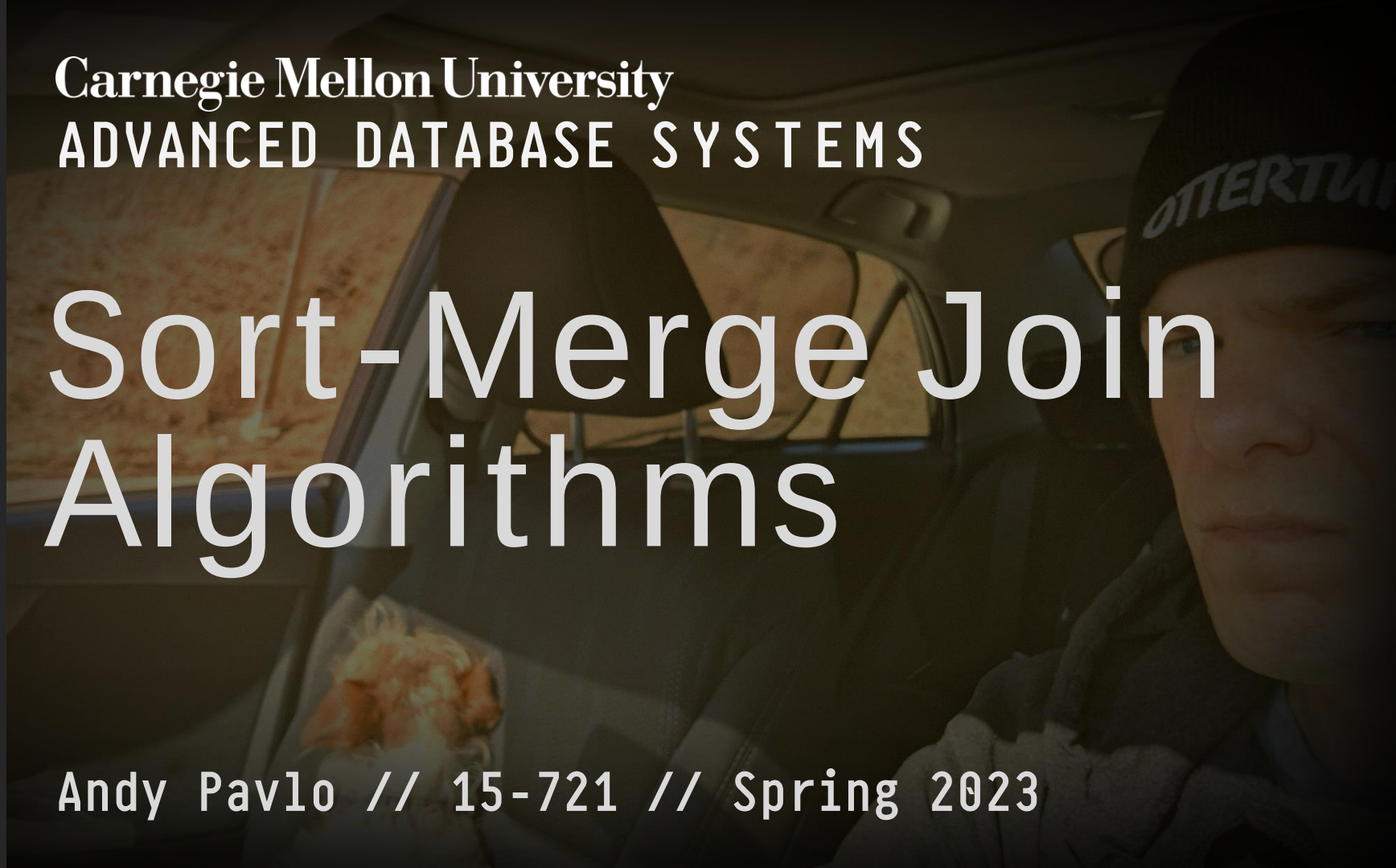
Lecture #12

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Sort-Merge Join Algorithms



Andy Pavlo // 15-721 // Spring 2023

ADMINISTRIVIA

Project #2:

→ Feedback Submission: **Saturday April 1st**
→ Final Submission: **Monday May 1st**
→ Sign up for a system if you haven't yet!

Project #3

→ Status Update Presentation: **Monday April 3rd**
→ Final Presentations: **Friday May 5th @ 5:30pm**

TODAY'S AGENDA

Background

Sorting Algorithms

Parallel Sort-Merge Join

Evaluation

SORT-MERGE JOIN (R⋈S)

Phase #1: Sort

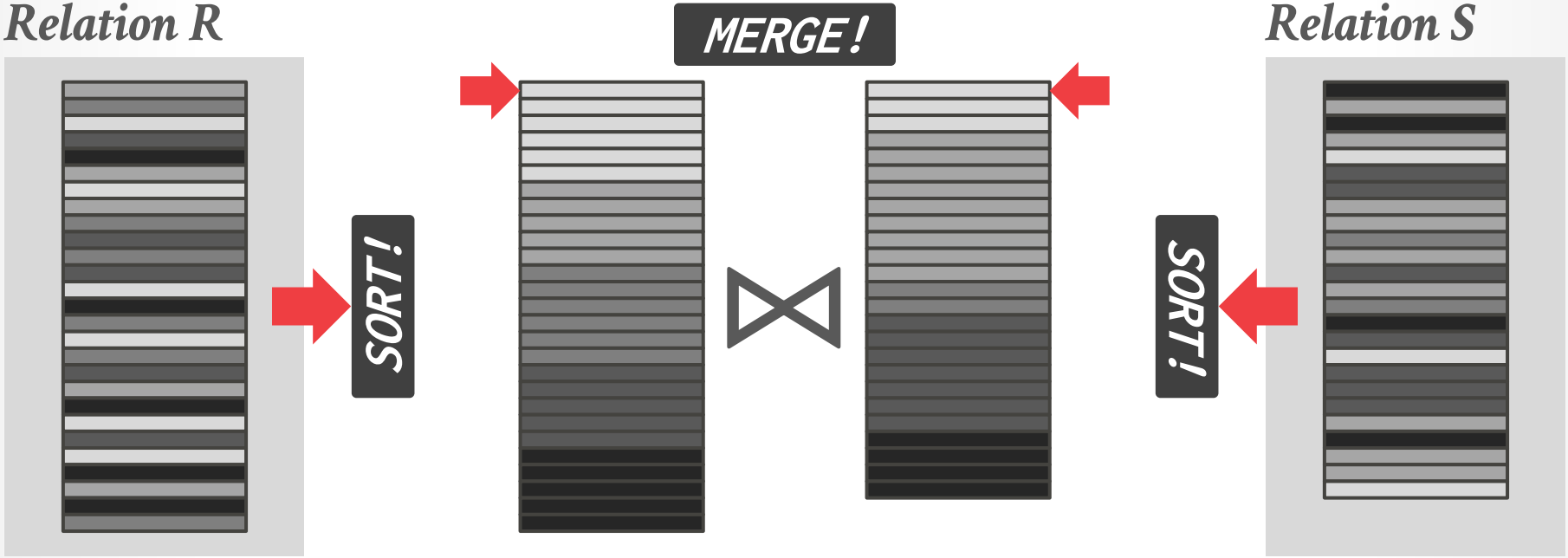
→ Sort the tuples of **R** and **S** based on the join key(s).

Phase #2: Merge

→ Maintain two iterators (one per sorted relation) and compare tuples at each position.

→ The outer relation **R** only needs to be scanned once.

SORT-MERGE JOIN (R⋈S)



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.

MULTI-CORE, MAIN-MEMORY JOINS:
SORT VS. HASH REVISITED
VLDB 2013

PARALLEL SORT-MERGE JOIN (R⋈S)

Phase #1: Partitioning (optional)

→ Partition **R** and assign them to workers / cores.
→ Can use the radix partitioning approach discussed last class.

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.

SORT PHASE

Quicksort is probably what most DBMSs will use.

Mergesort is good but requires O(N) additional storage for intermediate results.

We will first discuss a mergesort implementation that takes advantage of NUMA and parallel cores for in-memory data.

CACHE-CONSCIOUS SORTING

Level #1: In-Register Sorting

→ Sort runs that fit into CPU registers.

