Lecture #18

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Parallel Join Algorithms (Sorting)

@Andy_Pavlo // 15-721 // Spring 2020

PROJECT #2

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

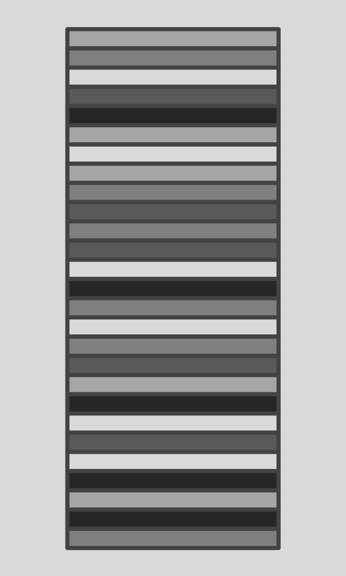
→ Sort the tuples of **R** and **S** based on the join key.

Phase #2: Merge

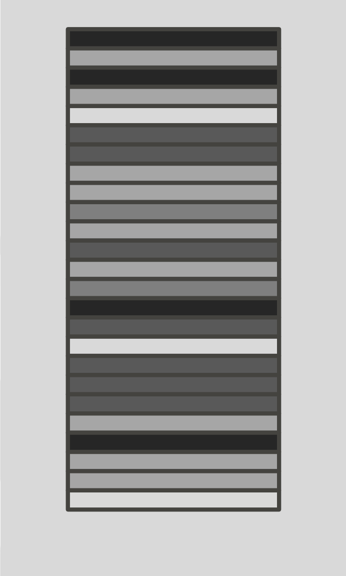
→ Scan the sorted relations and compare tuples.
→ The outer relation **R** only needs to be scanned once.

SORT-MERGE JOIN (R⋈S)

Relation R

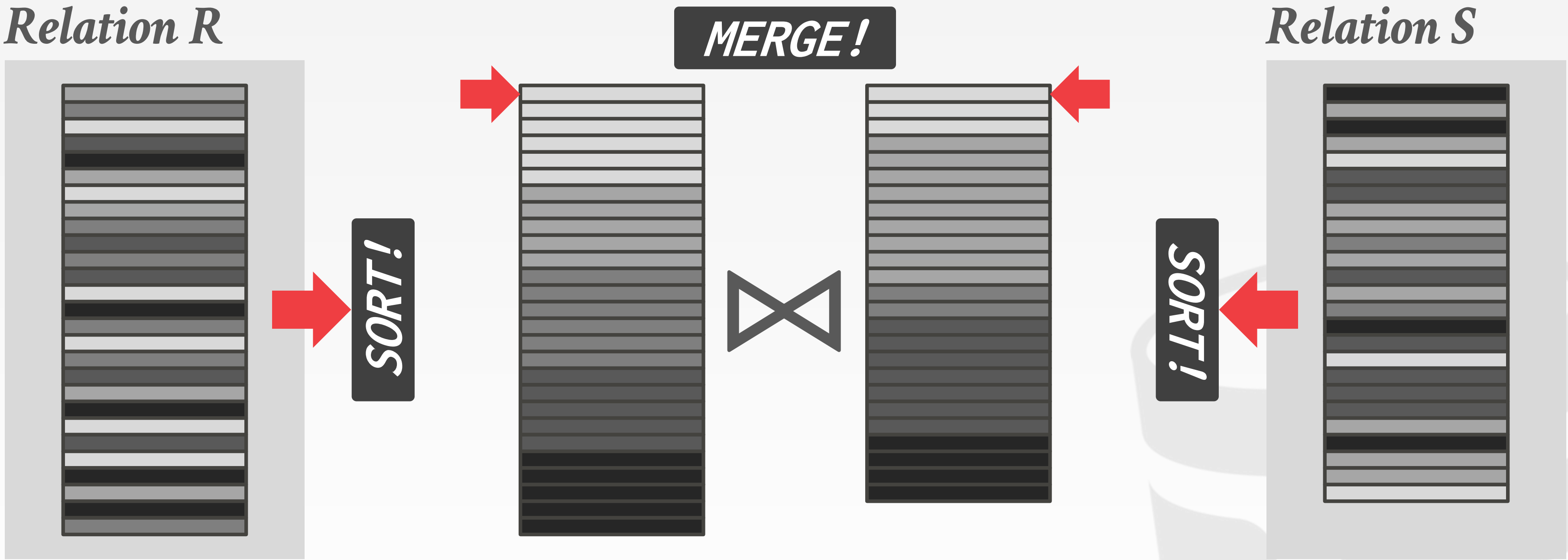


Relation S

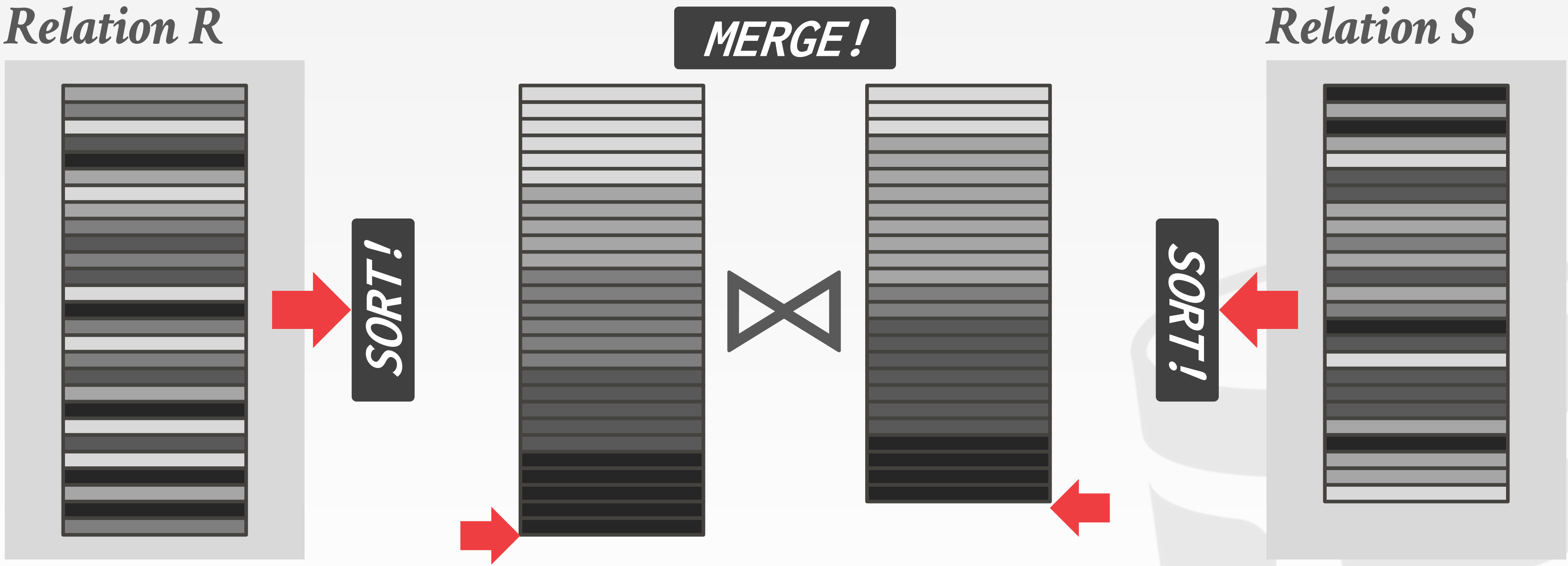


SORT-MERGE JOIN (R⋈S)

SORT-MERGE JOIN (R⋈S)



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.

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

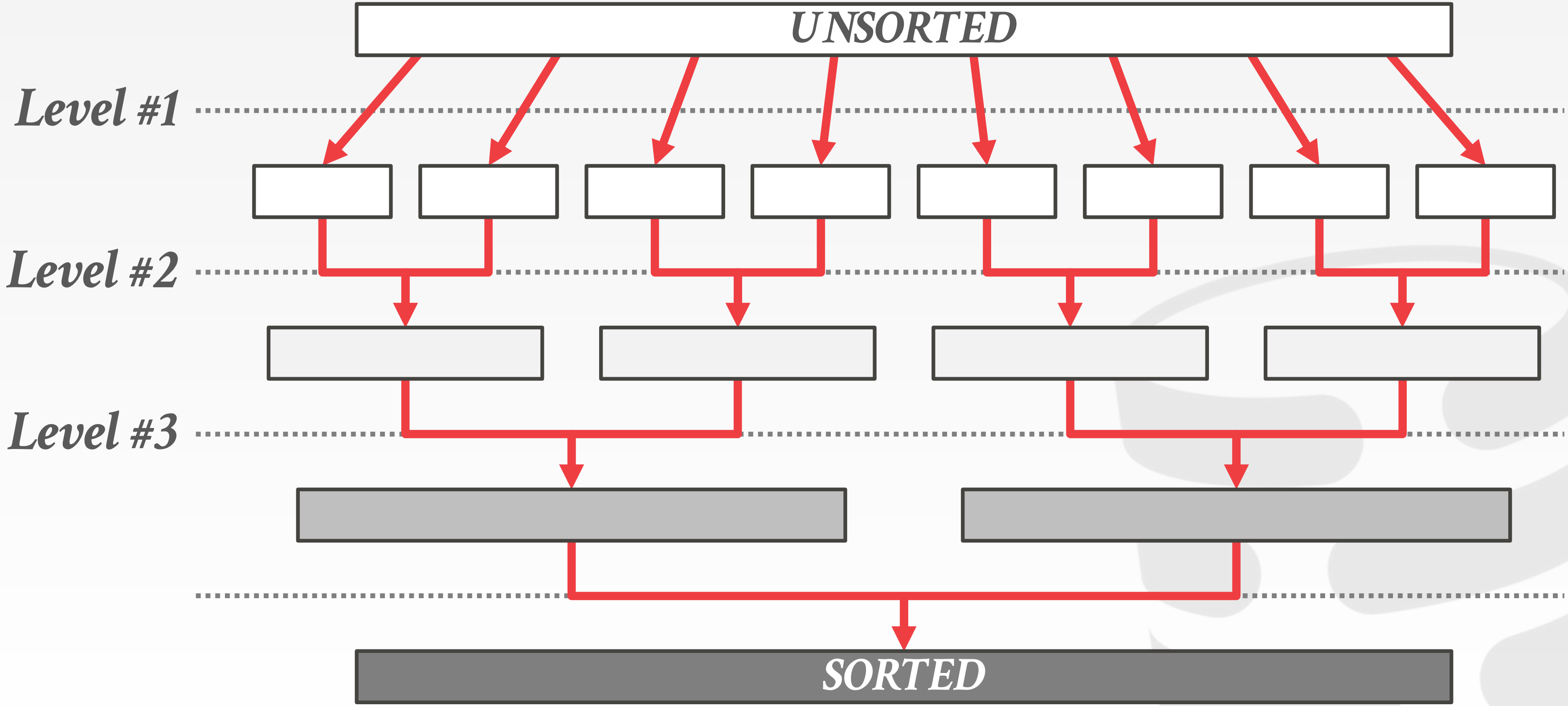
→ Merge Level #1 output into runs that fit into CPU caches.
→ Repeat until sorted runs are ½ cache size.

Level #3: Out-of-Cache Sorting

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

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

CACHE-CONSCIOUS SORTING

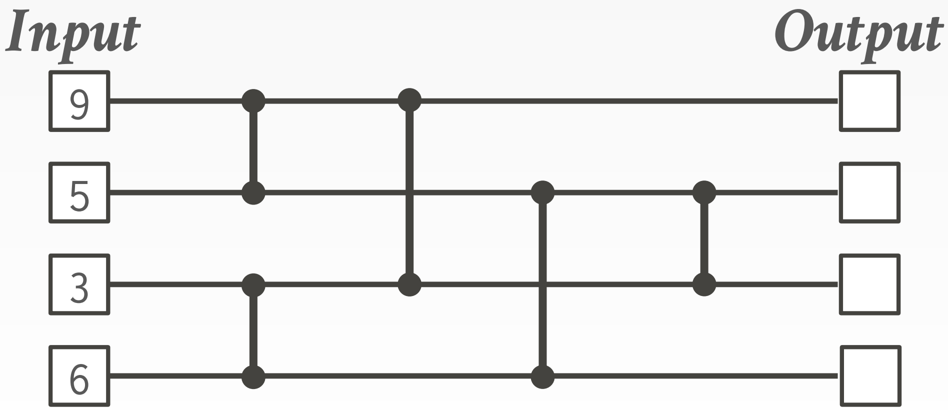


LEVEL #1 – SORTING NETWORKS

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.

