Multi-Core, Main-Memory Joins: Sort *vs.* Hash Revisited

Cagri Balkesen, Gustavo Alonso, Jens Teubner, M. Tamer Özsu

Presented by William Qian

2020 April 16

6.886 Spring 2020

Overview

1. Background
2. Parallel sort-merge joins
3. Parallel hash joins
4. Evaluation
5. Discussion

Sort-merge joins

```
SELECT * FROM R, S WHERE F(R.key) = G(S.key)
```

Sort phase: sort R's keys according to F and S's keys according to G
Merge phase: mergesort-style matching of keys from R and S

- Works for any comparator
- Requires sorting
- Sorting is known to be parallelizable
- Merging is much harder to parallelize

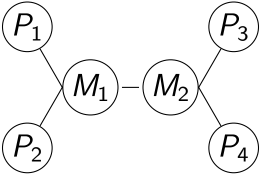
Hash joins

```
SELECT * FROM R, S WHERE F(R.key) = G(S.key)
```

Build phase: create base hashtable H from applying F to keys of R
Probe phase: apply G to keys in S and find matches in H to join

- Embarrassingly parallel
- Requires lots of memory to store H
- Frequently incurs cache misses for large tables
- Requires equijoins (which are fairly common)

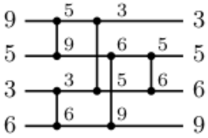
Non-uniform memory access



- P_1 can access M_1 easily, but M_2 is a little more costly
- Lots of data movement to "farther" memory increases bandwidth congestion

Parallel run-generation

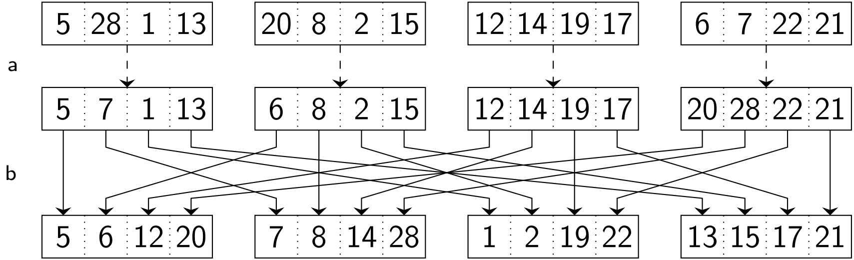
Sorting networks



- Few data dependencies
- No branching
- Only sorts across vectors

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

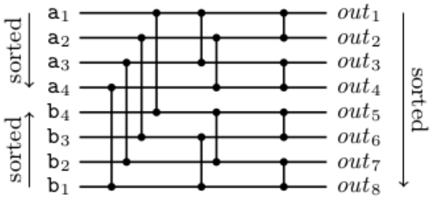
Parallel run-generation



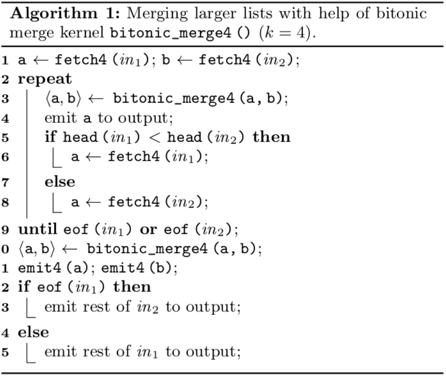
- Sorting network in (a) generates vectors sorted across positions
- Shuffling in (b) transposes vectors so that each vector is sorted

Parallel merge

Bitonic merge networks



- Scales poorly
- Used as a kernel sort



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

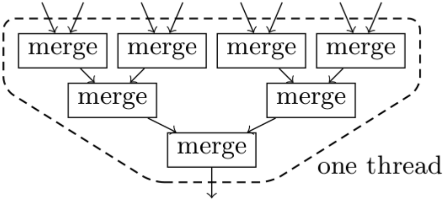
```
a ← fetch4 (in_1); b ← fetch4 (in_2);
repeat
    ⟨a, b⟩ ← bitonic_merge4 (a, b);
    emit a to output;
    if head (in_1) < head (in_2) then
        a ← fetch4 (in_1);
    else
        a ← fetch4 (in_2);
until eof (in_1) or eof (in_2);
⟨a, b⟩ ← bitonic_merge4 (a, b);
emit4 (a); emit4 (b);
if eof (in_1) then
    emit rest of in_2 to output;
else
    emit rest of in_1 to output;
```

