Parallel Join Algorithms

Iztok Savnik, FAMNIT

November, 2025.

Outline

- Introduction
- Hash joins
- Parallelizing sort with SIMD
- Sort-Merge joins
- Parting thoughts

Classical join algorithms

- Traditional database query processing relies on three types of algorithms for join
  - **Index nested loops** join exploits an index on its inner input.
  - **Sort-merge** join exploits sorted inputs.
  - **Hash join** exploits differences in the sizes of the join inputs.

New hardware

Large amounts of main memory is available.

- DBMS can be implemented inside main memory.

Modern CPUs have super-scalar architecture.

- Operations for different types of data.
- Multiple ALUs for integer and floating point numbers.

CPUs have L1, L2 and L3 level caches.

- Numbers for recent (2025) server CPUs.
- Sizes of caches in Intel Core i9 are 64KB (per core), 1.5MB (per core) and 30MB (per socket)
- L1 is 100-150 times faster than RAM, L2 30-50 times faster, and L3 3-8 times faster than RAM.

New hardware

CPUs include instructions and storage for vector processing.

- Intel AVX-512
- Size of vector registers is 4 or 8.
- Instructions: predication, arithmetic, reading and storing vectors from/to RAM, permute, shuffling vectors in CPU, etc.

History

Most architectural ideas of new CPUs are taken from super-compters in 1970’s and 1980’s.

- 1960, Atlas, Manchester University
- 1964, CDC 6600, Seymour Cray
- 1976, Cray-1, 1985, Cray-2 (multiple processors, pipelines, various forms of parallelism)
- 1980, Connection Machine, MIT (65536 processors)

Most ideas used in new DBMSs were present in 1980’s.

- Parallel DBMSs
- 1981, Grace, Tokyo University
- 1985, Gamma Database Machine, Wisconsin University
- 1988, Bubba, University of Texas

Outline

- Introduction
- Hash joins
- Parallelizing sort with SIMD
- Sort-Merge joins
- Parting thoughts

Parallel hash joins

1) No partitioning hash join
2) Partitioned hash join
3) Radix hash join
4) Parallel radix hash join

The primary source for this section is the paper:

Balkesen, et al., Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware, ICDE, 2013.

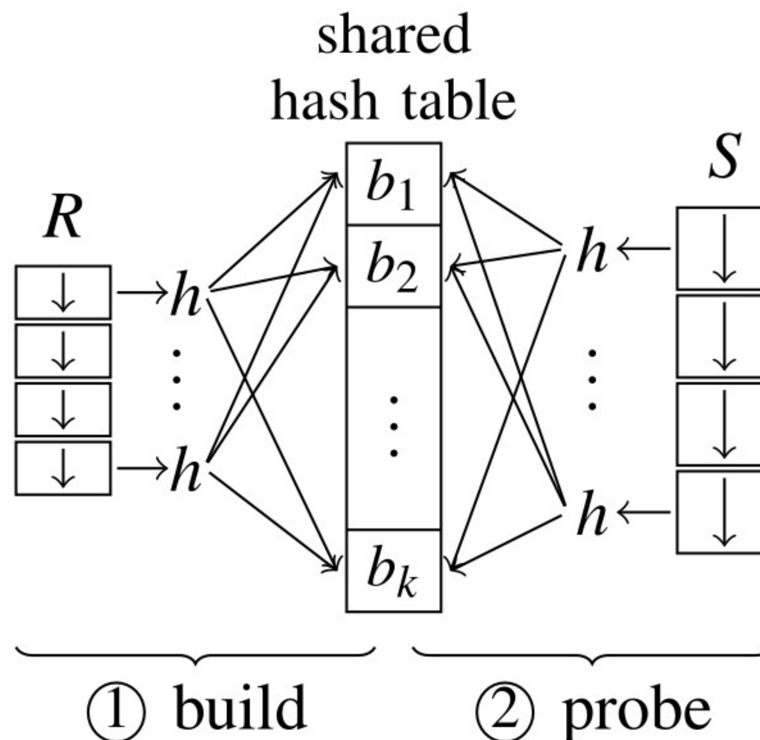
No Partitioning (Hash) Join

A direct parallel version of the canonical hash join

- It does not depend on any hardware-specific parameters

NPJ Algorithm (R⋈S)

- Both input relations are divided into equi-sized portions that are assigned to a number of worker threads

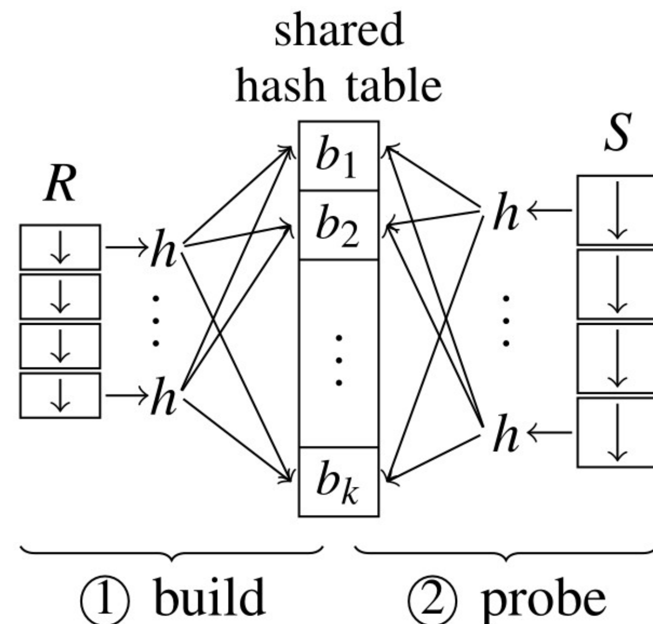


Blanas, et al., Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs, SIGMOD, 2011.

No Partitioning (Hash) Join

NPJ Algorithm (R⋈S)

- **Build** phase: R workers populate shared HT that all workers can access
- **Probe** phase: After synchron. barrier worker threads read from S and concurrently find matching join pairs from R



Concurrent insertions into HT must be synchronized!

- Latches (contention?)
- Compare-and-Swap (CAS)
- Probe: RO mode

Observations: Cache and TLB

Hashing results in cache misses (while accessing RAM)

- When the HT is larger than cache, almost every access to HT results in a cache miss.
- Therefore, partitioning HT into cache-sized chunks reduces cache misses and improves performance.

Shatdal, Kant, Naughton, Cache conscious algorithms for relational query processing, VLDB1994.
Boncz, Manegold, Kersten, Database Architecture Optimized for the new Bottleneck: Memory Access, VLDB1999.

Observations: Cache and TLB

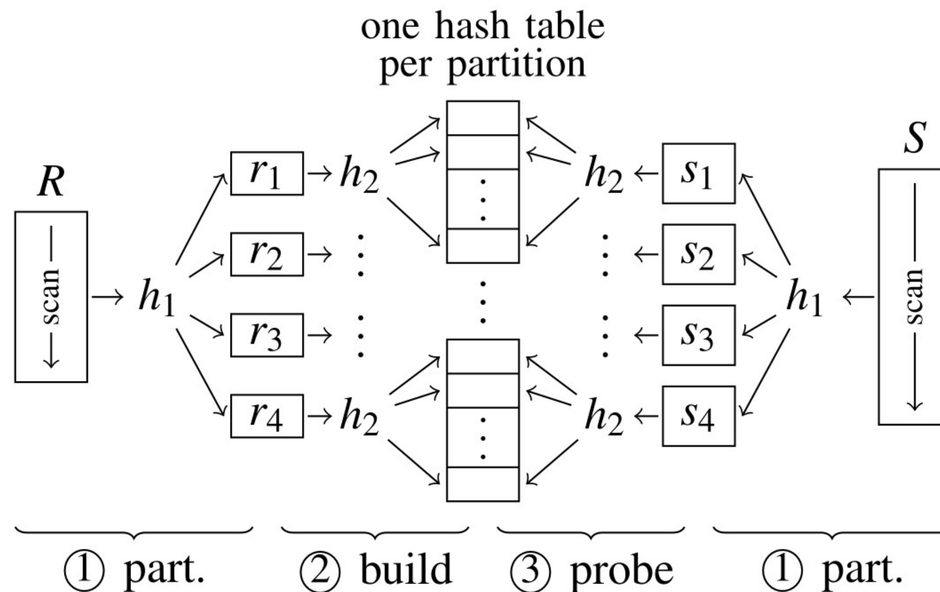
Effects of translation look-aside buffer (TLB) during partitioning phase

- TLB is a memory cache that stores the recent translations of virtual memory to physical memory (located in MMU)
- Data partitions typically reside on different memory pages with a separate entry for virtual memory mapping (TLB entry) required for each partition.
- # of TLB entries is **an upper bound** on # of partitions that can be created or accessed efficiently at the same time
- This led to **multi-pass partitioning**, now a standard component of radix join

Partitioned Hash Join

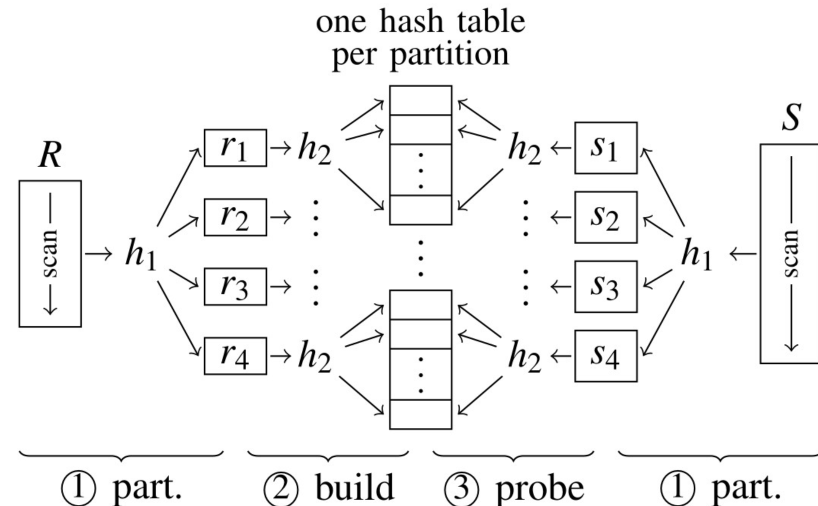
Grace hash join, disk version (University of Tokyo).