Level #2: In-Cache Sorting

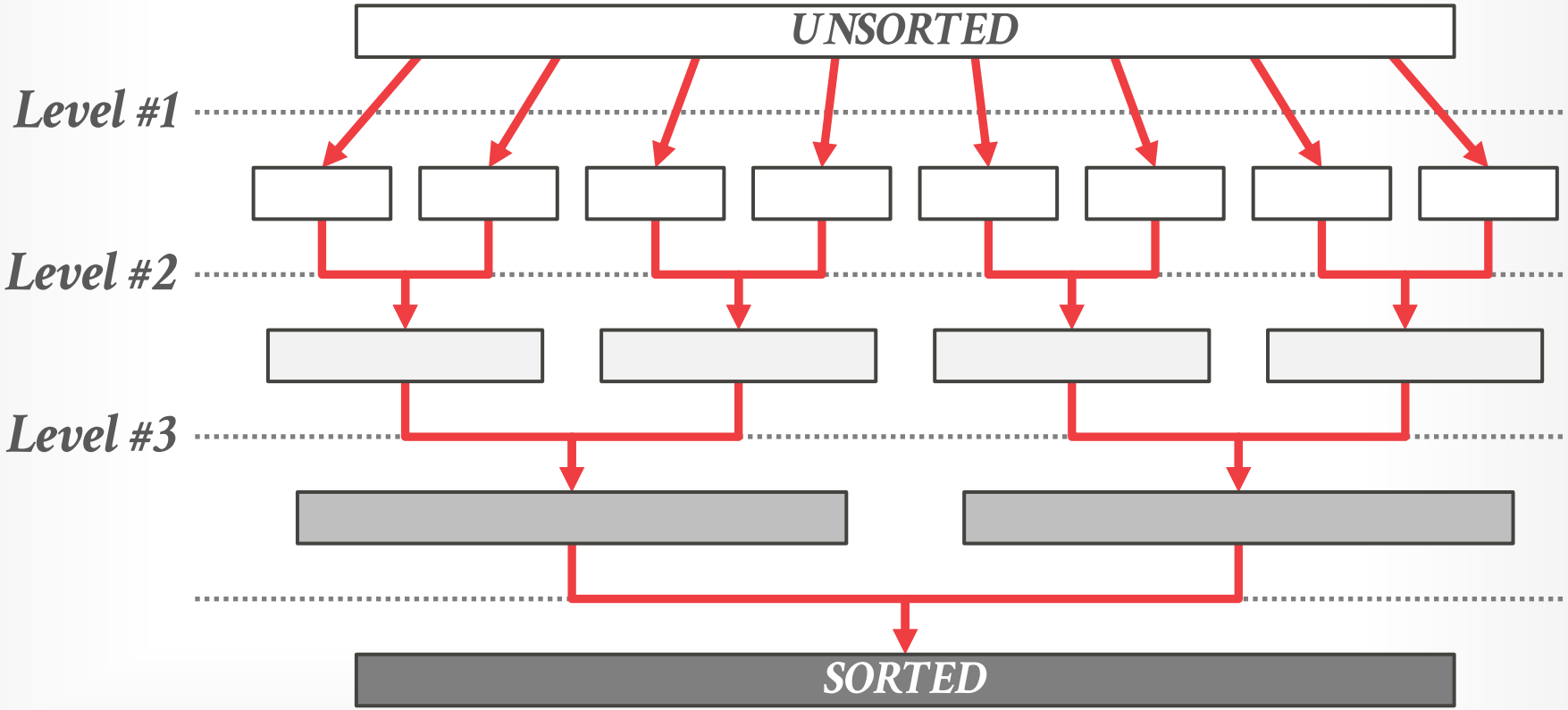
→ Merge Level #1 output into runs that fit into CPU caches.
→ Repeat until sorted runs are ½ cache size.

Level #3: Out-of-Cache Sorting

→ Used when the runs of Level #2 exceed the size of caches.

SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUS
VLDB 2009

CACHE-CONSCIOUS SORTING

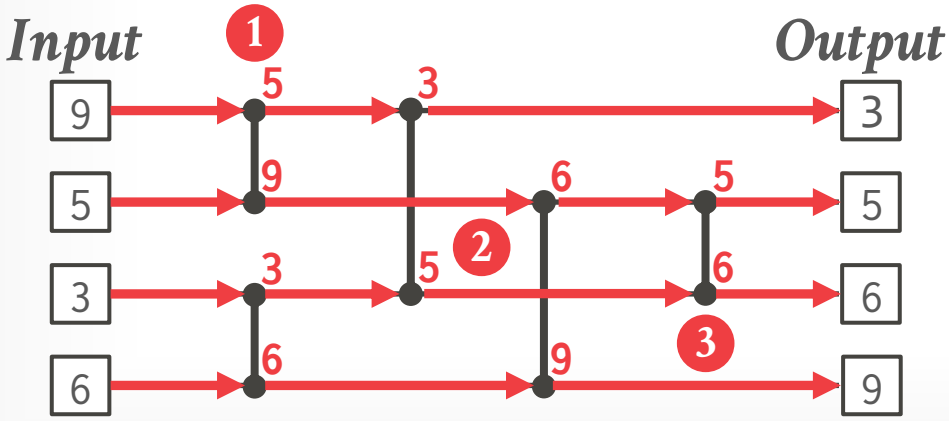


LEVEL #1 - SORTING NETWORKS

Abstract model for sorting keys.

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

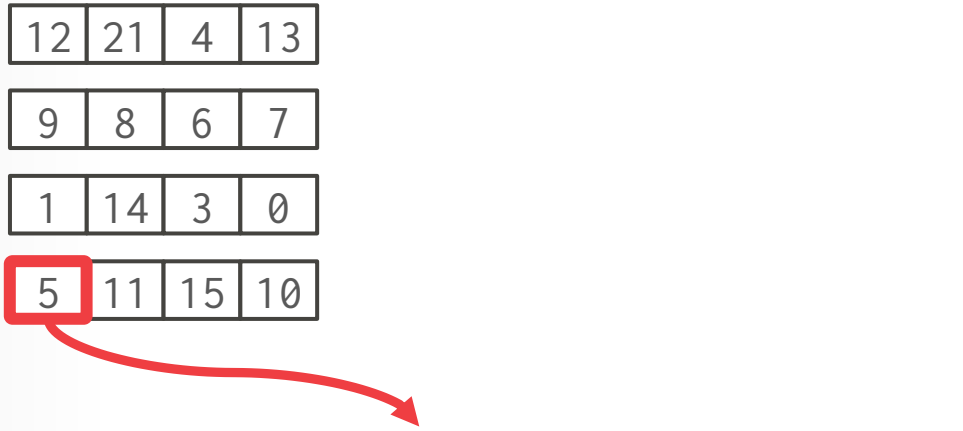
→ Efficient to execute on modern CPUs because of limited data dependencies and no branches.




```
orig = [9,5,3,6]

wires1[0] = min(orig[0], orig[1])
wires1[1] = max(orig[0], orig[1])
wires1[2] = min(orig[2], orig[3])
wires1[3] = max(orig[2], orig[3])

output[0] = min(wires1[0], wires1[2])
wires2[2] = max(wires1[0], wires1[2])
wires2[1] = min(wires1[1], wires1[3])
output[3] = max(wires1[1], wires1[3])

output[1] = min(wires2[1], wires2[2])
output[2] = max(wires2[1], wires2[2])
```

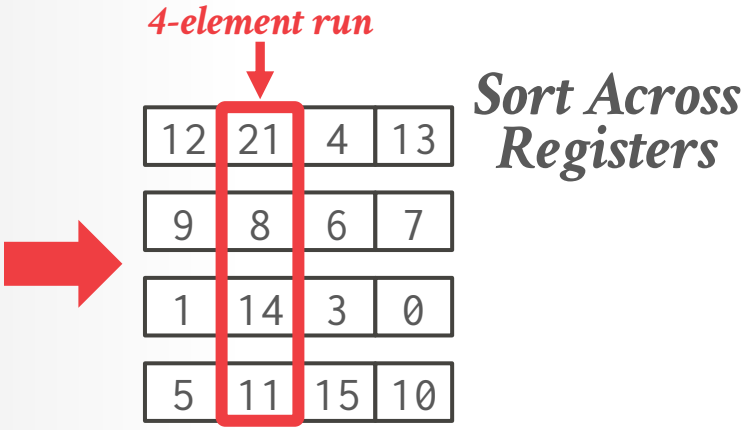
1 2 3

LEVEL #1 – SORTING NETWORKS



<64-bit Join Key, 64-bit Tuple Pointer>

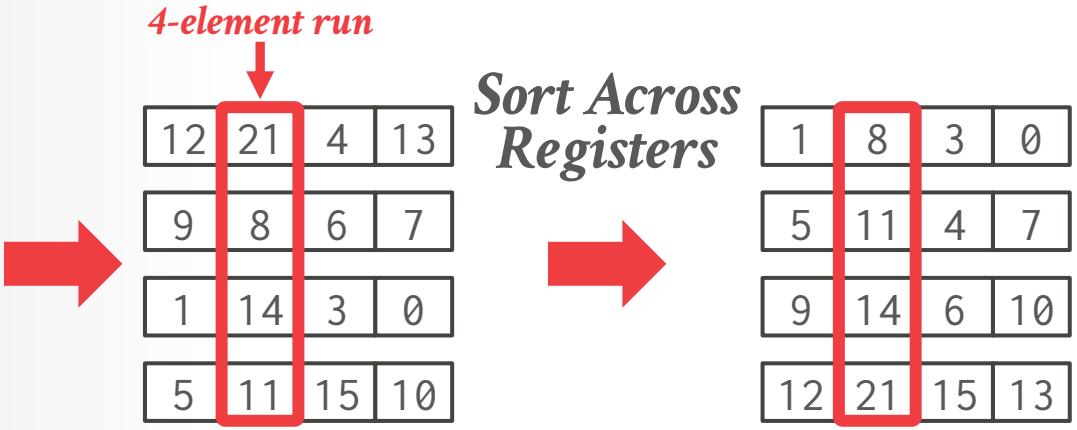
LEVEL #1 – SORTING NETWORKS



Instructions:

