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

Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Ozsu

Presented by: Anirudh Ramanathan
Motivation - I: What are joins?!

Think database tables in an RDBMS.

Try it for yourself: link
Motivation - II: Real-world perspective

- Advanced instruction sets, vector operations, multi-core architectures, and NUMA constraints present an ever-changing landscape when it comes to the choices between the various join algorithms.
- **MySQL** uses variants of the Nested-Loop Join Algorithm.
- **PostgreSQL** is more flexible and picks between nested loops, hash join, sorted-merge join, etc.
- Other DBMS have their own implementations of joins, which sometimes vary depending on the execution plan generated for that piece of SQL code.
Basic Approaches

Sort-Merge Join

/* Stage 1: Sorting */
sort R on R.A
sort Q on Q.B

/* Stage 2: Merging */
r = first tuple in R
q = first tuple in Q

while r ≠ EOR and q ≠ EOR do
    if r.A > q.B then
        q = next tuple in Q after q
    else
        if r.A < q.B then
            r = next tuple in R after r
        else
            put r \join q in the output relation

    /* output further tuples that match with r */
    q' = next tuple in Q after q
    while q' ≠ EOR and r.A = q'.B do
        put r \join q' in the output relation
        q' = next tuple in Q after q'
    od

    /* output further tuples that match with q */
    r' = next tuple in R after r
    while r' ≠ EOR and r'.A = q.B do
        put r' \join q in the output relation
        r' = next tuple in R after r'
    od

r = next tuple in R after r
q = next tuple in Q after q

Hash Join

/* Hash relation R */
foreach tuple r ∈ R do
    put r in bucket no. h(r.A)
od

/* Probe relation Q */
foreach tuple q ∈ Q do
    foreach tuple r in bucket no. h(q.A) do
        if r.A = q.B then
            put r \join q in the output relation
        fi
    od
od
Contributions

- The paper explores the relative performance of radix-hash vs. sort-merge join algorithms in main-memory, multicore settings.
- There is new research that with hardware that provides support for vector instructions that are sufficiently wide (SIMD with 256-bit AVX and wider), sort-merge joins would easily outperform radix-hash joins.
- This paper makes the following major contributions:
  - Shows that radix-hash join is still superior to sort-merge join in most cases;
  - Provides several insights on the implementation of data operators on modern processors
  - Presents the fastest algorithms available to date for both sort-merge — 2-3 times faster than available results and radix-hash join, demonstrating how to use modern processors to improve the performance of data operators.
Modern Improved Approaches: Multi-core Era

**Sort-Merge Join**
- **Multi-way** and Multi-pass sort-merge joins.
- MPSM (Massively Parallel Sort-Merge join): NUMA-friendly access patterns and avoids full sorting of the outer relation.

**Hash Join**
- No-partitioning Hash Join

**Performance improvements**
- Use of hardware assisted sorting techniques with SIMD and multiple cores.
- Operations on NUMA-aware regions to achieve better performance.
Sort-Merge Optimization 1: Cache consciousness

- in-register sorting with SIMD CPU registers using sorting networks.
  - Accelerated through SIMD instructions.
  - Generating four runs of four items requires 10 min/max instructions, 8 SIMD shuffles, 4 loads, and 4 stores.
  - Shuffling brings down speedup from 4-wide SIMD to ~2.7x instead of 4x.
  - Three stage approach.
  - On current Intel hardware, for k = 4, implementing a bitonic merge network for 2 × 4 input items requires 6 SIMD min/max instructions and 7-10 shuffles.
- out-of-cache sorting using multi-way merging and SIMD-optimized bitonic merging to save round-trips to memory.
Sort-Merge Optimization 1.5: Bitonic Merge

- Use of sorting networks naively is inefficient.
- They use the algorithm shown to the right, similar to scalar merge.
- No values moved between the scalar and vector execution units of the CPU.
- The vector size (k) for merging is chosen keeping mind complexity of merge network and the gains of amortizing branch mispredictions over that many elements.
- They chose 8, and each loop iteration requires 36 assembly instructions to produce 8 output items.

```c
// Algorithm 1: Merging larger lists with help of bitonic merge kernel bitonic_merge4() (k = 4).
1 a ← fetch4(in1); b ← fetch4(in2);
2 repeat
3   (a, b) ← bitonic_merge4(a, b);
4   emit a to output;
5   if head(in1) < head(in2) then
6       a ← fetch4(in1);
7   else
8       a ← fetch4(in2);
9 until eof(in1) or eof(in2);
10 (a, b) ← bitonic_merge4(a, b);
11 emit4(a); emit4(b);
12 if eof(in1) then
13    emit rest of in2 to output;
14 else
15    emit rest of in1 to output;
```
Sort-Merge Optimization 2: Bandwidth & NUMA