Joined tables are first partitioned. Then, the hash table is build and probing phase starts.



DeWitt, Gerber, Multiprocessor Hash-Based Join Algorithms, VLDB, 1985
Shatdal, Kant, Naughton, Cache conscious algorithms for relational query processing, VLDB1994.

Partitioned Hash Join



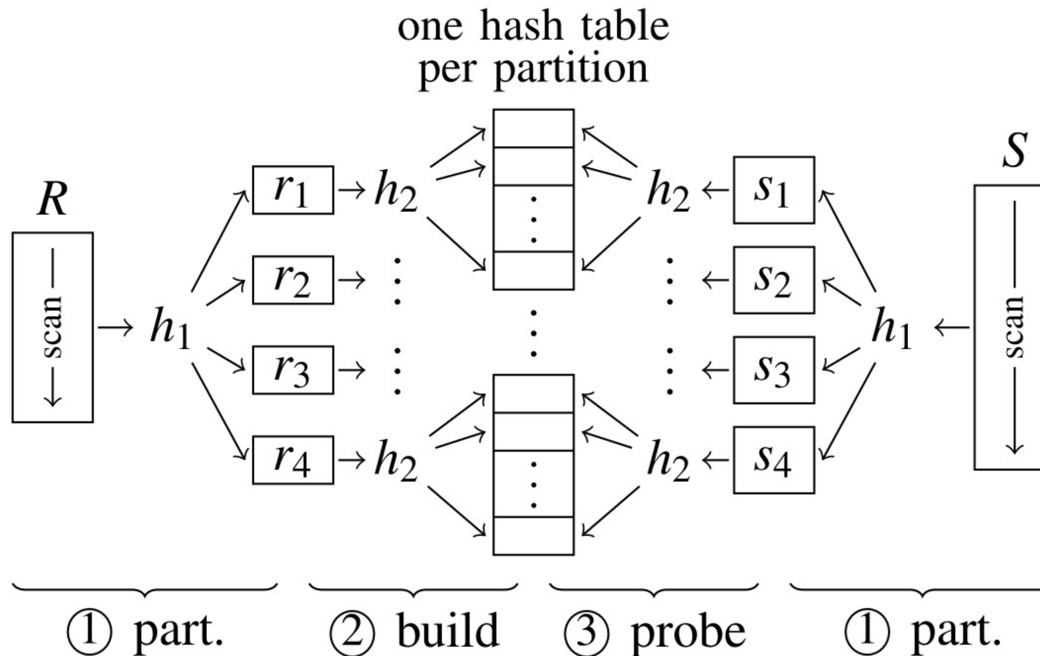
Build phase:

- Input rels R and S divided into partitions r_i and s_j.
- R and S tuples are divided using h1 on their join key values.
  - Consequences: $r_i \bowtie s_j = \varnothing$ $(i \neq j)$
- Partitions r_i of R are assigned to threats.
- A separate hash table is created for each r_i partition.
  - No synchronisation needed when creating hash tables.
- Hash tables for partitions r_i now **fit into CPU cache**!

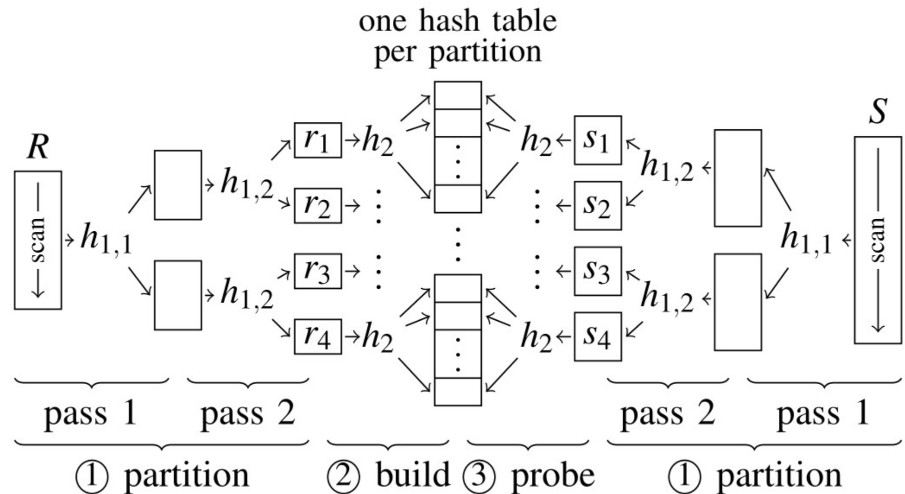
Partitioned Hash Join

Probe phase:

- s_j partitions are scanned and respective hash tables are probed for matching tuples using h2.
- One thread is used for one partition s_i.



Radix Hash Join



Excessive TLB misses can be avoided by partitioning input data in multiple passes

- ∀ pass j, partitions produced preceding pass j−1 are refined
- Partitioning **fan-out never exceeds the hardware limit** given by the number of TLB entries (often, large VM pages are set in OS!)
- Precomputing output memory ranges of each target partition by building histograms

Each pass looks at different set of bits from hash function h1 (radix partitioning).

- 2-3 passes sufficient to create partitions without problems with TLB limitations.
- Different targeted memory regions => no need for further synchronization!

Boncz, Manegold, Kersten, Database Architecture Optimized for the new Bottleneck: Memory Access, VLDB1999.

Radix Hash Join

Radix join R⋈S

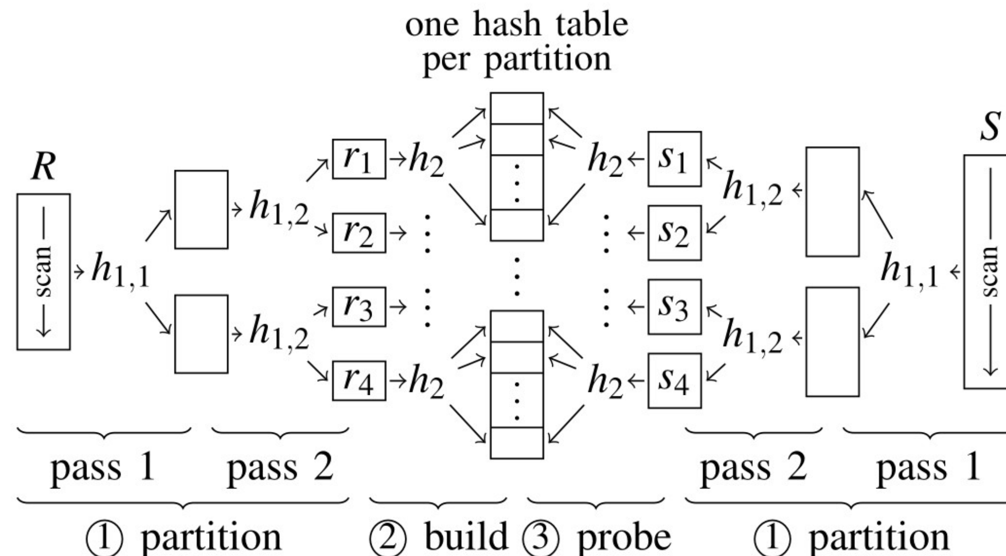
- Both inputs are partitioned using two-pass radix partitioning

Built phase:

- Hash tables are built over each r_i partition of input table R.

Probe phase:

- All s_i partitions are scanned and respective r_i partitions probed for join matches.



Radix Hash Join

Number of passes necessary is log(|R|) (or log(|S|))

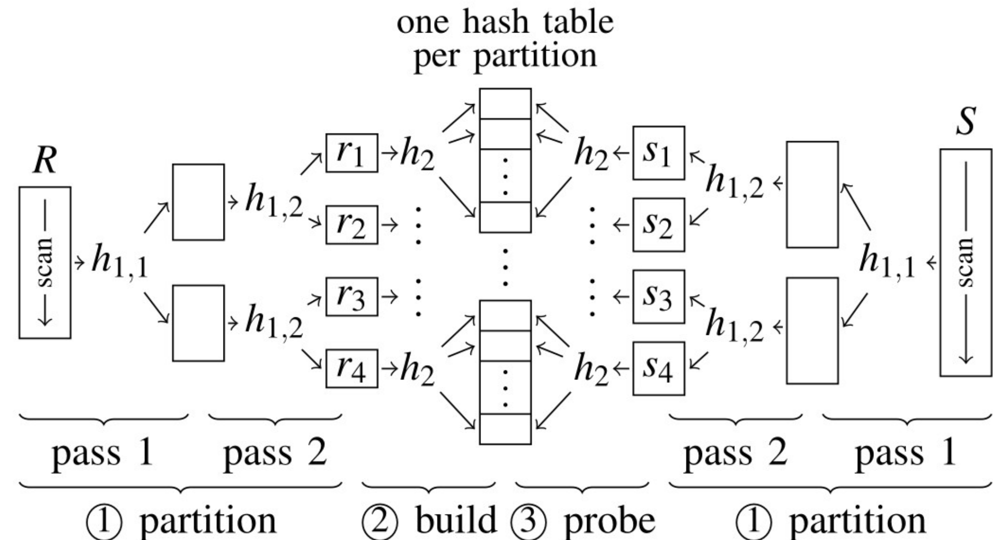
- log() basis is #TLB

Complexity of radix hash join is O((|R| + |S|) log |R|)

