This Week
→ Status Meetings

Wednesday April 8th
→ Code Review Submission
→ Update Presentation
→ Design Document
PARALLEL JOIN ALGORITHMS

Perform a join between two relations on multiple threads simultaneously to speed up operation.

Two main approaches:
→ Hash Join
→ Sort-Merge Join
TODAY’S AGENDA

Background
Sorting Algorithms
Parallel Sort-Merge Join
Evaluation
SORT-MERGE JOIN (R⨝S)

Phase #1: Sort
→ Sort the tuples of R and S based on the join key.

Phase #2: Merge
→ Scan the sorted relations and compare tuples.
→ The outer relation R only needs to be scanned once.
SORT-MERGE JOIN (R \( \bowtie \) S)

Relation R

Relation S
SORT-MERGE JOIN ($R \bowtie S$)
SORT-MERGE JOIN (R⨝S)

Relation R

MERGE!

Relation S

SORT!
SORT-MERGE JOIN (R \(\bowtie\) S)

Relation R

\[ \text{SORT!} \]

Relation S

\[ \text{SORT!} \]

MERGE!
PARALLEL SORT-MERGE JOINS

Sorting is the most expensive part.

Use hardware correctly to speed up the join algorithm as much as possible.
→ Utilize as many CPU cores as possible.
→ Be mindful of NUMA boundaries.
→ Use SIMD instructions where applicable.
PARALLEL SORT-MERGE JOIN (R⊗S)

Phase #1: Partitioning (optional)
→ Partition R and assign them to workers / cores.

Phase #2: Sort
→ Sort the tuples of R and S based on the join key.

Phase #3: Merge
→ Scan the sorted relations and compare tuples.
→ The outer relation R only needs to be scanned once.
PARTITIONING PHASE

Approach #1: Implicit Partitioning
→ The data was partitioned on the join key when it was loaded into the database.
→ No extra pass over the data is needed.

Approach #2: Explicit Partitioning
→ Divide only the outer relation and redistribute among the different CPU cores.
→ Can use the same radix partitioning approach we talked about last time.
SORT PHASE

Create **runs** of sorted chunks of tuples for both input relations.

It used to be that Quicksort was good enough and it usually still is.

We can explore other methods that try to take advantage of NUMA and parallel architectures …
CACHE-CONSCIOUS SORTING

Level #1: In-Register Sorting
→ Sort runs that fit into CPU registers.

Level #2: In-Cache Sorting
→ Merge Level #1 output into runs that fit into CPU caches.
→ Repeat until sorted runs are ½ cache size.

Level #3: Out-of-Cache Sorting
→ Used when the runs of Level #2 exceed the size of caches.
CACHE-CONSCIOUS SORTING

UNSORTED

Level #1

Level #2

Level #3

SORTED
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.

→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.

→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
Abstract model for sorting keys.

→ Fixed wiring “paths” for lists with the same # of elements.

→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input

| 9 | 5 | 3 | 6 |

Output

| 3 | 5 | 6 | 9 |

wires = [9,5,3,6]

wires[0] = min(wires[0], wires[1])
wires[1] = max(wires[0], wires[1])
wires[2] = min(wires[2], wires[3])
wires[3] = max(wires[2], wires[3])
wires[0] = min(wires[0], wires[2])
wires[2] = max(wires[0], wires[2])
wires[1] = min(wires[1], wires[3])
wires[3] = max(wires[1], wires[3])
wires[1] = min(wires[1], wires[2])
wires[2] = max(wires[1], wires[2])
LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.
→ Fixed wiring “paths” for lists with the same # of elements.
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.

Input

9

5

3

6

Output

1

5

3

6

wires = [9, 5, 3, 6]

wires[0] = min(wires[0], wires[1])
wires[1] = max(wires[0], wires[1])
wires[2] = min(wires[2], wires[3])
wires[3] = max(wires[2], wires[3])
wires[0] = min(wires[0], wires[2])
wires[2] = max(wires[0], wires[2])
wires[1] = min(wires[1], wires[3])
wires[3] = max(wires[1], wires[3])
wires[1] = min(wires[1], wires[2])
wires[2] = max(wires[1], wires[2])
LEVEL #1 – SORTING NETWORKS

<64-bit Join Key, 64-bit Tuple Pointer>
LEVEL #1 – SORTING NETWORKS

Instructions:
→ 4 LOAD
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD
LEVEL #1 – SORTING NETWORKS

