Carnegie Mellon University

Lecture #20

ADVANCED DATABASE SYSTEMS

Parallel Join Algorithms (Sorting)

@Andy_Pavlo // 15-721 // Spring 2018

ADMINISTRIVIA

Code Review Submission: April 11th

Project Status Meetings: April 11th

Project #3 Status Updates: April 16th

TODAY'S AGENDA

SIMD Background

Parallel Sort-Merge Join

Evaluation

Project #3 Code Review Guidelines

SINGLE INSTRUCTION, MULTIPLE DATA

A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

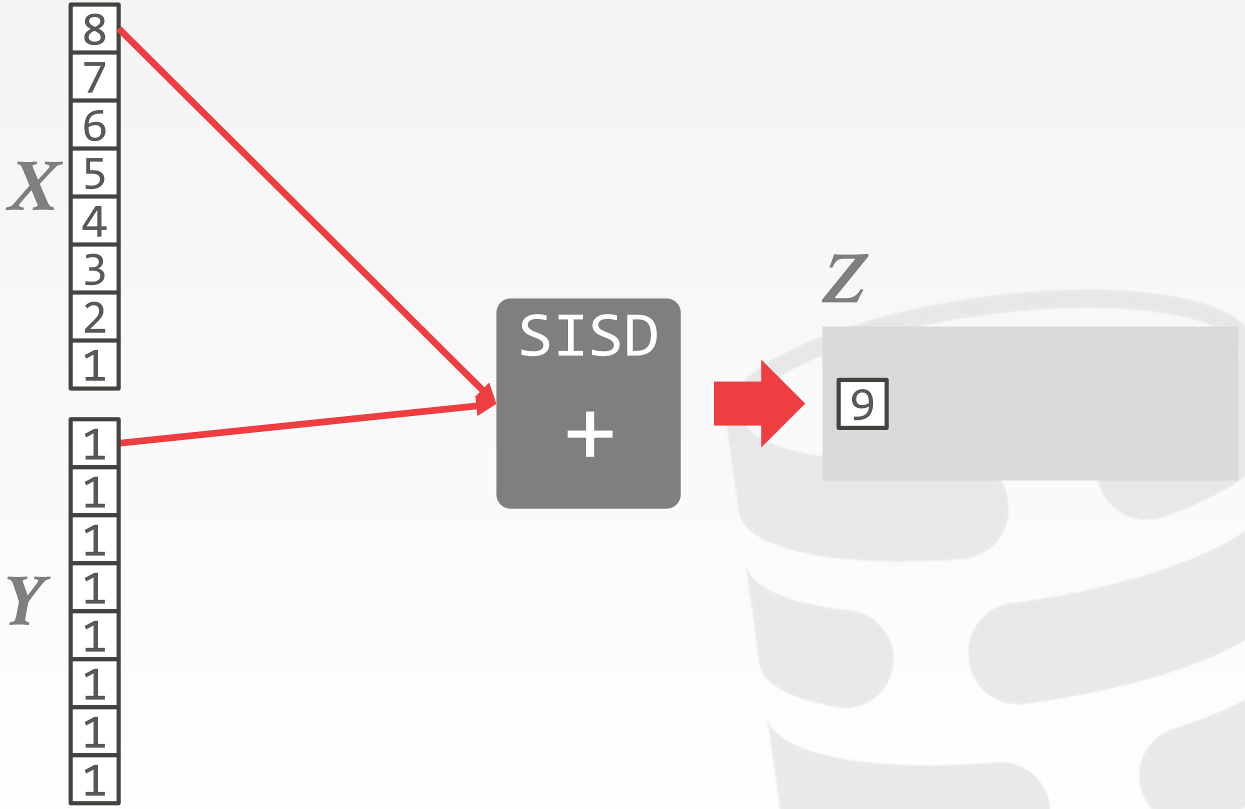
Both current AMD and Intel CPUs have ISA and microarchitecture support SIMD operations.
→ MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX

SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

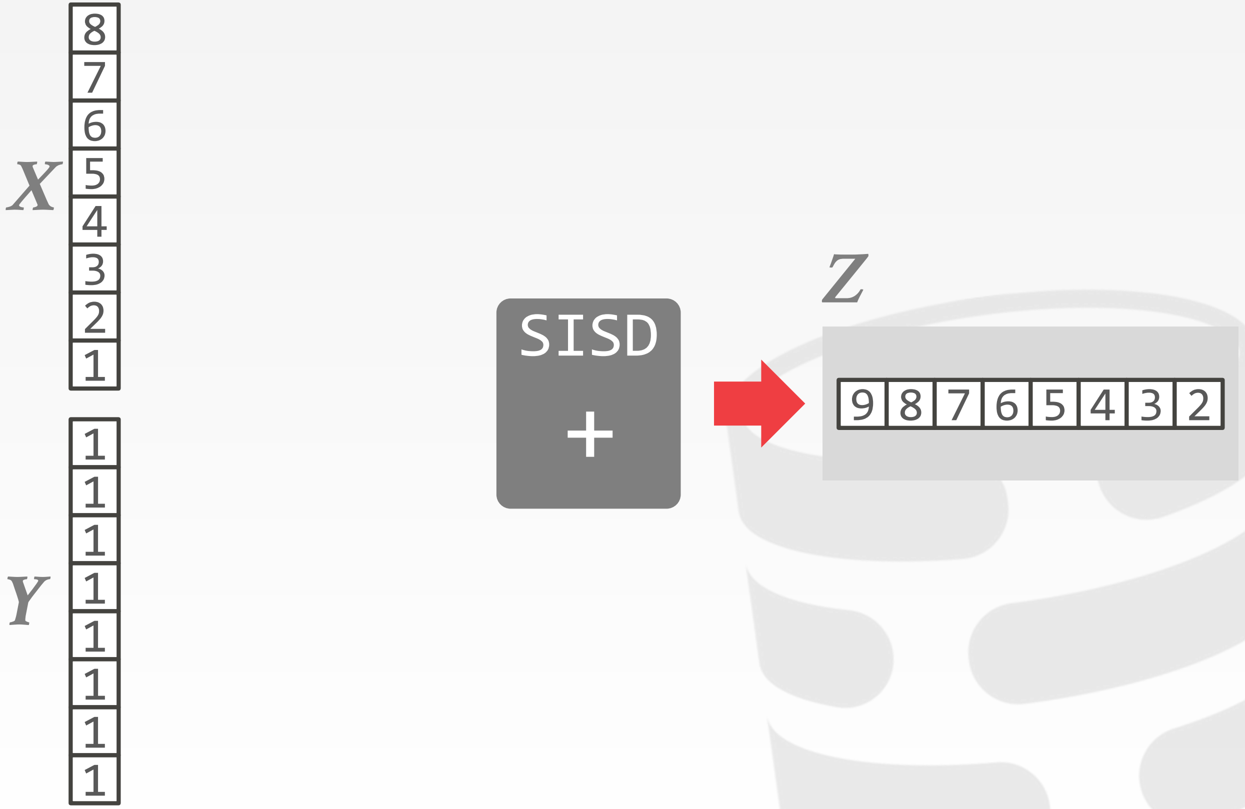


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

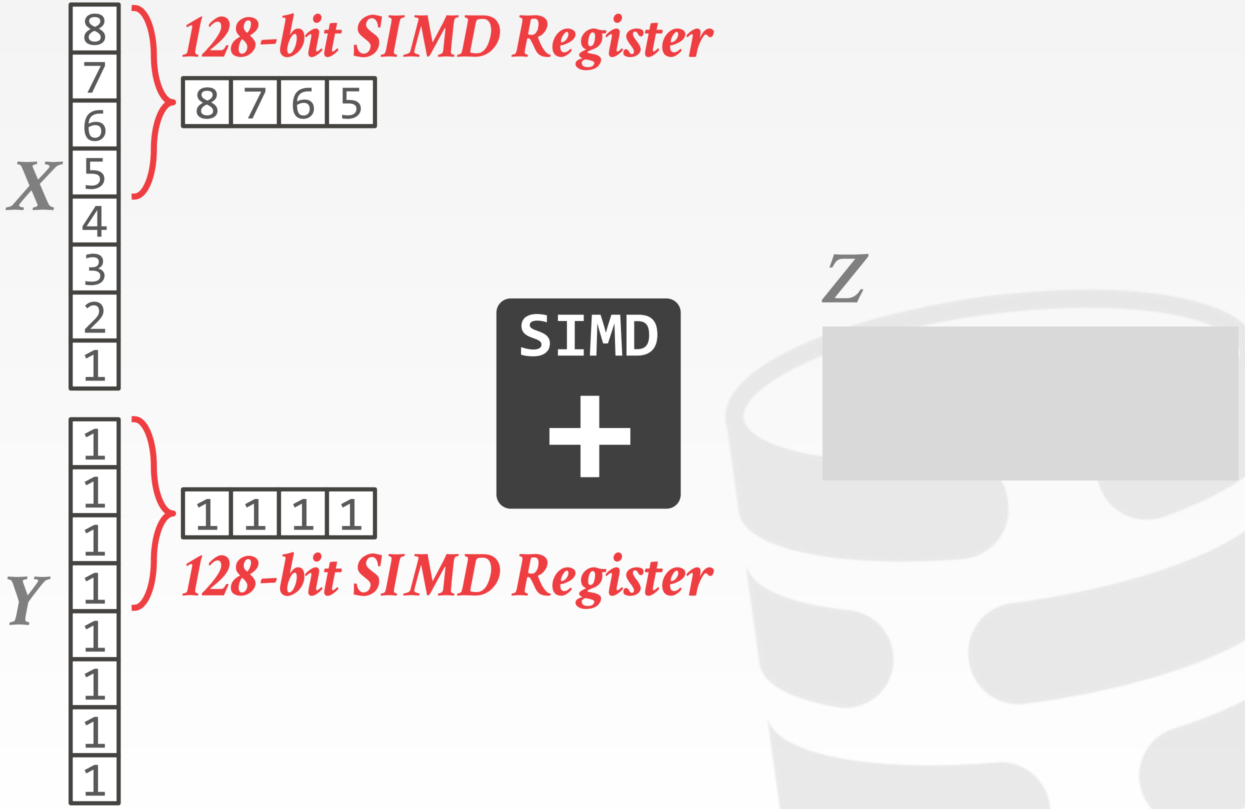


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

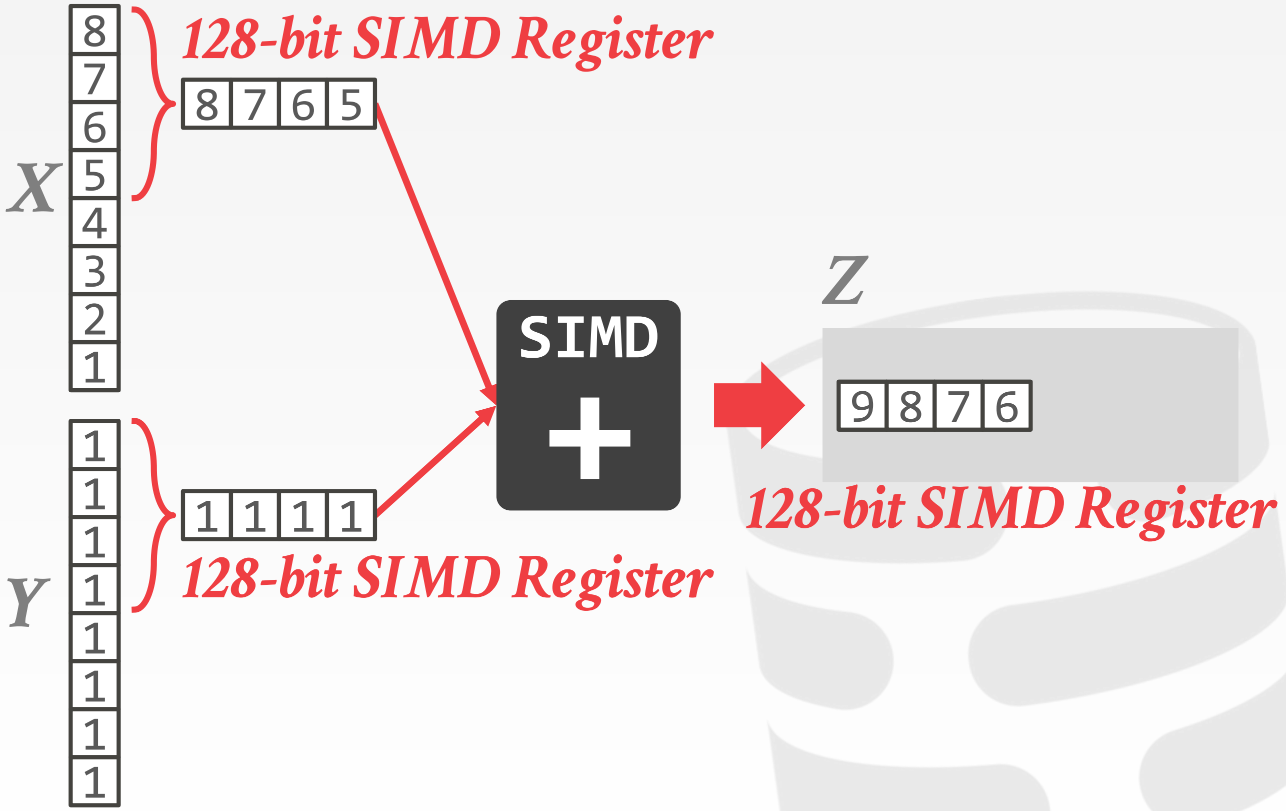


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

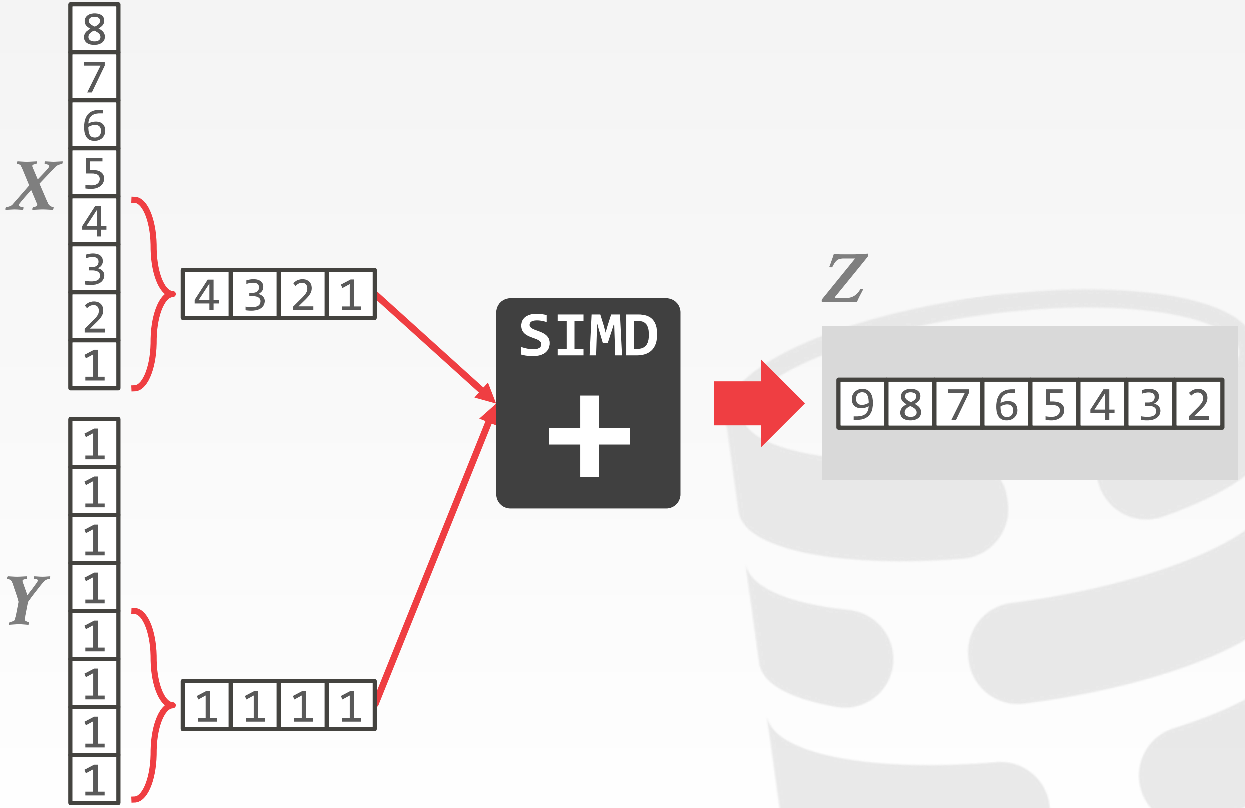


SIMD EXAMPLE

$X + Y = Z$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



SIMD TRADE-OFFS

Advantages:

→ Significant performance gains and resource utilization if an algorithm can be vectorized.

Disadvantages:

→ Implementing an algorithm using SIMD is still mostly a manual process.
→ SIMD may have restrictions on data alignment.
→ Gathering data into SIMD registers and scattering it to the correct locations is tricky and/or inefficient.

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

Phase #1: Sort

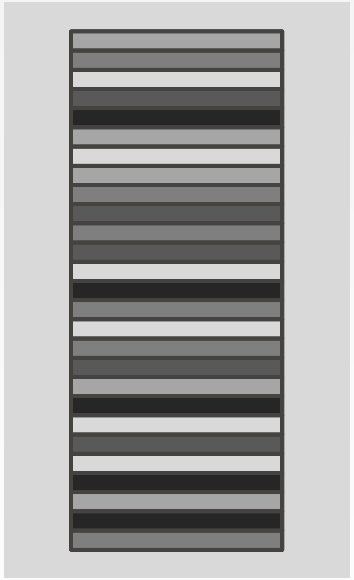
→ Sort the tuples of **R** and **S** based on the join key.

Phase #2: Merge

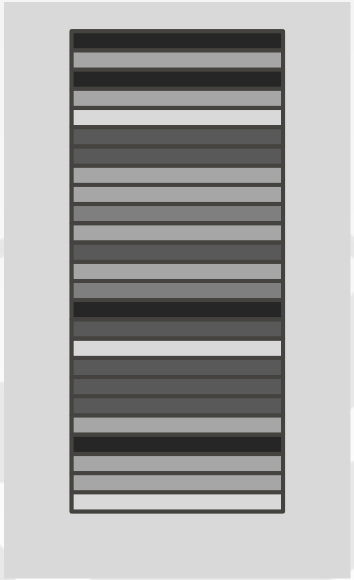
→ Scan the sorted relations and compare tuples.
→ The outer relation **R** only needs to be scanned once.

SORT-MERGE JOIN (R⋈S)

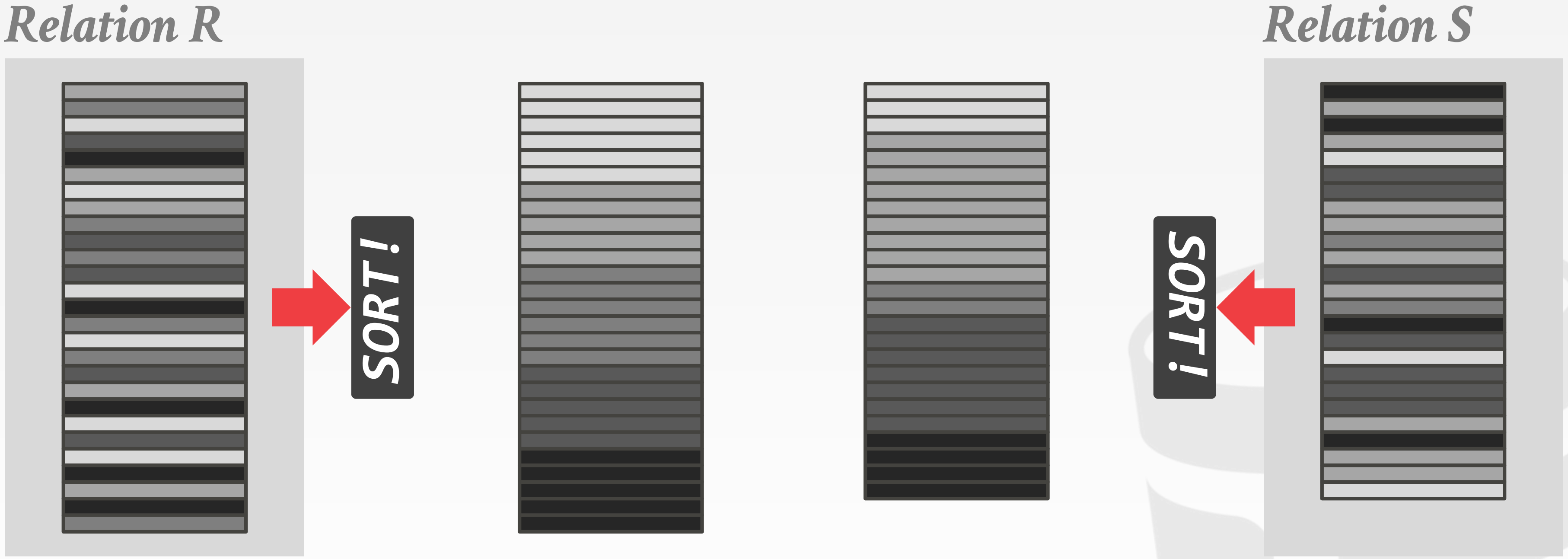
Relation R



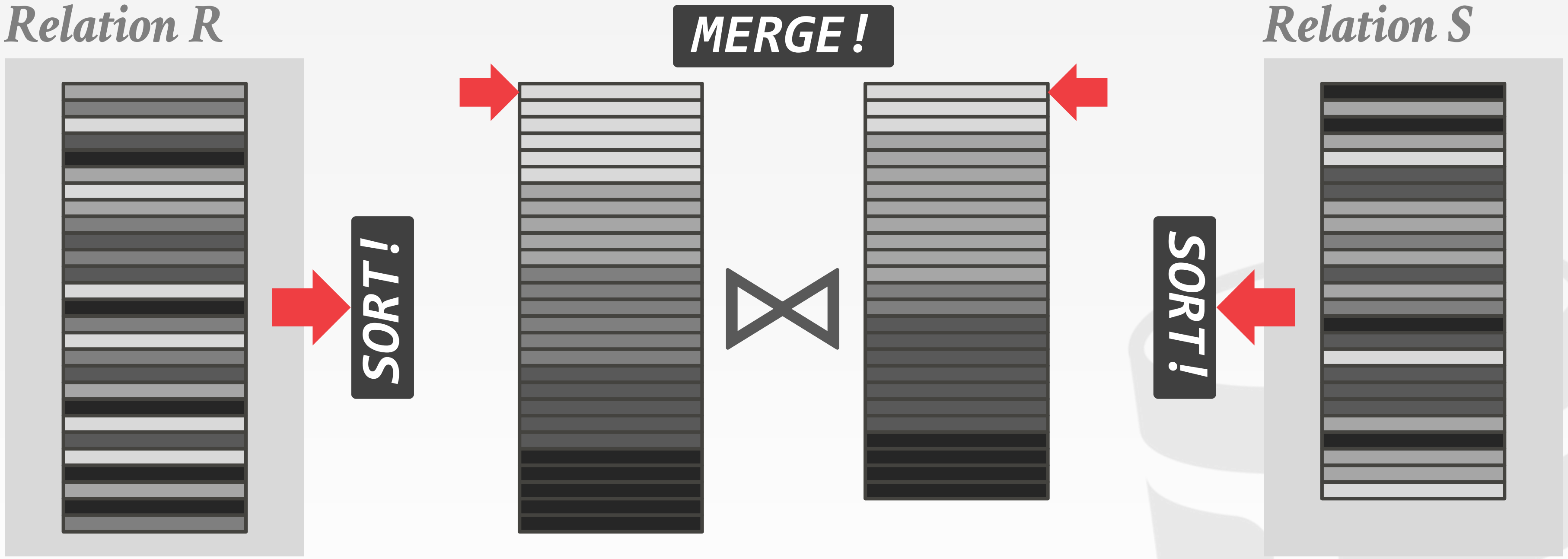
Relation S



SORT-MERGE JOIN (R⋈S)



SORT-MERGE JOIN (R⋈S)



PARALLEL SORT-MERGE JOINS

Sorting is always 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.

But NUMA and parallel architectures require us to be more careful…

CACHE-CONSCIOUS SORTING

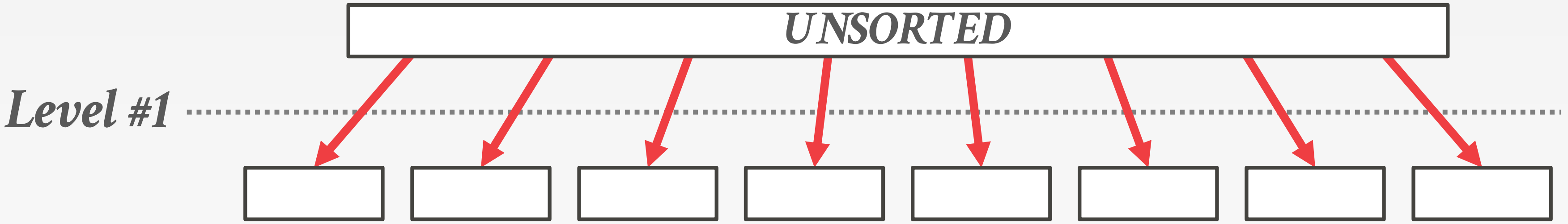
Level #1: In-Register Sorting
→ Sort runs that fit into CPU registers.