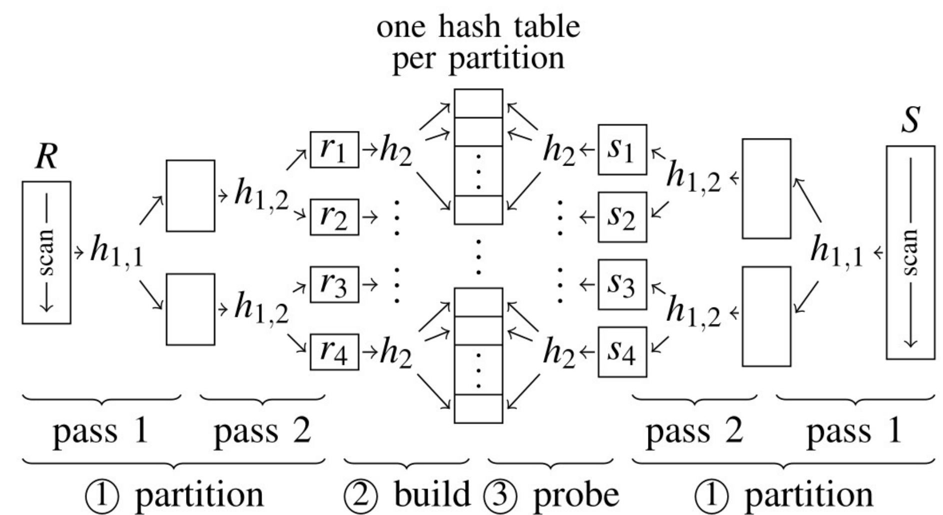
- R is typically smaller

Hardware parameters:

- Maximum fanout per radix pass is limited by #TLB;
- Resulting partition size is roughly the size of system's CPU cache.



Parallel RHJ



Mapping tasks to threads

- Dividing R and S into sub-relations that are assigned to individual threads.
- During the **first pass**, all threads create a shared set of partitions.
  - Number of partitions in this set is limited by hardware parameters (TLB) and typically small.
  - Partitions are accessed by potentially many execution threads, creating a contention problem.
- For each thread a **dedicated range** is reserved within each output partition.
  - Threads scan input relations and write to selected memory regions.

Parallel RHJ

Final partitions used in built and probe phases fit into CPU cache (L2)

- In building phase, threads are building the hash tables for each particular partition
  - **No contention** since there are different memory regions for different HTs
- In probing phase, many execution threads write the result table
  - **No contention** since each threat joins exactly two partitions (s_i and r_i)

Outline

- Introduction
- Hash joins
- Parallelizing sort with SIMD
- Sort-Merge joins
- Parting thoughts

Parallelizing sort with SIMD

Sorting Networks

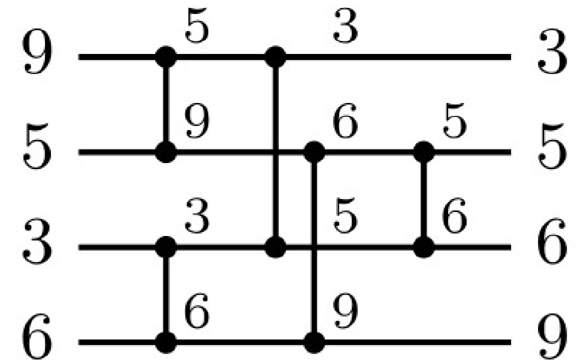
Merging Sorted Runs

AVX Sorting Algorithm

Balancing Computation & Bandwidth

Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Sorting Networks



Knuth's notation of sorting networks.

- Input ⟨9, 5, 3, 6⟩
- Comparator emits the smaller value at the top and larger value on the bottom.
- Traversing comparators from left to right.

Comparators can be implemented with min/max ops only.

- Five comparators from the above figure is converted into sequence of 10 min/max operations (no branching, fast!).

```
e = min (a, b)
f = max (a, b)
g = min (c, d)
h = max (c, d)
i = max (e, g)
j = min (f, h)
w = min (e, g)
x = min (i, j)
y = max (i, j)
z = max (f, h)
```


Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Sorting Networks with SIMD

Sorting networks are appealing because they can be accelerated through SIMD instructions.

- k=4: 4 SIMD vector registers with 4 elements
- Vectors are loaded in SIMD registers to represent **lanes**.
- Result need to be transposed to obtain sorted vectors.
  - Shuffle instructions are expensive!
- 4 vectors are sorted in the **time used for one** in CPU with SIMD
  - Speedup 2.7 because of shuffle operations.

Sorting Networks: SIMD

Input vectors:

```
a = [a1,a2,a3,a4]
b = [b1,b2,b3,b4]
c = [c1,c2,c3,c4]
d = [d1,d2,d3,d4]
```

Sorted columns of vectors:

```
x = [x1,x2,x3,x4]
y = [y1,y2,y3,y4]
w = [w1,w2,w3,w4]
z = [z1,z2,z3,z4]
```

Result: Transposed vectors

```
s1 = [x1,y1,w1,z1]
s2 = [x2,y2,w2,z2]
s3 = [x3,y3,w3,z3]
s4 = [x4,y4,w4,z4]
```

```
#include <immintrin.h>

inline void sort4x4(__m128& a, __m128& b,
                    __m128& c, __m128& d) {
  __m128 e = _mm_min_ps(a, b);
  __m128 f = _mm_max_ps(a, b);
  __m128 g = _mm_min_ps(c, d);
  __m128 h = _mm_max_ps(c, d);

  __m128 x = _mm_min_ps(e, g);
  __m128 i = _mm_max_ps(e, g);
  __m128 z = _mm_max_ps(f, h);
  __m128 j = _mm_min_ps(f, h);

  __m128 y = _mm_min_ps(i, j);
  __m128 w = _mm_max_ps(i, j);
}
```

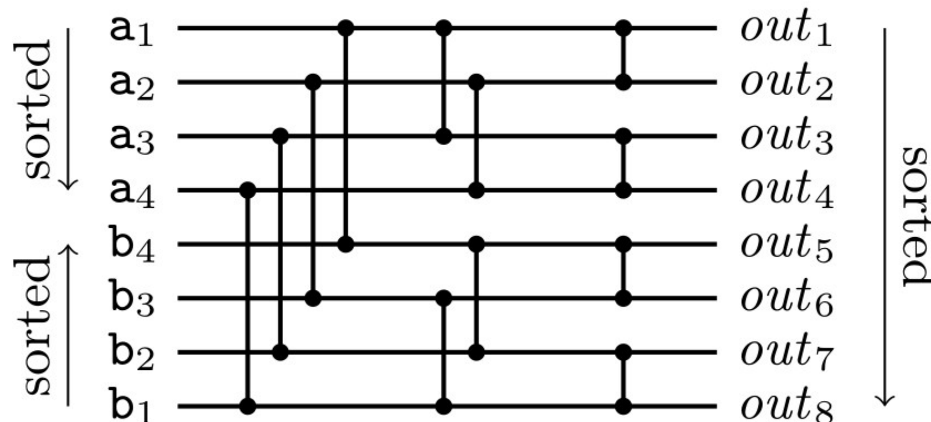
Merging Sorted Runs

Bitonic Merge Networks(*)

- Merging phase of SM join also benefits from SIMD acceleration.
- Networks that combine two pre-sorted inputs into sorted output.
  - This allows building larger networks.

A network that combines two input lists of size four.

- Using min and max operations, and shuffle operations in between comparators.

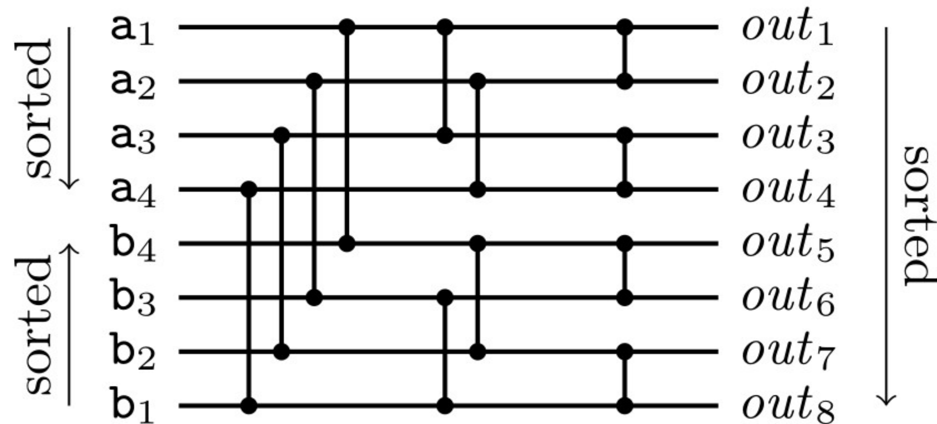


(*)Inoue, et al., AA-Sort_ A New Parallel Sorting Algorithm for Multi-Core SIMD Processors, PACT2007.

Merging Sorted Runs

Merging larger lists

- For larger input sizes, merge networks scale poorly.
  - $O(N \log^2 N)$
- Small merge networks can be used as a kernel within merging algorithm for larger lists.



Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Merging Sorted Runs

Merging larger lists

- There are two sorted runs as input.
- **bitonic_merge(a,b)** uses 2 registers of the size k=4.
- Merging algorithm uses vectors instead of scalar values (classical Alg. 1).
- Vector **a** with smaller values is emitted.
- A vector with the smaler head is fetched next.
  - There are at least k items that are smaller than the larger head!
  - In each iteration **a** includes the smallest values. (invariant)

Algorithm 1: Merging larger lists with help of bitonic merge kernel `bitonic_merge4()` ($k = 4$).

```
1  a ← fetch4(in_1); b ← fetch4(in_2);
2  repeat
3  |  ⟨a, b⟩ ← bitonic_merge4(a, b);
4  |  emit a to output;
5  |  if head(in_1) < head(in_2) then
6  |  |  a ← fetch4(in_1);
7  |  else
8  |  |  a ← fetch4(in_2);
9  until eof(in_1) or eof(in_2);
10 ⟨a, b⟩ ← bitonic_merge4(a, b);
11 emit4(a); emit4(b);
12 if eof(in_1) then
13 |  emit rest of in_2 to output;
14 else
15 |  emit rest of in_1 to output;
```

Sorting and Memory Hierarchy

Sorting using Advanced Vector Extensions (AVX) in CPUs

- AVX sorting algorithm (2.5 - 3 times faster then C++ sort)

Cache hierarchies in modern hardware require separating the overall sorting into several phases to optimize cache access.