- Adds branch predictions
- Avoids scalar-vector register movement

Out-of-cache sorting

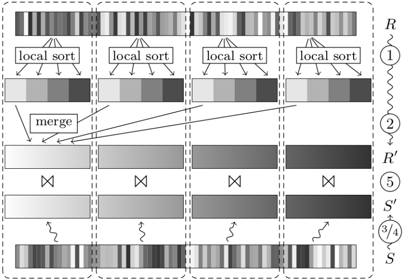
Multi-way merging

- Two-way merge units connected with FIFO buffers
- External memory bandwidth only at front of multi-way merge tree
- Helps combat NUMA



Sort-merge: choose your fighter

m-way



- NUMA-local partitions
- Tables sorted symmetrically
- Multiway merging for
- Single-pass merge join

m-pass

- Similar to m-pass
- Two-way bitonic merging instead of multiway merging

mpsm

- Globally partitions & sorts one table
- Partially sorts the other table
- Keys in S are a subset of keys in R
- First table merged w/ NUMA remote runs of second table

Radix partitioning

Problem: large hashtables result in many cache misses
Solution: radix partitioning

```
foreach input tuple $t$ do
    $k \leftarrow \mathrm{hash}(t)$;
    $p[k][\mathrm{pos}[k]] = t$;        // copy $t$ to target partition $k$
    $\mathrm{pos}[k]{+}{+}$;
```

- Moves tuples to destination partitions (pages)
- Reduces TLB miss effects during partitioning
- TLB size limits the fan-out of the partitioning step

Software-managed buffers

Problem: radix partitioning is limited by TLB sizes
Solution: buffer writes in cache

```
foreach input tuple $t$ do
    $k \leftarrow \mathrm{hash}(t)$;
    $\mathrm{buf}[k][\mathrm{pos}[k] \bmod N] = t$;        // copy $t$ to buffer
    $\mathrm{pos}[k]{+}{+}$;
    if $pos[k] \bmod N = 0$ then
        copy $\mathrm{buf}[k]$ to $p[k]$;        // copy buffer to part. $k$
```

- Extra copy step
- TLB fetch only needed once every *N* tuples in a partition
- More I/O reordering due to buffered writes & less TLB pressure
- Cache line-sized buffers can enable blind writes, which are faster

Hash: choose your fighter

radix

- Parallel radix-hash join
- Partitioned according to radix-hash
- Cache-local hash joins on partition pairs

n-part

- Emabarrassingly-parallelized hash join
- Tables sharded/striped across workers
- Build a shared hashtable based on one table
- Hash-and-match with the second table

1. Background
2. Parallel sort-merge joins
3. Parallel hash joins
4. Evaluation
5. Discussion

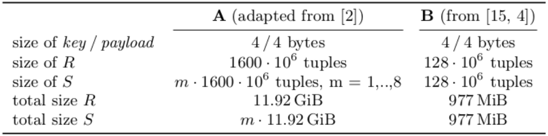
Setup

Benchmarks:

- m-way (sort-merge)
- m-pass (sort-merge)
- mpsm (sort-merge)
- radix (hash)
- n-part (hash)

Workloads:

- Column-store
- 4-byte keys and values, all integers
- Keys in S are a proper subset of keys in R
- Generally uniform key distribution in S



	A (adapted from [2])	**B** (from [15, 4])
size of *key* / *payload*	4 / 4 bytes	4 / 4 bytes
size of R	$1600 \cdot 10^6$ tuples	$128 \cdot 10^6$ tuples
size of S	$m \cdot 1600 \cdot 10^6$ tuples, m = 1,..,8	$128 \cdot 10^6$ tuples
total size R	11.92 GiB	977 MiB
total size S	$m \cdot 11.92$ GiB	977 MiB