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

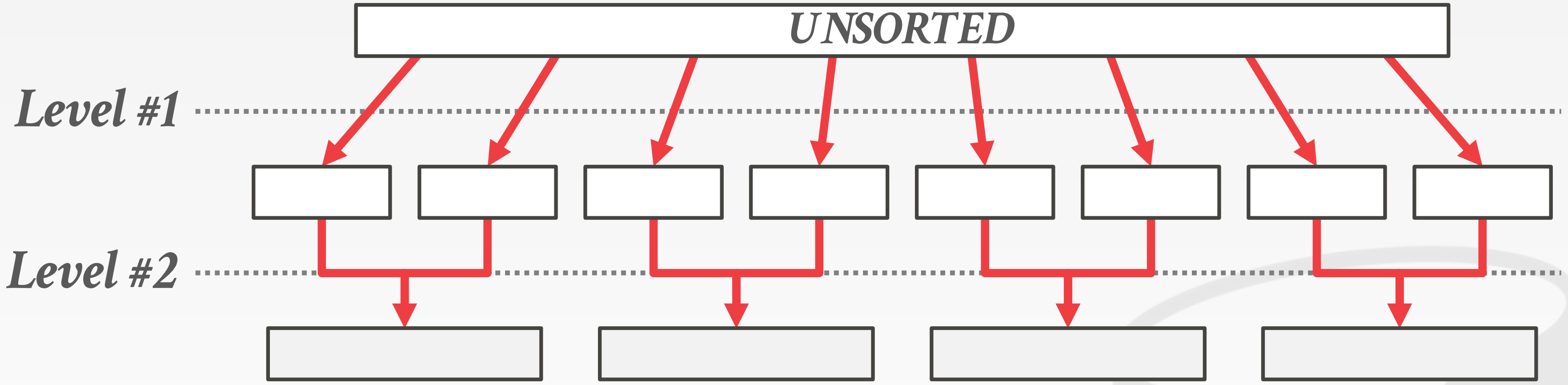
Level #3: Out-of-Cache Sorting
→ Used when the runs of Level #2 exceed the size of caches.

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

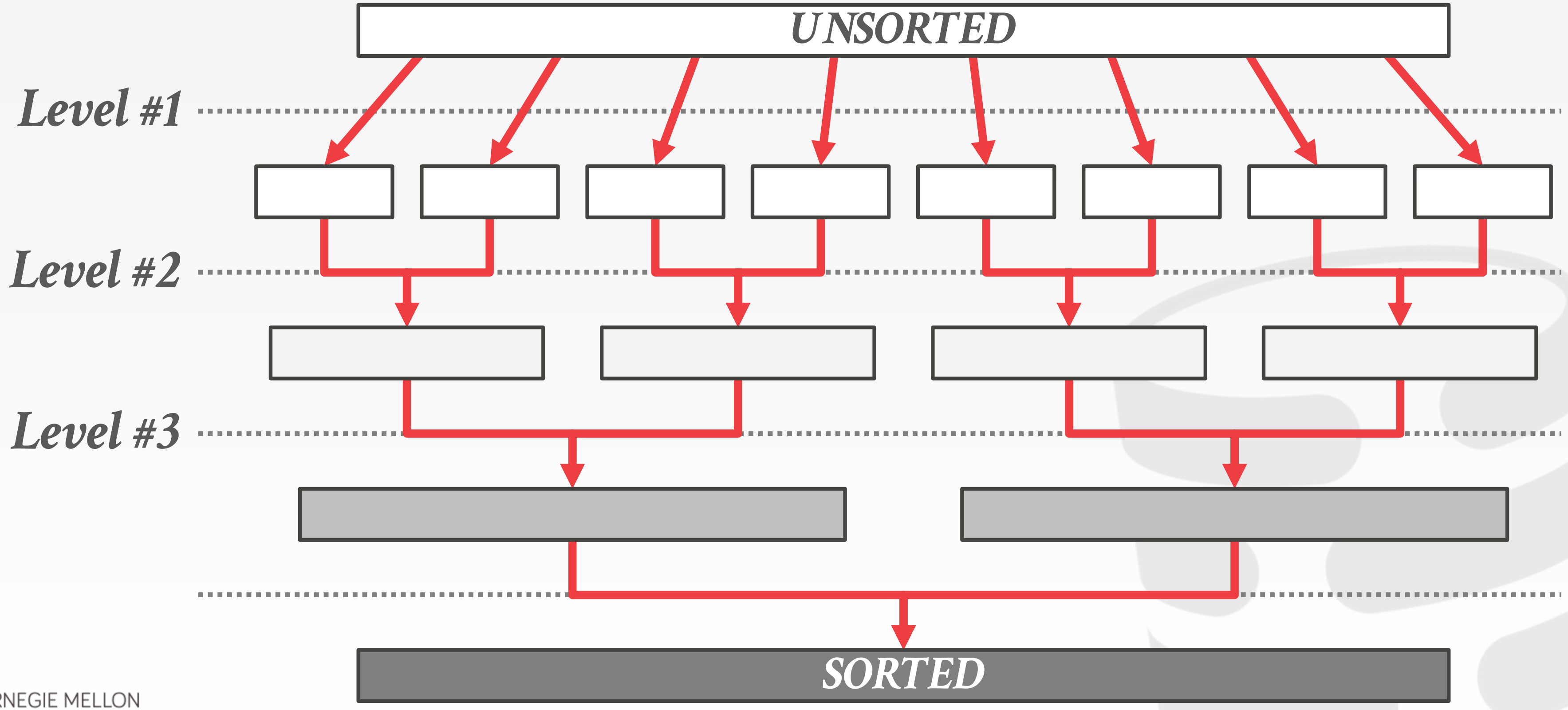
CACHE-CONSCIOUS SORTING



CACHE-CONSCIOUS SORTING



CACHE-CONSCIOUS SORTING

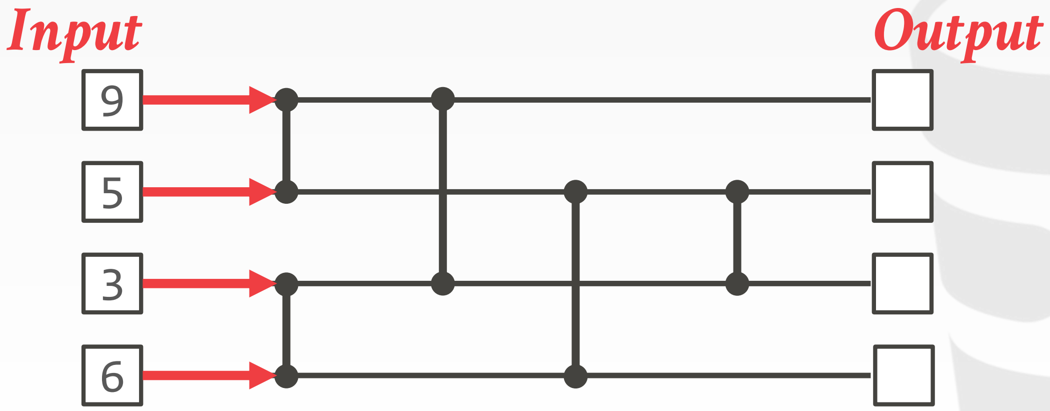


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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

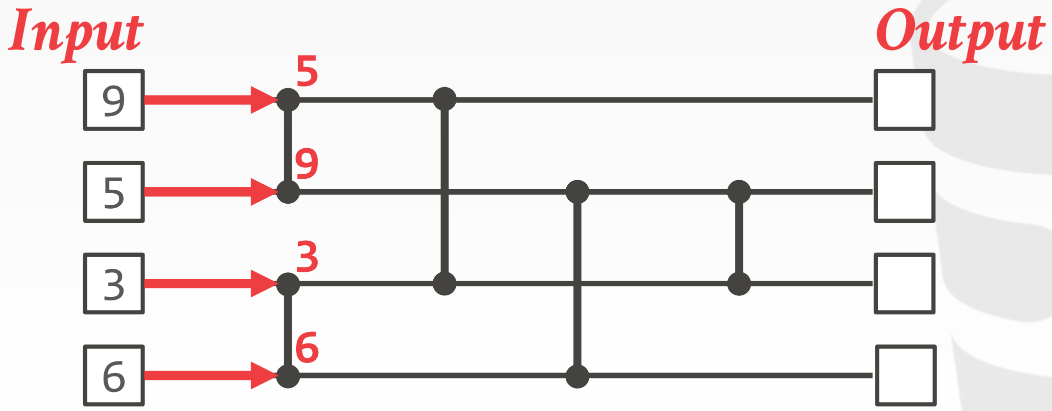


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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

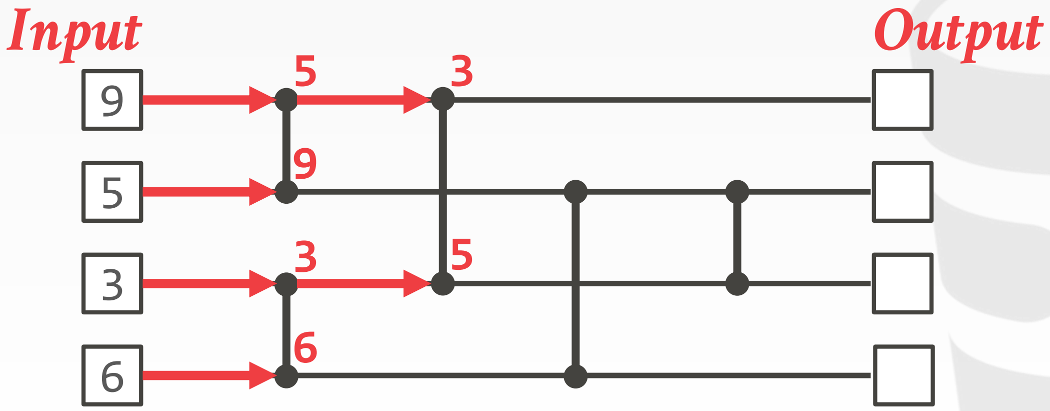


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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

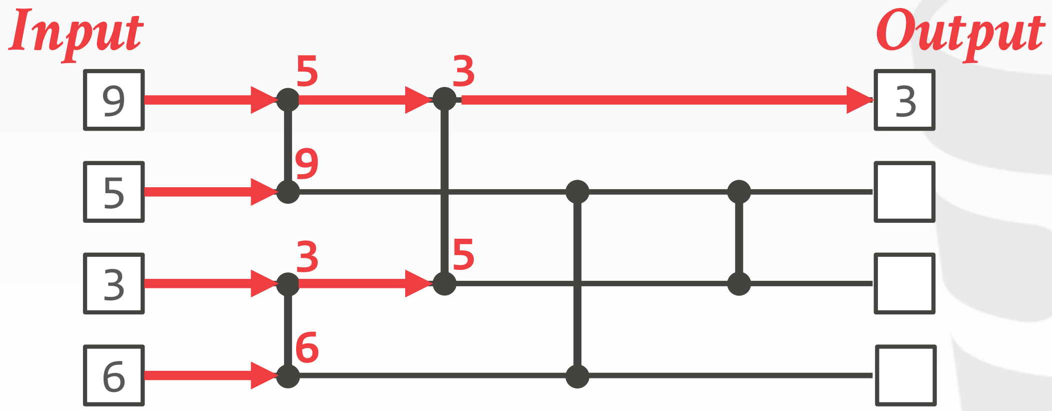


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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

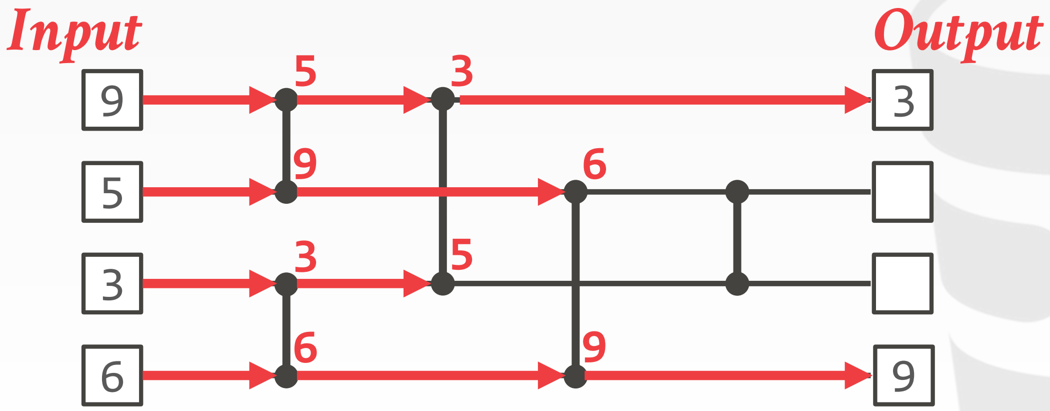


LEVEL #1 – SORTING NETWORKS

Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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

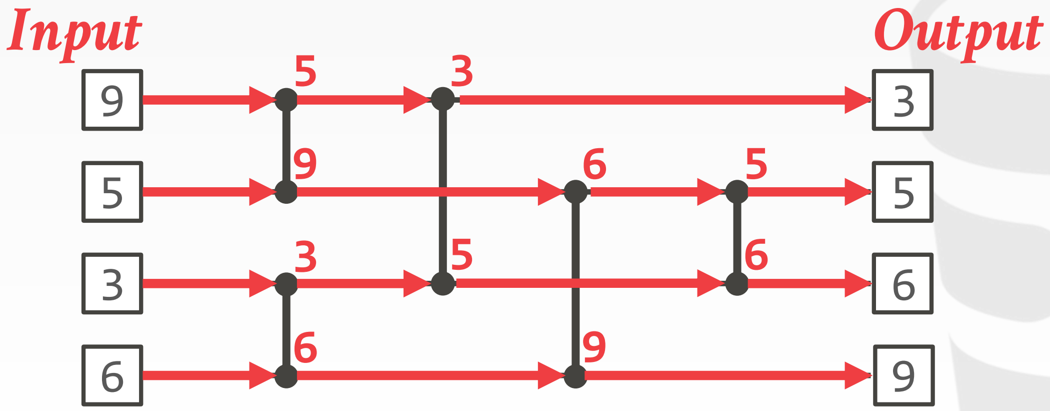


LEVEL #1 – SORTING NETWORKS

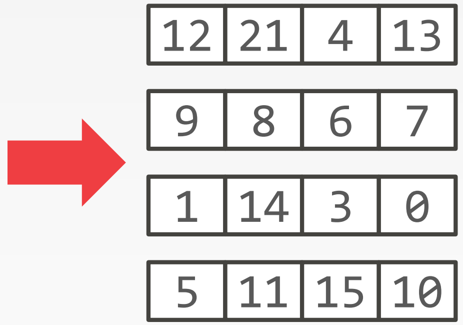
Abstract model for sorting keys.