- In-register sorting, with runs that fit into (SIMD) CPU registers
- In-cache sorting, where runs can still be held in a CPU-local cache
- Out-of-cache sorting, once runs exceed cache sizes

Sorting and Memory Hierarchy

In-register sorting

- Run-generation using sorting networks (initial sorted runs).

In-cache sorting

- Initial runs are merged until runs can no longer be contained within CPU caches.
- Multi-way merging using a hierarchy of Bitonic Merge Networks.

Out-of-Cache Sorting

- Continues merging until the data is fully sorted.
- Memory references will have to be fetched from off-chip memory.

Balancing Computation & Bandwidth

Accesses to off-chip memory makes sorting sensitive to the characteristics of the memory interface.

8-wide bitonic merge implementation requires 36 assembly instructions per 8 tuples being merged (29 CPU cycles)

- With a clock frequency of 2.4 GHz, this corresponds to a memory bandwidth 10.6 GB/s for a single merging thread
- This is more than existing memory interfaces support (+ 8 cores per CPU)
- Out-of-cache merging is thus **severely bound** by the memory bandwidth

Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

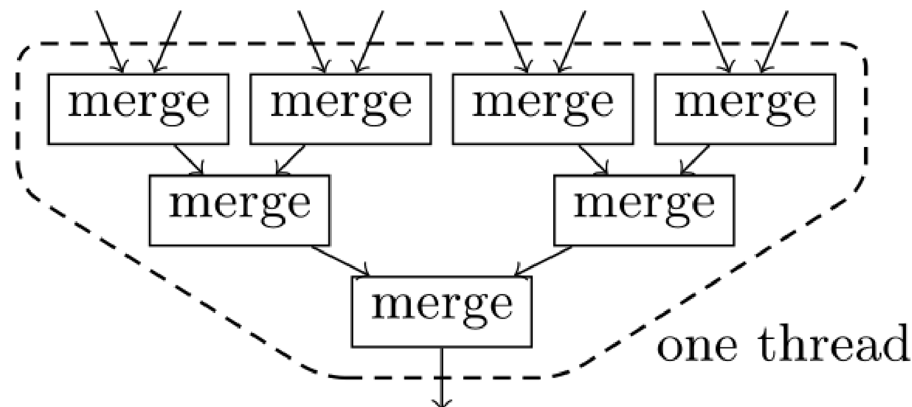
Balancing Computation & Bandwidth

Merging more than 2 runs at once (in case of multi-way merging) => Memory bandwidth demand reduced

- Saves round-trips to memory (memory bandwidth)
- Implemented with multiple two-way merge units

Two-way merge units are connected by **queues**

- FIFO queues are sized such that all queues together still fit into CPU cache



Outline

- Introduction
- Hash joins
- Parallelizing sort with SIMD
- <u>Sort-Merge joins</u>
- Parting thoughts

NUMA

Problem:

- Multi-processor system with RAM attached to bus
- One CPU can access memory at the given time
- CPUs starve memory access: **Von Neumann bottleneck**
- CPUs work much faster then data can be provided

Non-Uniform Memory Access

- Multi-processor systems have local memory
- The same address space is used for accessing local memory of other processors
- CPUs are connected by special interconnect

NUMA

NUMA evolved from symetric multiprocessing architectures (SMP)

- Access to local memory is faster than access to the memory of remote CPU
- **Latency:** local = ~70–90 ns and
  remote = ~130–170 ns
- **Bandwidth**: local = ~180–220 GB/s and
  remote = ~80–120 GB/s

NUMA **interconnect**

- AMD (Infinity Fabric), Intel UPI (Ultra Path Interconnect)

Sort-Merge joins

Sort-Merge Join

Multi-way Sort-Merge Join

Multi-pass Sort-Merge Join

Massively parallel Sort-Merge Join

Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Sort-Merge Join

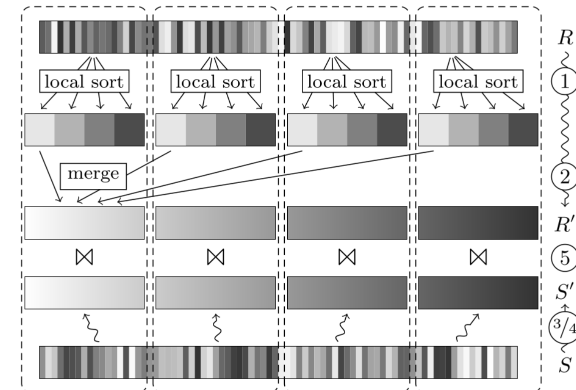
Sort R and S on their join keys and then **merge** sorted R' and S'.

- Sorting R and S is the dominant cost.
- (External) **merge sort** used for sort phase of SM join.
  - Merge sort is a basis of sort-merge join.
  - Merge sort can be combined with sort-merge join.
- **In-memory** sorting and merging in new architectures.
  - Sorting at different levels of memory hierarchy .
  - SIMD registers, CPU memory and RAM.

New hardware provides:

- SIMD registers in CPUs with the width of 4 or 8 words.
- Rich set of SIMD instructions that work on vectors.

Multi-way Sort-Merge Join



Highly parallel sort-merge join

- Relies on both data parallelism and thread parallelism.

Data parallelism

- Same operation on multiple data elements.
- SIMD extensions to standard ISA.

Thread parallelism

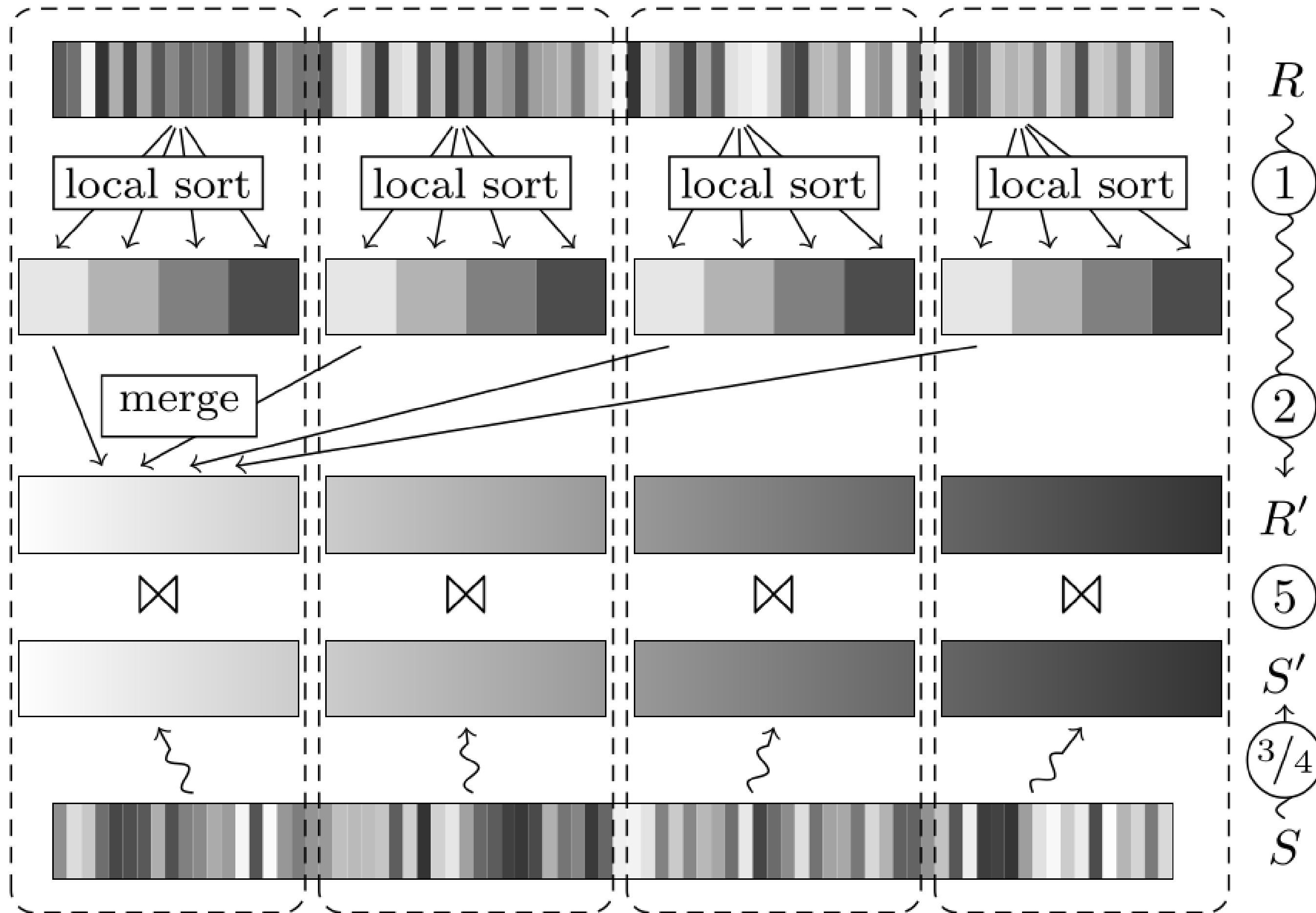
- Operator-level paralellism (also intra-operator) where different threads process different chunks of data for the same operator.

