Parallel Sort-Merge Join

Recap

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**

Hash Join

- **Phase 1: Partition (optional)**
  - Divide the tuples of **R** and **S** into sets using a hash on the join key.
- **Phase 2: Build**
  - Scan relation **R** and create a hash table on join key.
- **Phase 3: Probe**
  - For each tuple in **S**, look up its join key in hash table for **R**. If a match is found, output combined tuple.
- Reference

Hashing vs. Sorting

- 1970s – Sorting (External Merge-Sort)
- 1980s – Hashing (Database Machines)
- 1990s – Equivalent
- 2000s – Hashing (For Unsorted Data)
- 2010s – Hashing (Partitioned vs. Non-Partitioned)
- 2020s – ???

Today's Agenda

- Background
- Sort Phase
- Merge Phase
- Evaluation
- Retrospective

Background

Single Instruction Multiple Data (SIMD)

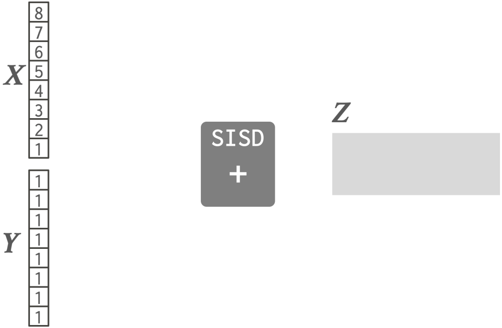
- A class of **CPU instructions** that allow the processor to perform the same operation on multiple data points simultaneously.
- All major ISAs have microarchitecture support SIMD operations.
- We first bring data into SIMD **registers** and then invoke the appropriate operation.
  - **x86:** MMX, SSE, SSE2, SSE3, SSE4, AVX
  - **PowerPC:** Altivec
  - **ARM:** NEON