→ 4 **LOAD**

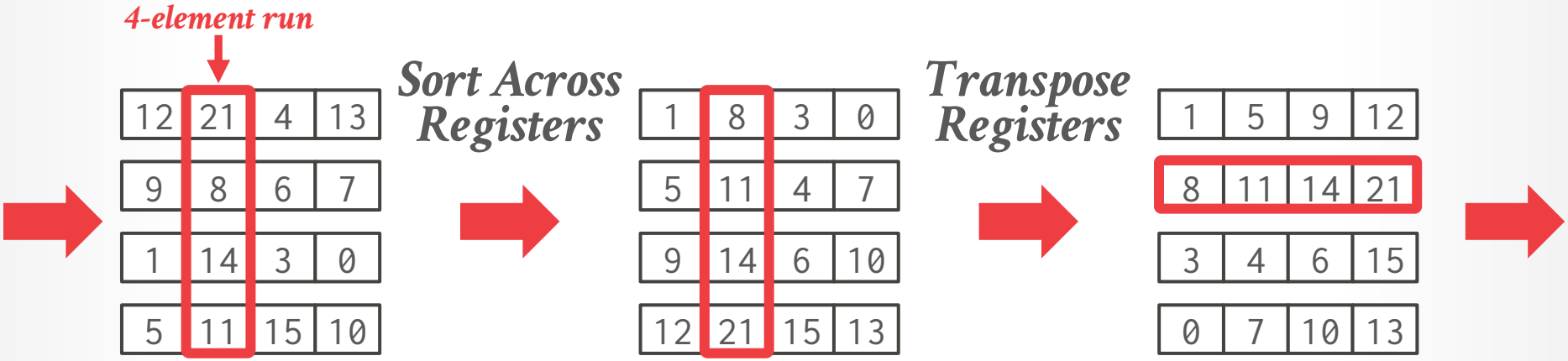
LEVEL #1 – SORTING NETWORKS



Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

LEVEL #1 – SORTING NETWORKS



Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

Instructions:
→ 8 **SHUFFLE**
→ 4 **STORE**

LEVEL #2 – BITONIC MERGE NETWORK

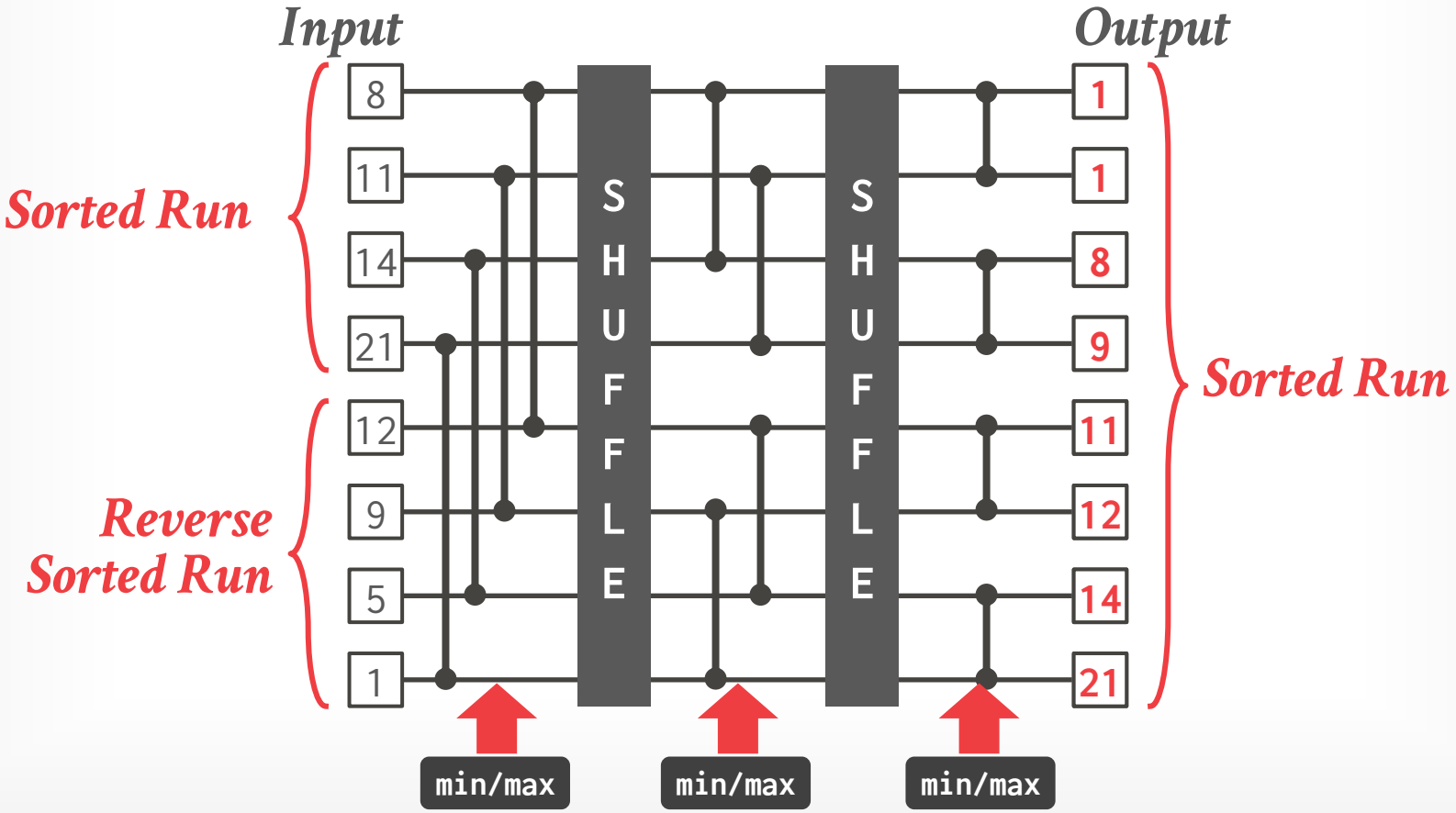
Similar technique as a Sorting Network but merges two locally-sorted lists into a globally-sorted list.

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.