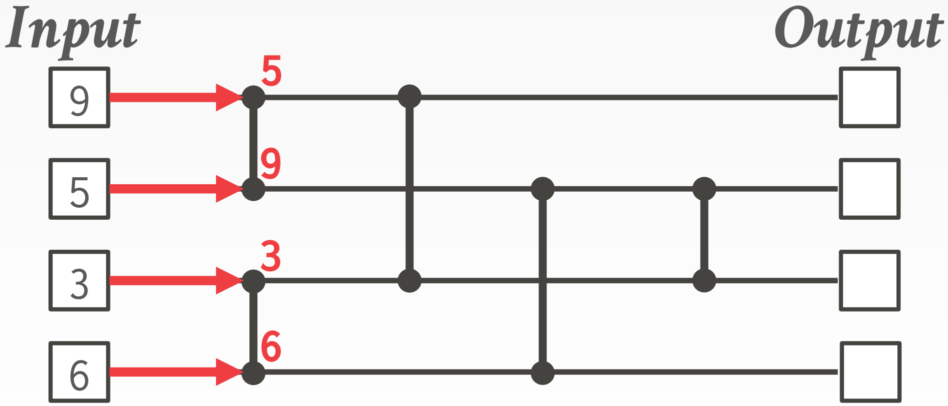


LEVEL #1 – SORTING NETWORKS

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.

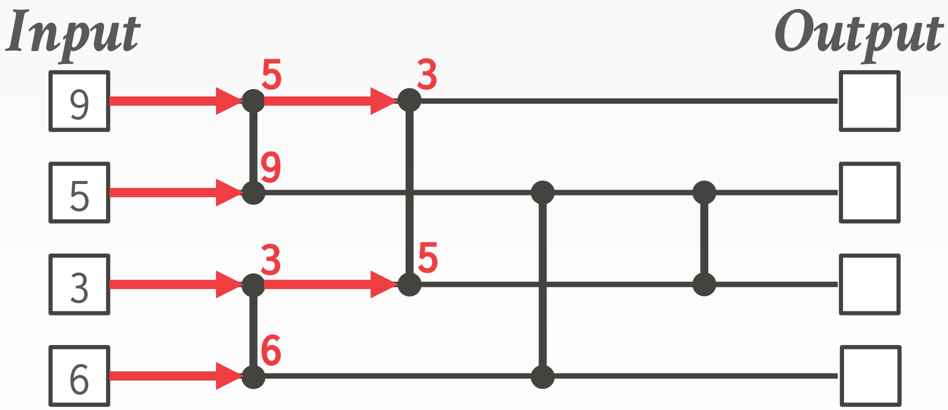


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

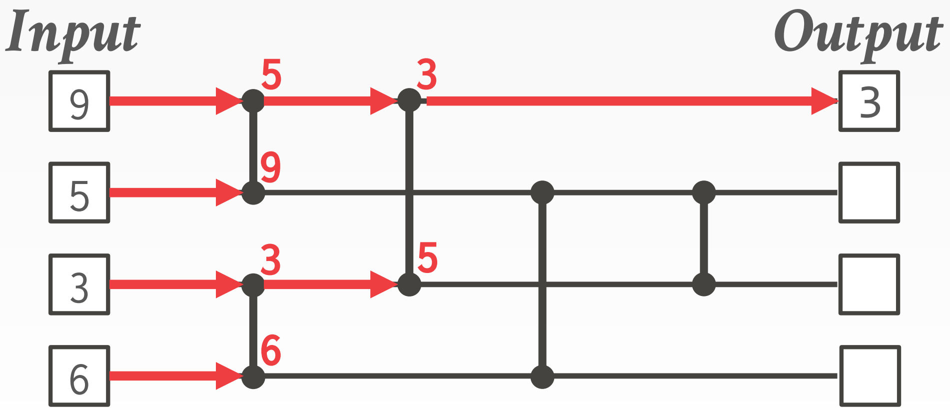
→ Fixed wiring "paths" for lists with the same # of elements.

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



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.

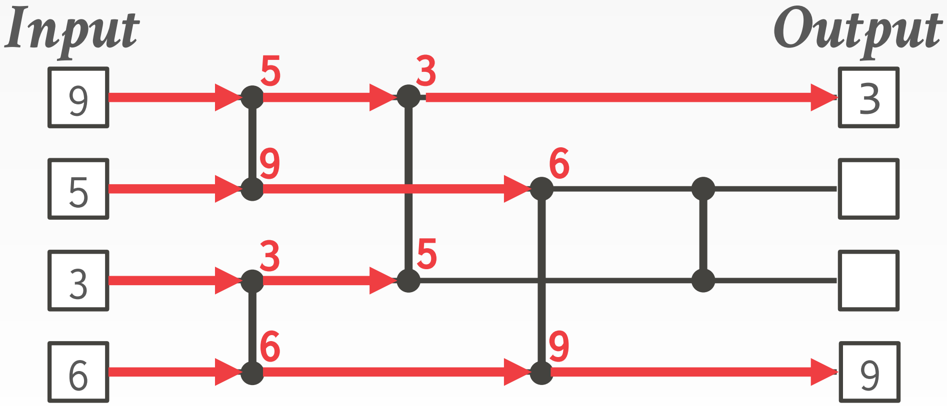


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

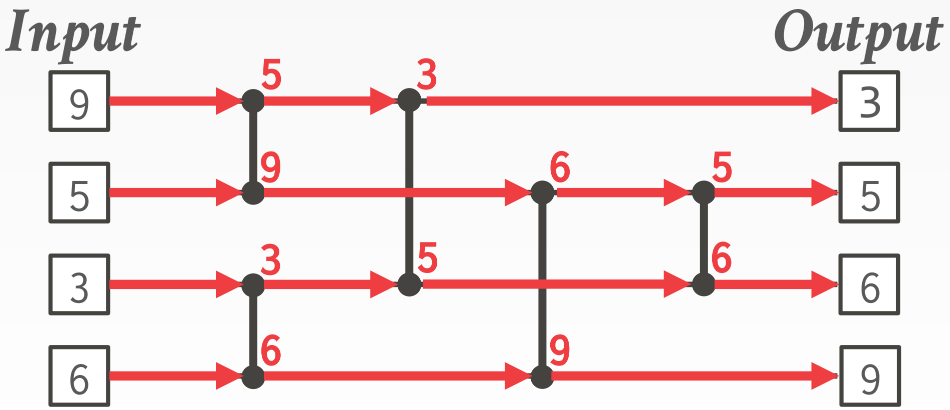
→ Fixed wiring "paths" for lists with the same # of elements.

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



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.

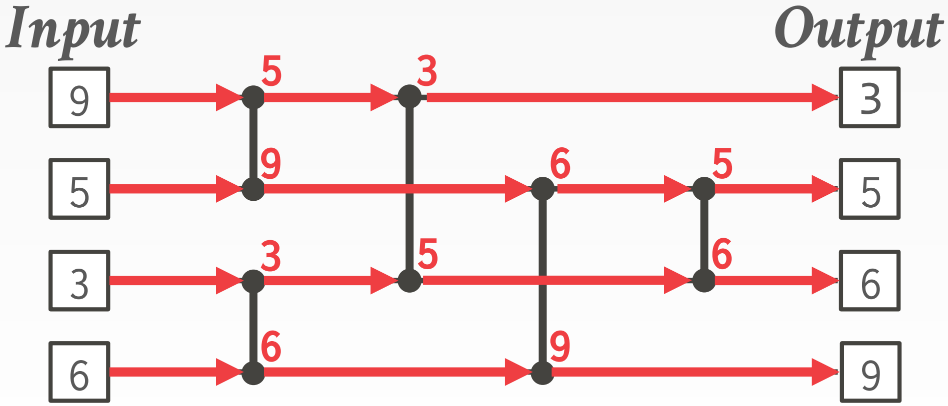


LEVEL #1 – SORTING NETWORKS

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.



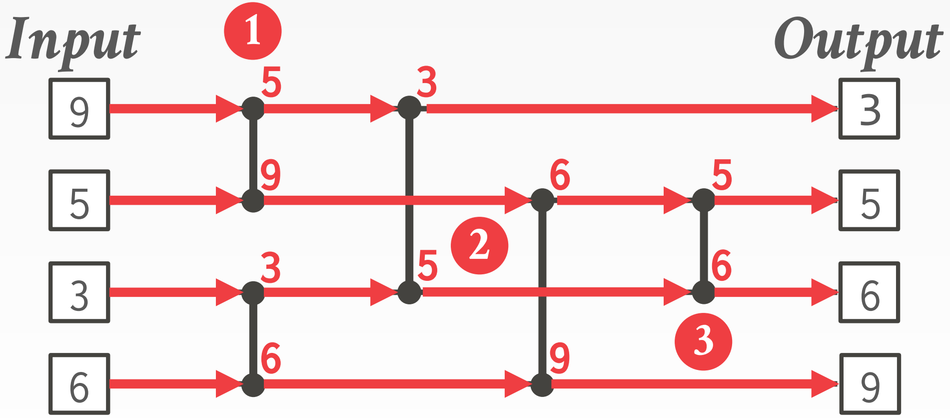

```
wires = [9,5,3,6]

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.

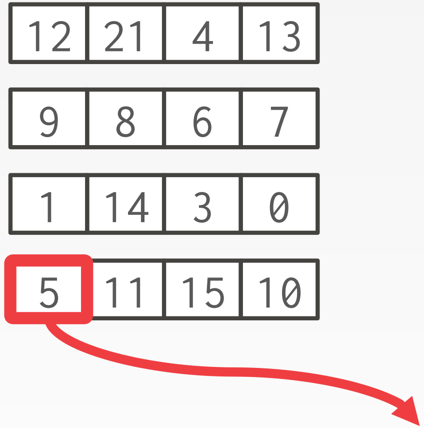
→ Efficient to execute on modern CPUs data dependencies and no branches.




