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

Cagri Balkesen, Gustavo Alonso
Systems Group, ETH Zurich
Switzerland
{name.surname}@inf.ethz.ch

Jens Teubner
TU Dortmund University
Germany
jens.teubner@cs.tu-dortmund.de

M. Tamer Özsu
University of Waterloo
Canada
tamer.ozsu@uwaterloo.ca

ABSTRACT

In this paper we experimentally study the performance of main-memory, parallel, multi-core join algorithms, focusing on sort-merge and (radix-)hash join. The relative performance of these two join approaches have been a topic of discussion for a long time. With the advent of modern multi-core architectures, it has been argued that sort-merge join is now a better choice than radix-hash join. This claim is justified based on the width of SIMD instructions (sort-merge outperforms radix-hash join once SIMD is sufficiently wide), and NUMA awareness (sort-merge is superior to hash join in NUMA architectures). We conduct extensive experiments on the original and optimized versions of these algorithms. The experiments show that, contrary to these claims, radix-hash join is still clearly superior, and sort-merge approaches to performance of radix only when very large amounts of data are involved. The paper also provides the fastest implementations of these algorithms, and covers many aspects of modern hardware architectures relevant not only for joins but for any parallel data processing operator.

1. INTRODUCTION

Modern processor architectures introduce many possibilities as well as challenges for the implementation of parallel data operators. Advanced instruction sets, vector operations, multi-core architectures, and NUMA constraints create a very rich design space where the effects of given design decisions on the performance of data operators are not always easy to determine. There is a need for developing a better understanding of the performance of parallel data operators on new hardware.

In this paper, we explore the relative performance of radix-hash vs. sort-merge join algorithms in main-memory, multi-core settings. Our main goal is to analyze the hypothesis raised by recent work claiming that sort-merge joins over new hardware are now a better option than radix-hash joins, the algorithm traditionally considered to be the fastest [2, 15]. Kim et al. [15] have suggested that, once hardware 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 claim was reinforced by recent results by Albutiu et al. [2] who report that their NUMA-aware implementation of sort-merge join is already superior to hash joins even without using SIMD instructions. Furthermore, there are a number of new optimizations for parallel radix join [3, 4, 5] that have not been considered in these studies, but that should be part of any analysis of the relative performance of the two options.

In this paper, we approach the question experimentally. We bring carefully-tuned implementations of all relevant, state-of-the-art join strategies (including *radix join* [4, 15, 19], *no-partitioning join* [5], *sort-merge join* [15], and *massively parallel sort-merge (MPSM) join* [2]) to a common and up-to-date hardware platform. We then compare the relative performance of all these algorithms under a wide range of experimental factors and parameters: algorithm design, data sizes, relative table sizes, degree of parallelism, use of SIMD instructions, effect of NUMA, data skew, and different workloads. Many of these parameters and combinations thereof were not foreseeable in earlier studies, and our experiments show that they play a crucial role in determining the overall performance of join algorithms.

Through an extensive experimental analysis, this paper makes several contributions: *(1)* we show that radix-hash join is still superior to sort-merge join in most cases; *(2)* we provide several insights on the implementation of data operators on modern processors; and *(3)* we present 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.

In addition, the paper sheds light on a number of relevant issues involving the processing of "big data" and the factors that affect the choice of the algorithm. These include:

Input Sizes. Our results show that the relative and absolute input table sizes have a big effect on performance. Moreover, as the data size grows, the duality between hashing and sorting becomes more pronounced, changing the assumption that only hashing involves several passes over the data when it is sufficiently large. We show that sort-merge joins also have to do multiple passes over the data, with their performance suffering accordingly.

Degree of Parallelism. Existing work has studied algorithms using a small degree of parallelism. As the number of available hardware contexts increases, contention in large merge trees for sorting also increases. This contention is not

Proceedings of the VLDB Endowment, Vol. 7, No. 1

visible in the four-core configurations used in earlier studies but it becomes a dominant factor in larger systems.