EFFICIENT IMPLEMENTATION OF SORTING ON MULTI-CORE
VLDB 2008

LEVEL #2 - BITONIC MERGE NETWORK



LEVEL #3 – MULTI-WAY MERGING

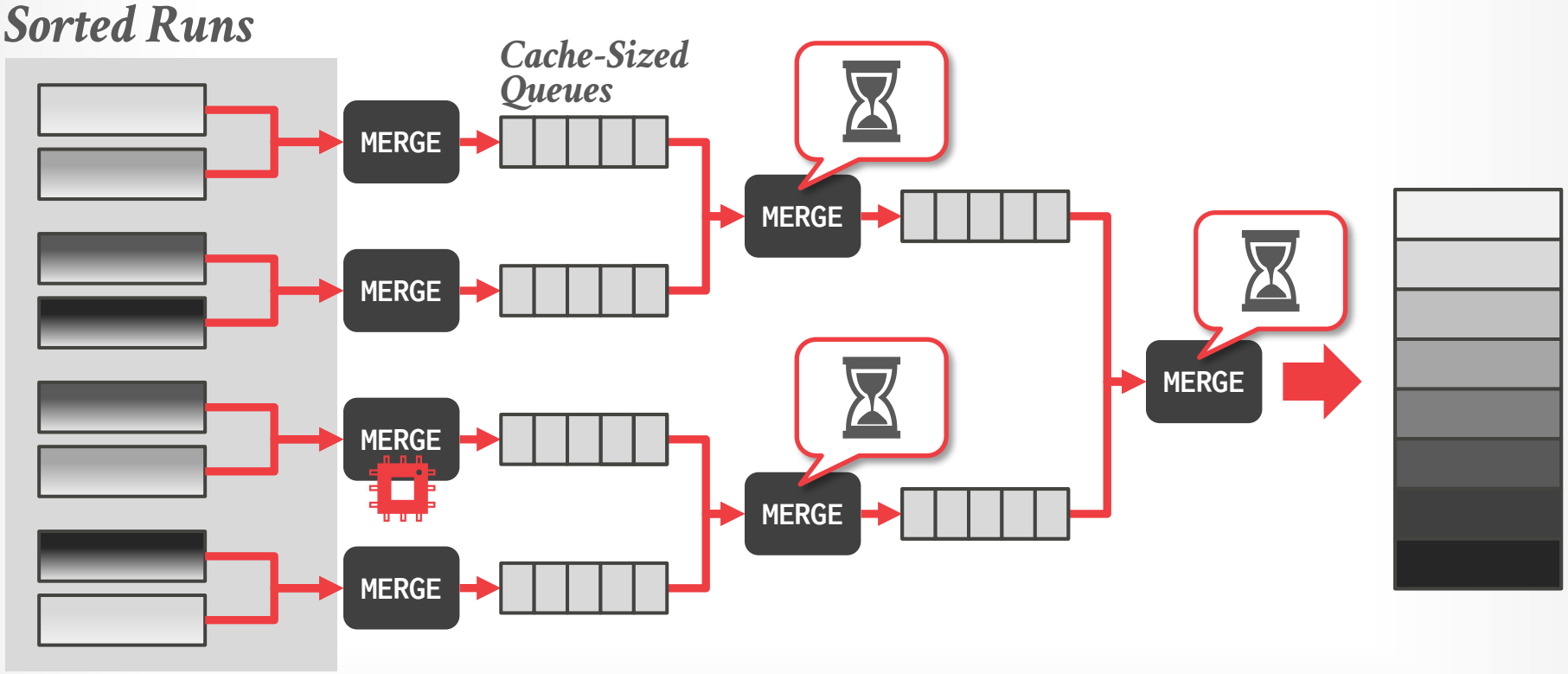
Use the Bitonic Merge Networks but split the process up into tasks.

→ 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



IN-PLACE SUPERSCALAR SAMPLESORT

The IPS^{4}o algorithm (2017) recursively partition relation by sampling keys to determine partition boundaries.

→ Copies data into output buffers during the partitioning phases.

→ When a buffer gets full, the DBMS writes it back into portions of its existing input buffers instead of allocating a new buffer.

This is the sorting algorithm that we used in CMU's NoisePage DBMS (RIP).

IN-PLACE PARALLEL SUPER SCALAR SAMPLESORT
ESA 2017

VECTORIZED QUICKSORT

Google vqsort (2022)

→ Use sorting network for less than 256 keys.
→ Based on Google Highway library to provide support for different ISAs and SIMD register sizes.
→ Claims to be 1.59x faster than IPS^{4}o.

Intel x86-simd-sort (2022)

→ Aggressive use of AVX512 instructions.

MERGE PHASE

Iterate through the outer table and inner table in lockstep and compare join keys.

May need to backtrack if there are duplicates.

The DBMS can execute this phase in parallel using multiple workers without synchronization if there are separate output buffers.

SORT-MERGE JOIN VARIANTS

Multi-Way Sort-Merge (M-WAY)

MULTI-CORE, MAIN-MEMORY JOINS:
SORT VS. HASH REVISITED
VLDB 2013

Multi-Pass Sort-Merge (M-PASS)

MULTI-CORE, MAIN-MEMORY JOINS:
SORT VS. HASH REVISITED
VLDB 2013

Massively Parallel Sort-Merge (MPSM)

MASSIVELY PARALLEL SORT-MERGE JOINS IN
MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012

MULTI-WAY SORT-MERGE

Outer Table

→ Each core sorts in parallel on local data (levels #1/#2).
→ Redistribute sorted runs across cores using range partitioning then perform **multi-way merge** (level #3).

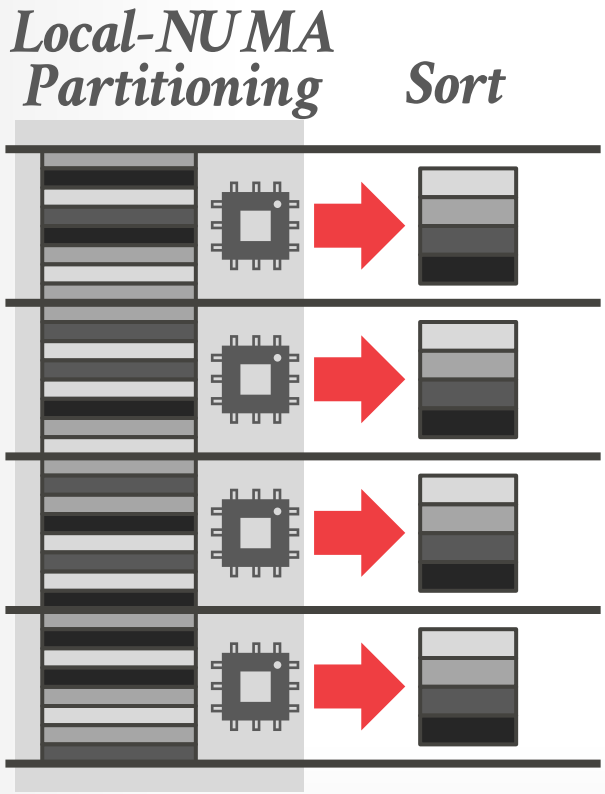
Inner Table

→ Same as outer table.

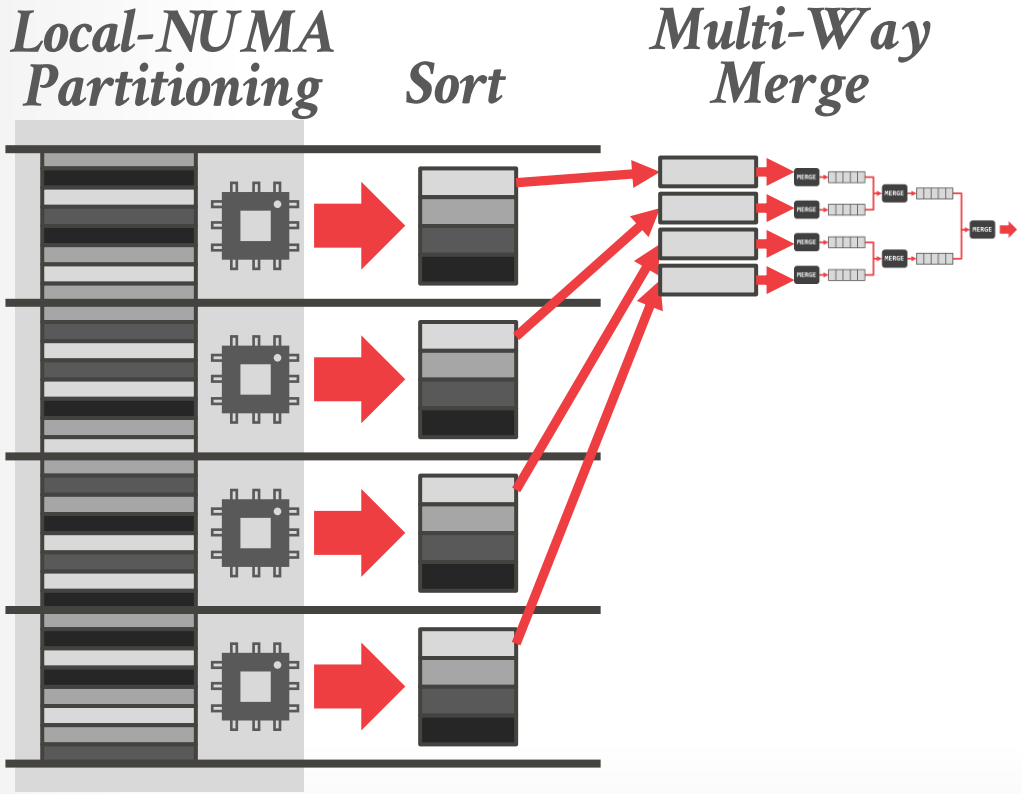
Merge phase is between matching pairs of chunks of outer/inner tables at each core.