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];
}
```

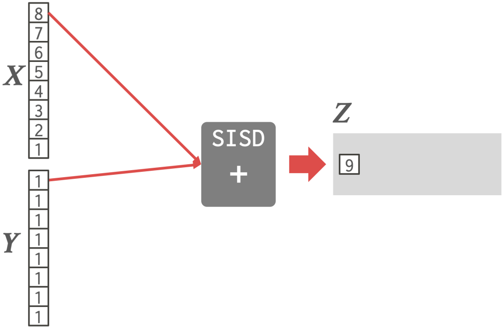


SIMD Example

$X + Y = Z$

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

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

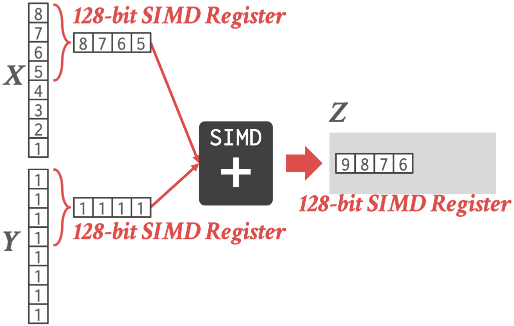


SIMD Example

$X + Y = Z$

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

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



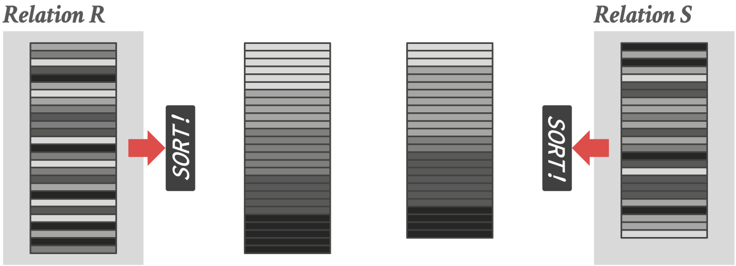
SIMD Trade-Offs

- **Advantages**
  - Significant performance gains and resource utilization if algorithm can be vectorized
- **Disadvantages**
  - Implementing an algorithm in SIMD is still mostly a manual process
  - SIMD may have restrictions on data alignment
  - Gathering data into SIMD registers and scattering to the correct location is tricky

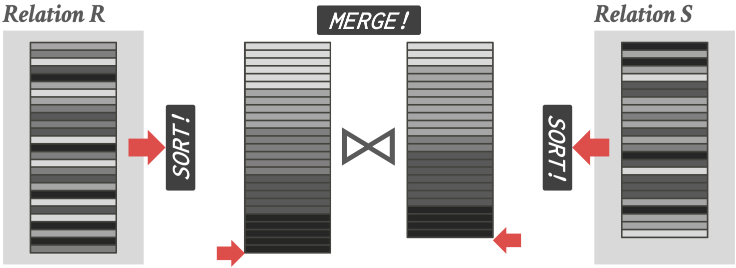
Sort-Merge Join

- **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



Sort-Merge Join



Parallel Sort-Merge Join

- Sorting is the most expensive part.
- **Warning:** We will be using **merge sort** for sorting the data.
- 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.
- These techniques also apply to the `ORDER BY` operator.
- Reference

Parallel Sort-Merge Join

- **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

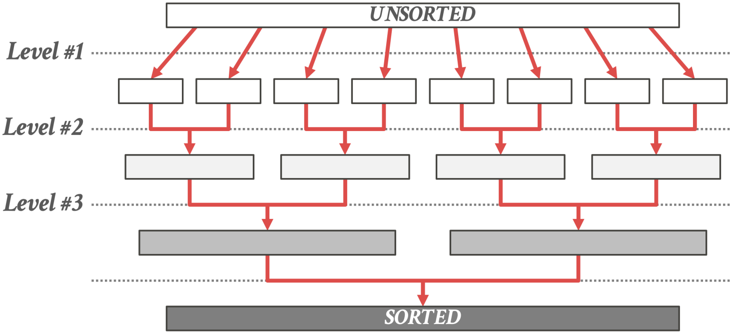
Sort Phase

- Create **runs** of sorted chunks of tuples for both input relations.
- It used to be that `quick-sort` was good enough in disk-centric DBMSs.
- 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 $\frac{1}{2}$ cache size.
- **Level 3: Out-of-Cache Sorting**
  - Used when the runs of Level 2 exceed the size of caches.

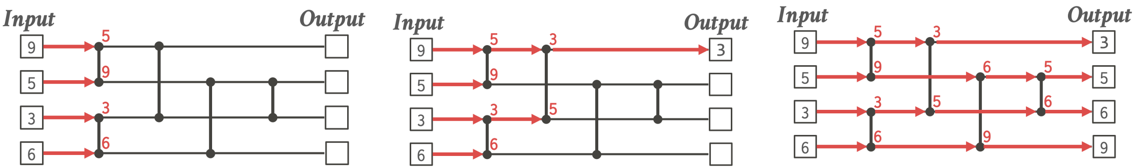
Cache-Conscious Sorting



Level 1 – Sorting Networks

- Abstract model for sorting keys.
  - 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.
- Reference

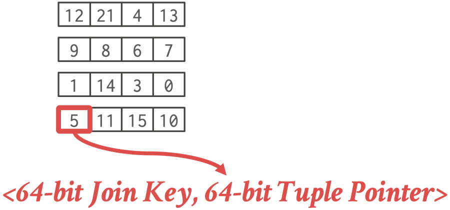
Level 1 – Sorting Networks



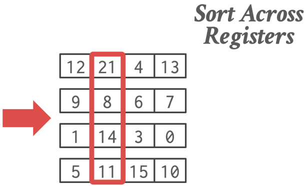
Level 1 – Sorting Networks