→ Always has fixed wiring "paths" for lists with the same number of elements.

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



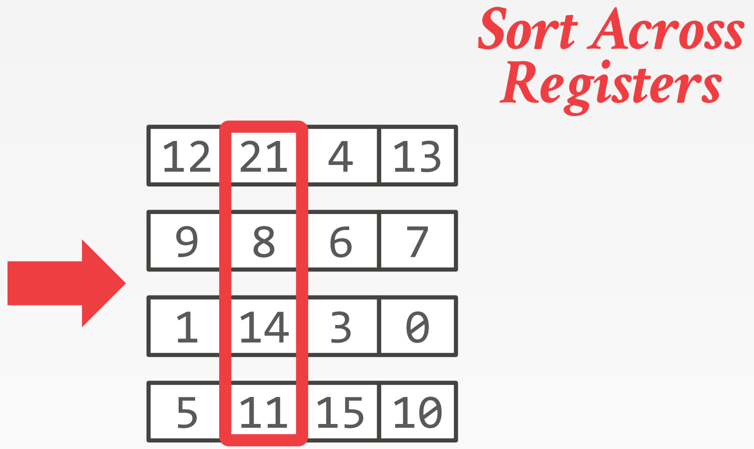
LEVEL #1 – SORTING NETWORKS



Instructions:

→ 4 **LOAD**

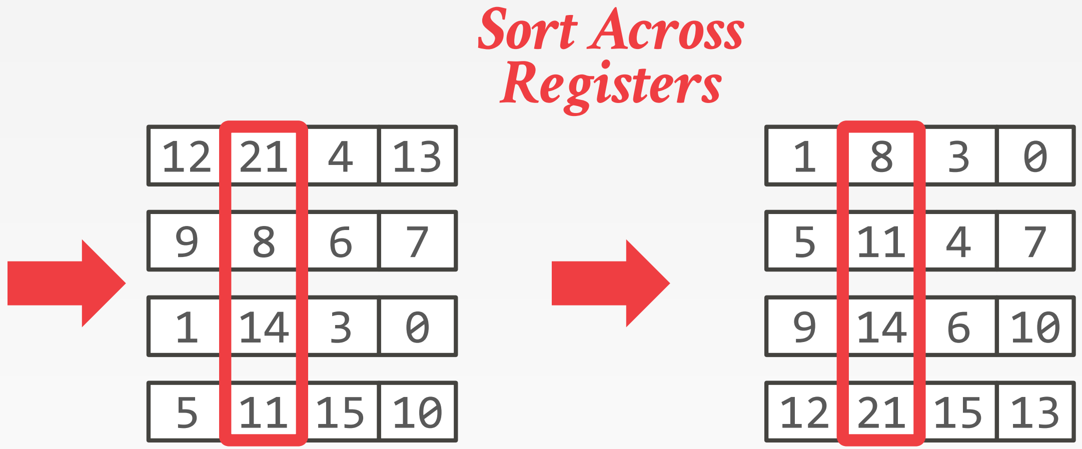
LEVEL #1 – SORTING NETWORKS



Instructions:

→ 4 **LOAD**

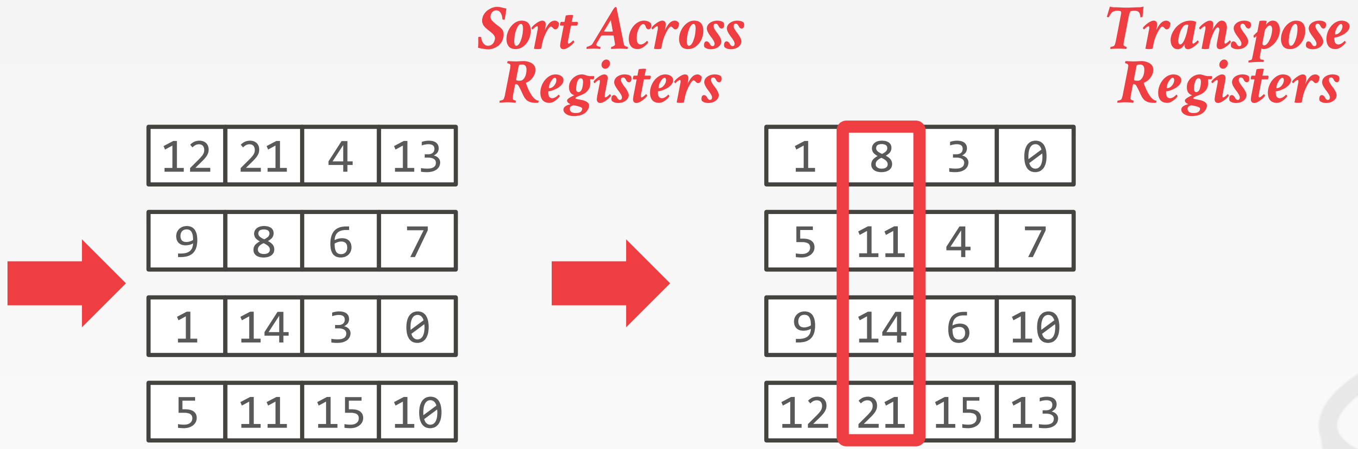
LEVEL #1 – SORTING NETWORKS



Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

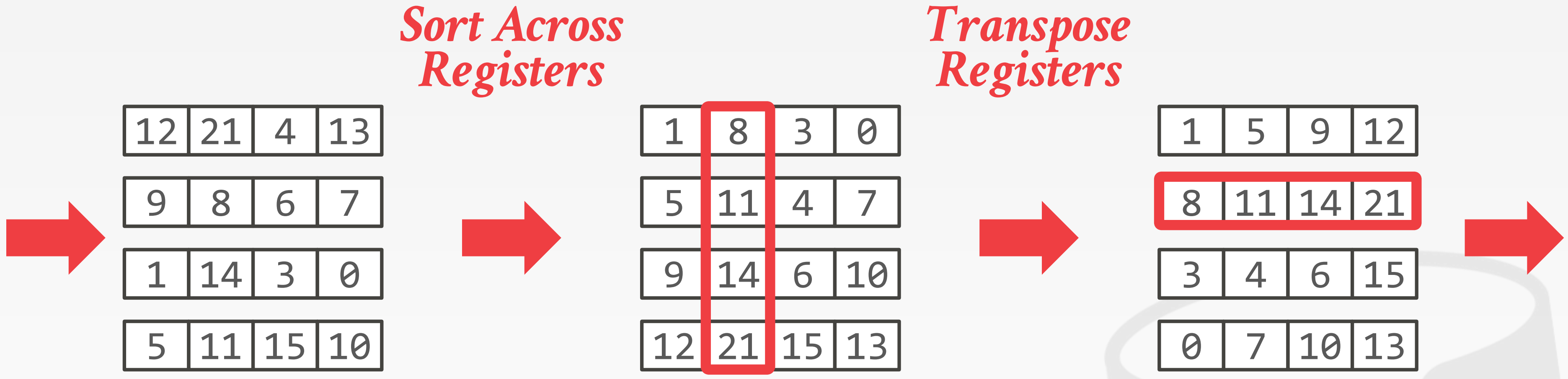
LEVEL #1 – SORTING NETWORKS



Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

LEVEL #1 – SORTING NETWORKS



Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

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.

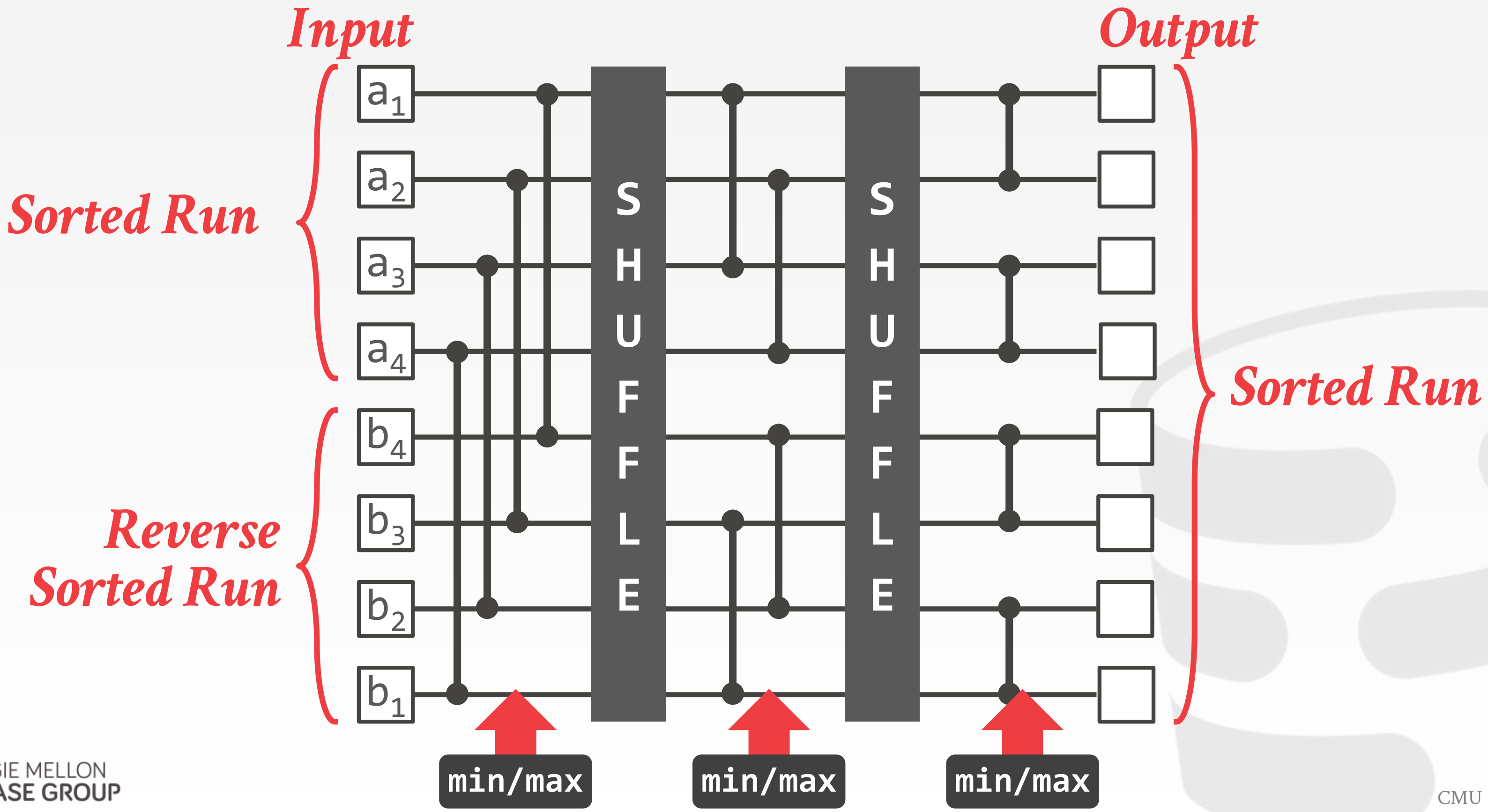
Can expand network to merge progressively larger lists (½ cache size).

Intel's Measurements

→ 2.25–3.5x speed-up over SISD implementation.