```
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])
```

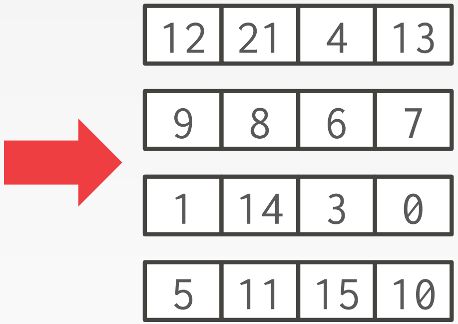
1 2 3

LEVEL #1 – SORTING NETWORKS



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

LEVEL #1 – SORTING NETWORKS

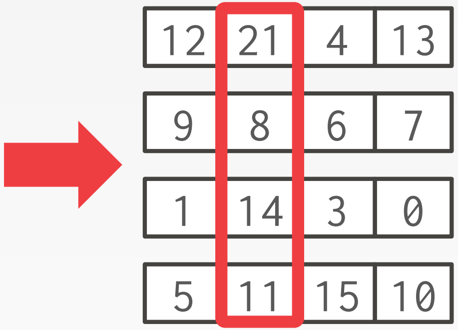


Instructions:

→ **4 LOAD**

LEVEL #1 – SORTING NETWORKS

Sort Across Registers



Instructions:

→ **4 LOAD**

LEVEL #1 – SORTING NETWORKS

Sort Across Registers

12	21	4	13
9	8	6	7
1	14	3	0
5	11	15	10

Instructions:

→ **4 LOAD**

1	8	3	0
5	11	4	7
9	14	6	10
12	21	15	13

Instructions:

→ **10 MIN/MAX**

LEVEL #1 – SORTING NETWORKS

12	21	4	13
9	8	6	7
1	14	3	0
5	11	15	10

Instructions:
→ **4 LOAD**

Sort Across Registers

1	8	3	0
5	11	4	7
9	14	6	10