```
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



Level 1 – Sorting Networks



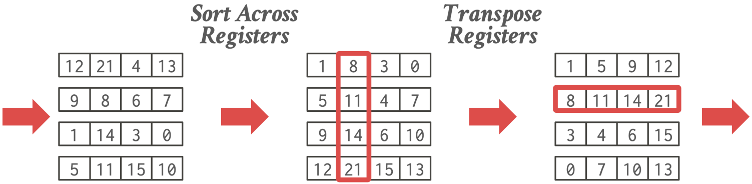
Instructions:
→ 4 **LOAD**

Level 1 – Sorting Networks

Instructions:
→ 4 **LOAD**

Instructions:
→ 10 **MIN/MAX**

Level 1 – Sorting Networks



Instructions:
→ 4 **LOAD**

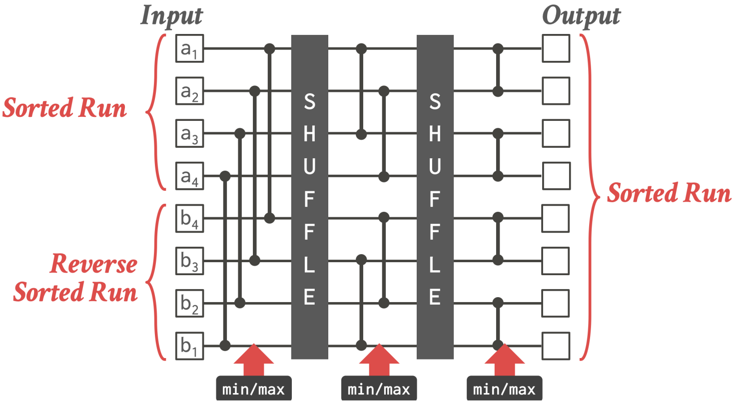
Instructions:
→ 10 **MIN/MAX**

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

Level 2 – Bitonic Merge Network

- 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 $\frac{1}{2}$ LLC size.
  - 2.25–3.5$\times$ speed-up over SISD implementation.

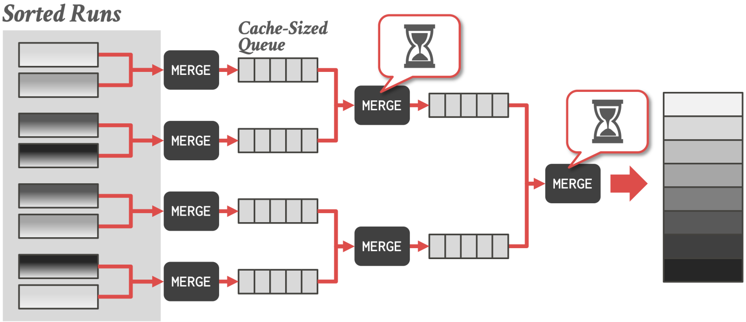
Level 2 – Bitonic Merge Network



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.
- A thread jumps around whenever work is available at an operator in the pipeline.

Level 3 – Multi-Way Merging



Merge Phase

Merge Phase

- Iterate through the outer table and inner table in lockstep and compare join keys.
- May need to backtrack if there are duplicates.
- Done in parallel at the different cores.

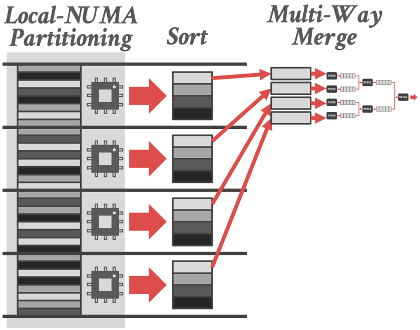
Sort-Merge Join Variants

- Multi-Way Sort-Merge (**M-WAY**)
- Multi-Pass Sort-Merge (**M-PASS**)
- Massively Parallel Sort-Merge (**MPSM**)