Environment

- 256-bit AVX (floating-point only)
- 64 threads = 4 sockets, 8 cores/socket, hyperthreading enabled
- L1/L2/L3 cache sizes: 32KiB/256KiB/20MiB
- L3 is socket-local
- Cache line size: 64B
- TLB1: 64 entries for 64KiB pages; 32 entries for 2MiB pages
- TLB2: page size 4KiB, 512 entries per TLB1 entry

Experiments

Sorting baseline

Merging baseline

Partitioning

Alternative merges

m-way factors

Input size

Data skew

Scalability

Sorting baselines

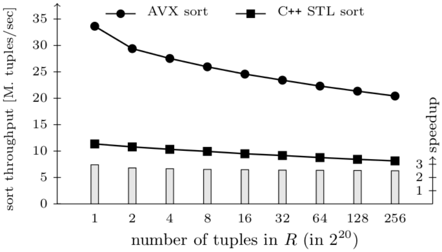


Figure 5: Single-threaded sorting performance where input table size varies from 8 MiB to 2 GiB.

- Evaluating single-threaded performance
- Confirm that AVX sorting is efficient

Merging

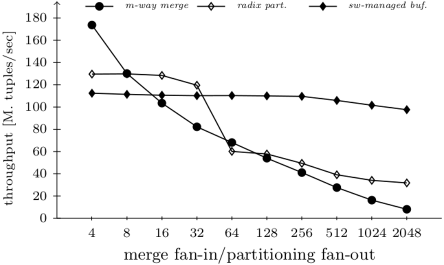


Figure 6: Impact of fan-in/fan-out on multi-way merging/partitioning (1-pass and single-thread).

- Larger merging fan-ins lead to smaller buffers
- Software managed buffers perform stably
- Idea: partition instead of merge

Merging

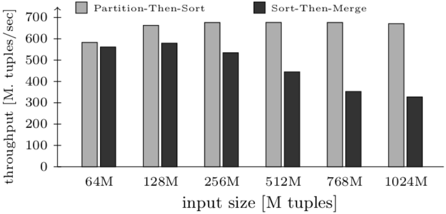


Figure 7: Impact of input size on different multi-threaded sorting approaches (using 64 threads).

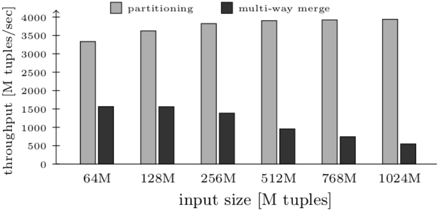


Figure 8: Trade-off between partitioning and merging (using 64 threads).

- Partition-then-sort: range-partition, sort, concatenate
- Sort-then-merge: what we've been discussing
- Partitioning doesn't degrade like merging does!

Sort-merge champion: m-way

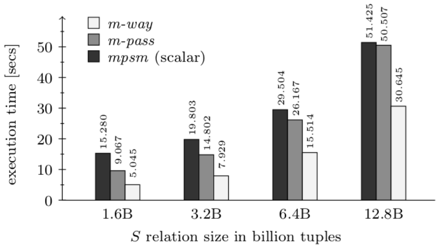


Figure 10: Execution time comparison of sort-merge join algorithms. Workload A, 64 threads.

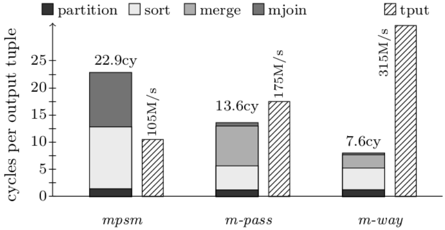


Figure 11: Performance breakdown for sort-merge join algorithms. Workload A. Throughput metric is output tuples per second, *i.e.* $|S|/\text{execution time}$.

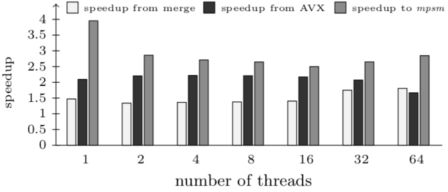


Figure 12: Speedup of *m-way* due to parallelism from AVX and efficiency from multi-way merge.