12	21	15	13

Instructions:
→ **10 MIN/MAX**

Transpose Registers

1	5	9	12
8	11	14	21
3	4	6	15
0	7	10	13

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

LEVEL #2 – BITONIC MERGE NETWORK

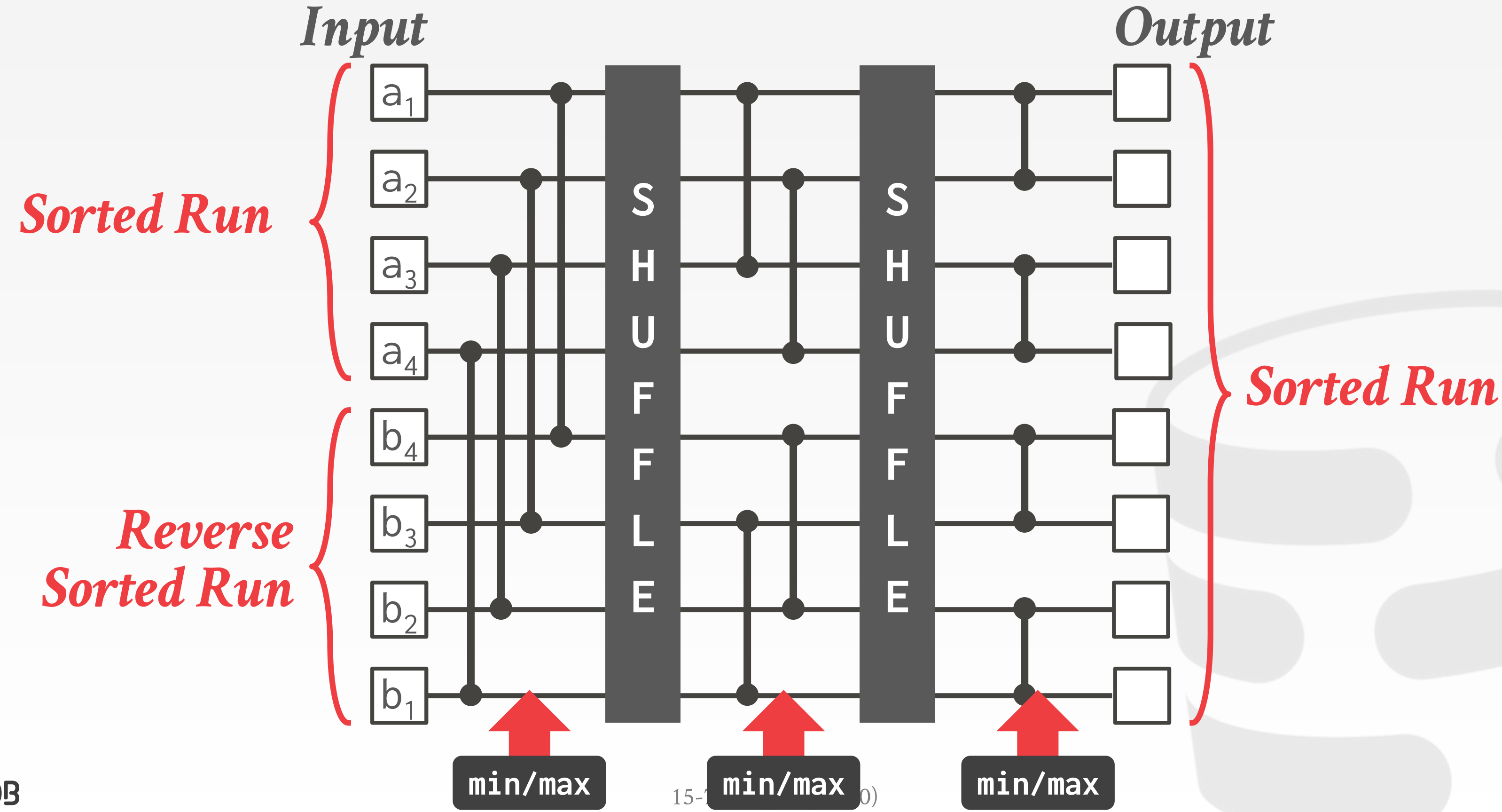
Like a Sorting Network but it can merge two locally-sorted lists into a globally-sorted list.

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.

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.

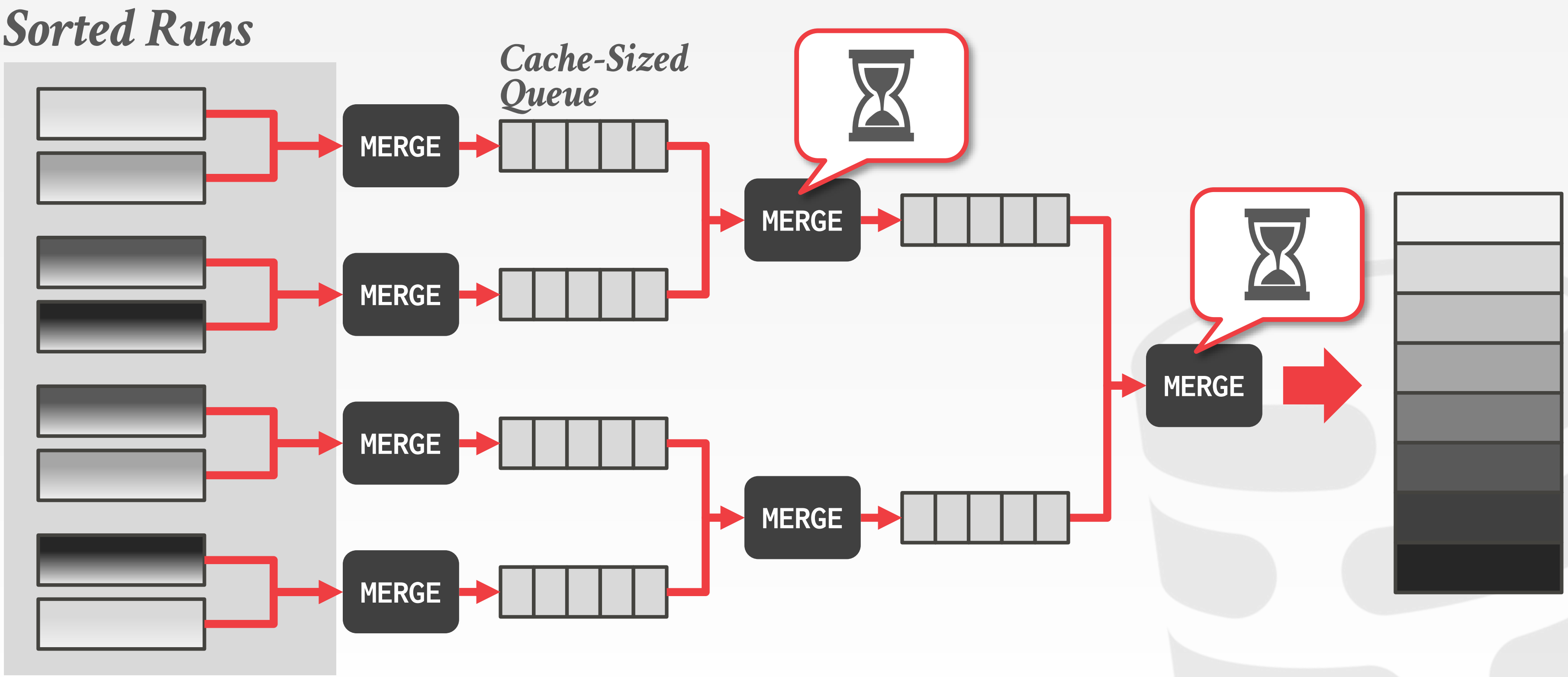
→ Still one worker thread per core.

→ 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

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.

IN-PLACE PARALLEL SUPER SCALAR SAMPLESORT
ESA 2017

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