Bandwidth-Compute Balancing:

- CPU overhead because of task switching in multiway merge with a FIFO in between.
- Overhead increases when the FIFO queues between merging tasks is small.
- This offers us a handle to balance bandwidth and compute.

Impact of NUMA:

- In practice, at least some merging passes will inevitably cross NUMA boundaries.
- With multi-way merging, this can be combated.
Hash-Join Optimization 1: Partitioning

Hashing results in random access to memory, which leads to a lot of cache misses when hash table size > cache size.

- Takes all input tuples one-by-one and writes them to their corresponding destination partition.
- Generally, partitions far apart and on separate VM pages.
- Avoid TLB misses by keeping partition fanout same as number of entries TLB can hold.

```
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]++;         // keep partition fanout same
```
Hash-Join Optimization 2: Software Buffering

- The TLB miss limitations on maximum fan-out can be reduced, when writes are buffered inside the cache first.
- Buffering leads to additional copy overhead. However, for sufficiently small $N$, all buffers will fit into a single memory page and into L1 cache.
- TLB misses become infrequent.
- The authors configure $N$ such that one buffer will exactly fill one cache line (64 bytes), allowing them to also use non-temporal writes optimization.

```plaintext
1  foreach input tuple $t$ do
2    $k \leftarrow$ 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. $k$
```
Analysis of Join Algorithms: m-way

Sort-merge join with multi-way merging

1. Phase 1: Each thread assigned its own NUMA-local chunk of relation R. They range-partition their local chunks in parallel. Each local sorted using the AVX sorting algorithm.
2. Phase 2: Multi-way merging across NUMA regions.
3. Phase 3,4: Same steps applied to relation S.
4. Phase 5: Each thread concurrently evaluates the join between NUMA-local sorted runs using a single-pass merge join.
Analysis of Join Algorithms: m-pass

Sort-merge join with multi-pass naive merging

- Differs in phase 2/4 from m-way
- Applies successive two-way bitonic merging.
- The first iteration of merging of sorted runs is done as the data is transferred to the local memory from NUMA-remote.
Analysis of Join Algorithms: mpsm

Massively Parallel Sort-Merge Join

1. Globally range-partitions one relation R.
2. Each thread independently sorts its partition, resulting in a globally-sorted R.
3. S is sorted only partially. Each thread sorts its NUMA-local chunk of S without a prior partitioning.
4. During the last phase, a run of R must be merge-joined with all the NUMA-remote runs of relation S.
5. Where S >> R in size, avoiding the global partitioning-sorting may pay off.
Analysis of Join Algorithms: radix

Parallel radix hash join

1. Partition both input relations using buffers in cache.
2. Run cache-local hash join on partition pairs.
No-Partitioning Hash Join: Direct parallel version of the canonical hash join.

1. Both input relations are divided into equi-sized portions that are assigned to a number of worker threads.
2. In the build phase, all worker threads populate a shared hash table with all tuples of R.
3. After synchronization via a barrier, all worker threads enter the probe phase and concurrently find matching join partners for their assigned S portions.
Experimental Setup

Workloads:

- A column-oriented storage model.
- To realize the value of vectorized instructions, key and payload are four bytes each.
- Treating integer keys as floats when operating with AVX.
- There is a foreign key relationship from S to R.
- Uniform distribution of key values from R in S

System:

- Intel Sandy Bridge with a 256-bit AVX instruction set.
- Four-socket configuration: each CPU socket containing 8 physical/16 hyperthreaded cores.
- Cache line size: 64 bytes. TLB1: 64/32 entries when using 4 KiB/2 MiB pages (respectively)
- 512 TLB2 entries (page size 4 KiB)
- Total memory available is 512 GiB.
Observation: STL sort v/s AVX Sort

- AVX sort between 2.5 and 3 times faster than the C++ sort.
- With size of the input, both algorithms suffer due to the increasing trips to the main-memory.