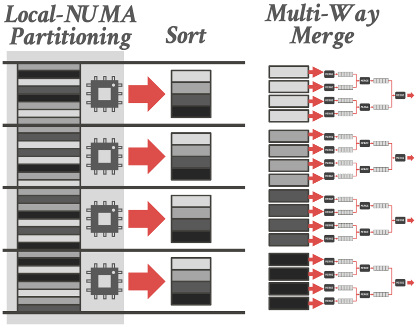
Multi-Way Sort-Merge

- **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.
- Reference

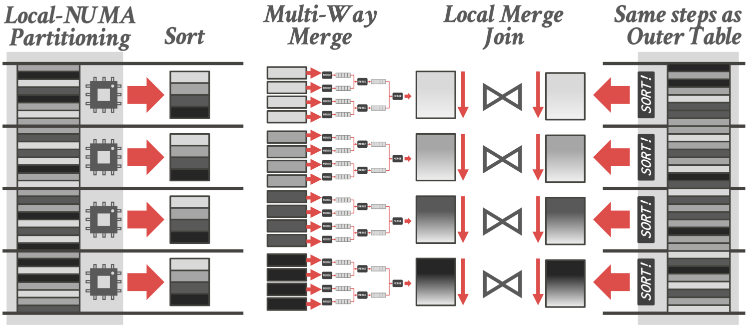
Multi-Way Sort-Merge



Multi-Way Sort-Merge



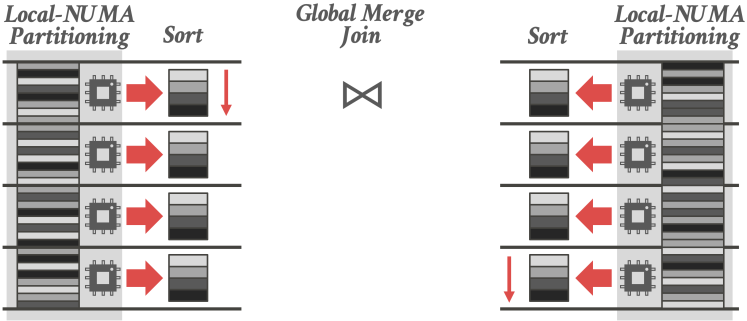
Multi-Way Sort-Merge



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.
- The **hardware prefetcher** masks the latency penalty of going over NUMA regions.