Cache Contention. Cache-conscious approaches make a significant difference in both hash and sort-merge joins. Cache consciousness becomes the decisive factor in choosing the best hash join algorithm, favoring the radix join approach of [4, 15] over the no-partitioning approach of [5]. Similarly, for efficient merging, Kim et al. [15] assume that entire merge trees can fit into a single last-level cache. As the degree of parallelism, data sizes, and merge fan-ins increase, this assumption may no longer hold, which calls for a closer look at the implementation of multi-way merge trees.

SIMD Performance. There are many ways to exploit SIMD in all algorithms. Our results show, however, that hardware peculiarities play a big role, and it turns out that the width of the SIMD registers is not the only relevant factor. The complex hardware logic and signal propagation delays inherent to more complex SIMD designs may result in latencies larger than one cycle, limiting the advantages of SIMD.

The paper shows that 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 our results show that exploiting such advances will also benefit radix join algorithms. Furthermore, new processor features such as *memory gather* support in Intel's upcoming Haswell series may play a bigger role in improving hash joins than the factors considered so far.

2. BACKGROUND AND RELATED WORK

2.1 Sort *vs.* Hash—Early Work

"Sort or hash" has long been a point of discussion in databases. Initially, sort-merge join was the preferred option [20]. Later, the invention of hashing-based techniques [6, 16] changed the balance. Schneider et al. [23] compared hash-based with sort-merge joins and concluded that hash-based joins were superior unless memory was limited. Hash join was also the main choice in most of the early parallel database systems [8, 9, 23].

Changes in hardware, memory, and data volumes prompted researchers to revisit the "sort or hash" question regularly over the years. Graefe et al. [12] provided an extensive study of sort- and hash-based query processing algorithms. They outlined the dualities and similarities among the approaches and concluded that performance only differs by percentages rather than factors if both algorithms are implemented in a careful and equally-optimized manner. Moreover, the study pointed out that there are cases when one of the algorithms would be preferable over the other and therefore both algorithms would be needed in a database system. Subsequently, Graefe et al. [11] used histogram-based partitioning to improve hash-based joins and finally concluded that sort-merge joins only win in a number of corner cases.

2.2 Sort *vs.* Hash—Multi-core Era

Multi-core has changed things once again. Kim et al. [15] compared parallel radix-hash join with a sorting-based join exploiting both SIMD data parallelism and multiple threads. They concluded that wider SIMD registers will soon make sort-merge a better option.

Albutiu et al. [2] presented a "massively parallel sort-merge join" (MPSM) tailored for modern multi-core and multi-socket NUMA processors. MPSM is claimed to observe NUMA-friendly access patterns and avoids full sorting of the outer relation. In their experiments on a 32-core, four socket machine, they report that sort-merge join is faster than the "no-partitioning" hash join implementation of Blanas et al. [5] (see below). Unlike the projections of Kim et al. [15], the claim is that sort-merge join is faster than hash join even without using SIMD parallelism.

On the hash join side, cache-aware, partitioning-based algorithms such as "radix join" provide the best performance [19, 24]. More recently, Blanas et al. [5] introduced the "no-partitioning" idea and advocated its simplicity, hardware obliviousness, and efficiency. Recent work has studied these hash join algorithms and showed that hardware-conscious, parallel radix join has the best overall performance [3, 4]. As the code for the no-partitioning and radix join algorithms is available, we use these algorithms in this study. We also refer to the literature for detailed descriptions of these algorithms and focus here mainly on sort-merge joins.