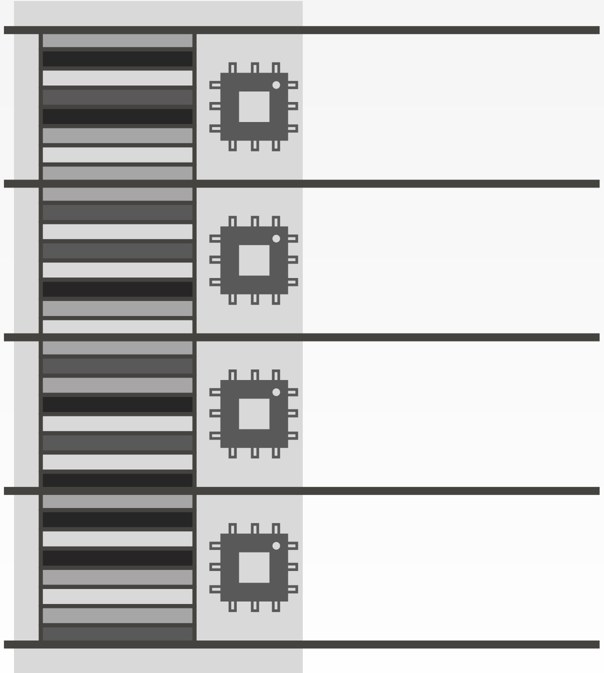
→ Same as outer table.

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

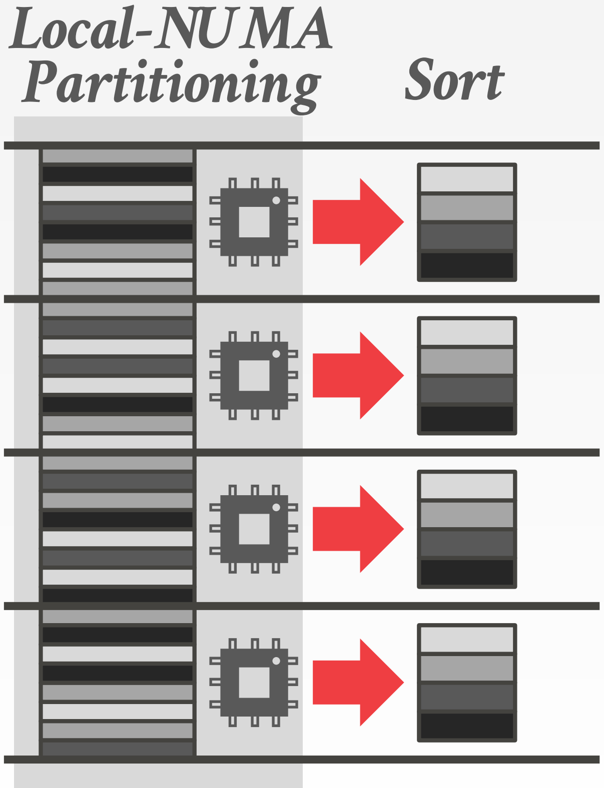
MULTI-CORE, MAIN-MEMORY JOINS:
SORT VS. HASH REVISITED
VLDB 2013

MULTI-WAY SORT-MERGE

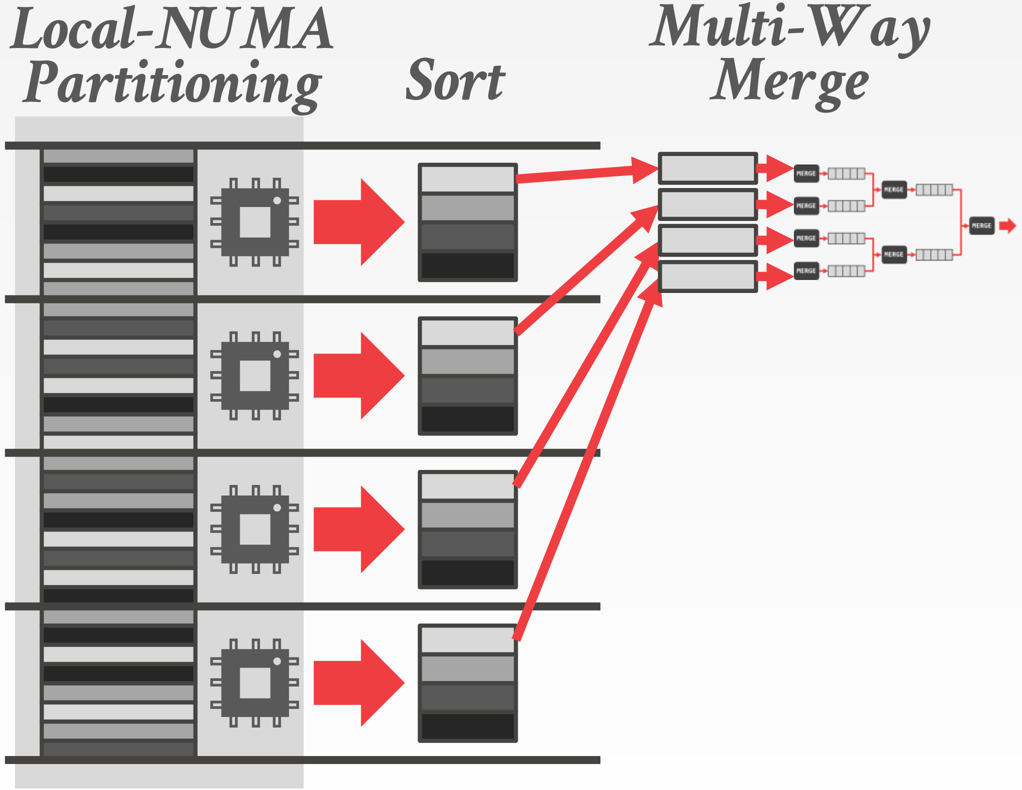
Local-NUMA Partitioning



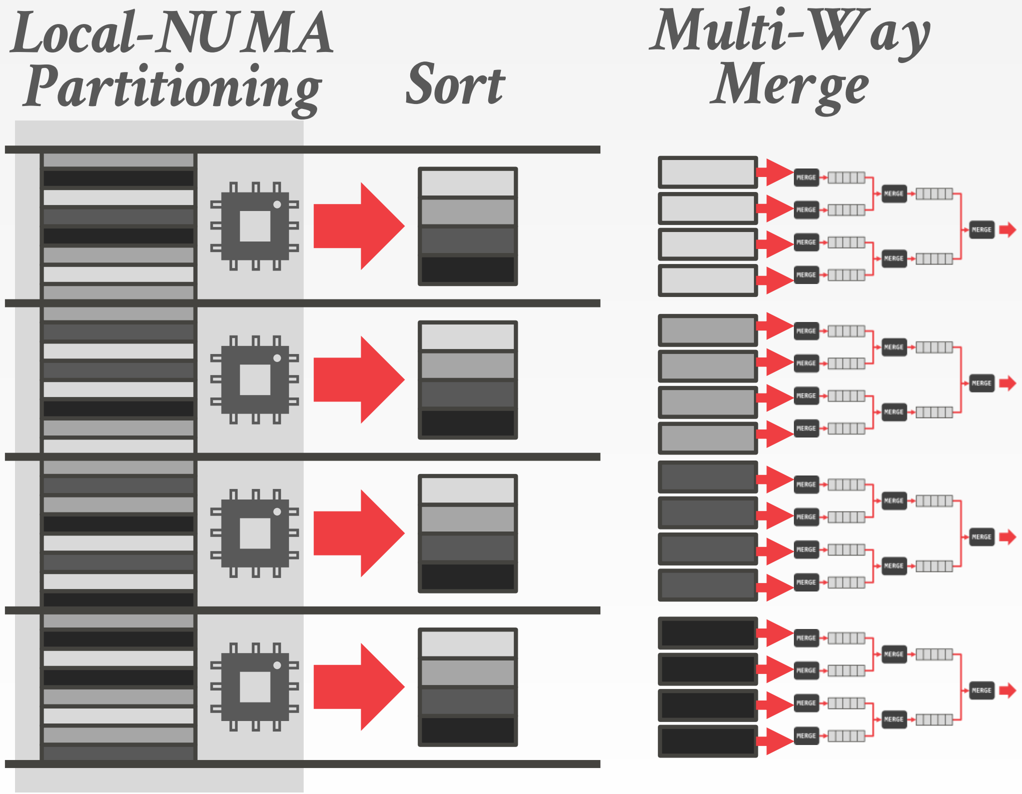
MULTI-WAY SORT-MERGE



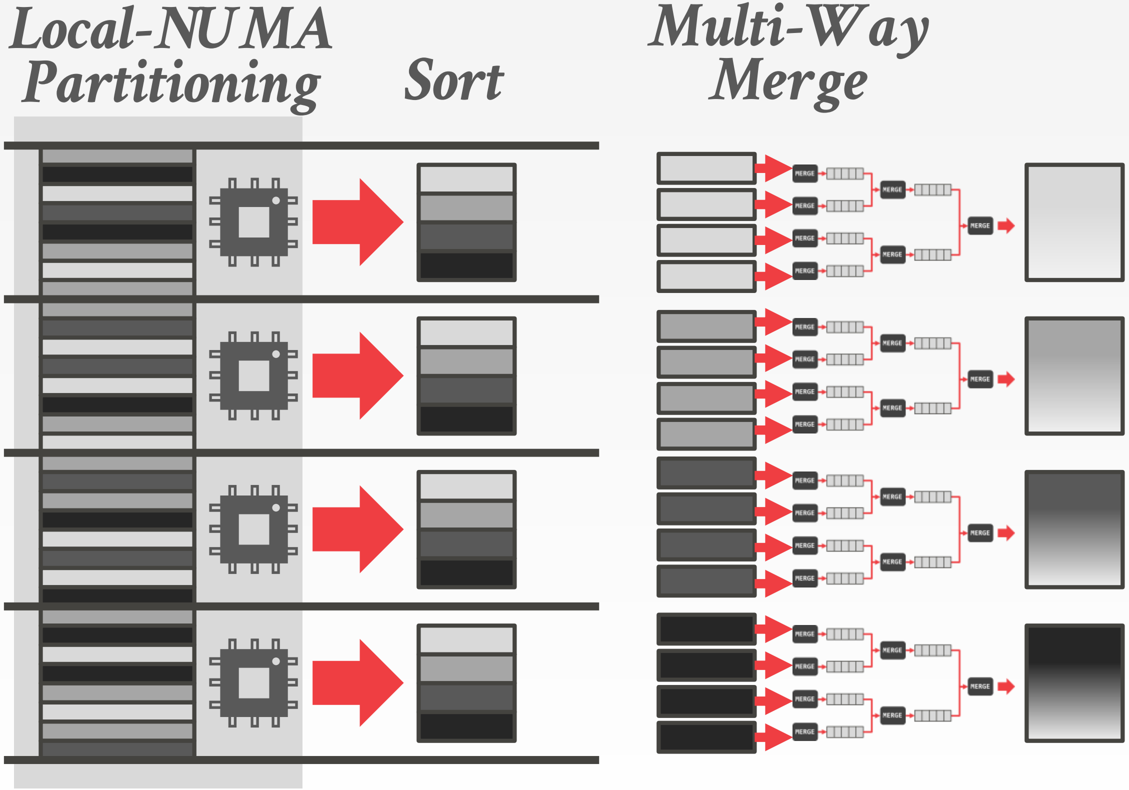
MULTI-WAY SORT-MERGE



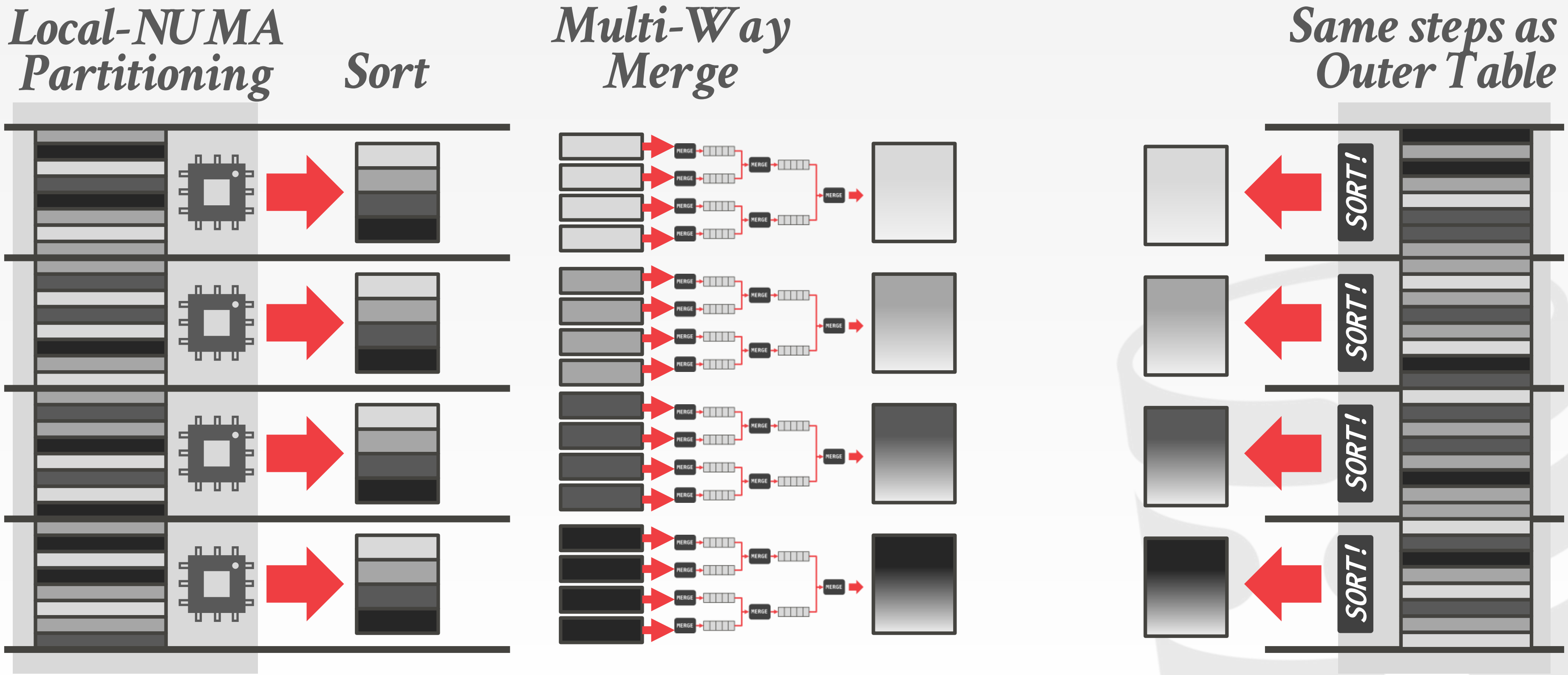
MULTI-WAY SORT-MERGE



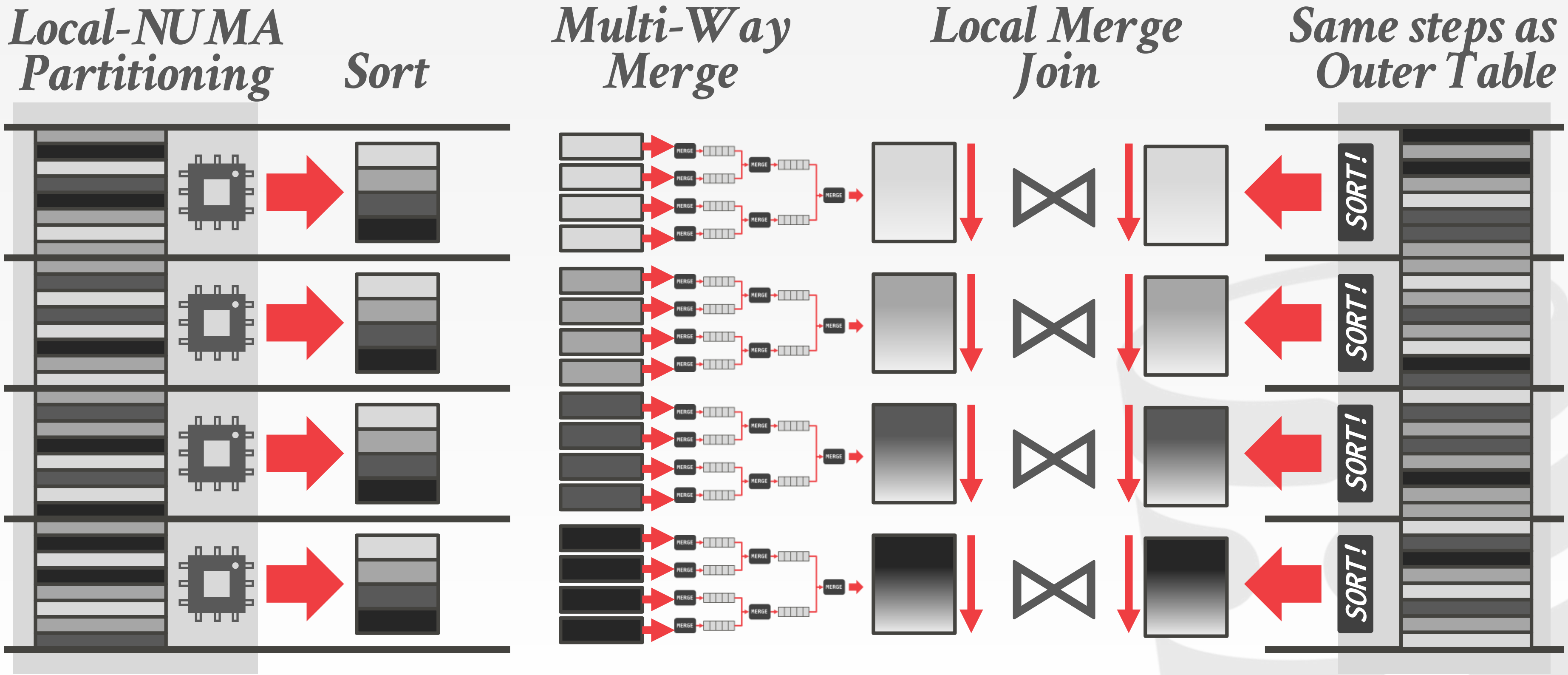
MULTI-WAY SORT-MERGE



MULTI-WAY SORT-MERGE



MULTI-WAY SORT-MERGE



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.

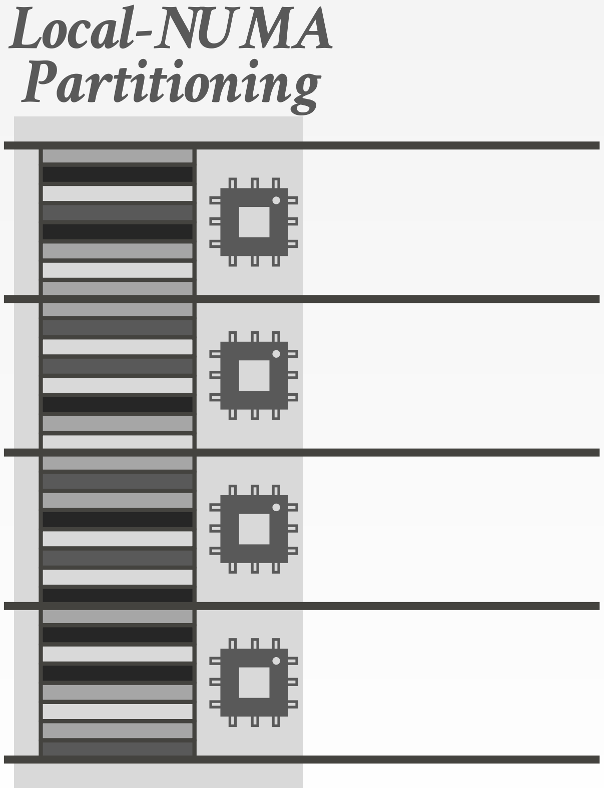