EFFICIENT IMPLEMENTATION OF SORTING ON MULTI-CORE
VLDB 2008

LEVEL #2 – BITONIC MERGE NETWORK



LEVEL #3 – MULTI-WAY MERGING

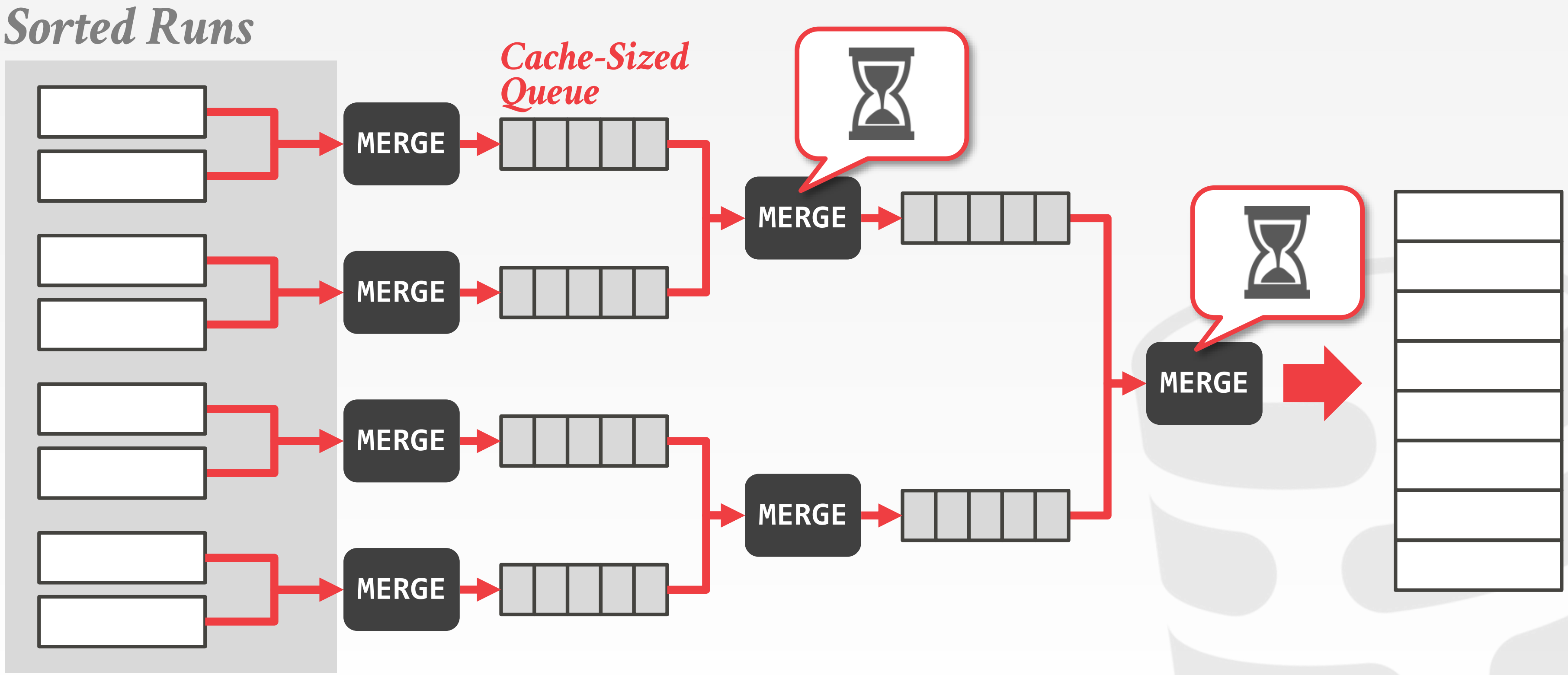
Use the Bitonic Merge Networks but split the process up into tasks.

→ 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



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).

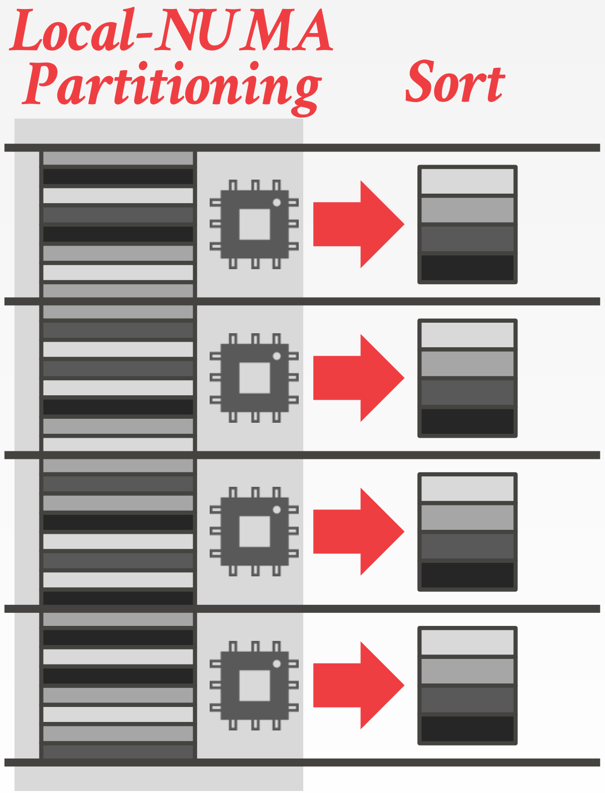
Inner Table

→ Same as outer table.

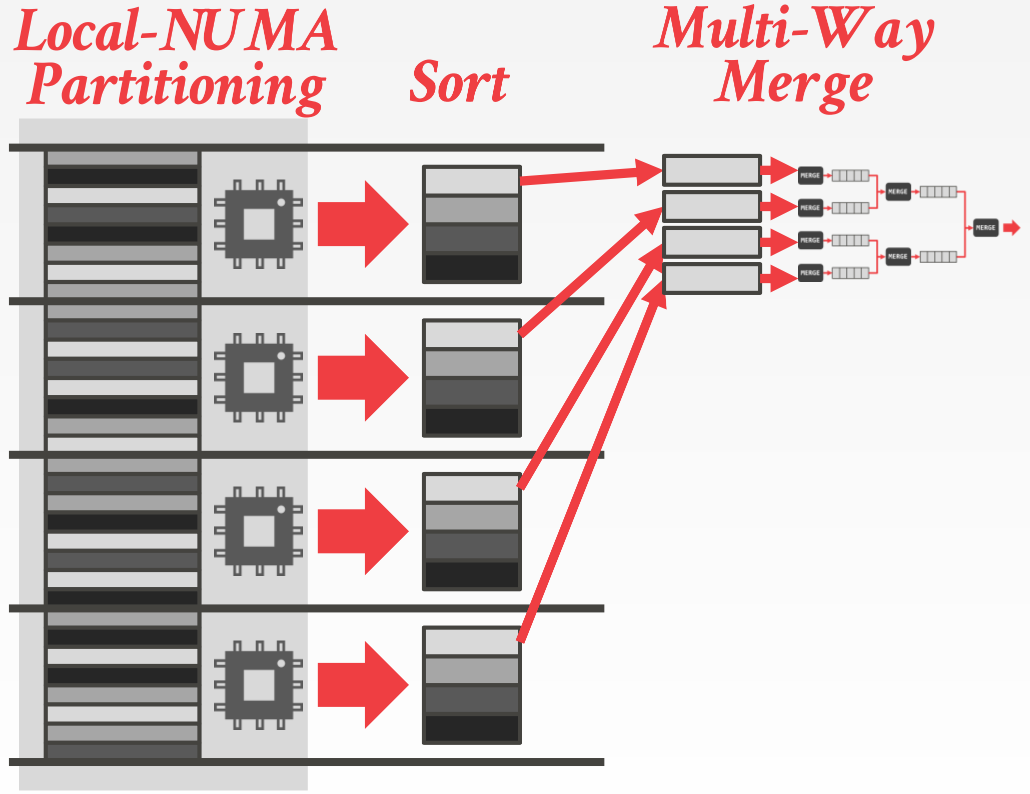
Merge phase is between matching pairs of chunks of outer/inner tables at each core.