Carefully optimized toward NUMA.

- Favors data processing in NUMA local memory.
- Accesses NUMA remote memory if necessary.

Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Multi-way Sort-Merge Join



Multi-way Sort-Merge Join

General description of the algorithm

- Initially, input relations R and S are equally distributed across NUMA regions
- **1st phase**: each thread is assigned its NUMA-local chunk of R and all the threads range-partition their local chunks in parallel
  - Allowing threads in the subsequent phases to work independently without any synchronization.
  - Partitioning fan-out is usually on the order of the number of threads (64–128)
  - Then each local partition is sorted locally (NUMA-local) using the AVX sorting algorithm
  - Different threads can sort different partitions independently!

Multi-way Sort-Merge Join

General description of the algorithm

– **2nd phase:** multi-way merging all partitions from partitioned chunks simultaneously.
  - One core merges tuples from a given range of keys by selecting repeatedly the minimal elements among the current heads of each partition.
  - The only phase that requires shuffling data between NUMA regions (thru interconnect bandwidth).
  - Multi-way merging overlaps data transfer and merging.
    – They should be always in balance!
  - Outcome of this phase is a globally sorted copy of R, indicated as R’

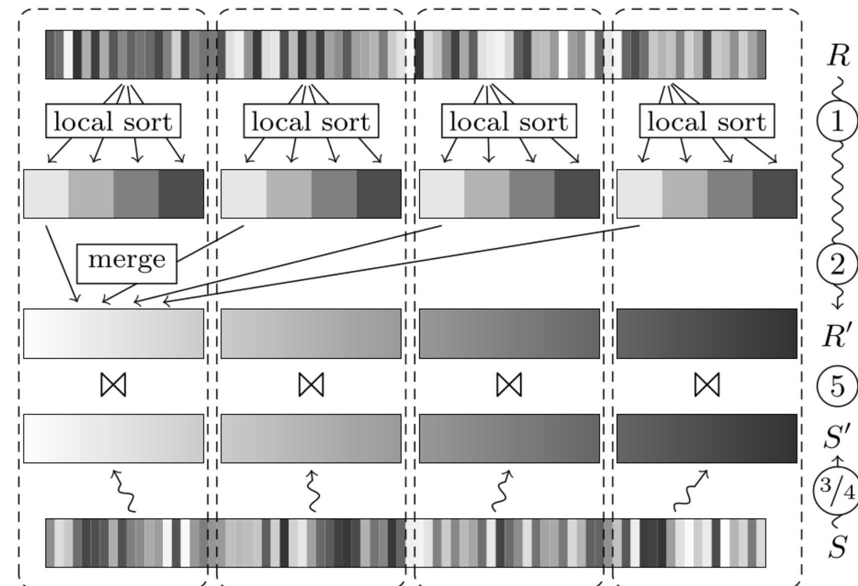
Multi-way Sort-Merge Join

- **3rd and 4th phase**:
  - The same steps 1 and 2 are applied to relation S!
- **5th phase**:
  - R’ and S’ are stored in the NUMA-local memory of each thread
  - Each thread concurrently evaluates the join between NUMA-local sorted runs using a single-pass merge join

Multi-Pass Sort-Merge Join

Algorithm differs from m-way SMJ solely in Phase 2.

- Instead of applying a multi-way merge for merging NUMA-remote runs, m-pass applies successive <u>two-way bitonic merging</u>.
- First iteration of merging of sorted runs is done as data is transferred to local memory
  - The number of runs reduces to 1/2 of the initial total number of runs.
- The rest of merging done in local memory, using multi-pass merging technique.



Massively parallel SMJ (MPSM)

There are two versions of Massively Parallel Sort-Merge join.

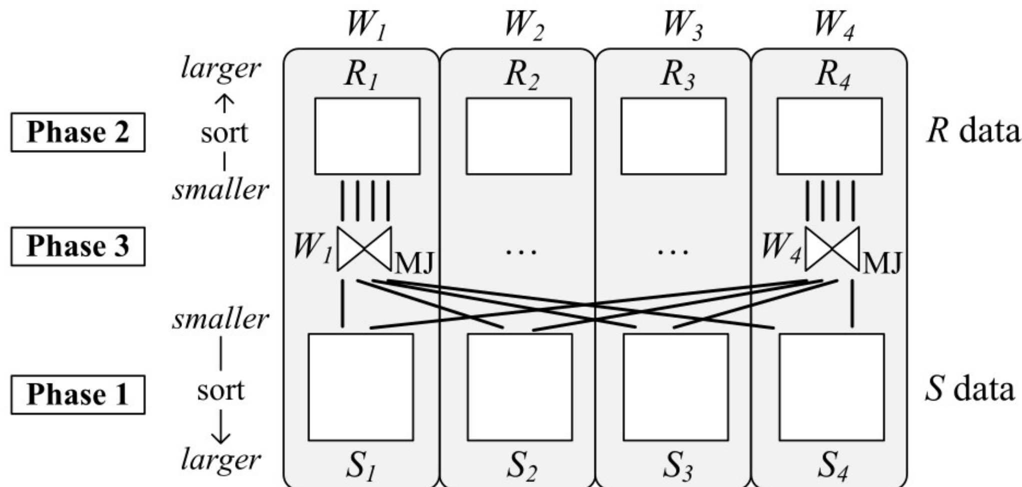
- **Basic MPSM** algorithm starts by generating sorted runs in parallel (B-MPSM).
  - These runs are not merged as doing so would heavily reduce the “parallelization power” of modern multi-core machines.
  - Instead, the sorted runs are simply joined in parallel.
- **Improved MPSM** version based on range partitioning of the input by join keys (P-MPSM).
  - Different ranges of R are assigned to different NUMA-regions/threads.

Albutiu, et al., Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems, VLDB, 2012.

Basic MPSM join

Input data is chunked into equally sized chunks among workers.

- Phase 1: Worker W_i is assigned a chunk R_i of R and S_i of S.
- Phase 2: Each worker generates sorted runs of the input data in parallel.
- Phase 3: Worker then processes only its own chunk of R but sequentially scans all chunks of S.



Comments on B-MPSM

- In phase 1 and 2, each worker thread handles (sorts) an equal share of R and S.
  - This phases do not require any synchronization.
  - They are performed in local NUMA memory (good for sort).
- Phase 3 requires reading non-local memory, but only sequentially!
  - Each worker joins its run of R with the sorted runs of S.
  - Sequential scans heavily profit from prefetching and cache locality and therefore do not affect performance significantly.
- B-MPSM algorithm is absolutely skew resistant!
- AVX algorithm for sorting can not be utilized because of large join keys and payloads.

Range-Partitioned MPSM

P-MPSM adds to B-MPSM a prologue phase to range partition R and assign R partitions to the workers.

Phase 1: Relation S is chunked into S_1-S_m and sorted locally.

Phase 2: Relation R is chunked into C_1-C_n and those chunks are range partitioned.

Histograms are used to ensure that the range partitions are balanced even for skewed data distributions.

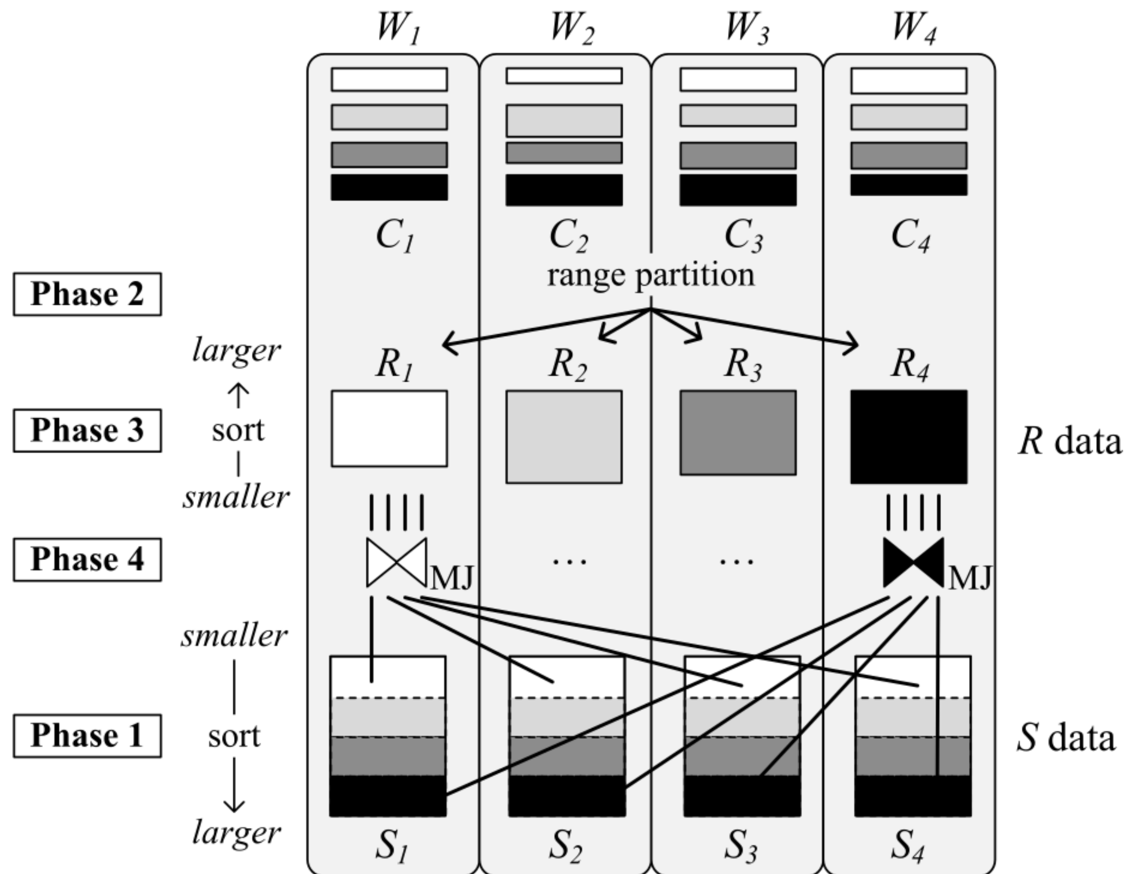
Each worker then scatters its input chunk to the partitions, guaranteeing synchronization freeness.