MULTI-CORE, MAIN-MEMORY JOINS:
SORT VS. HASH REVISITED
VLDB 2013

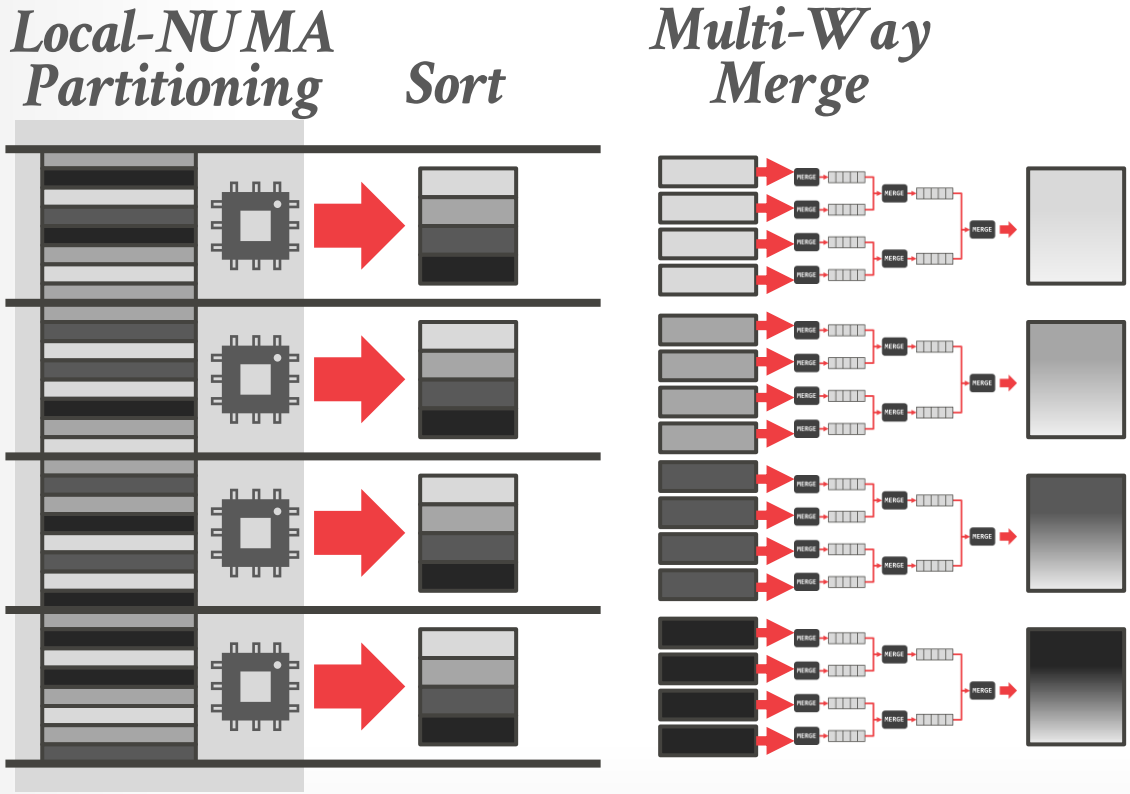
MULTI-WAY SORT-MERGE



MULTI-WAY SORT-MERGE



MULTI-WAY SORT-MERGE



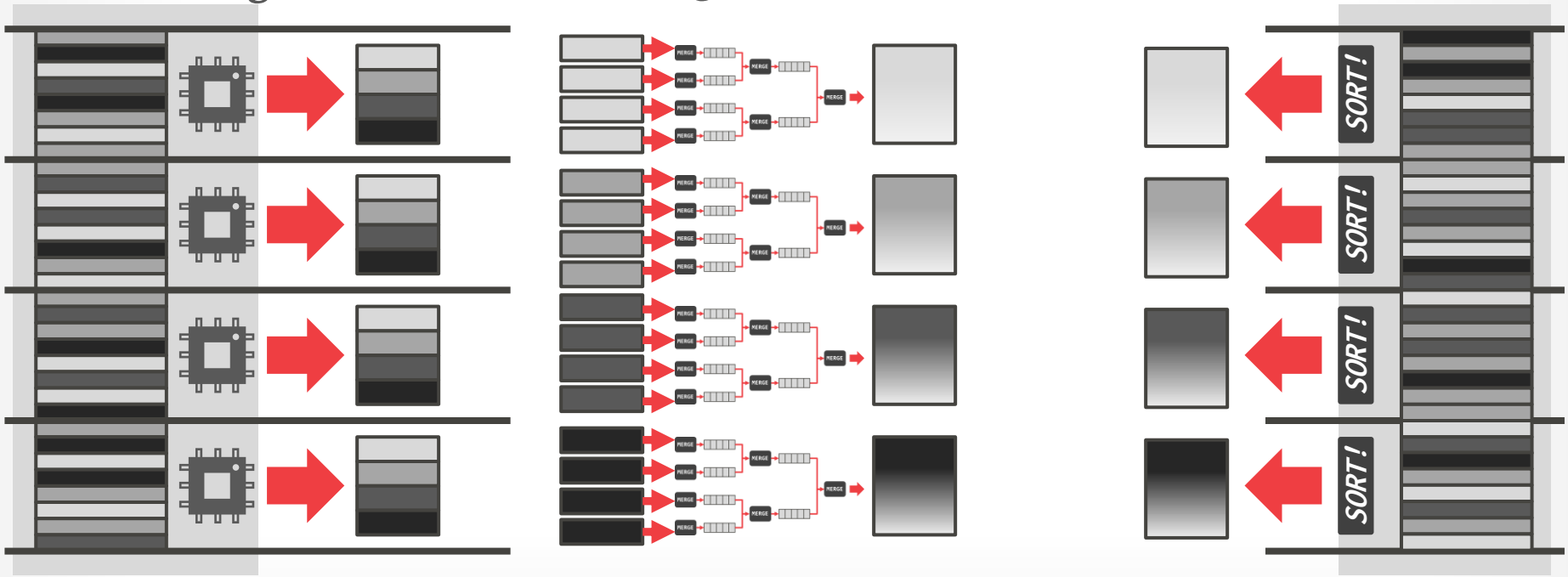
MULTI-WAY SORT-MERGE

Local-NUMA Partitioning

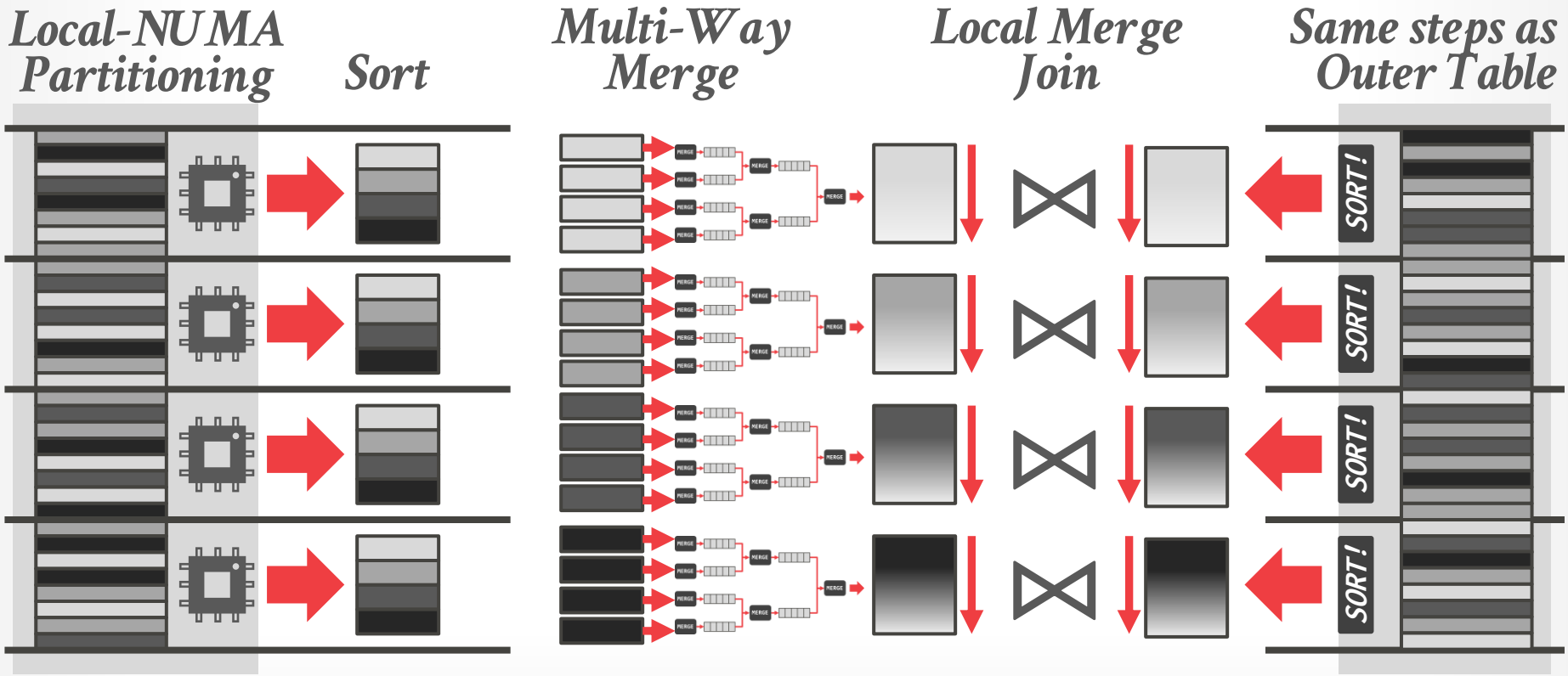
Sort

Multi-Way Merge

Same steps as Outer Table



MULTI-WAY SORT-MERGE



MULTI-PASS SORT-MERGE

Outer Table

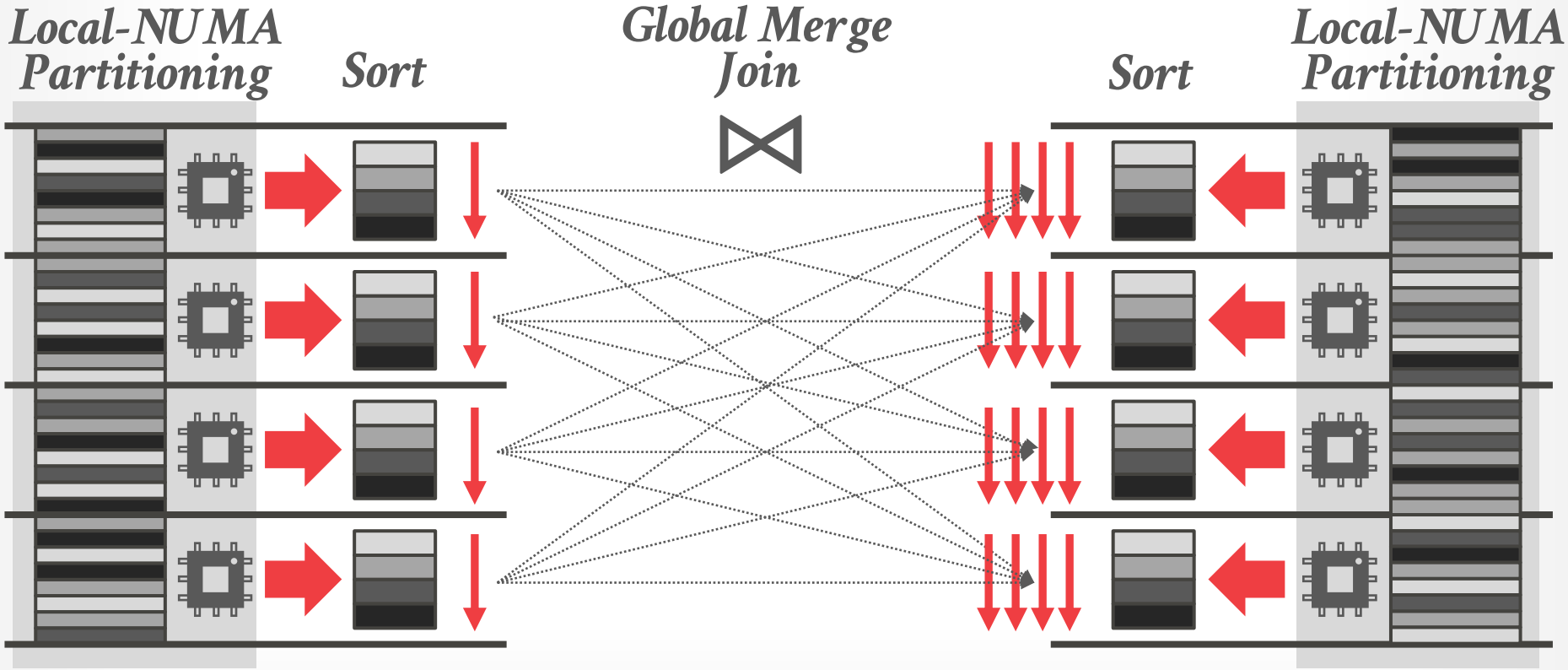