Inner Table

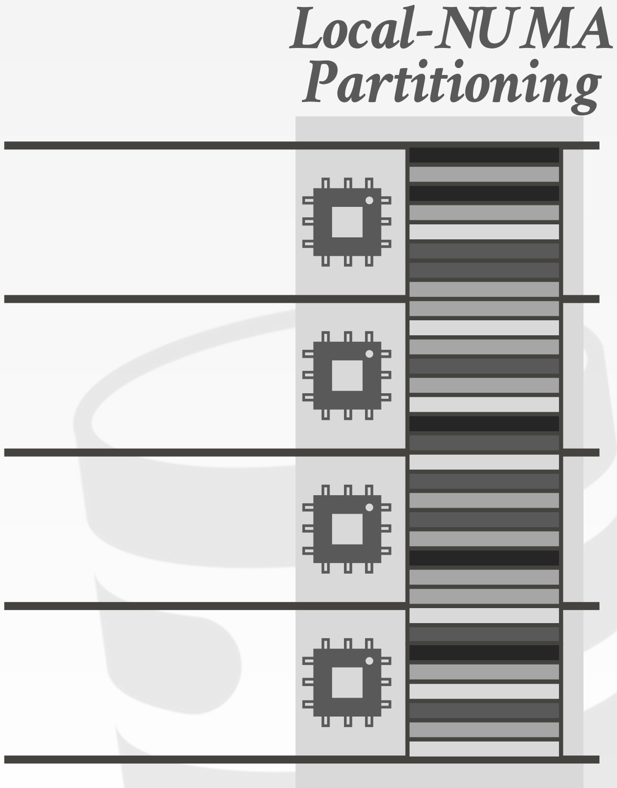
→ Same as outer table.

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

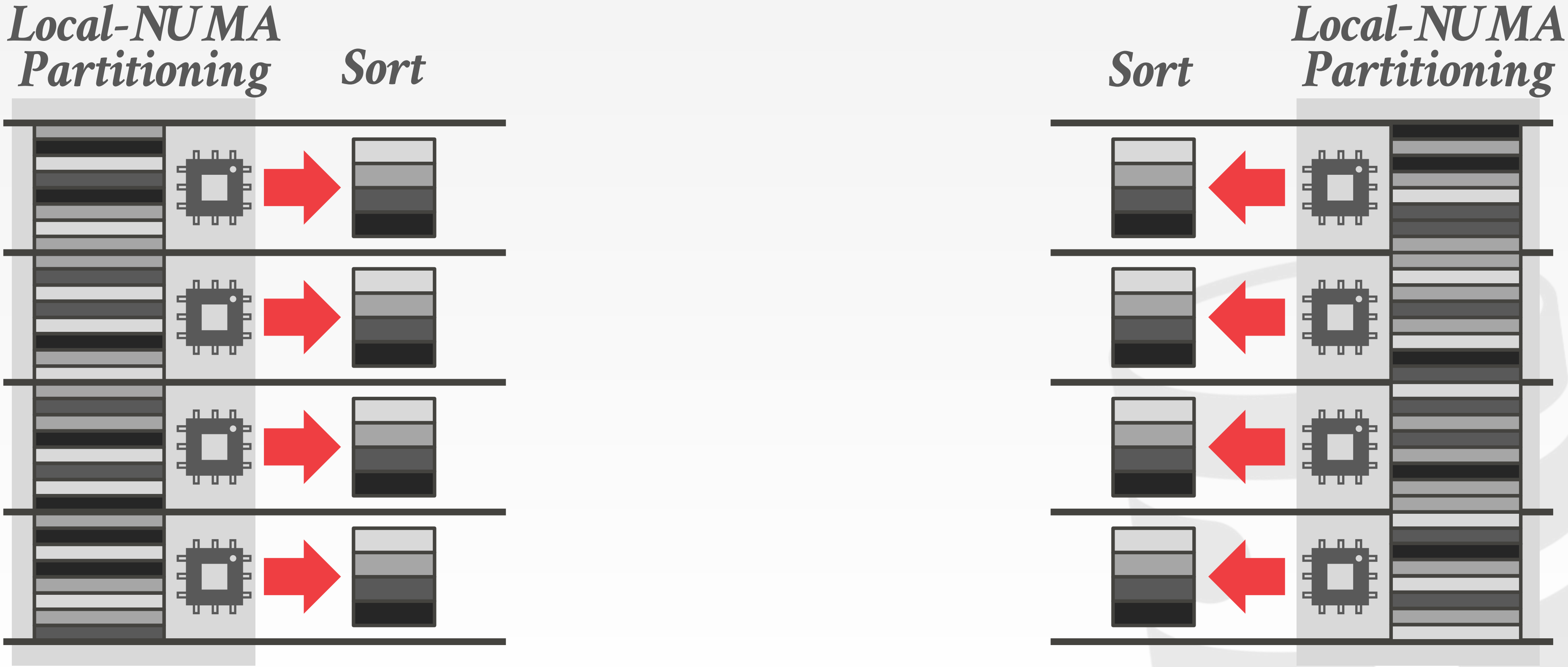
MULTI-CORE, MAIN-MEMORY JOINS: SORT VS. HASH REVISITED
VLDB 2013

MULTI-PASS SORT-MERGE

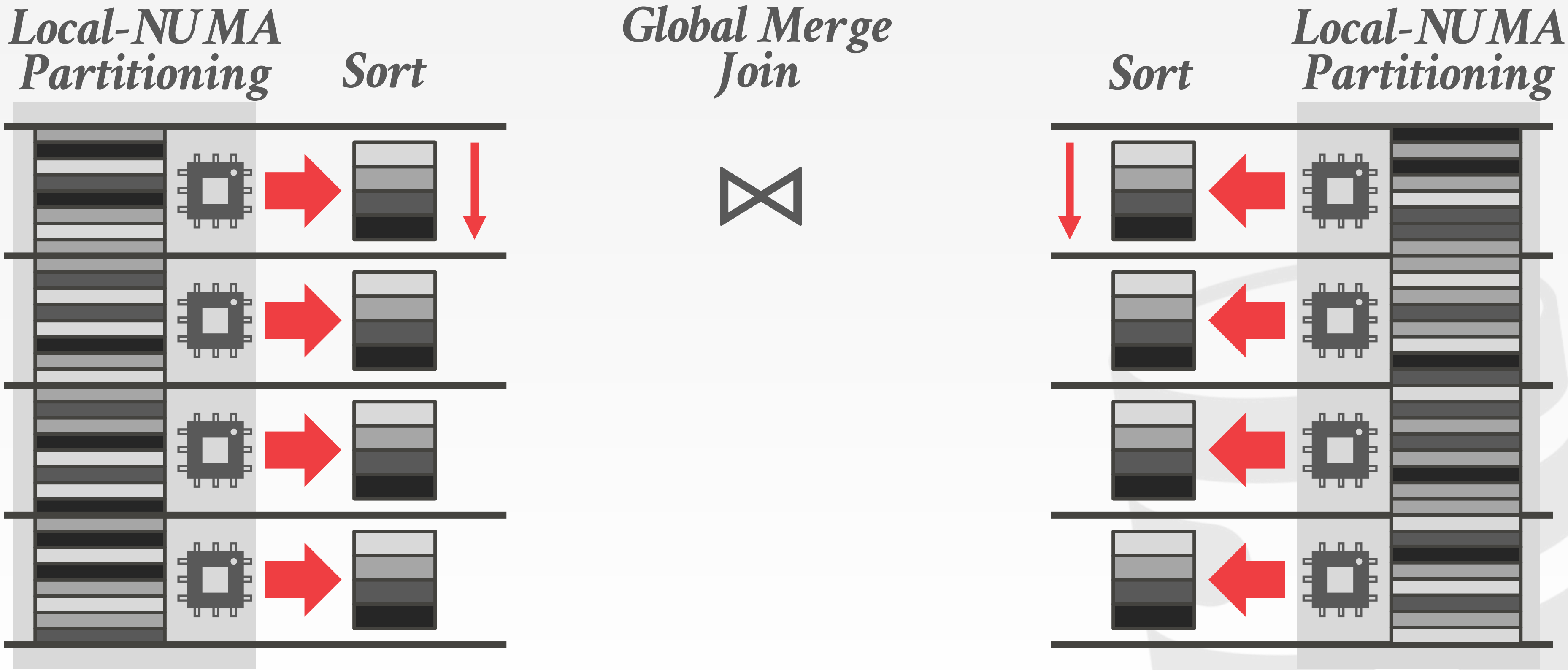




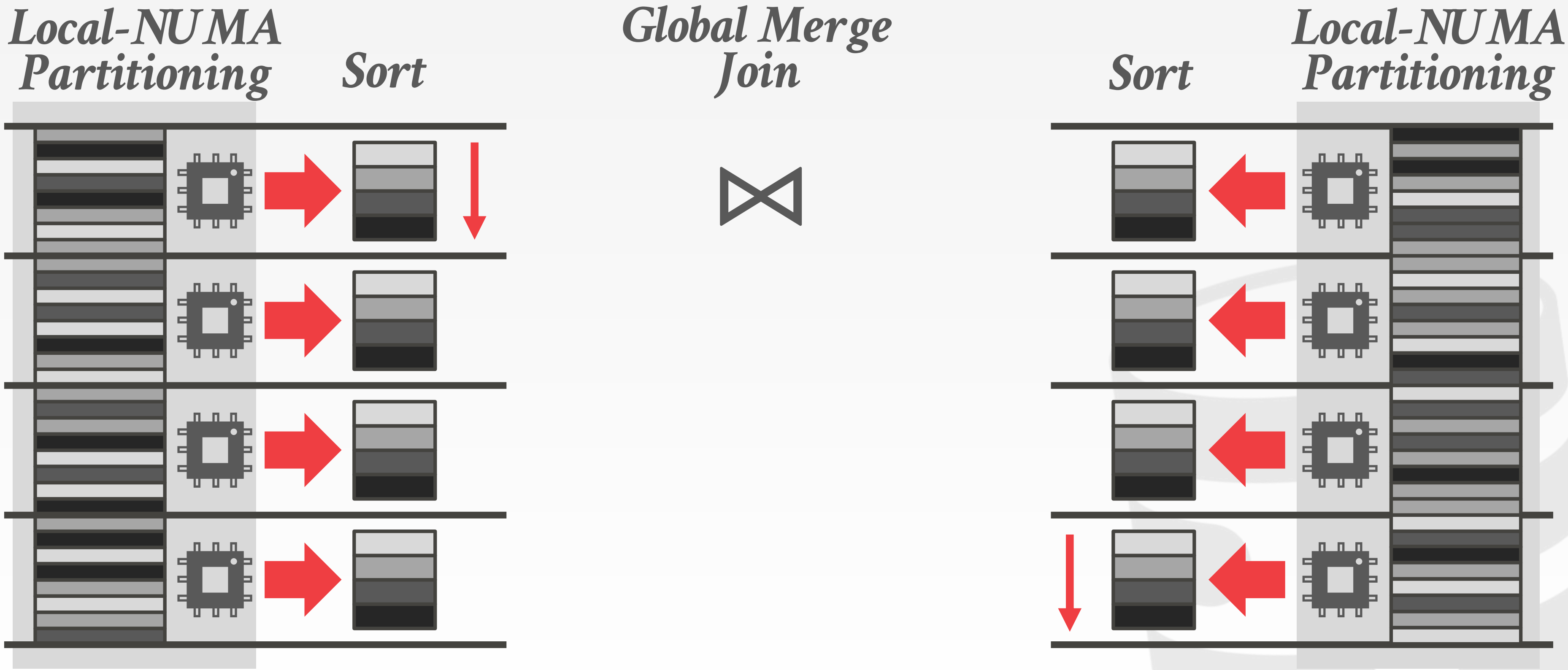
MULTI-PASS SORT-MERGE



MULTI-PASS SORT-MERGE



MULTI-PASS SORT-MERGE



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.

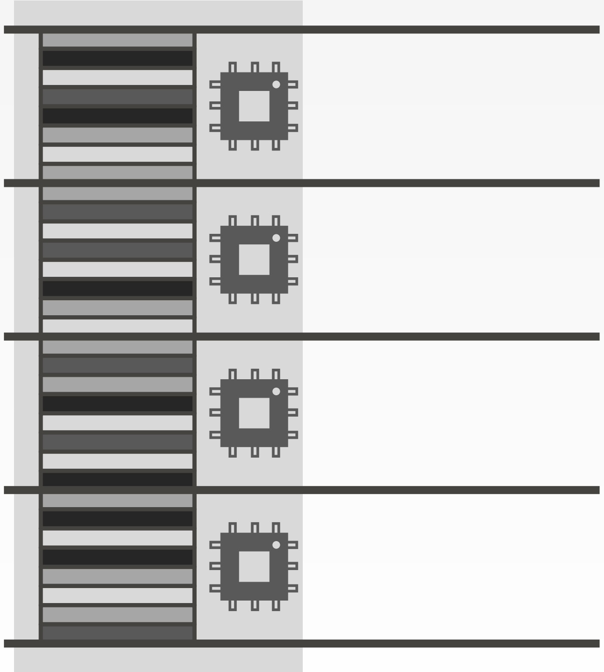
→ Each core sorts its local data.

Merge phase is between entire sorted run of outer table and a segment of inner table.

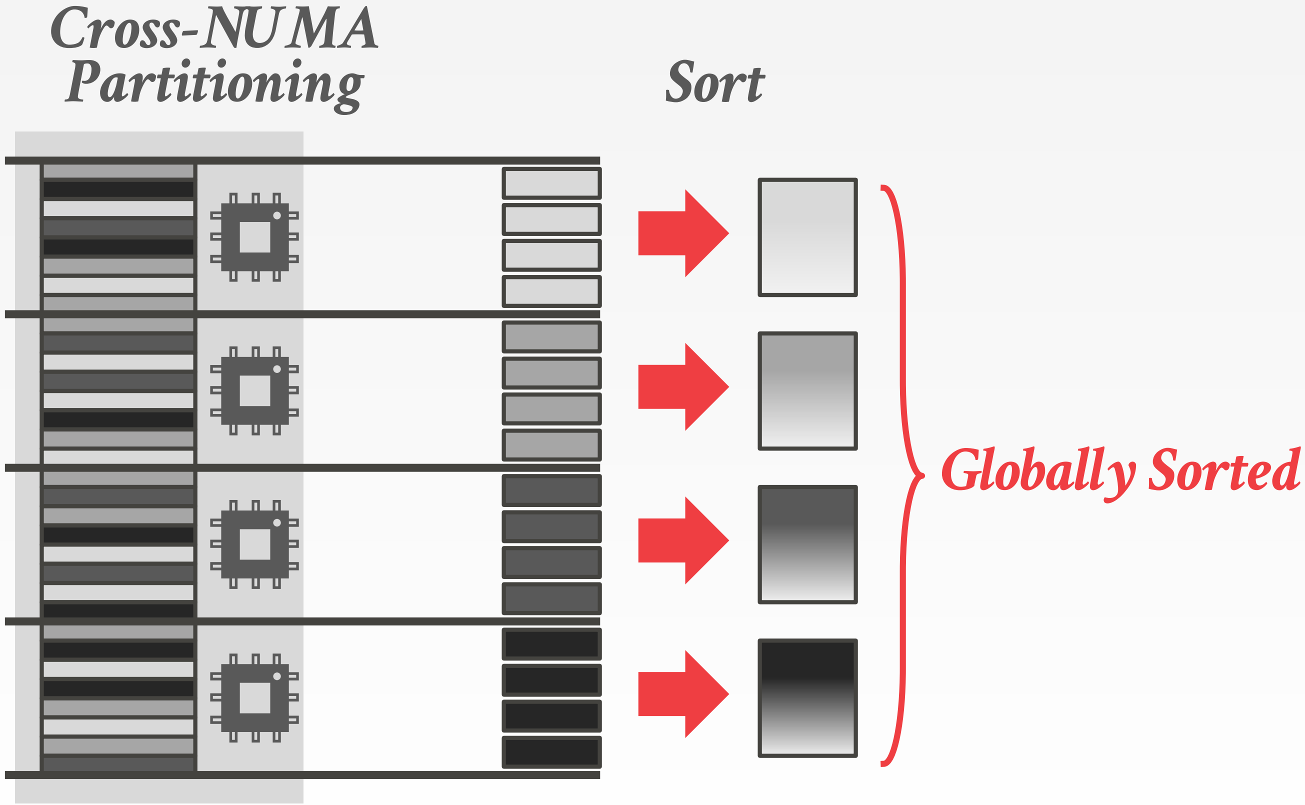
MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012