- Multi-way merge helps when memory is contended
- AVX benefit is persistent

Hash champion: radix-hash

Radix-hash with software-managed buffers [2]

Sort vs. Hash: Input size

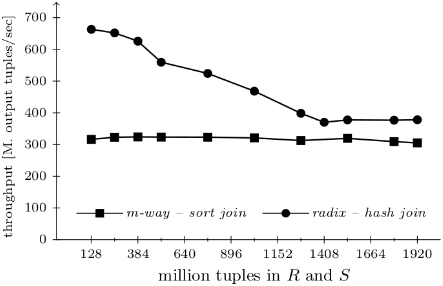


Figure 15: Sort vs. hash with increasing input table sizes ($|R| = |S|$). Throughput metric is total output tuples per second, *i.e.* $^{|S|}/_{\text{execution time}}$.

- Radix-hash wins at smaller sizes
- Radix-hash degrades quickly with larger sizes
- m-way doesn't degrade with table size, but
- m-way performs ≈radix-hash at best

Sort vs. Hash: Skew

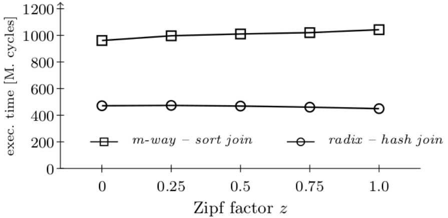


Figure 16: Join performance when foreign key references follow a Zipfian distribution. Workload B.

Radix-hash

- Fine-granular task decomposition [2, 3]
- Redistributes “hotter” partitions to all threads

m-way

- Multi-way merging’s two-step approach:
  1. Sub-task merges, split in NUMA region
  2. Special handling for heavy hitters

Sort vs. Hash: Scalability

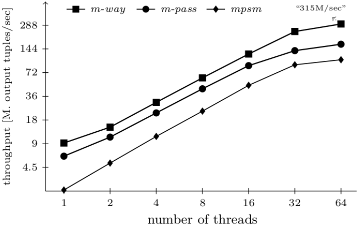
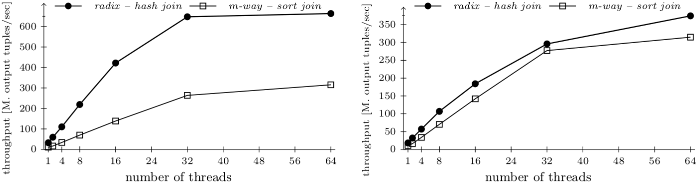


Figure 13: Scalability of sorting-based joins. Workload A, (11.92 GiB ⋈ 11.92 GiB). Throughput metric is output tuples per second, *i.e.* $|S|$/execution time.

- Sort-merge algorithms all scale
- Radix-hash scales as well



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

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

Figure 17: Scalability of sort vs. hash join. Throughput is in output tuples per second, *i.e.* $|S|$/execution time.

Sort vs. Hash

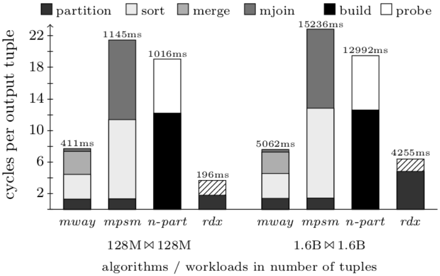


Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.

- Radix-hash works well
- m-way is about similar for larger joins

Hash joins are still the winners

References

Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu.
Multi-core, main-memory joins: Sort vs. hash revisited.
Proceedings of the VLDB Endowment, 7(1):85–96, 2013.

Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu.
Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware.
In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey.
Sort vs. hash revisited: Fast join implementation on modern multi-core cpus.
Proceedings of the VLDB Endowment, 2(2):1378–1389, 2009.

Feedback

Positive:

- Paper layout is very readable!
- Lots of appropriate data visuals
- Thorough work on minimizing effects of external factors
- Good balance of self and cross-system comparisons

Constructive:

- Throughput vs execution time graphs can be confusing
- Hyperthread scaling cap for memory-restricted workloads is well-known
- Generally should avoid benchmarking with hyperthreads

Discussion

1. How could multi-way merging benefit from advances with (parallel) funnelsort?
2. How would a non-NUMA architecture affect these results?
3. How could these results translate to other database data layouts?
    - Delta encodings
    - Bit vector layouts