→ Same level #1/#2 sorting as previous Multi-Way Merge.
→ But instead of redistributing data across cores, perform a global **multi-pass naïve merge** on sorted runs.

Inner Table

→ Same as outer table.

Merge phase is between matching pairs of chunks of outer table and inner table.

MULTI-PASS SORT-MERGE



MASSIVELY PARALLEL SORT-MERGE

Outer Table

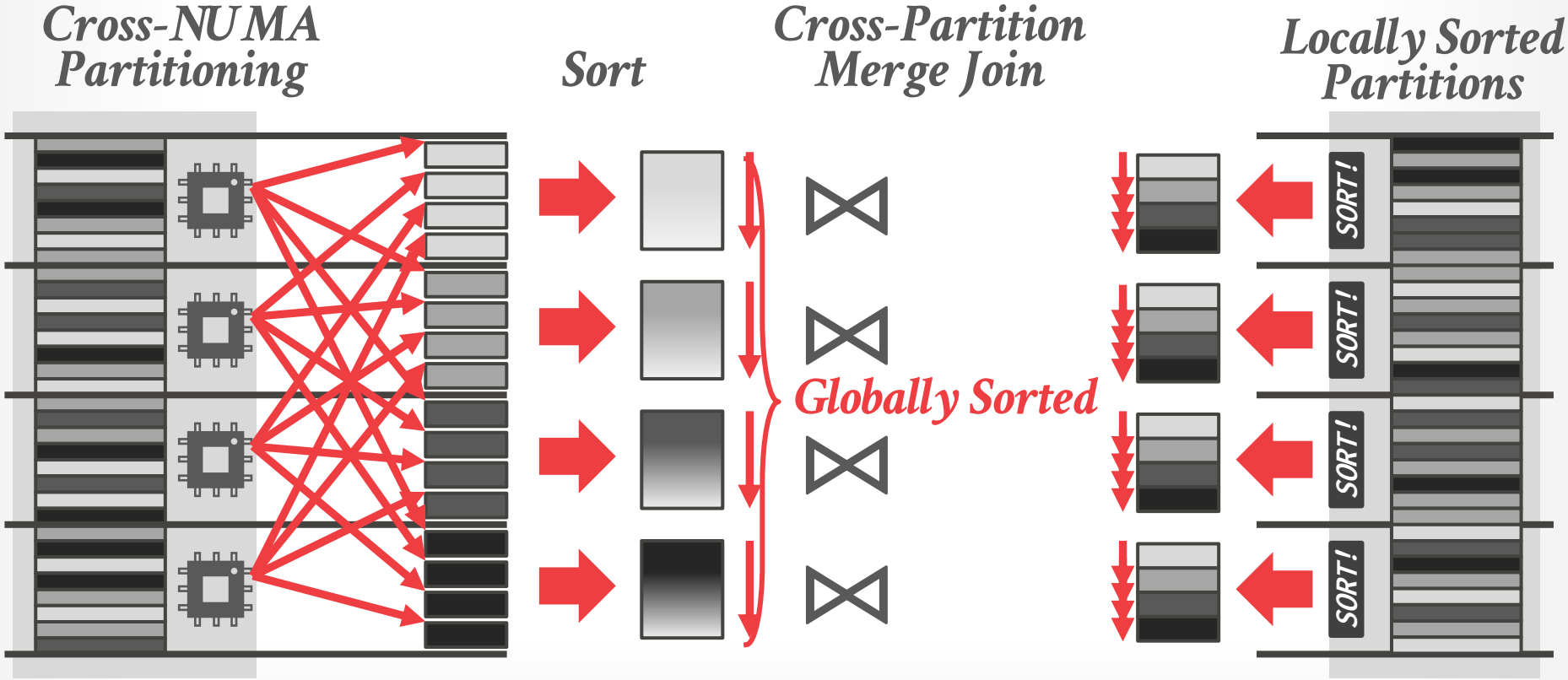
→ Range-partition outer table and redistribute to cores.
→ Each core sorts then in parallel on their local partitions.