MASSIVELY PARALLEL SORT-MERGE

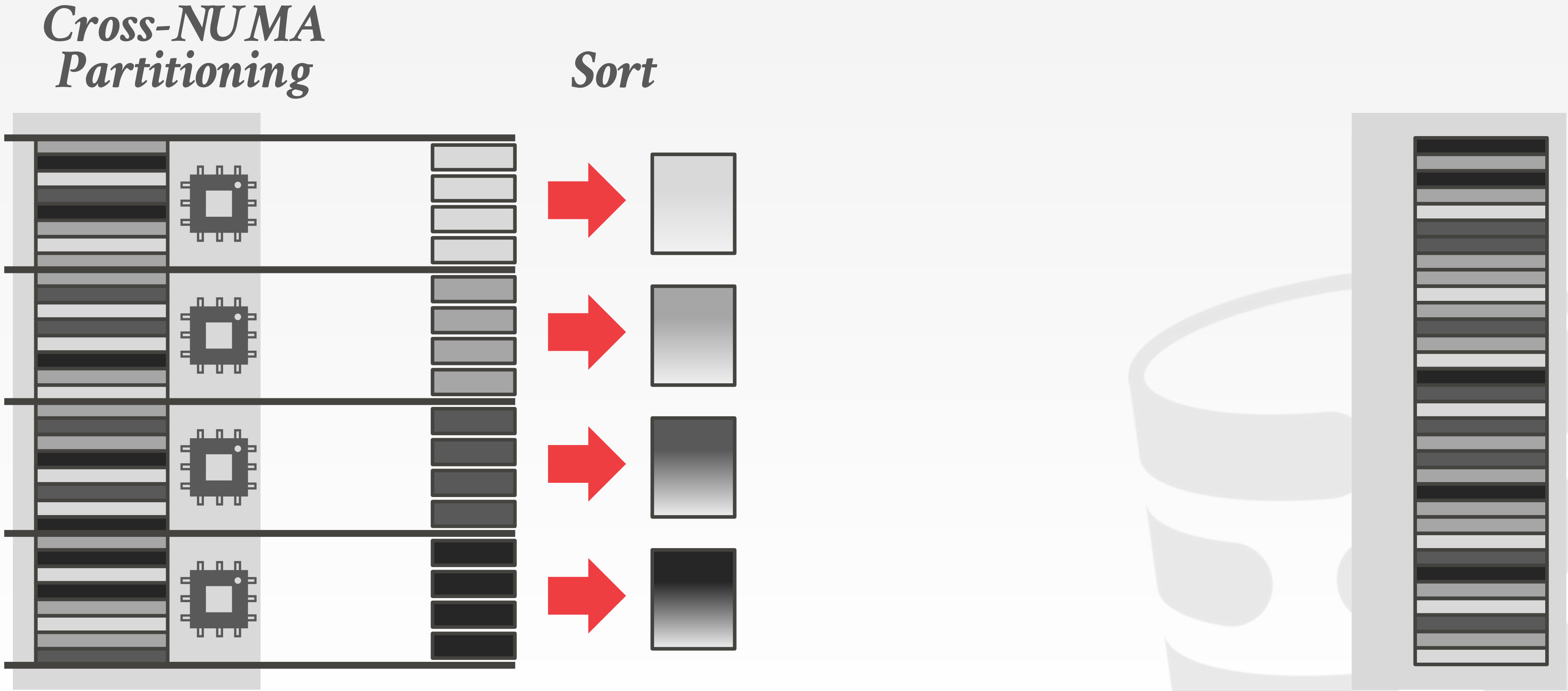
Cross-NUMA Partitioning



MASSIVELY PARALLEL SORT-MERGE



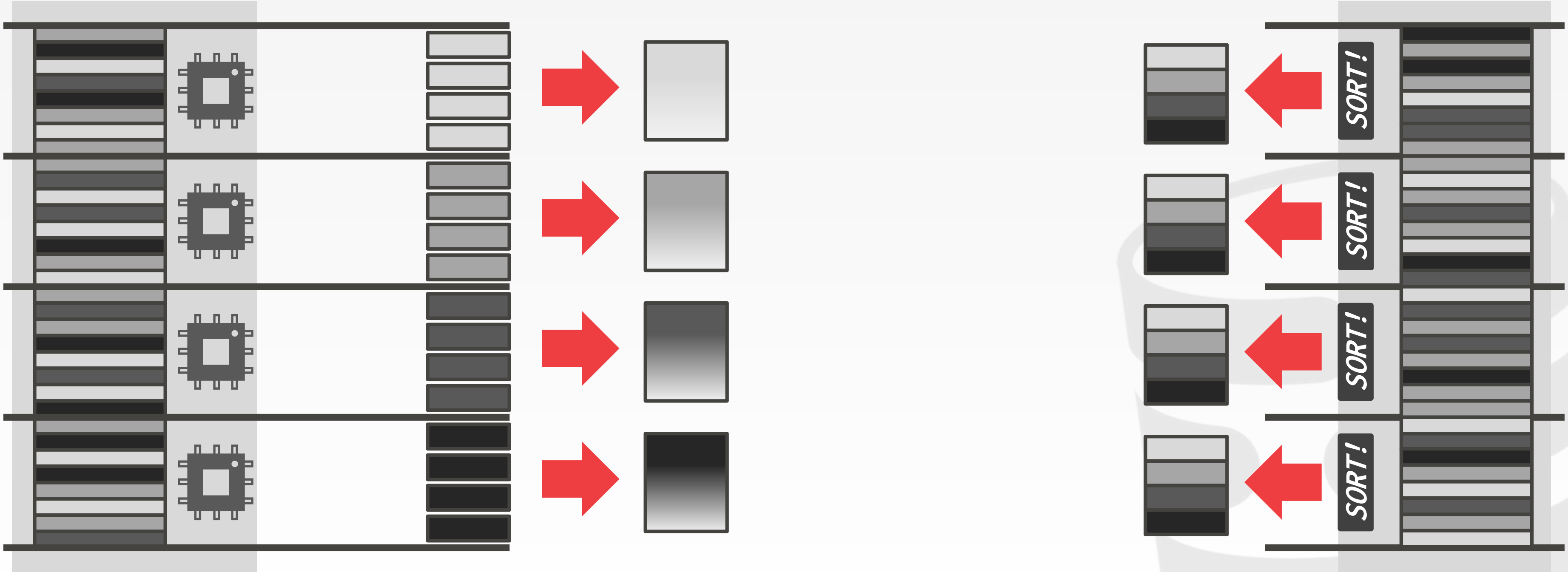
MASSIVELY PARALLEL SORT-MERGE



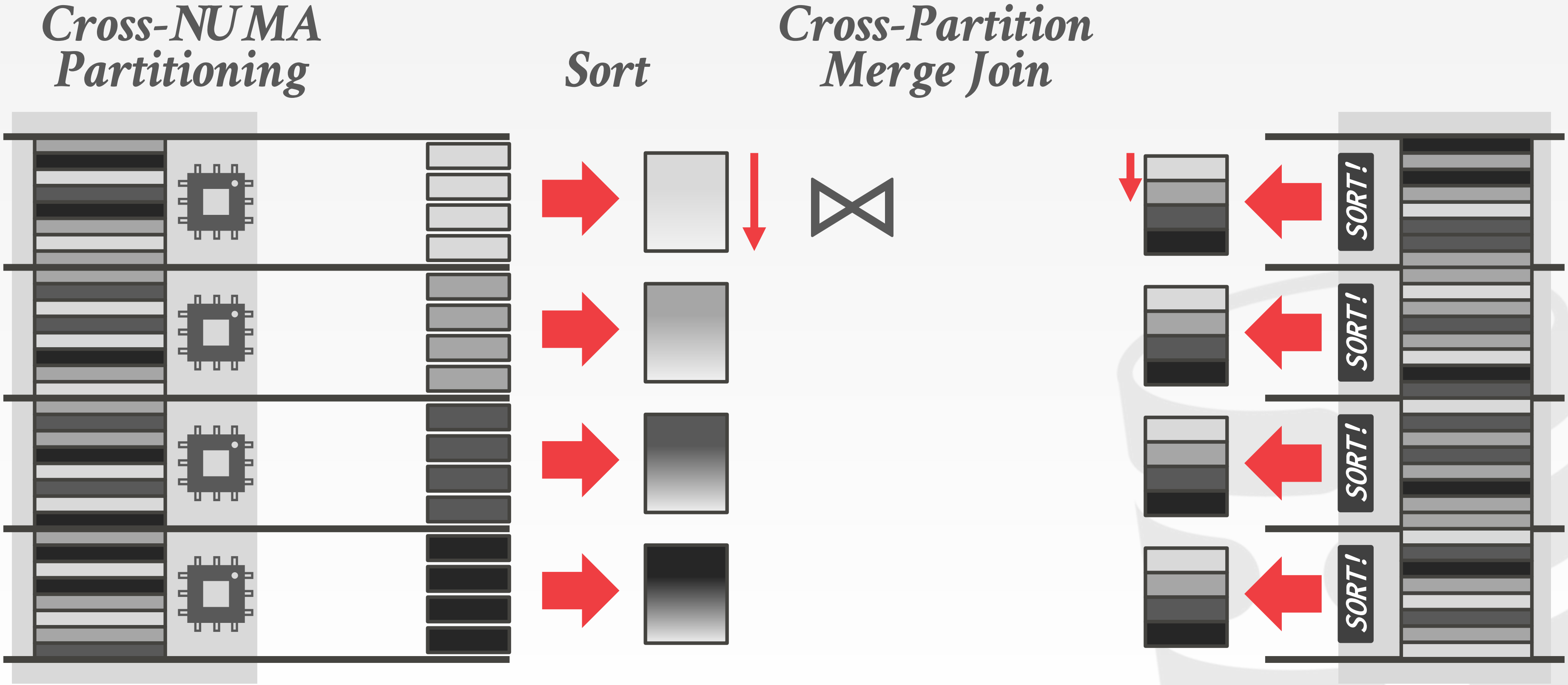
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

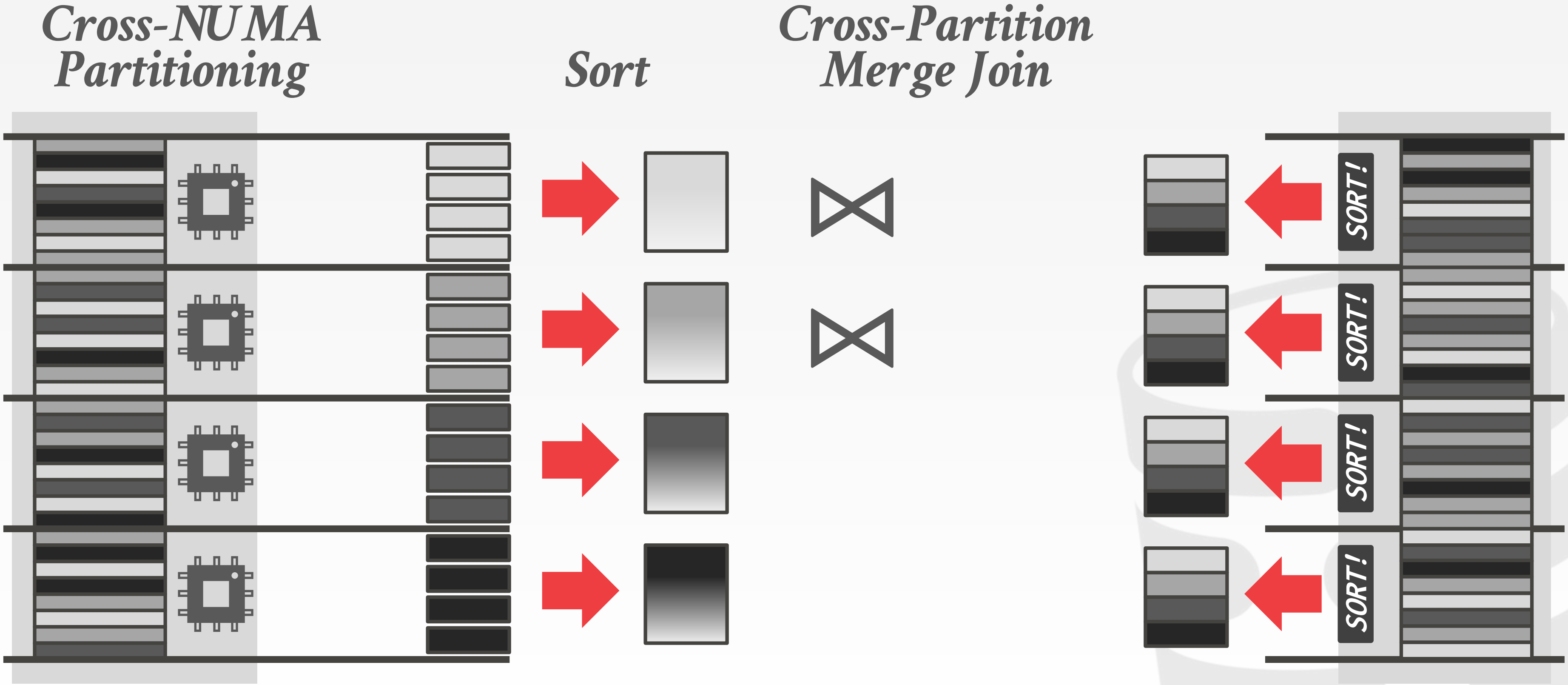
Sort



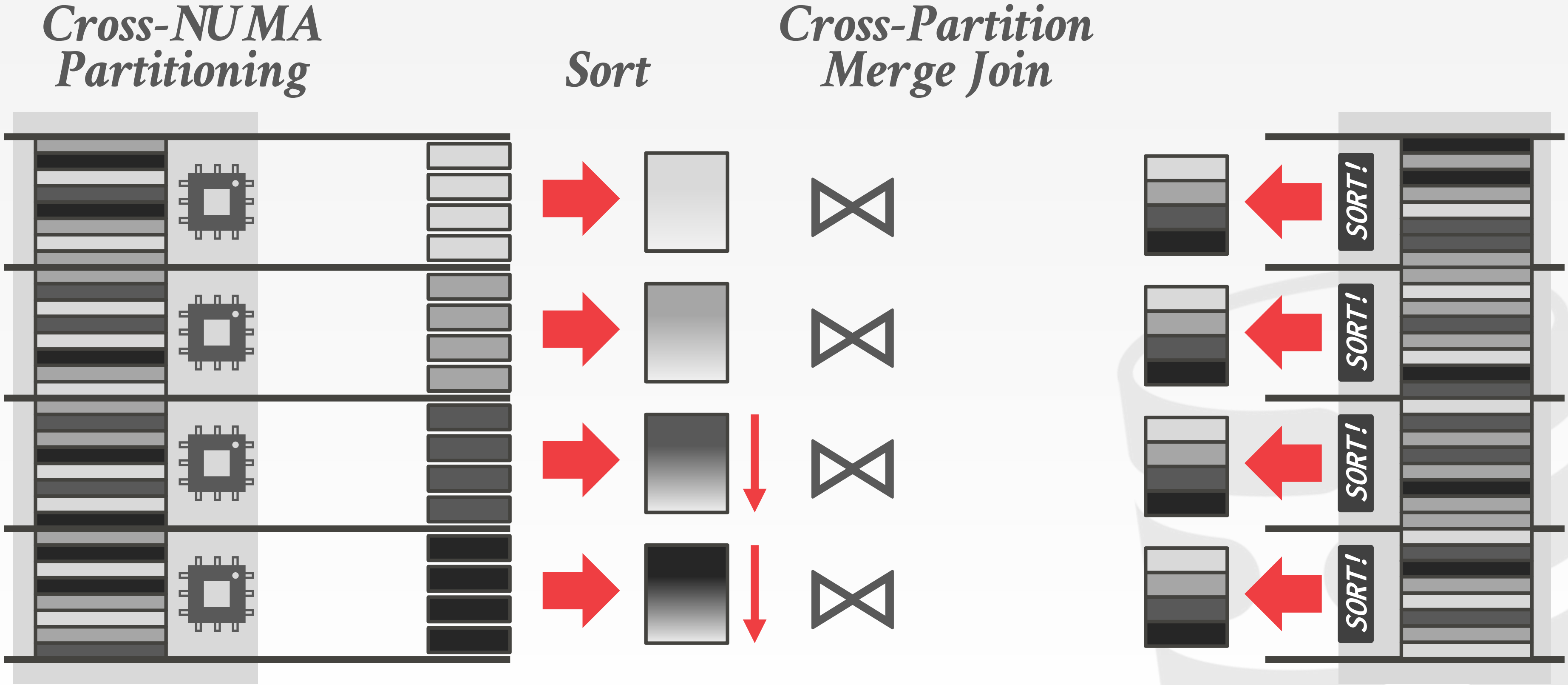
MASSIVELY PARALLEL SORT-MERGE



MASSIVELY PARALLEL SORT-MERGE



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

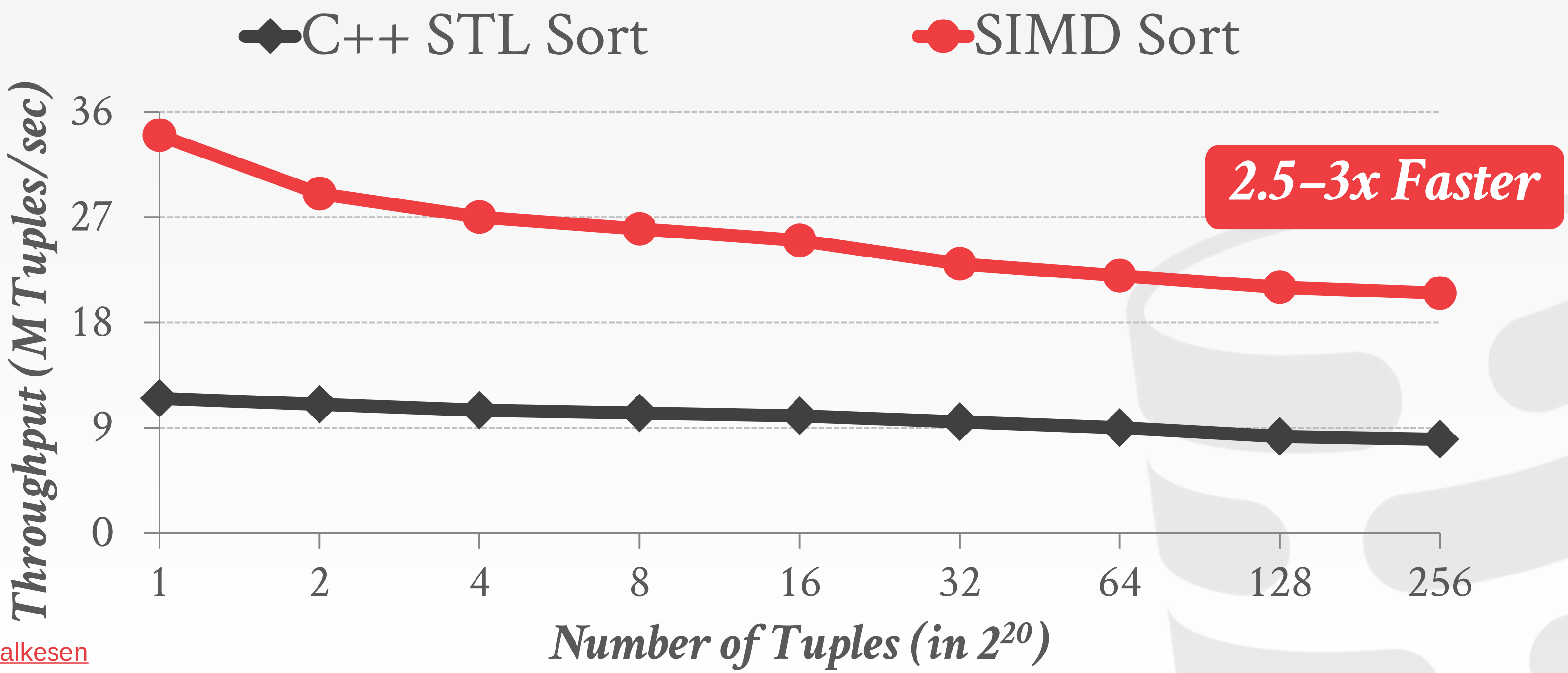
Hardware:

→ 4 Socket Intel Xeon E4640 @ 2.4GHz
→ 8 Cores with 2 Threads Per Core
→ 512 GB of DRAM

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