Sort Across Registers

Instructions:
→ 4 LOAD

Instructions:
→ 10 MIN/MAX
# LEVEL #1 – SORTING NETWORKS

<table>
<thead>
<tr>
<th>Sort Across Registers</th>
<th>Transpose Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 21 4 13</td>
<td>1 8 3 0</td>
</tr>
<tr>
<td>9 8 6 7</td>
<td>5 11 4 7</td>
</tr>
<tr>
<td>1 14 3 0</td>
<td>9 14 6 10</td>
</tr>
<tr>
<td>5 11 15 10</td>
<td>12 21 15 13</td>
</tr>
</tbody>
</table>

**Instructions:**
- $\rightarrow 4$ LOAD
- $\rightarrow 10$ MIN/MAX
- $\rightarrow 8$ SHUFFLE
- $\rightarrow 4$ STORE
Like a Sorting Network but it can merge two locally-sorted lists into a globally-sorted list.

Can expand network to merge progressively larger lists up to ½ LLC size.

Intel’s Measurements  
→ 2.25–3.5× speed-up over SISD implementation.
LEVEL #2 – BITONIC MERGE NETWORK

Input

\[ \begin{align*}
& a_1 \\
& b_1 \\
& a_2 \\
& b_2 \\
& a_3 \\
& b_3 \\
& a_4 \\
& b_4
\end{align*} \]

Sorted Run

\[ \begin{align*}
& S \ H \ U \ F \ E \\
& S \ H \ U \ F \ E
\end{align*} \]

Reverse Sorted Run

\[ \begin{align*}
& \text{min/max} \\
& \text{min/max} \\
& \text{min/max}
\end{align*} \]

Output

\[ \begin{align*}
& \text{min/max} \\
& \text{min/max} \\
& \text{min/max}
\end{align*} \]
LEVEL #3 – MULTI-WAY MERGING

Use the Bitonic Merge Networks but split the process up into tasks.
→ Still one worker thread per core.
→ Link together tasks with a cache-sized FIFO queue.

A task blocks when either its input queue is empty, or its output queue is full.
Requires more CPU instructions but brings bandwidth and compute into balance.
LEVEL #3 – MULTI-WAY MERGING

Sorted Runs

Cache-Sized Queue

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

MERGE

15-721 (Spring 2020)
IN-PLACE SUPERSCALAR SAMPLESORT

Recursively partition the table by sampling keys to determine partition boundaries.

It copies data into output buffers during the partitioning phases. But when a buffer gets full, it writes it back into portions of the input array already distributed instead of allocating a new buffer.
MERGE PHASE

Iterate through the outer table and inner table in lockstep and compare join keys. May need to backtrack if there are duplicates.

Can be done in parallel at the different cores without synchronization if there are separate output buffers.
SORT-MERGE JOIN VARIANTS

Multi-Way Sort-Merge (M-WAY)
Multi-Pass Sort-Merge (M-PASS)
Massively Parallel Sort-Merge (MPSM)
MULTI-WAY SORT-MERGE

Outer Table
→ Each core sorts in parallel on local data (levels #1/#2).
→ Redistribute sorted runs across cores using the multi-way merge (level #3).

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer/inner tables at each core.
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge

[Diagram showing the process of multi-way sort-merge with local NUMA partitioning and sort stages]
# Multi-Way Sort-Merge

## Local-NUMA Partitioning

<table>
<thead>
<tr>
<th></th>
<th>Sort</th>
<th>Multi-Way Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local-NUMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partitioning</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing the process of Multi-Way Sort-Merge](image_url)
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning | Sort | Multi-Way Merge
--- | --- | ---

CMU-DB

15-721 (Spring 2020)
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge  Same steps as Outer Table

Local-NUMA Partitioning

Sort

Multi-Way Merge

Same steps as Outer Table
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning  Sort  Multi-Way Merge  Local Merge Join  Same steps as Outer Table

Local Merge Join

Same steps as Outer Table
MULTI-PASS SORT-MERGE

Outer Table
→ Same level #1/#2 sorting as Multi-Way.
→ But instead of redistributing, it uses a multi-pass naïve merge on sorted runs.

Inner Table
→ Same as outer table.

Merge phase is between matching pairs of chunks of outer table and inner table.
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning

Local-NUMA Partitioning
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning | Sort | Local-NUMA Partitioning
---|---|---

1. Local-NUMA Partitioning
2. Sort
3. Local-NUMA Partitioning
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning → Sort → Global Merge Join → Sort → Local-NUMA Partitioning
MULTI-PASS SORT-MERGE

Local-NUMA Partitioning  Sort  Global Merge Join  Sort  Local-NUMA Partitioning
MASSIVELY PARALLEL SORT-MERGE

Outer Table
→ Range-partition outer table and redistribute to cores.
→ Each core sorts in parallel on their partitions.