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



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:

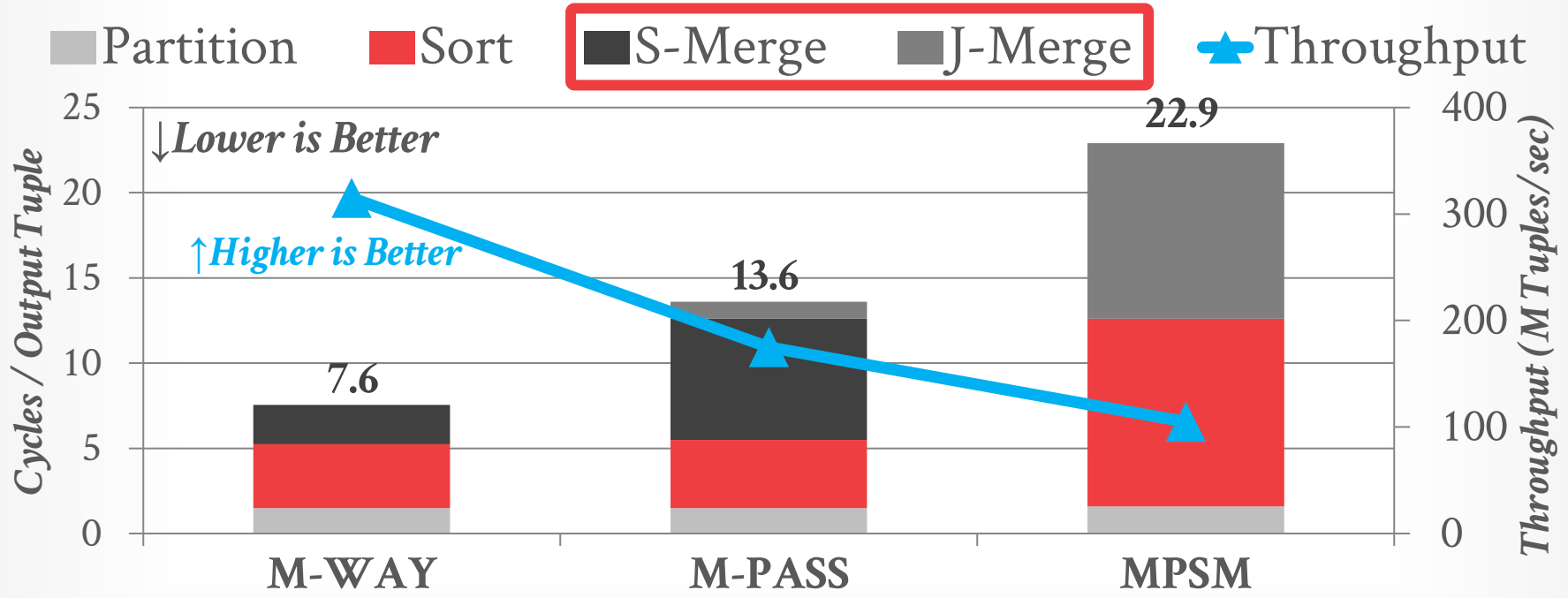
→ 4 Socket Intel Xeon E4640 @ 2.4GHz

→ 8 Cores with 2 Threads Per Core

→ 512 GB of DRAM

COMPARISON OF SORT-MERGE JOINS

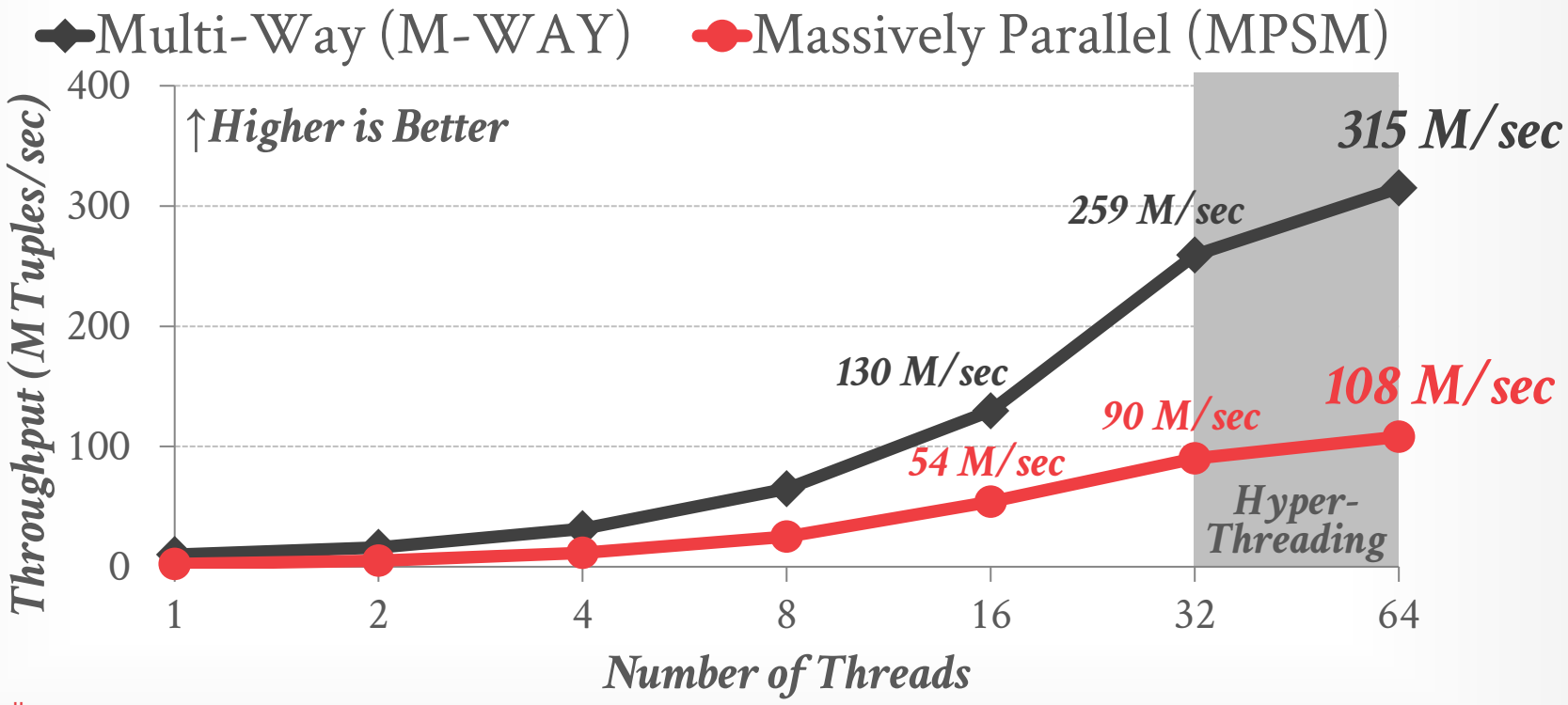
Workload: 1.6B ⋈ 128M (8-byte tuples)



Source: Cagri Balkesen

M-WAY JOIN VS. MPSM JOIN

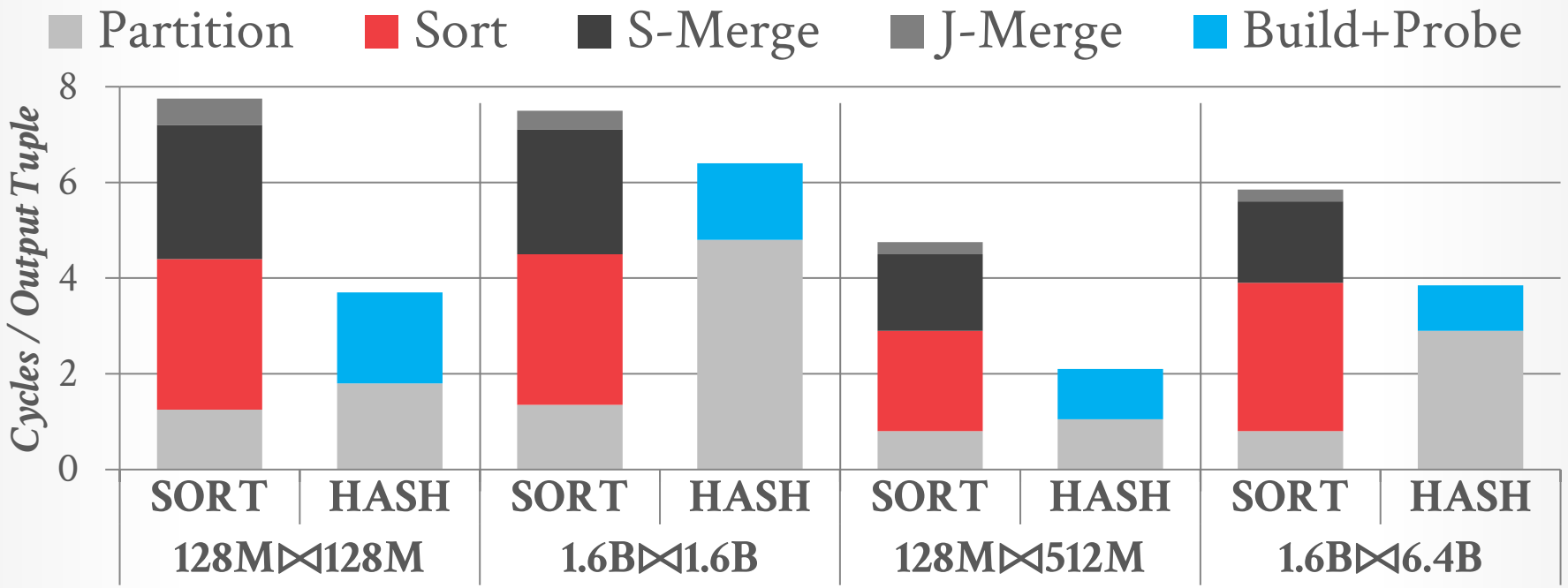
Workload: 1.6B ⋈ 128M (8-byte tuples)