Thereby, R is partitioned into disjoint key ranges.

Phase 3: Each worker then sorts its range partition of R.

Phase 4: Merge joins its local R_i with all runs S_j from S.

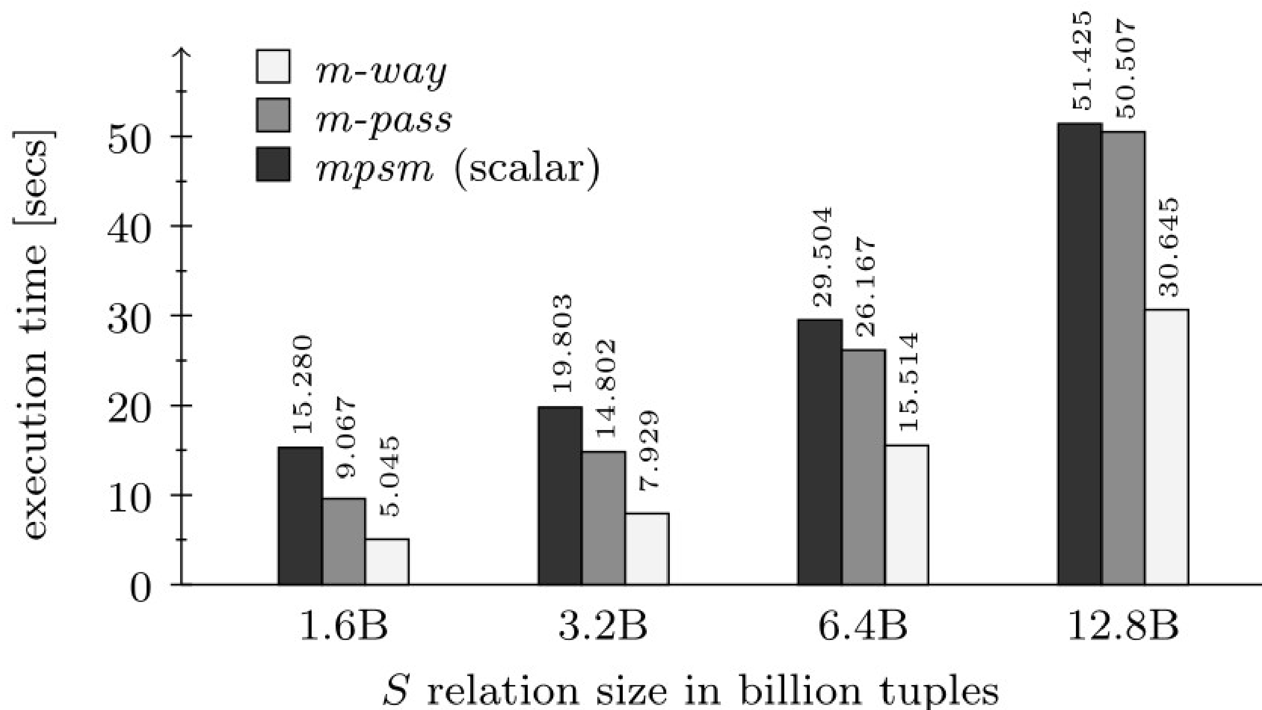
Range-Partitioned MPSM



Albutiu, et al., Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems, VLDB, 2012.

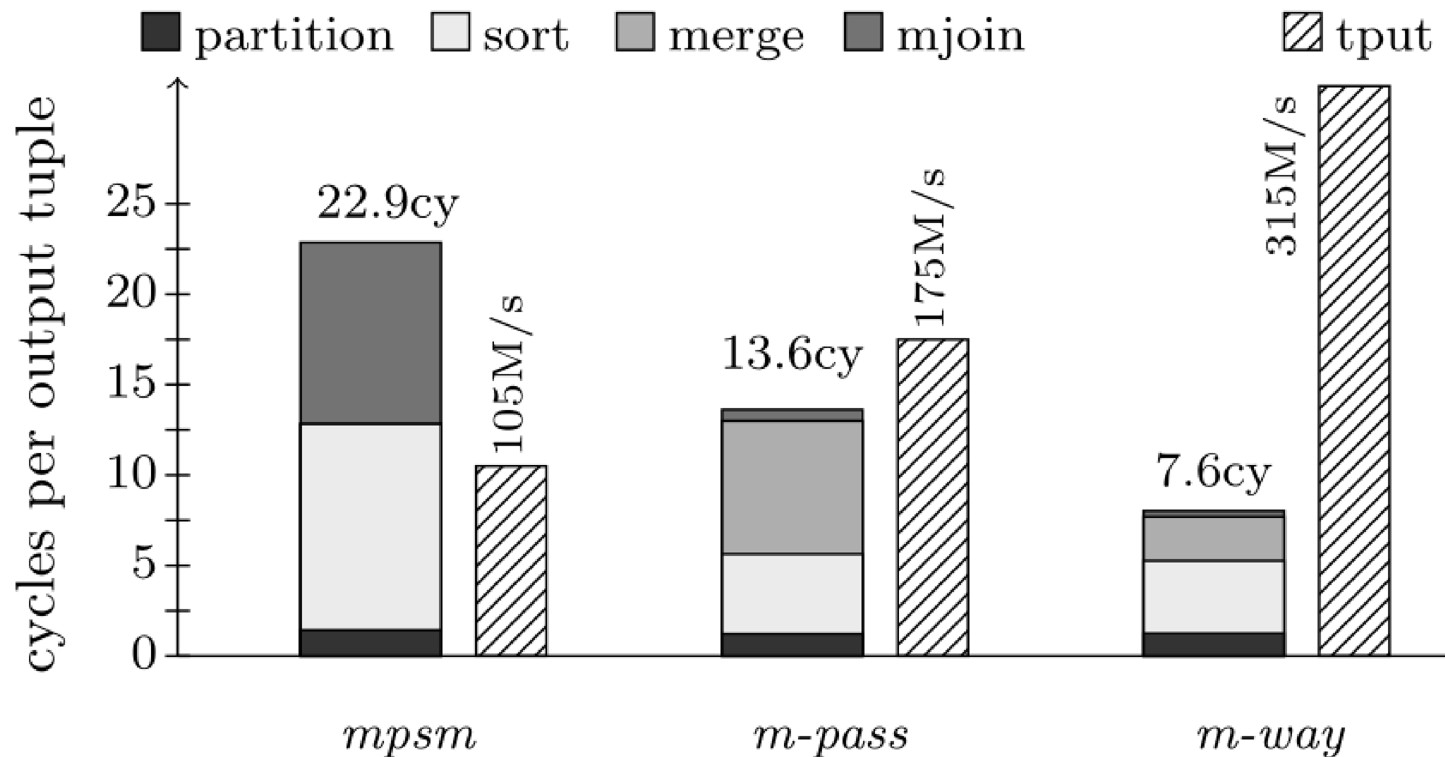
Experimental evaluation: Execution time of SM join algorithms

Intel Sandy Bridge, 256-bit AVX IS, 4 sockets, 8 hw cores, 16 thread contexts, L1=32KB, L2=256KB, L3=20MB, Cache line=64B, TLB1=64 entries, TLB2=512, mem=512GB (DDR3).



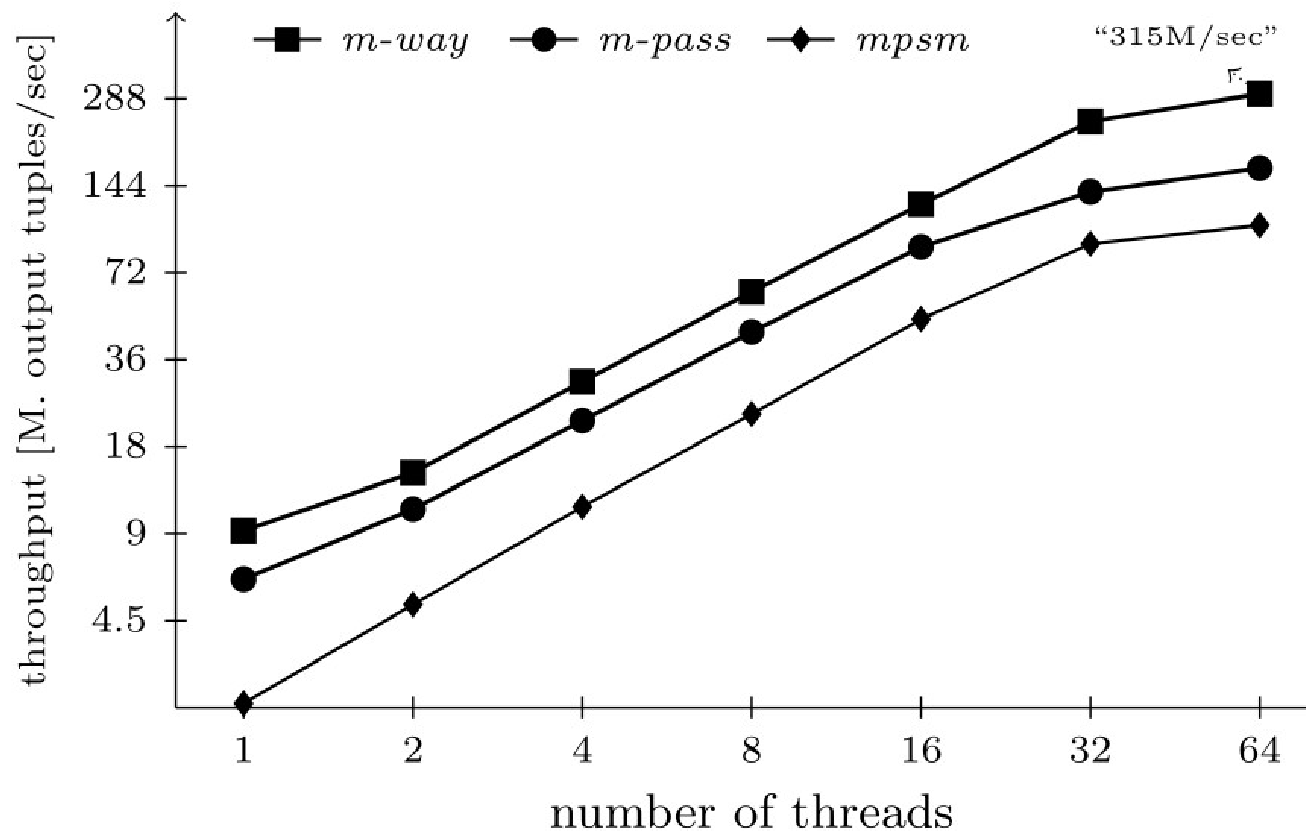
Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Experimental evaluation: Perform. breakdown for SM join algorithms



Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

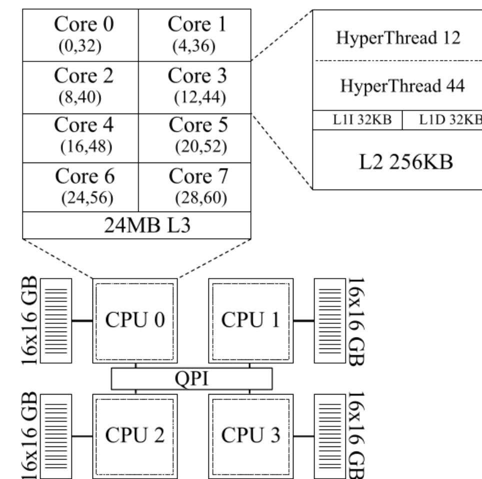
Experimental evaluation: Scalability of SM joins (11.92GBx11.92GB)

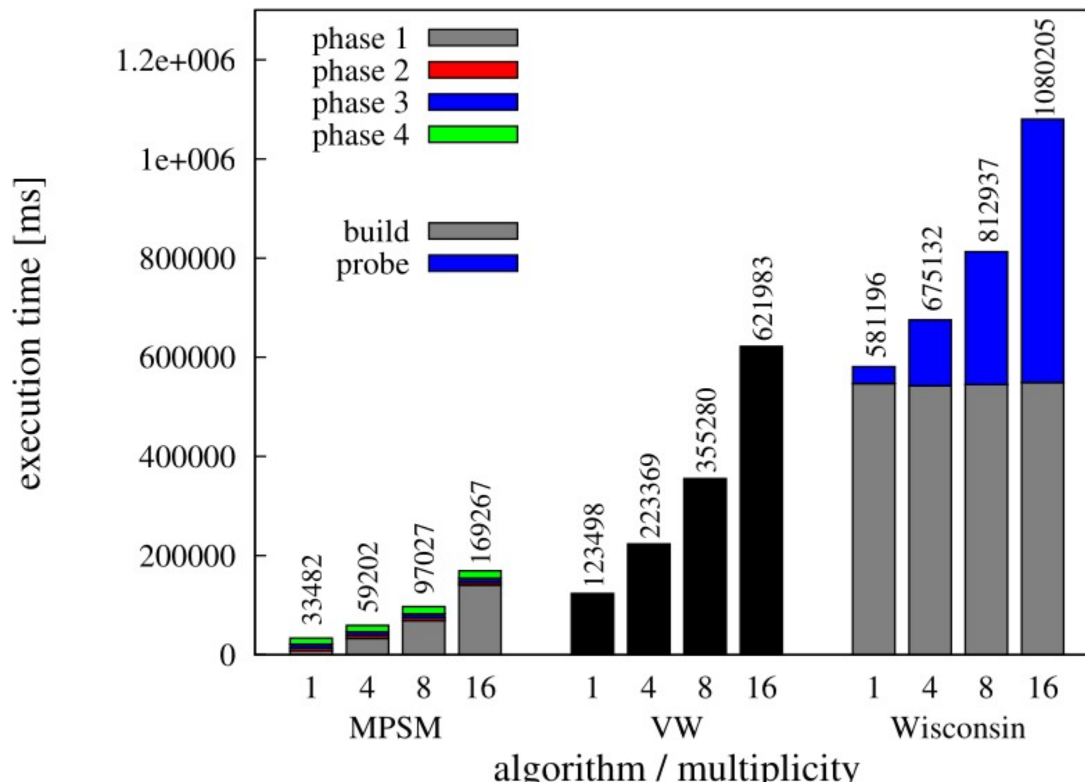


Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Experimental evaluation: MPSM vs. VW and Wisconsin

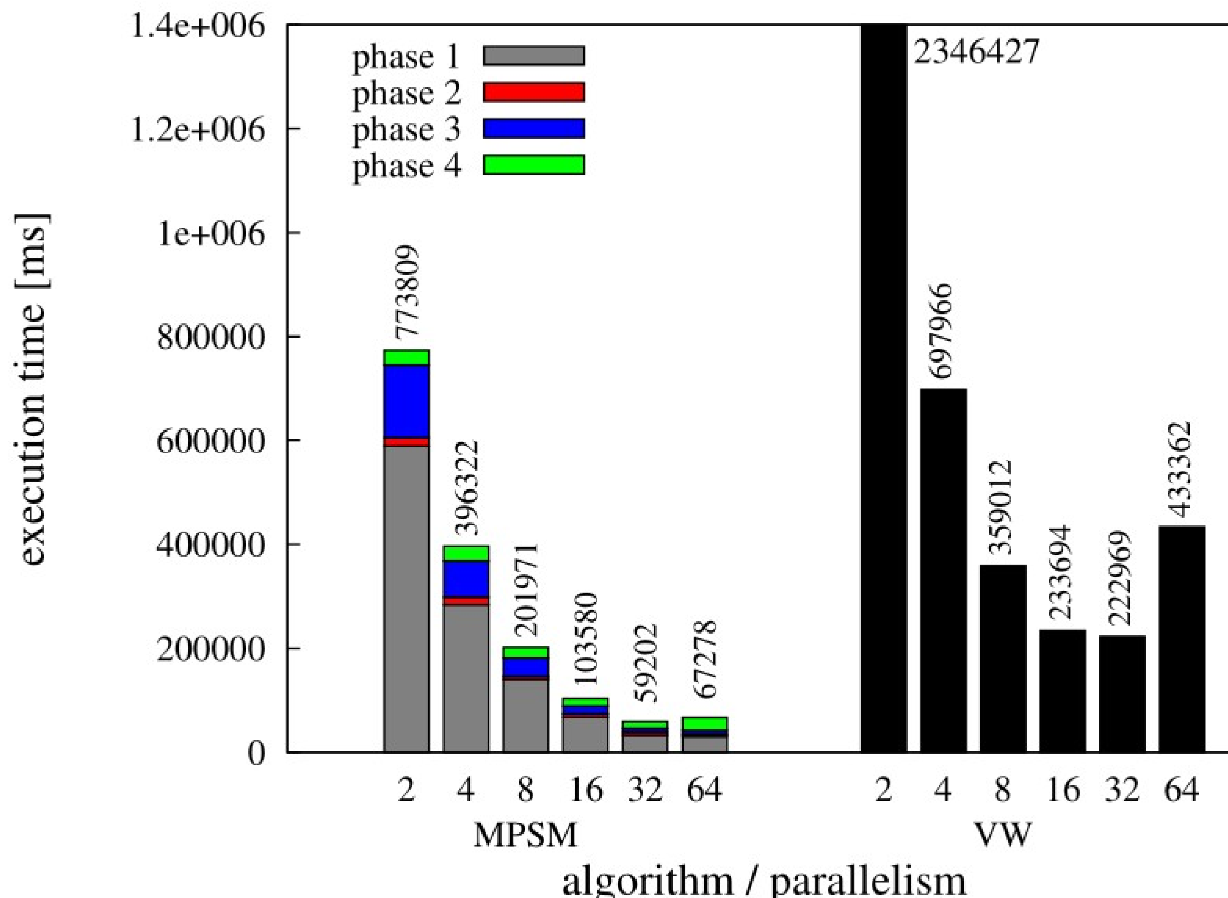
Linux, 1TB RAM, Xeon X7560 CPUs, 2.27GHz
4 CPUs, 8 cores (16 hardware contexts) each.
Datasets: |R|=1600M, S=1*|R| - 16*|R|, **uniform data**





Albutiu, et al., Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems, VLDB, 2012.