Figure 5: Single-threaded sorting performance where input table size varies from 8 MiB to 2 GiB.
Observation: Merge fan-in v/s algorithm

- Fan-in (F) of the multi-way merge equals the number of 1/2-L2-sized sorted runs.
- Merge tree must reside in the L3 cache.
- Shared L3 must be further divided by the number of threads.
- The cache contention problem in multi-way merging raises question against the assumption that multi-way merge can be done efficiently.
- Software-managed buffers strategy for high fan-outs is clearly a big improvement over the normal radix partitioning.
Observation: Partition-then-Sort v/s Sort-then-Merge

- Partition-then-Sort approach first range-partitions the input with efficient software-managed radix partitioning (using most-significant radix bits).

- Sort-then-Merge, creates cache-sized sorted runs using AVX sort. These sorted runs are then merged to form a complete sorted output using the AVX multi-way merge routine.

Figure 7: Impact of input size on different multi-threaded sorting approaches (using 64 threads).
The cooperative m-way approach follows the original idea by Chhugani et al. where there is a single multiway merge tree and multiple threads cooperatively merge does not scale well beyond 4 threads.

- It does not scale for a variety of reasons: contention for upper-level nodes in the merge tree, idle threads due to lack of enough work, and synchronization overhead.

Observation: Cooperative m-way v/s Partition-then-Sort

![Graph showing comparison between independent sort and cooperative m-way sort throughput.](image)
Observation: Comparing merge algorithms

- The merge phase in m-way is 3 times faster than m-pass with bandwidth-aware multi-way merging.

- The “mjoin” phase is a linear merge-join operation on NUMA-local sorted runs everywhere except mpsm and overhead of that phase becomes negligible.

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

- All the algorithms exhibit linear scaling behavior up to 16 physical CPU cores.
- Algorithms are cache and CPU resource sensitive, the scaling with hyper threads is rather limited.

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 v/s Hash

- When input table sizes are in the hundred millions, radix hash join is more than 2 times faster than m-way sort-merge join.

- The speed difference is maintained even when the outer table size becomes significantly larger than the primary key table.

- For significantly larger workloads, the picture becomes more favorable for sort-merge joins.

Figure 14: Comparison of best sort vs. best hash join algorithms with cycles per output tuple metric under different workloads. Using 64 threads.
Sort v/s Hash: Input Size

- Radix join performance degrades with the increasing input size due to the increasing number of passes for the partitioning phase.

- The optimized radix partitioning very efficient up to 9-bits/512-way partitioning since the entire partitioning buffer can reside in L1 cache (32 KiB = 512 × 64-bytes-cache-lines).

- Even higher radix-bits/fan-outs such as a maximum of 12/4,096 can be tolerated when backed up by the L2 cache (256 KiB = 4,096 × 64-bytes).

Figure 15: Sort vs. hash with increasing input table sizes ($|R| = |S|$). Throughput metric is total output tuples per second, i.e. $|S|$/execution time.
Sort v/s Hash: Skew

- Both have techniques to handle skew.
- Parallel radix hash join handles skew by a fine-granular task decomposition method: parallelize the processing of larger partitions through a refined partitioning.
- Multi-way merging, is prone to skew, create multiple merge tasks by finding boundaries within each of the sorted runs with binary search.
Both algorithms show almost linear scalability up to the physical number of cores.

Within the SMT region, algorithms scale poorly.

SMT scalability for radix join are different in the two workloads mainly because of the optimal partitioning configuration.
Conclusions

- Input Sizes: Sort-merge joins also have to do multiple passes over the data, with their performance suffering accordingly as data sizes increase.
- Degree of Parallelism: As the number of available hardware contexts increases, contention in large merge trees for sorting also increases.
- Cache-conscious approaches make a significant difference in both hash and sort-merge joins.
- Hash joins still have an edge over sort-merge alternatives unless the amount of data involved is very large.
- Some advances in hardware will eventually favor sort-merge joins, e.g., wider SIMD registers and higher memory bandwidth, but results show that exploiting such advances will also benefit radix join algorithms.
Questions?