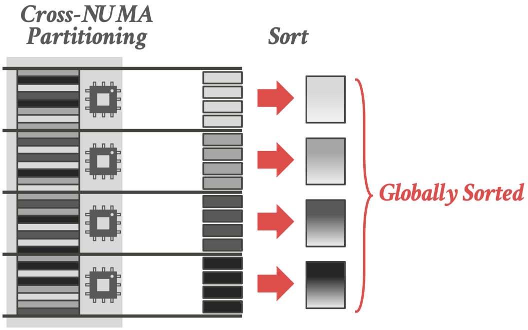
Multi-Pass Sort-Merge



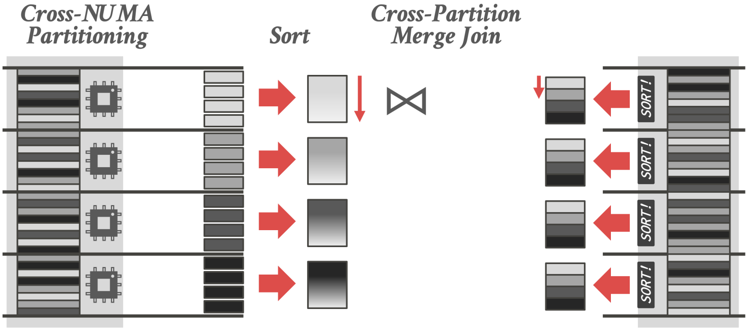
Massively Parallel Sort-Merge

- **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.
- Reference

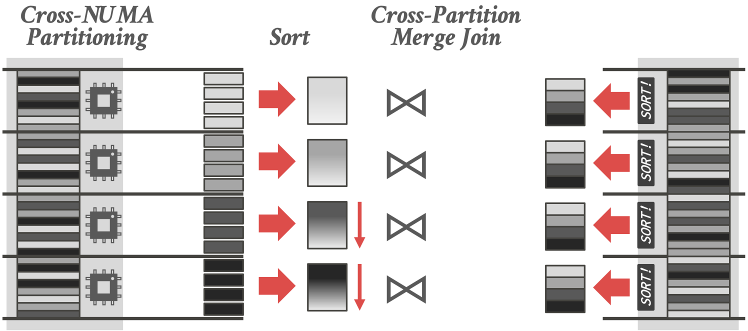
Massively Parallel Sort-Merge



Massively Parallel Sort-Merge



Massively Parallel Sort-Merge



Rules for Parallelization

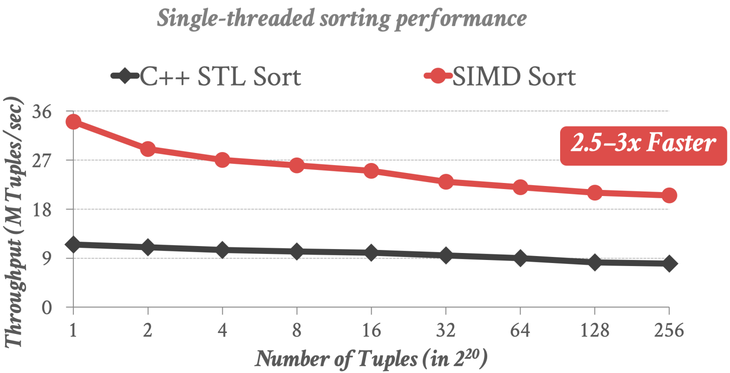
- **Rule 1:** No random writes to non-local memory
  - Chunk the data, redistribute, and then each core sorts/works on local data.
- **Rule 2:** Only perform sequential reads on non-local memory
  - This allows the hardware prefetcher to hide remote access latency.
- **Rule 3:** No core should ever wait for another
  - Avoid fine-grained latching or sync barriers.

Evaluation

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

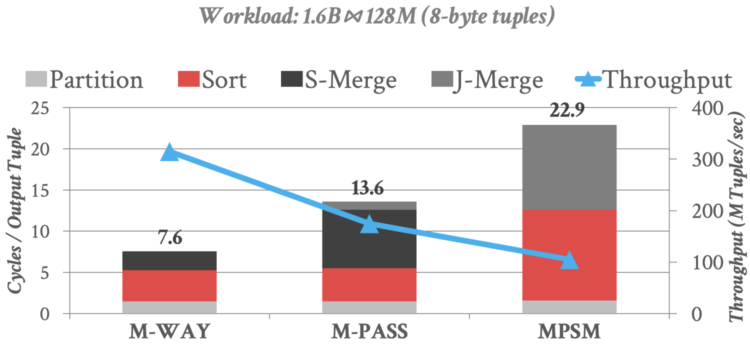
Raw Sorting Performance



Raw Sorting Performance

- STL's `sort` is a hybrid algorithm
- Quicksort in the beginning, and then switches over to Heapsort.

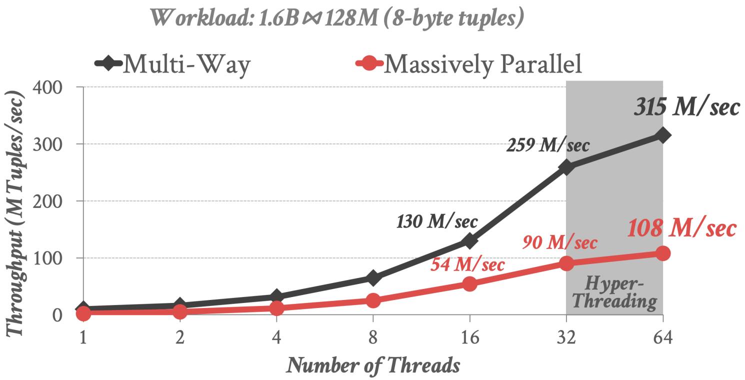
Comparison of Sort-Merge Joins



Comparison of Sort-Merge Joins

- Multi-way performs the best.
- Does more work to redistribute data.
- But it enables better cache locality $\implies$ higher number instructions per cycle.