2.3 Hardware-Assisted Sorting

Recent work on sorting has explored the use of SIMD data parallelism [7, 10, 13, 22]. Inoue et al. [13] introduced AA-Sort which utilized both SIMD and thread-level parallelism. AA-Sort eliminates unaligned loads for maximum utilization of SIMD where unaligned accesses cause performance bottlenecks in architectures such as PowerPC and Cell processors. Gedik et al. [10] also investigated parallel sorting for the Cell processor and implemented an efficient sorting algorithm with bitonic sorting and merging using SIMD parallelism. Chhugani et al. [7] provided a multi-core SIMD sorting implementation over commodity `x86` processors. Their algorithm, based on merge sort, extensively used bitonic sort and merge networks for SIMD parallelism, following the ideas introduced by Inoue et al. [13] for in-register sorting. However, the machine they used had only four cores and further scalability was based on projections. Satish et al. [22] have analyzed comparison and non-comparison-based sorting algorithms on modern CPU/GPU architectures. Their study provided some of the fastest sorting implementations and found that non-comparison-based scalar sorting such as radix sort is faster with smaller keys. Moreover, they showed that SIMD-based merge sort is more competitive with larger keys and will be even more favorable with the future hardware trends such as larger SIMD width. Kim et al. [14] implemented distributed in-memory sorting over a cluster of multi-core machines. While their main solution focuses on overlapping computation and inter-node communication, their approach also makes extensive use of parallel SIMD sorting on each of the machines.

2.4 The Role of NUMA

For better performance in NUMA systems, algorithms must be hardware conscious by taking the increasingly more complex NUMA topologies into consideration [2, 18]. Li et al. [18], for instance, showed that a hardware-conscious "ring-based" data shuffling approach across NUMA regions achieves a much better interconnect bandwidth and improves the performance of sort-merge join algorithm of Albutiu et al. [2]. Therefore, we followed a similar approach and made our algorithms NUMA aware.

3. PARALLELIZING SORT WITH SIMD

The dominant cost in sort-merge joins is *sorting* the input relations. We thus now discuss strategies to implement sorting in a hardware-conscious manner. Typically, sort-merge joins use *merge sort*—a tribute to the latency/bandwidth gap in modern system architectures. Both building blocks of merge sort, *(a)* initial *run generation* and *(b)* the *merging* of pre-sorted runs, benefit from SIMD.

3.1 Run Generation

For initial run generation, many chunks with a *small* number of tuples need to be sorted. This favors sorting algorithms that can process multiple chunks in parallel over ones that have a good asymptotic complexity with respect to the tuple count. *Sorting networks* provide these characteristics and fit well with the SIMD execution model of modern CPUs [7, 10, 21].

3.1.1 Sorting Networks

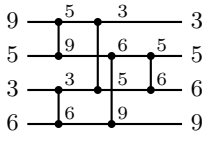


Figure 1: Even-odd network for four inputs.

Figure 1 on the left illustrates, in the notation of Knuth [17, Section 5.3.4], a sorting network for four input items. A set of four items $\langle 9, 5, 3, 6\rangle$ enters the network on the left and travels toward the right through a series of *comparators* :. Every comparator emits the smaller of its two input values at the top, the larger on the bottom. After traversing the five comparators, the data set is sorted.

The beauty of sorting networks is that comparators can be implemented with help of `min`/`max` operators only. Specifically, the five comparators in Figure 1 compile into a sequence of ten `min`/`max` operations as illustrated here on the right (input variables `a`, ..., `d` and output variables `w`, ..., `z`). Limited data dependencies and the absence of branching instructions make such code run very efficiently on modern hardware.

```
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)
```

Sorting networks are also appealing because they can be accelerated through SIMD instructions. When all variables in the code on the right are instantiated with SIMD vectors of κ items and all `min`/`max` calls are replaced by SIMD calls, κ sets of items can be sorted in approximately the same time that a single set would require in scalar mode (suggesting a κ-fold speedup through SIMD).

3.1.2 Speedup Through SIMD

However, the strategy illustrated above will sort input items *across* SIMD registers. That is, for each vector position i, the sequence $\mathtt{w}_i, \mathtt{x}_i, \mathtt{y}_i, \mathtt{z}_i$ will be sorted, but *not* the sequence of items within one vector (*i.e.*, $\mathtt{w}_i, \ldots, \mathtt{w}_\kappa$ is in undefined order). Only full SIMD vectors can be read or written to memory consecutively. Before writing back initial runs to main-memory, SIMD register contents must thus be *transposed*, so items within each vector become sorted (*i.e.*, $\mathtt{w}_2$ must be swapped with $\mathtt{x}_1$, $\mathtt{w}_3$ with $\mathtt{y}_1$, etc.).