RAW SORTING PERFORMANCE

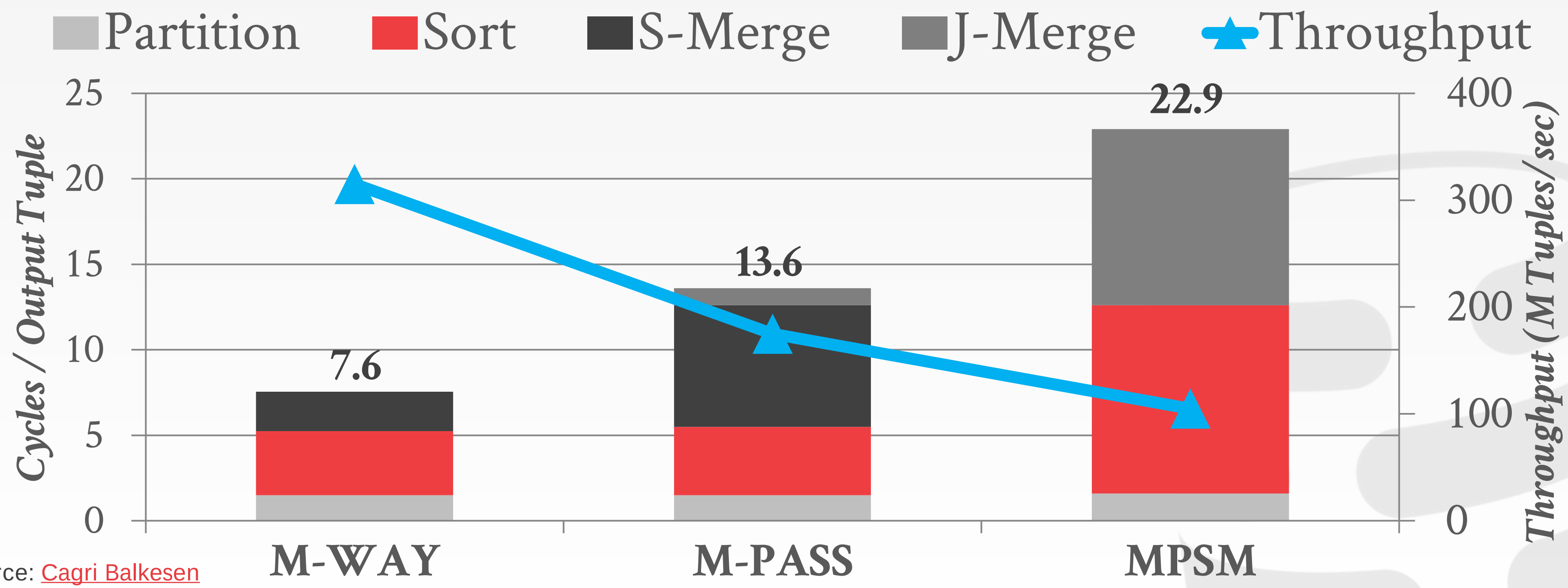
Single-threaded sorting performance



Source: Cagri Balkesen

COMPARISON OF SORT-MERGE JOINS

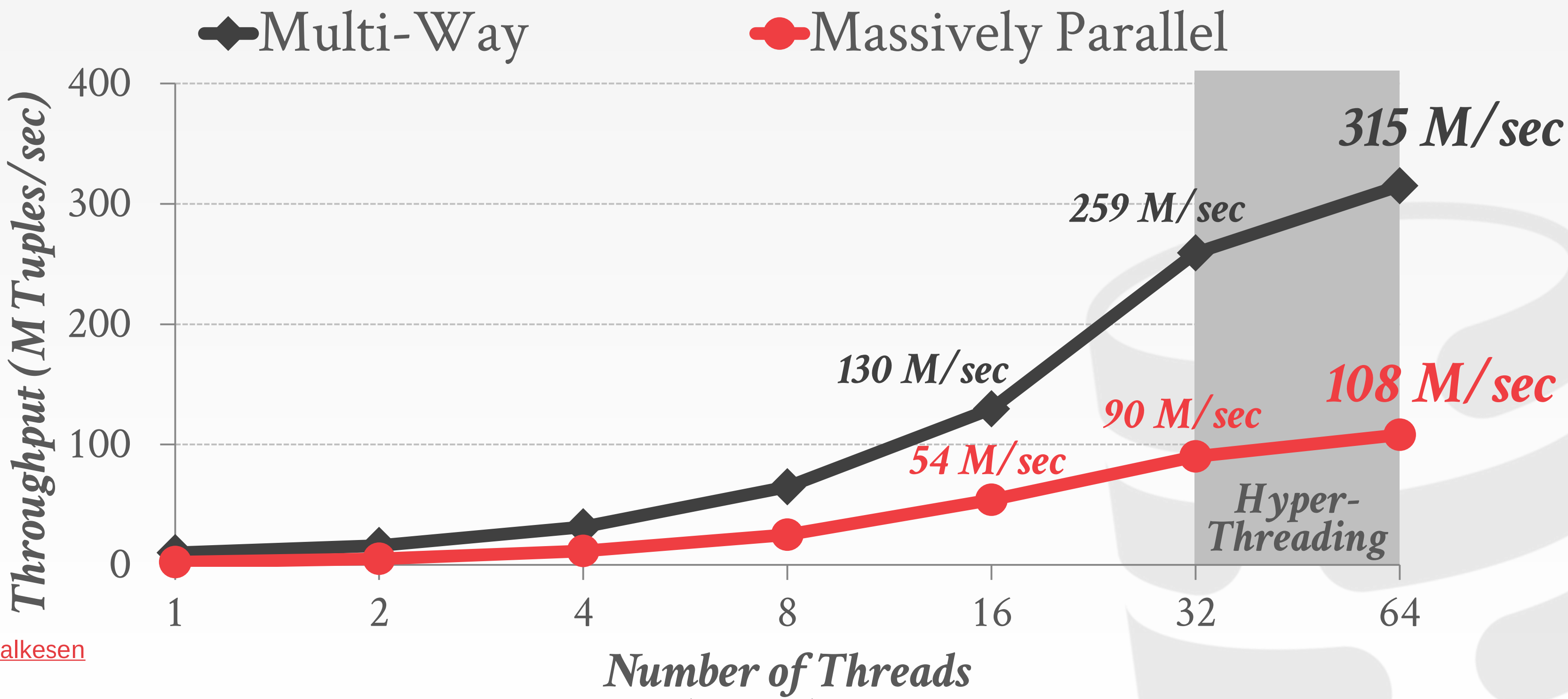
Workload: 1.6B ⋈ 128M (8-byte tuples)



Source: Cagri Balkesen

M-WAY JOIN VS. MPSM JOIN

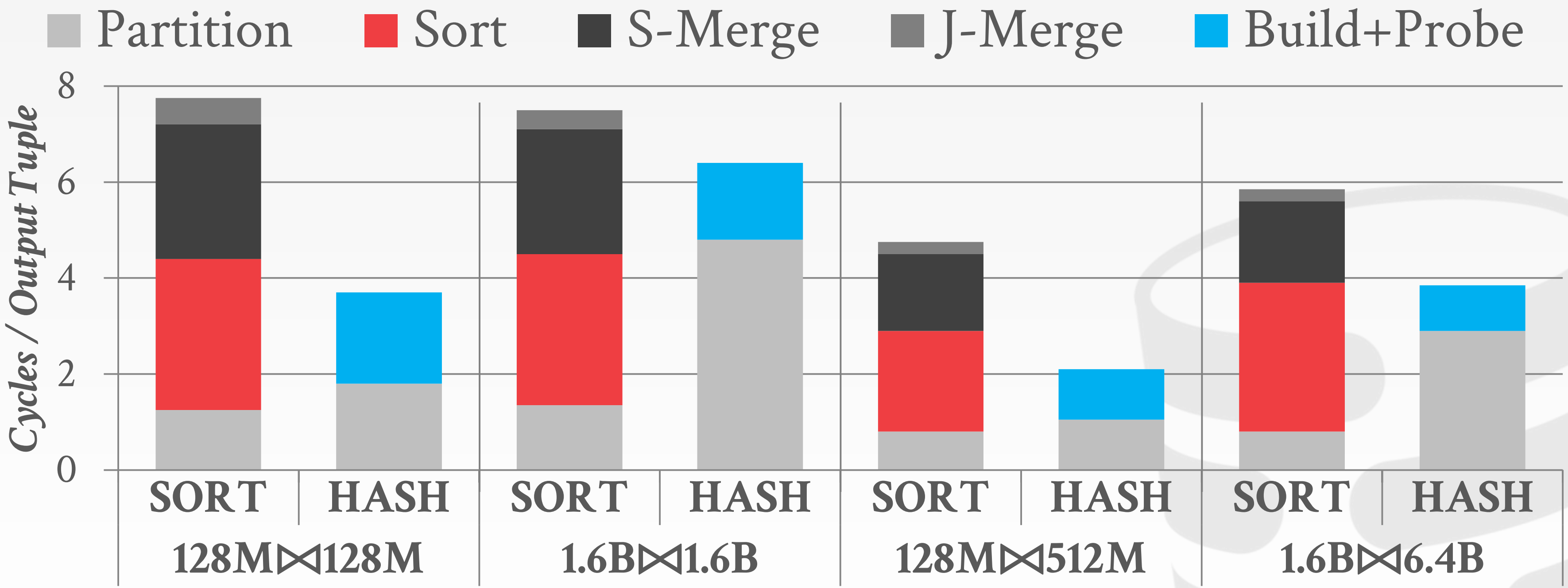
Workload: 1.6B⋈128M (8-byte tuples)



Source: Cagri Balkesen

SORT-MERGE JOIN VS. HASH JOIN

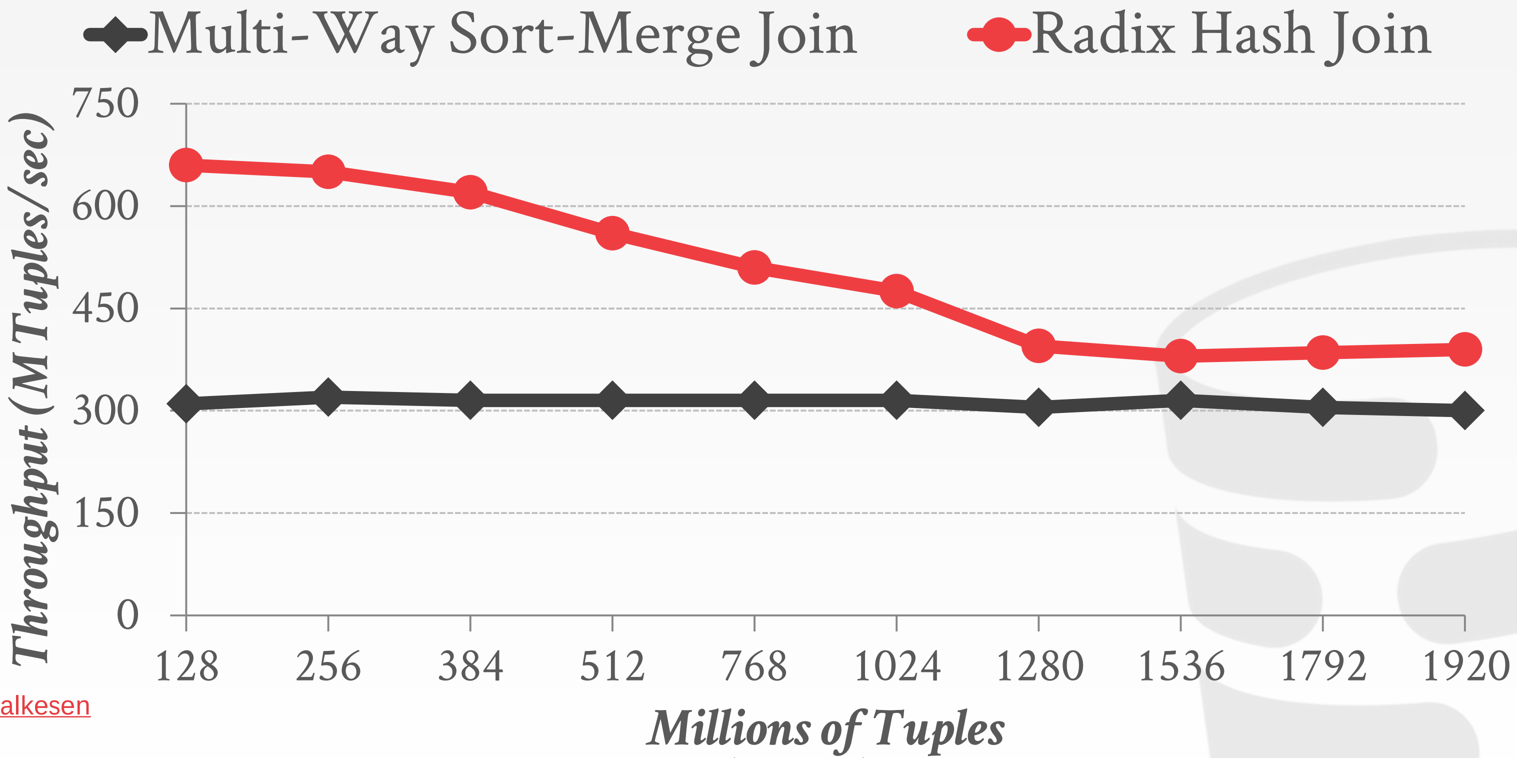
Workload: Different Table Sizes (8-byte tuples)



Source: Cagri Balkesen

SORT-MERGE JOIN VS. HASH JOIN

Varying the size of the input relations



Source: Cagri Balkesen

PARTING THOUGHTS

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