Source: Cagri Balkesen

SORT-MERGE JOIN VS. HASH JOIN

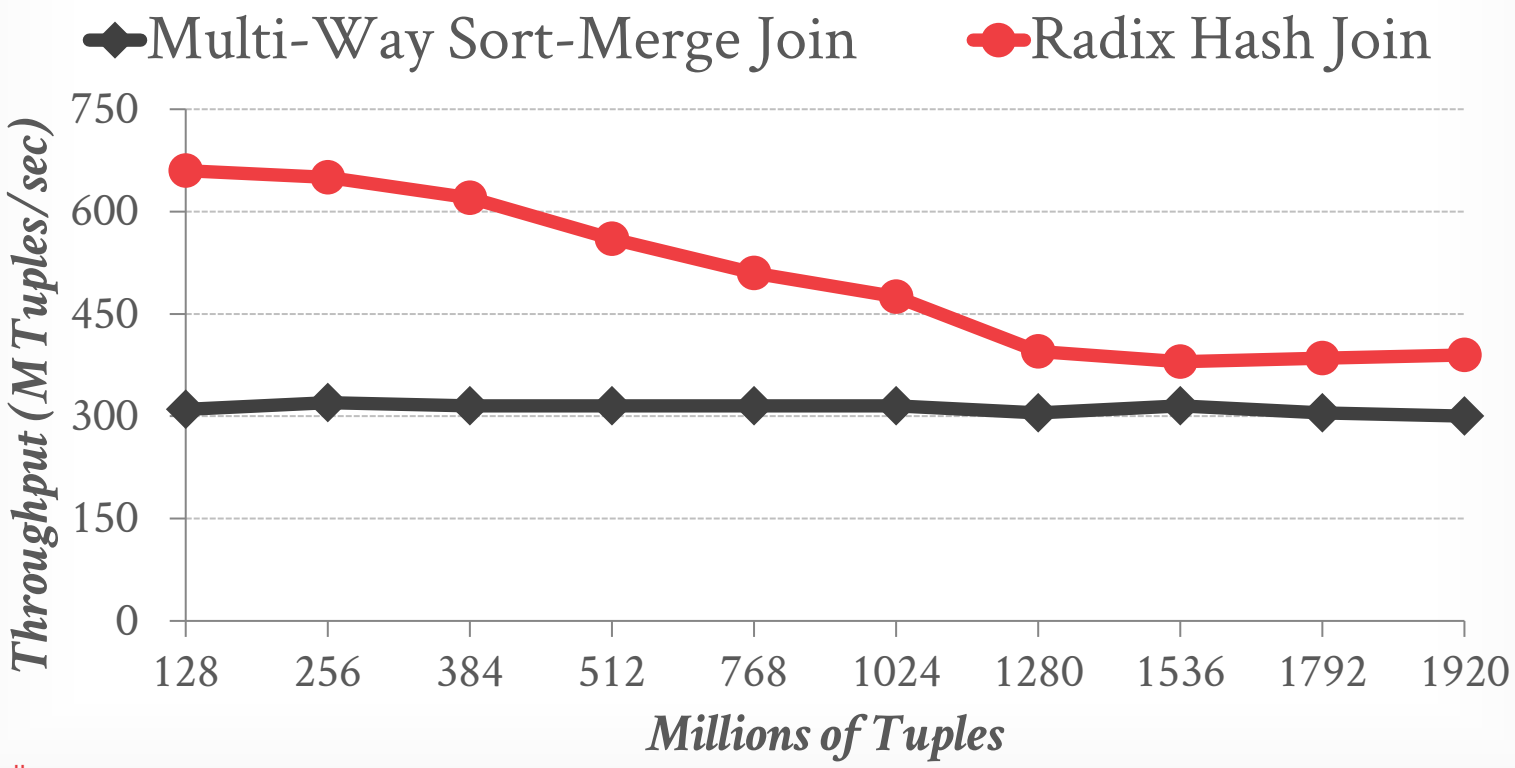
Workload: Different Table Sizes (8-byte tuples)



Source: Cagri Balkesen

SORT-MERGE JOIN VS. HASH JOIN

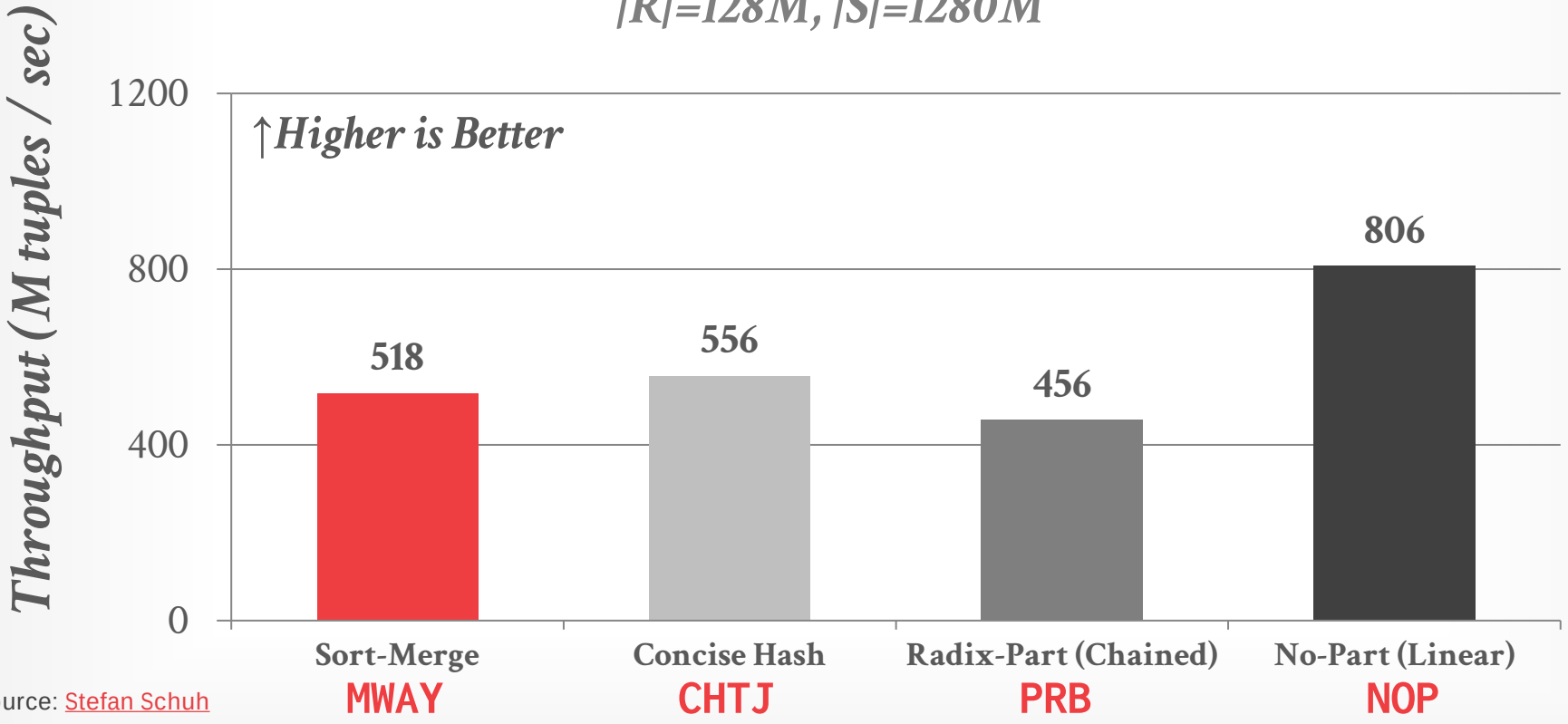
Varying the size of the input relations



Source: Cagri Balkesen

JOIN COMPARISON (R⋈S)

4× Intel Xeon CPU E7-4870v2 (Only 32 cores)
|R|=128M, |S|=1280M



Source: Stefan Schuh

PARTING THOUGHTS

Hash join is (almost) always the superior choice for a join algorithm on modern hardware.
→ Most enterprise OLAP DBMS support both.

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

We will see sort-merge joins again next class…

NEXT CLASS

Worst-Case Optimal Joins (aka multi-way joins)