Transposition can be achieved through SIMD *shuffle* instructions that can be used to move individual values within and across SIMD registers. A common configuration in the context of join processing is to generate runs of four items with $\kappa = 4$. Eight shuffle instructions are then needed to transpose registers. That is, generating four runs of four items each requires 10 `min`/`max` instructions, 8 shuffles, 4 loads, and 4 stores. Shuffle operations significantly reduce the effective SIMD speedup for run generation from optimal $\kappa = 4$ to about 2.7.

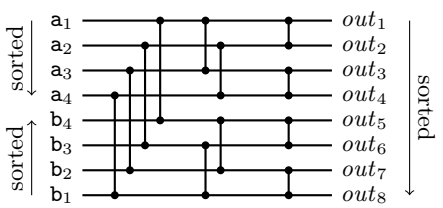


Figure 2: Bitonic merge network.

3.2 Merging Sorted Runs

3.2.1 Bitonic Merge Networks

Although sequential in nature, merging also benefits from SIMD acceleration. The basic idea comes from Inoue et al. [13] and has been used for sorting [7] and joins [15].

Looking back to the idea of sorting networks, larger networks can be built with help of *merging networks* that combine two pre-sorted inputs into an overall sorted output. Figure 2 shows a network that combines two input lists of size four. The network in Figure 2 is a sequence of three stages, each consisting of four comparator elements :. Each stage can thus be implemented using one `max` and one `min` SIMD instruction (assuming $\kappa = 4$). Shuffle instructions in-between stages bring vector elements into their proper positions (for instance, if `a` and `b` are provided as one SIMD register each, `b` must be reversed using shuffles to prepare for the first `min`/`max` instruction pair).

On current Intel hardware, for $\kappa = 4$, implementing a bitonic merge network for 2×4 input items requires 6 SIMD `min`/`max` instructions and 7–10 shuffles. The exact number of shuffles depends on the bit width of the input items and the instruction set offered by the hardware (SSE, AVX, AVX2).

3.2.2 Merging Larger Lists using Bitonic Merge

For larger input sizes, merge networks scale poorly [21]: sorting networks for N input items require $\mathcal{O}\left(N \log^2 N\right)$ comparators—clearly inferior to alternative algorithms. But small merge networks can be used as a *kernel* within a merging algorithm for larger lists [13]. The resulting merging algorithm (Algorithm 1) uses a working set of $2 \times k$ data items (variables `a` and `b`, both implemented as SIMD registers). In each iteration of the algorithm's loop body, that working set is sorted (using the merge kernel `bitonic_merge4()` and knowing that `a` and `b` themselves are sorted already) and the smaller k items are emitted to the merge result.

The emitted SIMD vector is then replaced by fresh data from the input. As in the classical scalar merge algorithm, the two head elements of the input runs are used to decide which new data to load (line 5 in Algorithm 1). Unlike in the classical algorithm, however, the decision is used to load an *entire vector* into the working set. The rationale is that the resulting working set still contains at least k items that are smaller than the larger of the two head items, and only k items will be emitted in the next loop iteration.

In terms of performance, the separation between control flow and merge kernel operations in Algorithm 1 fits well