Inner Table
→ Not redistributed like outer table.
→ Each core sorts its local data.

Merge phase is between entire sorted run of outer table and a segment of inner table.
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning
MASSIVELY PARALLEL SORT - MERGE

Cross-NUMA Partitioning

Sort

Globally Sorted
MASSIVELY PARALLEL SORT - MERGE

Cross-NUMA Partitioning

Sort

15-721 (Spring 2020)
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

<table>
<thead>
<tr>
<th>Sort 1</th>
<th>Sort 2</th>
<th>Sort 3</th>
<th>Sort 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15-721 (Spring 2020)
# Massively Parallel Sort-Merge

## Cross-NUMA Partitioning

<table>
<thead>
<tr>
<th>Cross-NUMA Partitioning</th>
<th>Sort</th>
<th>Cross-Partition Merge Join</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Sorting" /></td>
<td><img src="image2" alt="Merge Join" /></td>
</tr>
</tbody>
</table>

- **Cross-NUMA Partitioning**: Each node in a NUMA system is assigned a portion of the data to be sorted. This involves distributing the data across the memory nodes to optimize locality and reduce communication overhead.

- **Sort**: The sorting process is performed on each node independently. This is typically done using parallel algorithms like quicksort or mergesort, which are optimized for parallel execution.

- **Cross-Partition Merge Join**: After sorting, the sorted partitions from different nodes are merged to produce the final sorted result. This merge step is critical for ensuring the data is correctly ordered across the entire system.

The diagram illustrates the flow of data through these steps, showing how the data is efficiently sorted and merged across the NUMA nodes to achieve high performance and scalability in parallel computing environments.
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning  Sort  Cross-Partition Merge Join

[Diagram showing the process of Cross-NUMA Partitioning, followed by Sort and Cross-Partition Merge Join, with arrows indicating the flow of data.]
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning | Sort | Cross-Partition Merge Join
--- | --- | ---

Diagram showing the process of cross-NUMA partitioning, sorting, and cross-merge join.
HYPER's RULES FOR PARALLELIZATION

Rule #1: No random writes to non-local memory
→ Chunk the data, redistribute, and then each core sorts/works on local data.

Rule #2: Only perform sequential reads on non-local memory
→ This allows the hardware prefetcher to hide remote access latency.

Rule #3: No core should ever wait for another
→ Avoid fine-grained latching or sync barriers.

Source: Martina-Cezara Albutiu
EVALUATION

Compare the different join algorithms using a synthetic data set.
→ **Sort-Merge**: M-WAY, M-PASS, MPSM
→ **Hash**: Radix Partitioning

**Hardware:**
→ 4 Socket Intel Xeon E4640 @ 2.4GHz
→ 8 Cores with 2 Threads Per Core
→ 512 GB of DRAM
RAW SORTING PERFORMANCE

Single-threaded sorting performance

Throughput (M Tuples/sec) vs. Number of Tuples (in $2^{20}$)

- C++ STL Sort
- SIMD Sort

Source: Cagri Balkesen

2.5–3x Faster
COMPARISON OF SORT-MERGE JOINS

Workload: 1.6B ⋈ 128M (8-byte tuples)

Source: Cagri Balkesen
**M-WAY JOIN VS. MPSM JOIN**

**Workload: 1.6B ⋈ 128M (8-byte tuples)**

- **Multi-Way**
- **Massively Parallel**

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Throughput (M Tuples/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>108 M/sec</td>
</tr>
<tr>
<td>2</td>
<td>130 M/sec</td>
</tr>
<tr>
<td>4</td>
<td>259 M/sec</td>
</tr>
<tr>
<td>8</td>
<td>315 M/sec</td>
</tr>
<tr>
<td>32</td>
<td>90 M/sec</td>
</tr>
<tr>
<td>64</td>
<td>54 M/sec</td>
</tr>
</tbody>
</table>

Source: Cagri Balkesen

15-721 (Spring 2020)
SORT-MERGE JOIN VS. HASH JOIN

Workload: Different Table Sizes (8-byte tuples)

- Partition
- Sort
- S-Merge
- J-Merge
- Build+Probe

Source: Cagri Balkesen
SORT-MERGE JOIN VS. HASH JOIN

Varying the size of the input relations

- Multi-Way Sort-Merge Join
- Radix Hash Join

Source: Cagri Balkesen
Both join approaches are equally important. Every serious OLAP DBMS supports both.

We did not consider the impact of queries where the output needs to be sorted.
NEXT CLASS

Optimizers – The Hardest Topic in Databases