❖ On the same expensive join
Real Workload Time Experiments

- Projected (accurately) to 10 Gbit Ethernet
  - CPU vs. network analogous to commercial platforms
  - DBMS profiling platform: ~2.8X network & ~2.2X CPU
  - Schedule generation is fast (insignificant in workload 2)

- Original real workload 1
- Shuffled real workload 1
- Original real workload 2
- Shuffled real workload 2
Conclusions

❖ We introduced **Track Join**
  ❖ For distributed joins
    ❖ Not a hash join
    ❖ Not a broadcast join

❖ Optimize **network traffic**
  ❖ **Track** matching keys using hash join
  ❖ Works at join **key granularity** (not at hash groups)
  ❖ Generate **optimal** Cartesian join schedules **fast** (and in linear time)

❖ Experimental results
  ❖ Reduces network traffic significantly
  ❖ **CPU** time penalty is modest
  ❖ Better with data **locality**
Questions