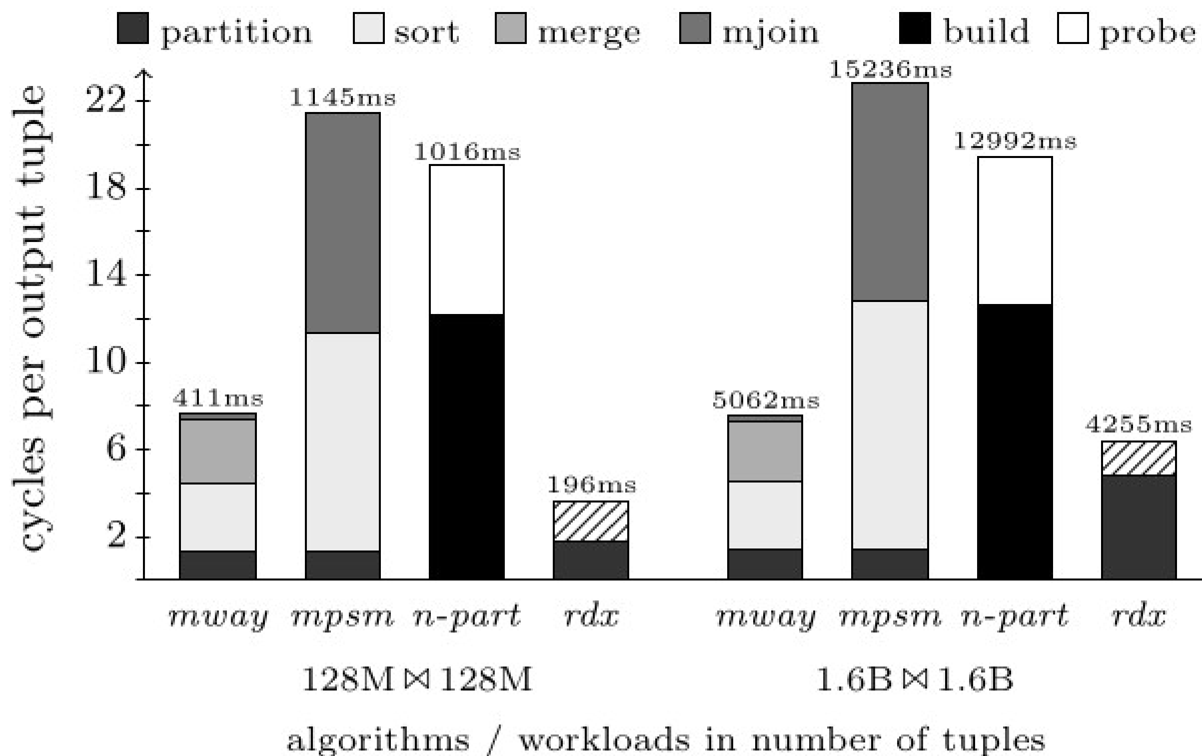
Experimental evaluation (MPSM): Scalability in number of cores



Albutiu, et al., Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems, VLDB, 2012.

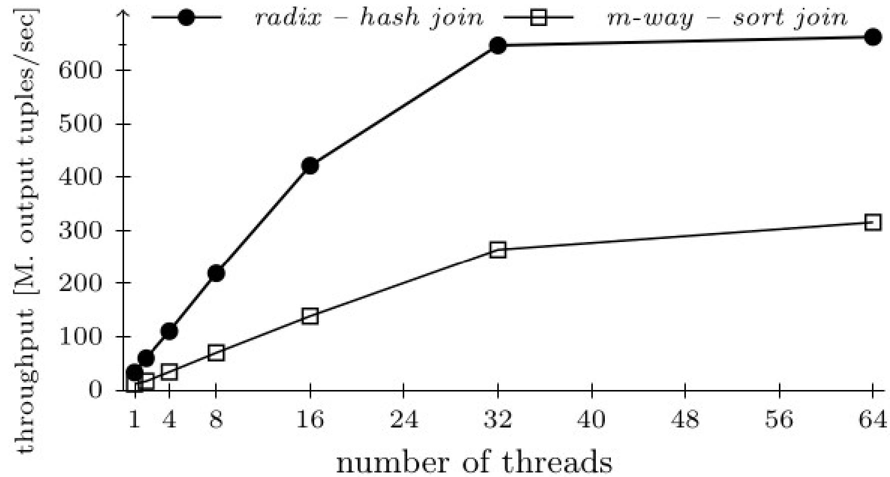
Experimental evaluation: Sort vs. hash join comparison

Intel Sandy Bridge, 256-bit AVX IS, 4 sockets, 8 hw cores, 16 thread contexts, L1=32KB, L2=256KB, L3=20MB, Cache line=64B, TLB1=64 entries, TLB2=512, mem=512GB (DDR3).

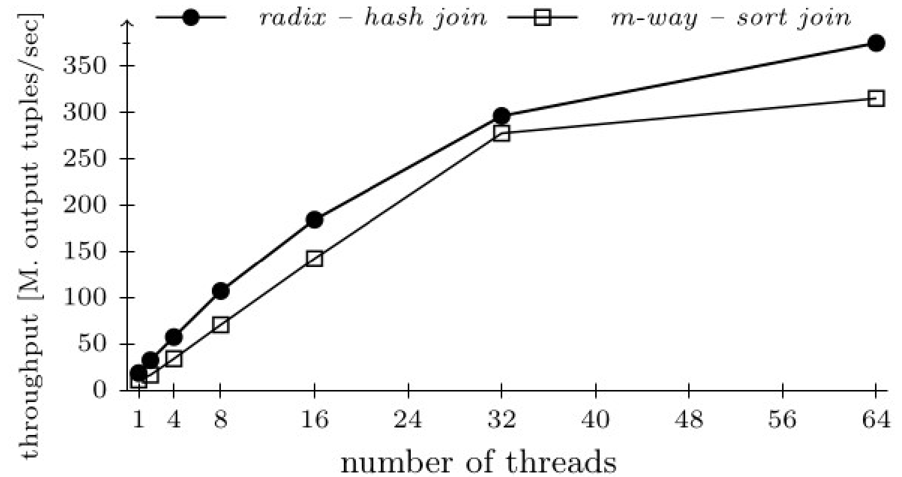


Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Experimental evaluation: Scalability of sort vs. hash join



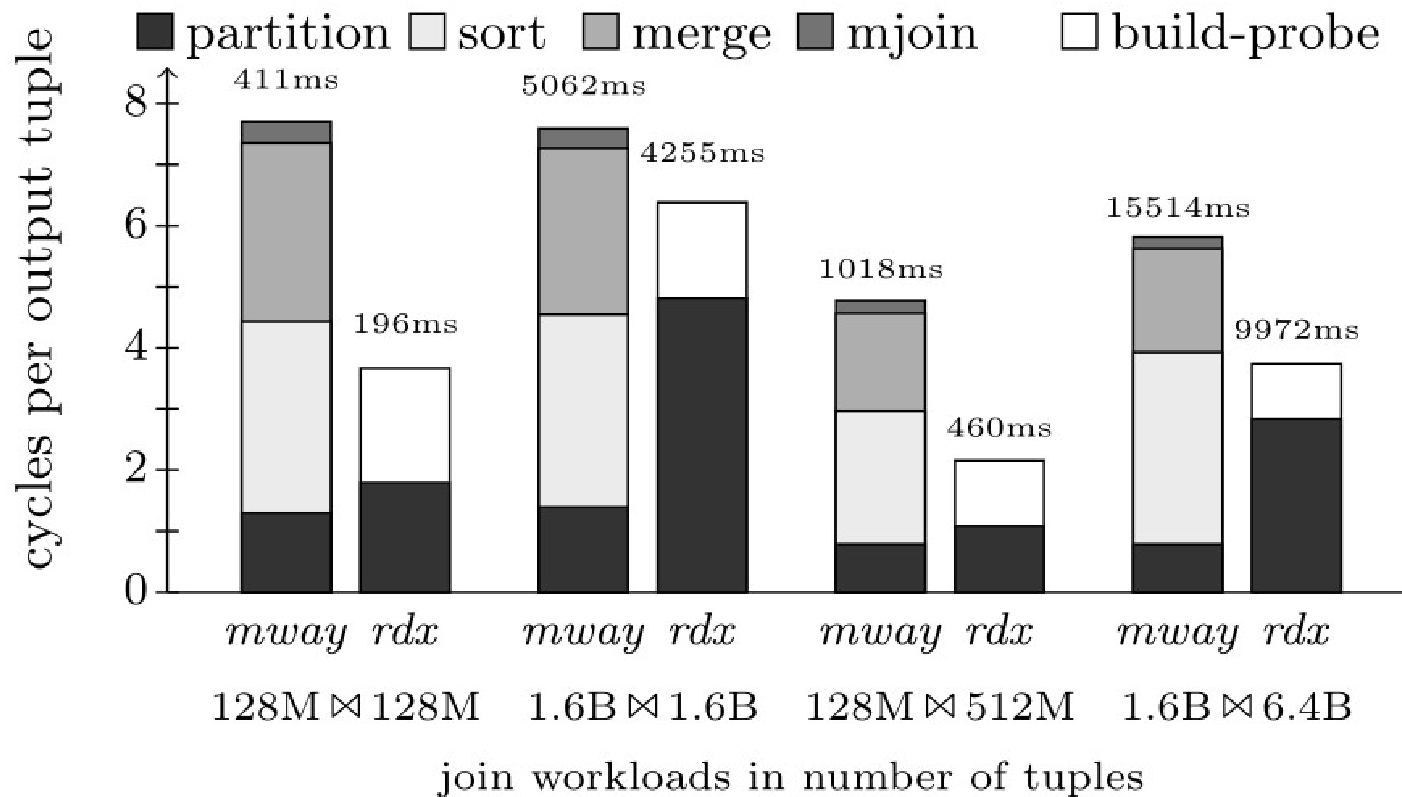
(a) 977 MiB ⋈ 977 MiB (128 million 8-byte tuples)



(b) 11.92 GiB ⋈ 11.92 GiB (1.6 billion 8-byte tuples)

Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Experimental evaluation: Best sort vs. best hash join algorithms



Balkesen, et al., Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited, VLDB, 2013.

Outline

- Introduction
- Hash joins
- Parallelizing sort with SIMD
- Sort-Merge joins
- Parting thoughts

Parting thoughts

Hash join and Sort-Merge join are prevailing classes of join algorithms in new DBMSs.

A consensus on sort vs. hash join has not been reached yet.

Novel CPUs are (almost) stand-alone computers.

Multiple "computers" (cores, CPUs) in a computer.

DB research is one of the frontiers in CPU development.