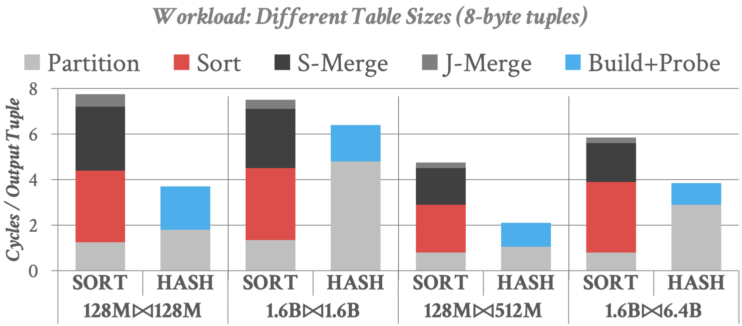
M-way Join vs. MPSM Join



M-way Join vs MPSM Join

- **M-WAY:** Extra instructions used for the multi-way sort in Level 3 pays off.
- **MPSM:** Overhead of reading data across NUMA regions hurts performance
- Hardware prefetcher is unable to help in this case.

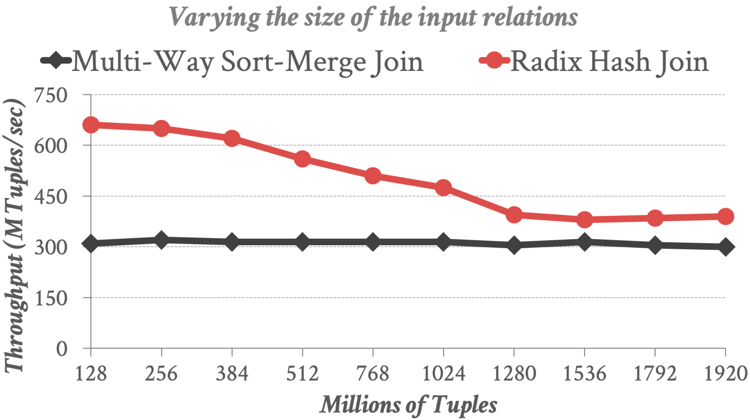
Sort-Merge Join vs. Hash Join



Sort-Merge Join vs. Hash Join

- Hash join works well in all settings.
- Radix partitioning overhead is high since the tables are large.
- No partitioning scheme should do even better.

Sort-Merge Join vs. Hash Join



Sort-Merge Join vs. Hash Join

- Radix hash needs more passes with larger tables.
- Performance gap shrinks due to partitioning overhead.
- No partitioning scheme should do even better.

Summary

- Both join algorithms are equally important.
- Every serious OLAP DBMS supports both.
- Sort-merge join is useful when the output needs to be sorted.

Retrospective

What did we learn

- You are tired of systems programming
- You are exhausted
- Let's take a step back and think about what happened

Lessons learned

- Systems programming is hard
- Become a better programmer through the study of database systems internals
- Going forth, you should have a good understanding how systems work

Big Ideas

- Database systems are awesome – but are not magic.
- Elegant abstractions are magic.
- Declarativity enables usability and performance.
- Building systems software is more than hacking
- There are recurring motifs in systems programming.
- CS has an intellectual history and you can contribute.

What Next?

- We have barely scratched the surface. Follow-on course: CS 8803 (DBMS Implementation - Part II)
  - Query Compilation + Vectorization
  - Query Optimization
  - Concurrency Control
  - Logging and Recovery Methods
- Stay in touch
  - Tell me when this course helps you out with future courses (or jobs!)
  - Ask me cool DBMS questions

Parting Thoughts

- You have surmounted several challenges in this course.
- You make it all worthwhile.
- Please share your feedback via CIOS.