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

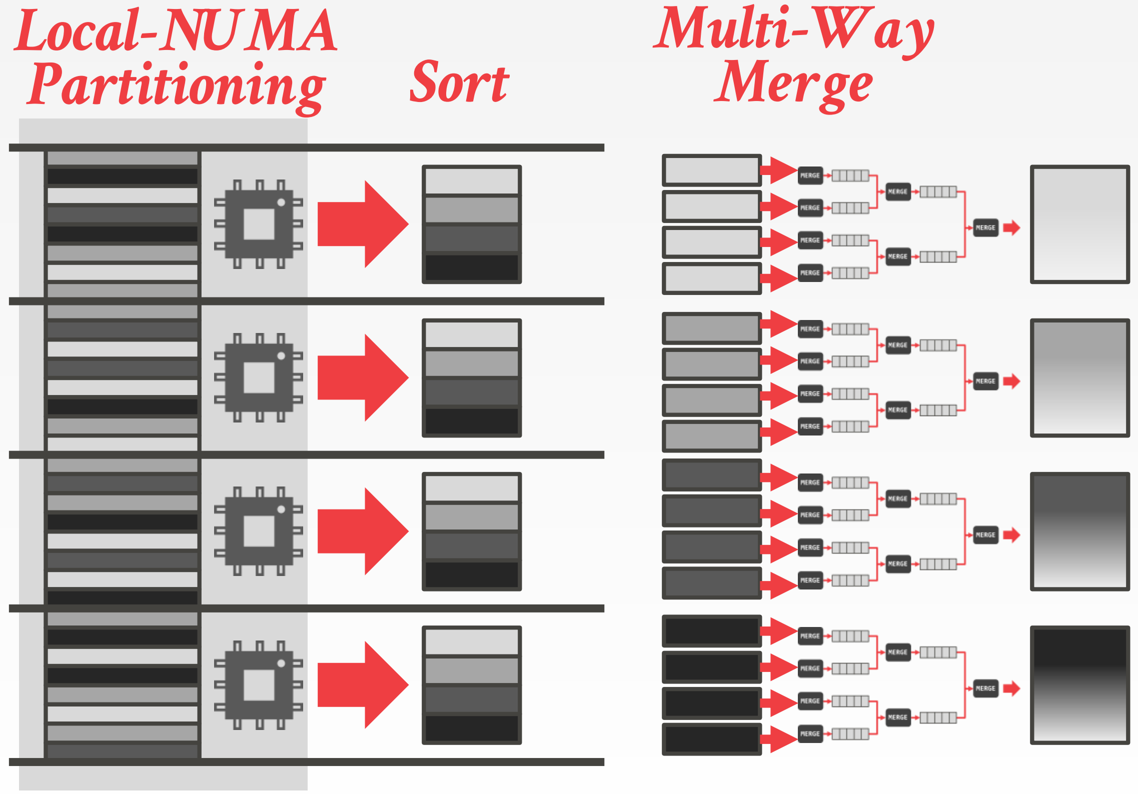
MULTI-WAY SORT-MERGE



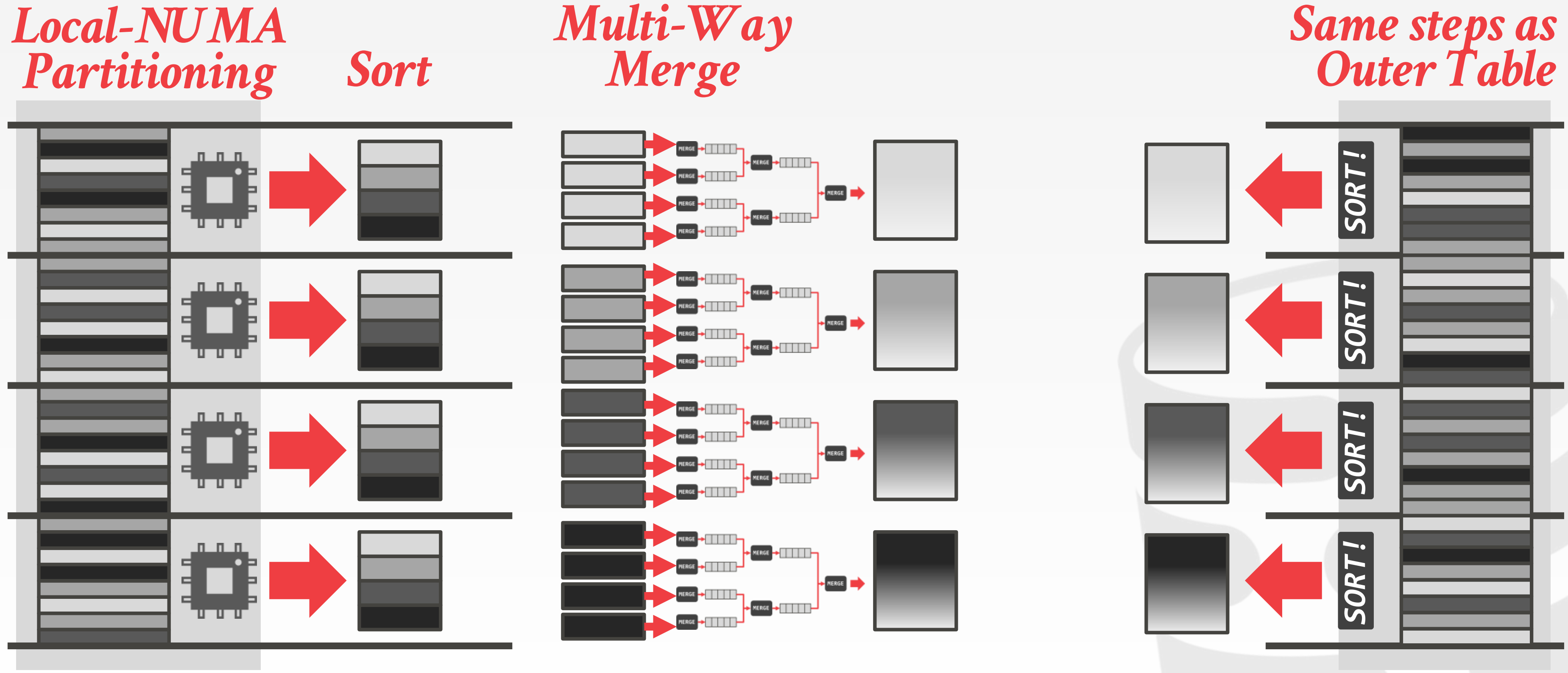
MULTI-WAY SORT-MERGE



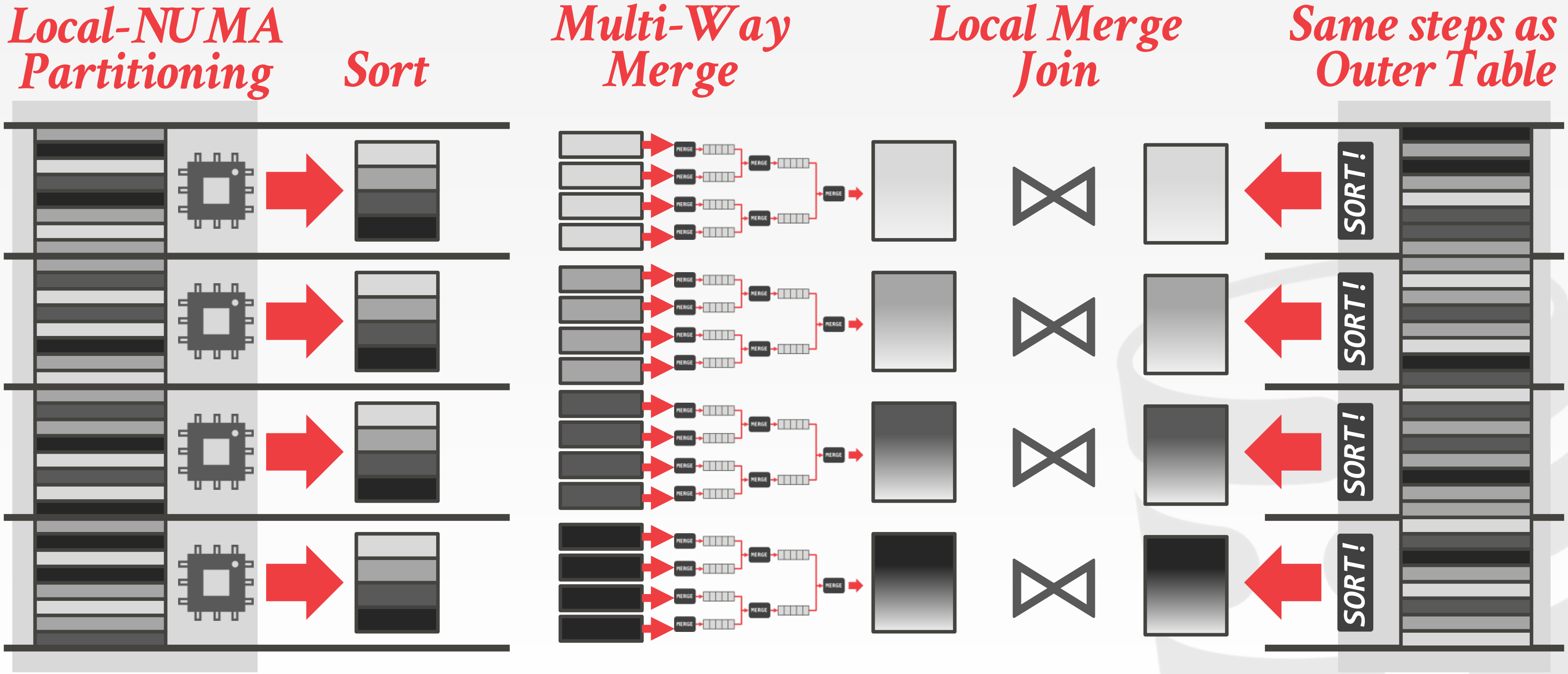
MULTI-WAY SORT-MERGE



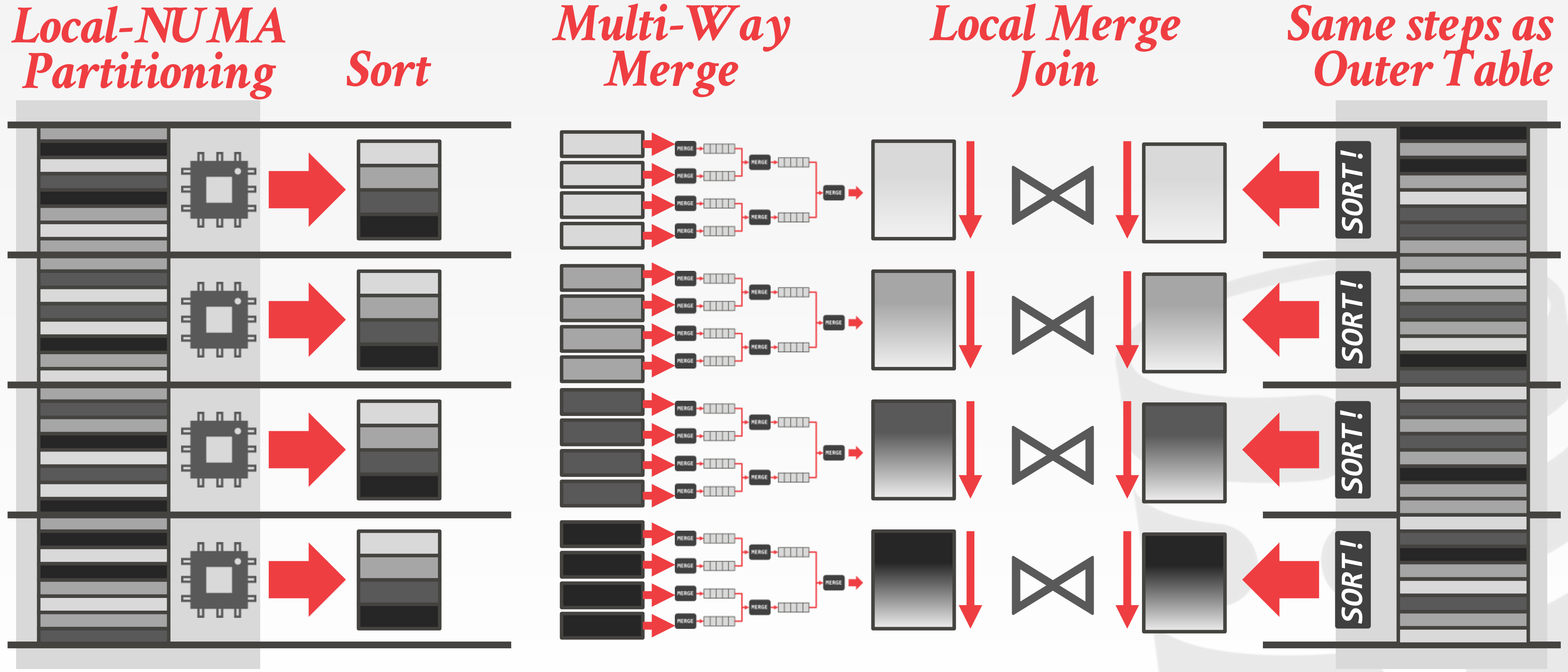
MULTI-WAY SORT-MERGE



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

MASSIVELY PARALLEL SORT-MERGE

Outer Table

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

Inner Table

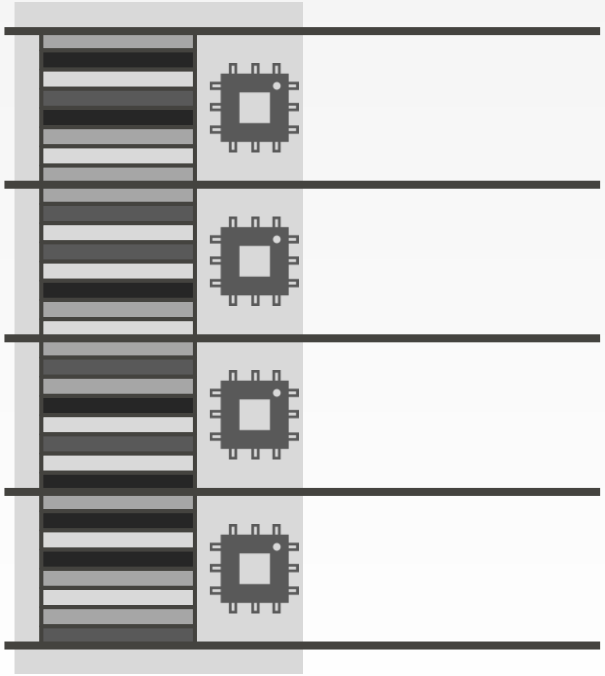
→ Not redistributed like outer table.
→ Each core sorts its local data.

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

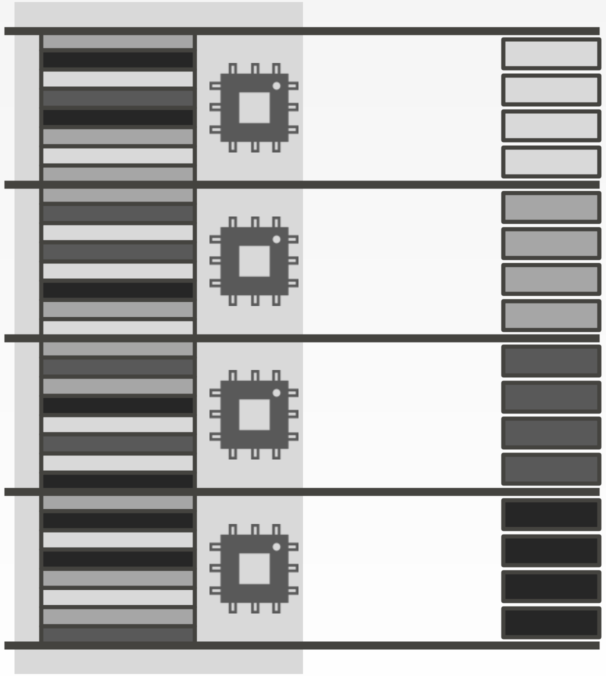
MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning



MASSIVELY PARALLEL SORT-MERGE

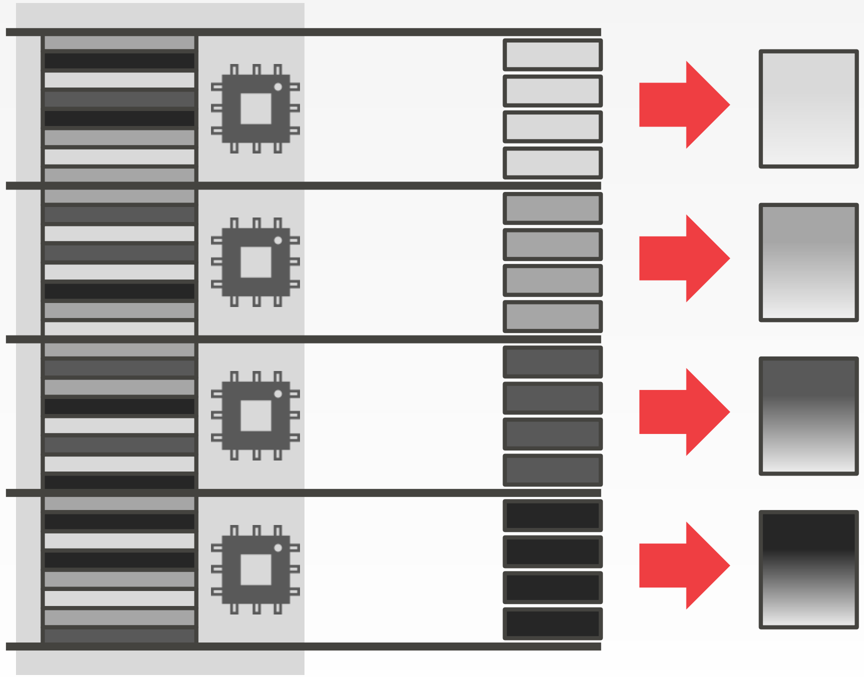
Cross-NUMA Partitioning



MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

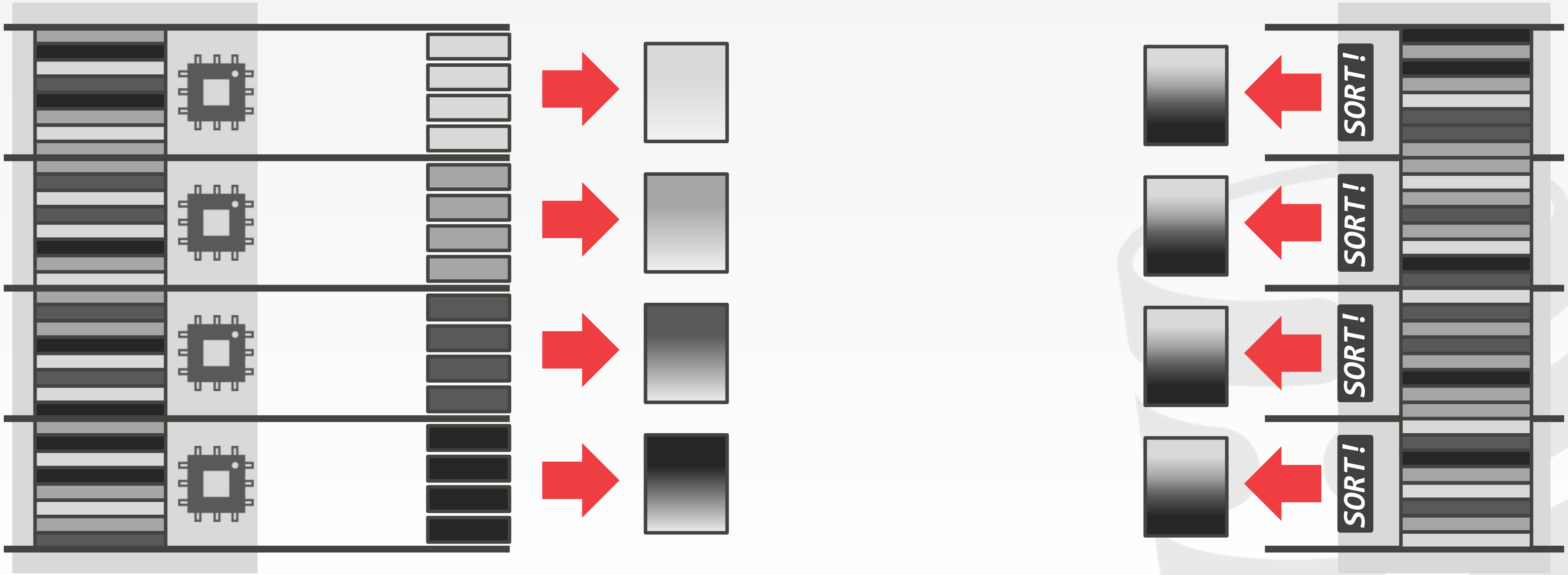
Sort



MASSIVELY PARALLEL SORT-MERGE

Cross-NUMA Partitioning

Sort

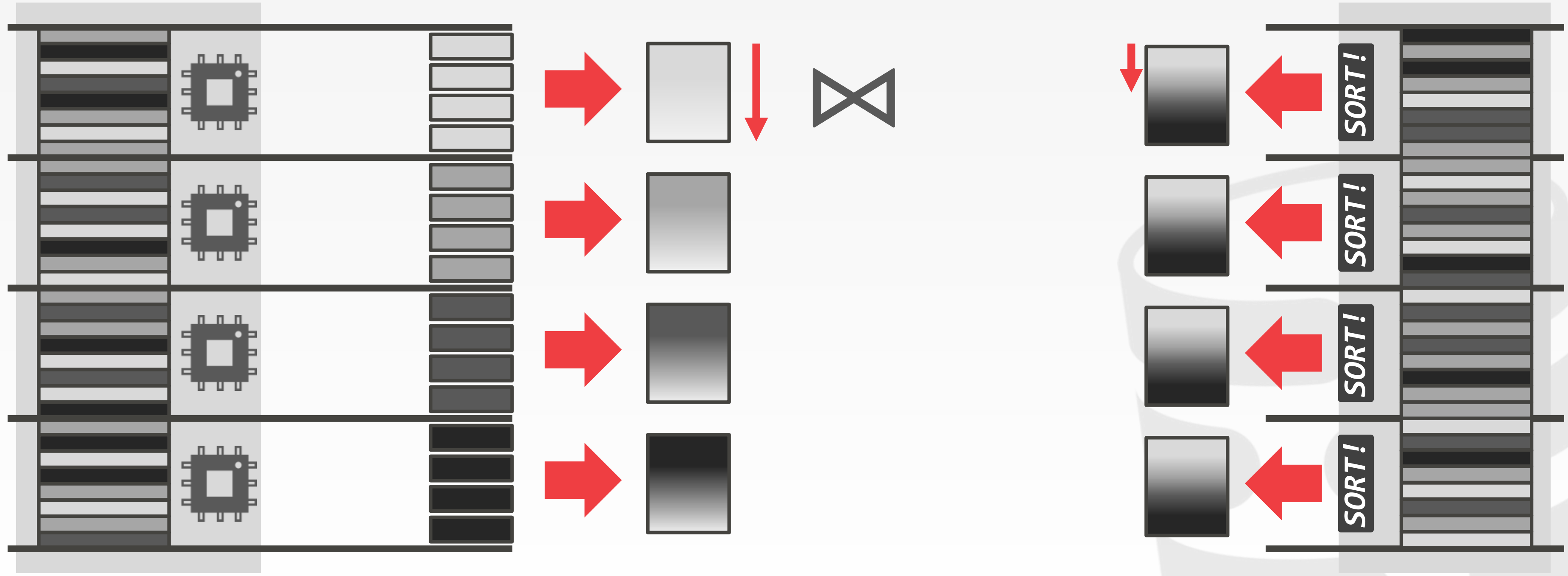


MASSIVELY PARALLEL SORT-MERGE

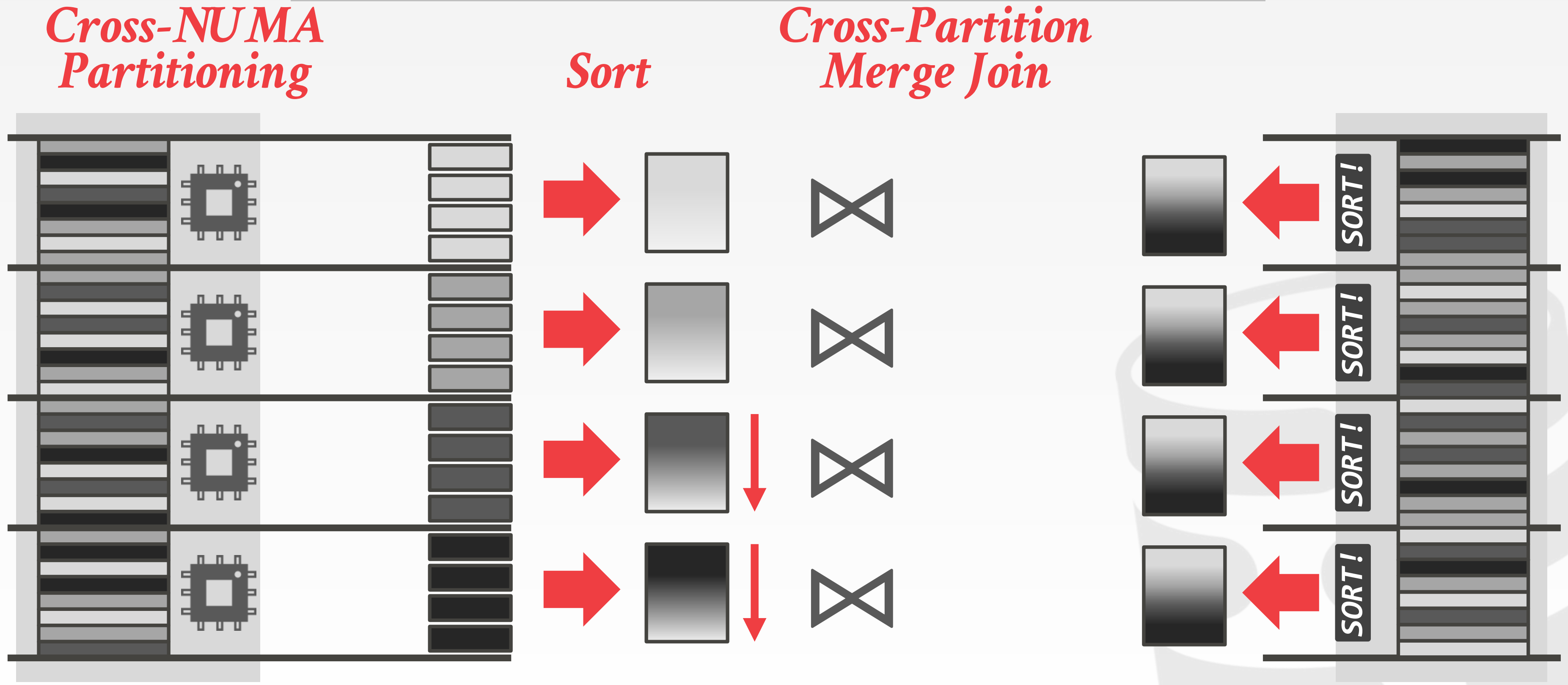
Cross-NUMA Partitioning

Sort

Cross-Partition Merge Join



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

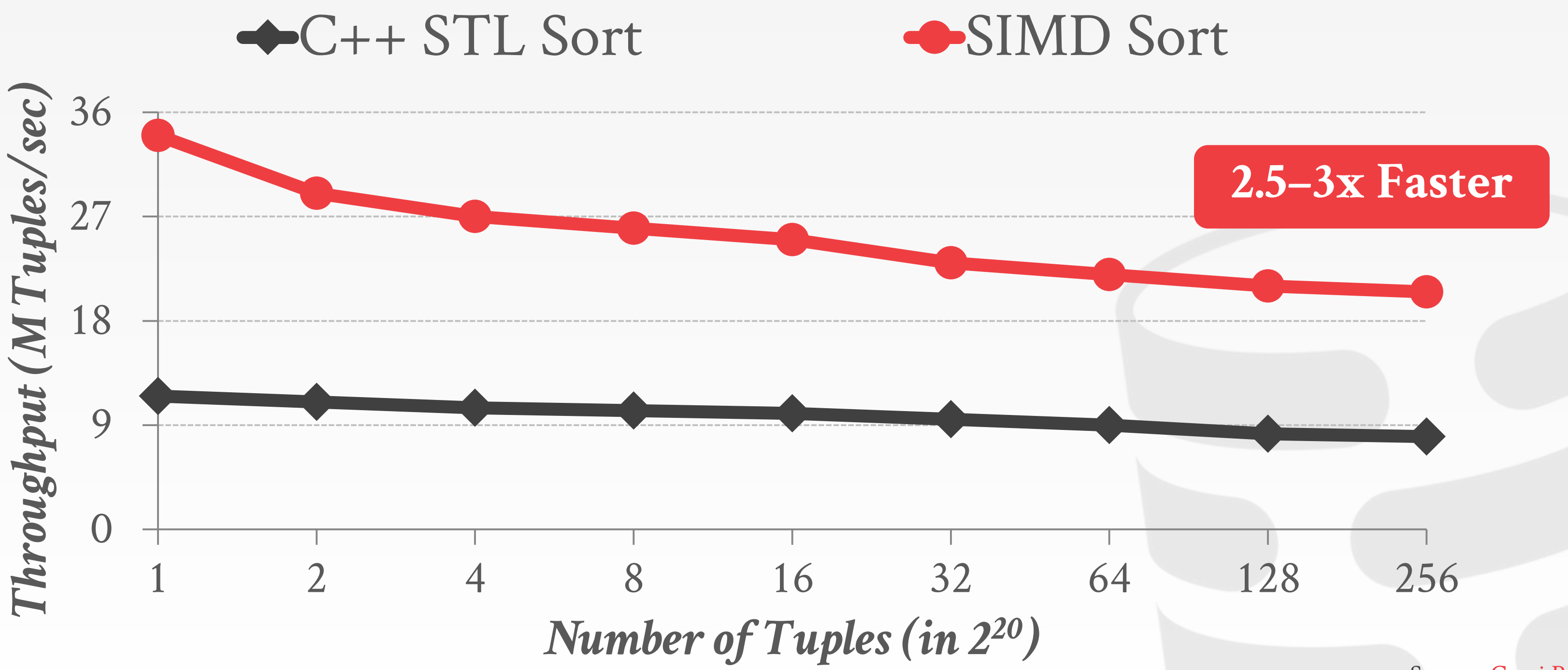
EVALUATION

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

RAW SORTING PERFORMANCE

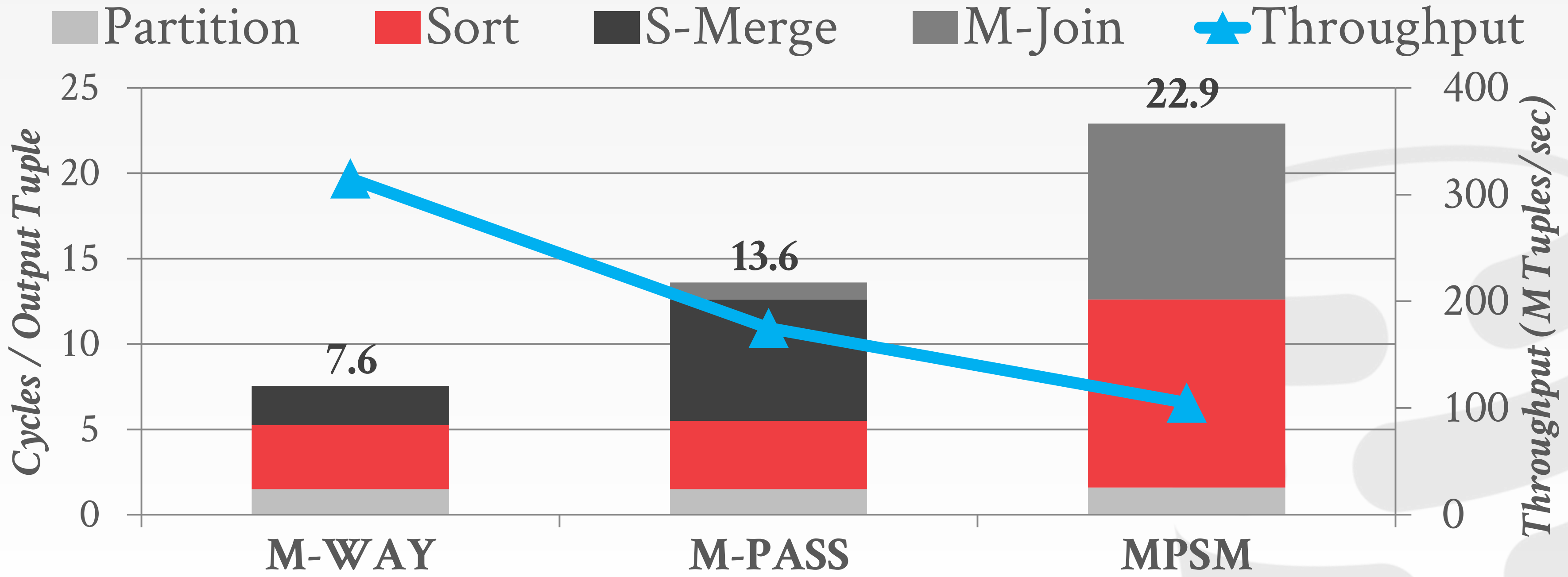
Single-threaded sorting performance



Source: Cagri Balkesen

COMPARISON OF SORT-MERGE JOINS

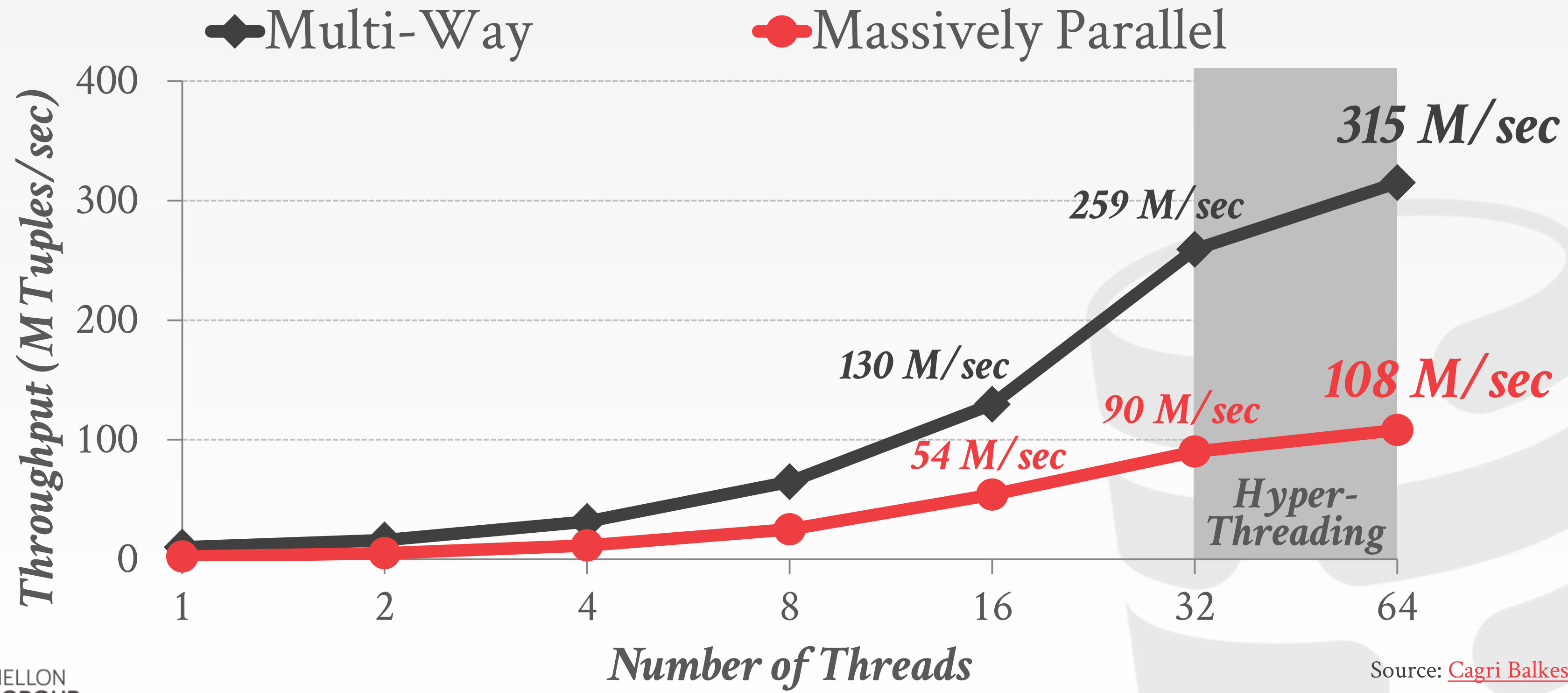
Workload: 1.6B⋈128M (8-byte tuples)



Source: Cagri Balkesen

M-WAY JOIN VS. MPSM JOIN

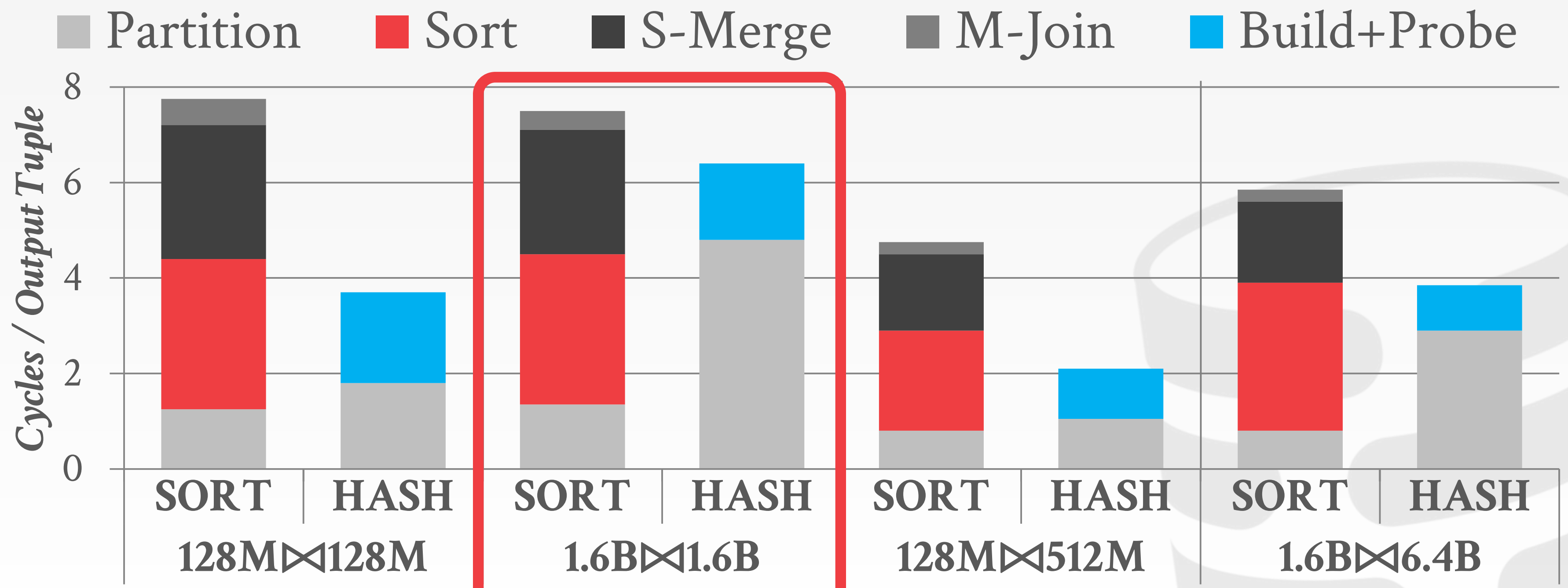
Workload: 1.6B⋈128M (8-byte tuples)



Source: Cagri Balkesen

SORT-MERGE JOIN VS. HASH JOIN

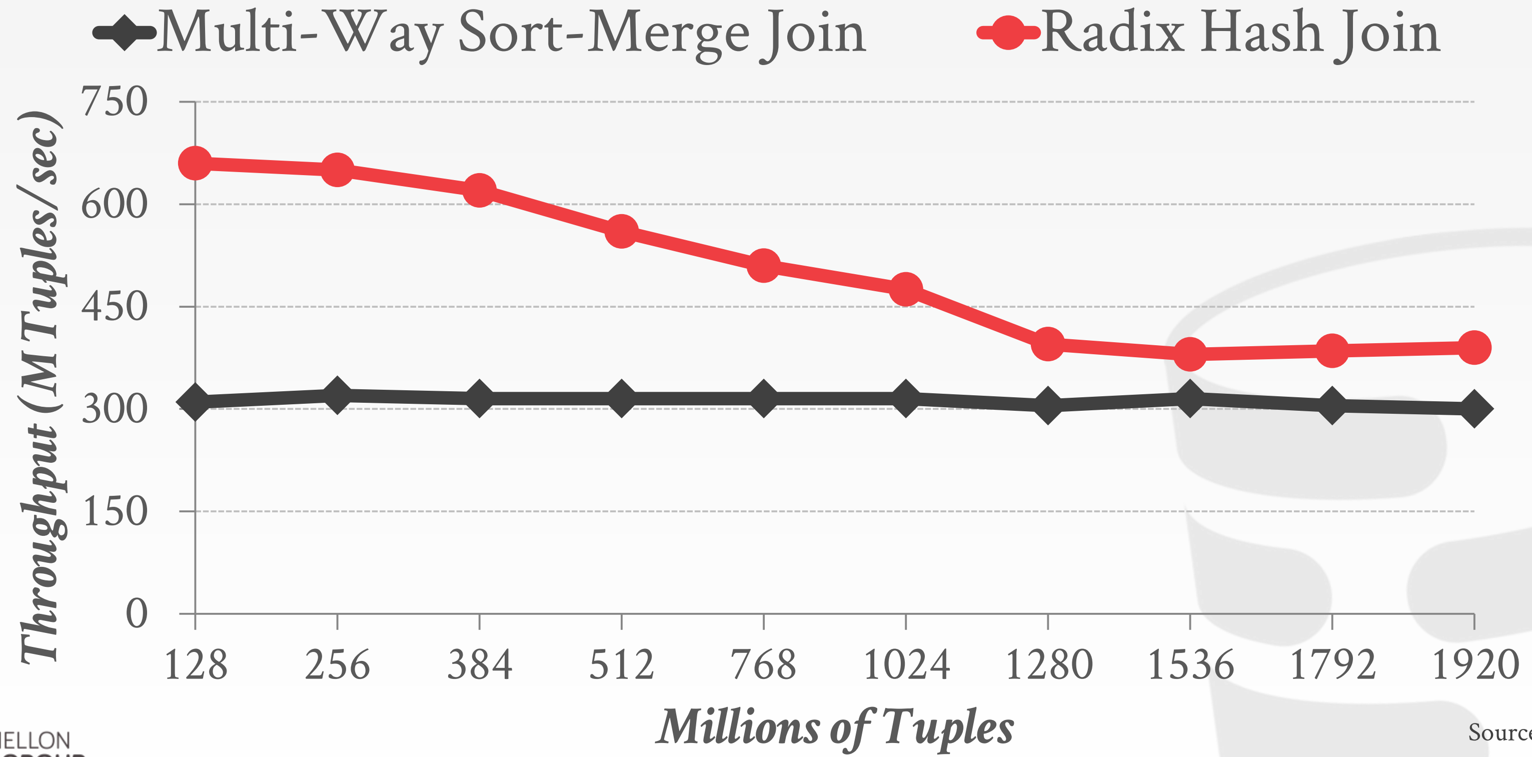
4 Socket Intel Xeon E4640 @ 2.4GHz
8 Cores with 2 Threads Per Core



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.

CODE REVIEWS

Each group will send a pull request to the CMU-DB master branch.

→ This will automatically run tests + coverage calculation.
→ PR must be able to merge cleanly into master branch.
→ Reviewing group will write comments on that request.
→ Add the URL to the Google spreadsheet and notify the reviewing team that it is ready.

Please be helpful and courteous.

GENERAL TIPS

The dev team should provide you with a summary of what files/functions the reviewing team should look at.

Review fewer than 400 lines of code at a time and only for at most 60 minutes.

Use a **checklist** to outline what kind of problems you are looking for.

Source: SmartBear

CHECKLIST – GENERAL

Does the code work?

Is all the code easily understood?

Is there any redundant or duplicate code?

Is the code as modular as possible?

Can any global variables be replaced?

Is there any commented out code?

Is it using proper debug log functions?

Source: Gareth Wilson

CHECKLIST – DOCUMENTATION

Do comments describe the intent of the code?

Are all functions commented?

Is any unusual behavior described?

Is the use of 3rd-party libraries documented?

Is there any incomplete code?

Source: Gareth Wilson

CHECKLIST – TESTING

Do tests exist and are they comprehensive?

Are the tests actually testing the feature?

Are they relying on hardcoded answers?

What is the code coverage?

Source: Gareth Wilson

NEXT CLASS

No lecture on Wednesday April 11th.

Reminder: First Code Review
April 11th @ 11:59pm