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
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

\[
\text{SELECT } * \text{ FROM } R, S \text{ WHERE } F(R.\text{key}) = G(S.\text{key})
\]

**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.
1 Background

2 Parallel sort-merge joins

3 Parallel hash joins

4 Evaluation

5 Discussion
Parallel run-generation

Sorting networks

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

\[
\begin{align*}
e &= \min(a, b) \\
f &= \max(a, b) \\
g &= \min(c, d) \\
h &= \max(c, d) \\
i &= \min(e, g) \\
j &= \min(f, h) \\
w &= \min(e, g) \\
x &= \min(i, j) \\
y &= \max(i, j) \\
z &= \max(f, g)
\end{align*}
\]
Parallel run-generation

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

Parallel merge

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

1. $a \leftarrow \text{fetch4}(in_1); b \leftarrow \text{fetch4}(in_2);$  
2. repeat
3. $\langle a, b \rangle \leftarrow \text{bitonic_merge4}(a, b);$  
4. emit $a$ to output;  
5. if head($in_1$) < head($in_2$) then  
6. $\langle a \leftarrow \text{fetch4}(in_1);$  
7. else  
8. $\langle a \leftarrow \text{fetch4}(in_2);$  
9. until eof($in_1$) or eof($in_2$);  
10. $\langle a, b \rangle \leftarrow \text{bitonic_merge4}(a, b);$  
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;

- Scales poorly
- Used as a kernel sort
- 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

```plaintext
1 foreach input tuple t do
2     k ← hash(t);
3     p[k][pos[k]] = t; // copy t to target partition k
4     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

```plaintext
1 foreach input tuple t do
2     k ← hash(t);
3     buf[k][pos[k] mod N] = t;  // copy t to buffer
4     pos[k]++;
5     if pos[k] mod N = 0 then
6         copy 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 $S$

<table>
<thead>
<tr>
<th>size of key / payload</th>
<th>A (adapted from [2])</th>
<th>B (from [15, 4])</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of $R$</td>
<td>4 / 4 bytes</td>
<td>4 / 4 bytes</td>
</tr>
<tr>
<td>size of $S$</td>
<td>1600 · $10^6$ tuples</td>
<td>128 · $10^6$ tuples</td>
</tr>
<tr>
<td>$m$ · 1600 · $10^6$ tuples, $m = 1, ..., 8$</td>
<td>11.92 GiB</td>
<td>977 MiB</td>
</tr>
<tr>
<td>total size $R$</td>
<td>11.92 GiB</td>
<td>977 MiB</td>
</tr>
<tr>
<td>total size $S$</td>
<td>$m$ · 11.92 GiB</td>
<td></td>
</tr>
</tbody>
</table>
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
Alternative merges
Data skew

Merging baseline
m-way factors

Partitioning
Input size
Scalability
Evaluation

Sorting baselines

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

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

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

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

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

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).
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}.

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

Figure 12: Speedup of m-way due to parallelism from AVX and efficiency from multi-way merge.
Hash champion: radix-hash

Radix-hash with software-managed buffers [2]
Sort vs. Hash: Input size

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 \( \approx \) radix-hash at best

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

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

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

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

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.

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

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

Hash joins are still the winners

Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.
Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu.
Multi-core, main-memory joins: Sort vs. hash revisited.

Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu.
Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware.

Sort vs. hash revisited: Fast join implementation on modern multi-core cpus.
1 Background

2 Parallel sort-merge joins

3 Parallel hash joins

4 Evaluation

5 Discussion
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