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;
```

with the execution model of modern CPUs. In particular, no values have to be moved between the scalar and vector execution units of the CPU (a costly operation in many architectures). Branch mispredictions will still occur, but their effect will now be amortized over k input elements. Also, optimizations such as predication or conditional moves can be applied in the same way as in scalar merge implementation.

The vector size used for merging is a configuration parameter, typically $k = 2^p \times \kappa$ (where κ is the hardware SIMD width). For the hardware that we used in our experimental study, we found $k = 8$ to be a sweet spot that amortizes branching cost well, while not suffering too much from the complexity of large merge networks. Each loop iteration requires 36 assembly instructions to produce 8 output items.

4. CACHE CONSCIOUS SORT JOINS

4.1 Sorting and the Memory Hierarchy

The cache hierarchies in modern hardware require separating the overall sorting into several phases to optimize cache access: *(i) in-register sorting*, with runs that fit into (SIMD) CPU registers; *(ii) in-cache sorting*, where runs can still be held in a CPU-local cache; and *(iii) out-of-cache sorting*, once runs exceed cache sizes.

In-Register Sorting. Phase *(i)* corresponds to run-generation as discussed in Section 3.1.

In-Cache Sorting. In Phase *(ii)*, runs are then merged until runs can no longer be contained within CPU caches. In-cache sorting corresponds to Algorithm 1 (bitonic merging) in Section 3.2.2. It is backed up by a bitonic merge kernel such as `bitonic_merge4 ()`. Using bitonic merging, runs are repeatedly combined until runs have reached $1/2$ cache size (since in- and output runs must fit into cache).

Out-of-Cache Sorting. Phase *(iii)* continues merging until the data is fully sorted. Once runs have exceeded the size of the cache, however, all memory references will have to be fetched from off-chip memory.

4.2 Balancing Computation and Bandwidth

Accesses to off-chip memory make out-of-cache sorting sensitive to the characteristics of the memory interface. As mentioned in Section 3.2.2, an 8-wide bitonic merge implementation requires 36 assembly instructions per eight tuples being merged—or per 64 bytes of input data.

We analyzed the 36-instructions loop of our code using the Intel Architecture Code Analyzer [1]. The tool considers the super-scalar instruction pipeline of modern Intel processors and infers the expected execution time for a given instruction sequence. For our code, the tool reported a total of 29 CPU cycles per 64 bytes of input data. The tool does *not* consider potential pipeline stalls due to memory accesses (bandwidth and/or latency).

With a clock frequency of 2.4 GHz, this corresponds to a memory bandwidth of $2 \times 5.3\,\mathrm{GB/s}$ (read+write) or $10.6\,\mathrm{GB/s}$ for a single merging thread. This is more than existing interfaces support, considering also that typical CPUs contain eight or more cores per chip. Out-of-cache merging is thus severely bound by the memory bandwidth.

Memory bandwidth demand can be reduced by merging more than two runs at once. *Multi-way merging* saves round-trips to memory and thus precious memory bandwidth.

To still benefit from CPU-efficient, SIMD-optimized bitonic merging, we implement multi-way merging using multiple two-way merge units, as illustrated in Figure 3. Two-way merge units are connected using FIFO queues; FIFO queues are sized such that all queues together still fit into the CPU cache. Thus, external memory bandwidth is needed only at the front of the multi-way merging tree.

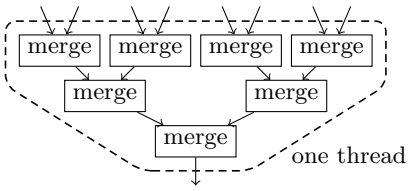


Figure 3: Multi-way merging.

As indicated in Figure 3, a complete multi-way merging is realized by a single operating system thread. This thread will treat two-way merges as individual tasks and switch between tasks whenever a task is blocked by a FIFO over- or underrun.

Compute vs. Bandwidth. Task switching will cause new CPU overhead that is not necessary in an implementation that merges two runs from memory until completion. This overhead will increase when the FIFO queues between merging tasks are small, because more task switches are needed then to perform a full multi-way merge. Since we size FIFO buffers such that all buffers in one thread fill a CPU cache, CPU overhead increases together with the merge fan-in.

This offers us a handle to *balance* bandwidth and compute. Merging only two runs is bound by memory bandwidth, with plenty of stalled CPU cycles that could be spent on additional CPU instructions. As we increase merge fan-in, memory pressure becomes reduced until the system becomes CPU-bound. At that point, larger fan-in will degrade performance again because of increasing CPU load.

Impact of NUMA. In practice, at least some merging passes will inevitably cross NUMA boundaries (if not, NUMA crossing has to be performed in the later join phase, which in this regard behaves like a two-way merge). As pointed out by Li et al. [18]—and confirmed by our measurements—multi-socket systems show an increasing asymmetry, where the NUMA interconnect bandwidth stays further and further behind the aggregate memory bandwidth that the individual memory controllers could provide. With multi-way merging,

we can combat this problem in an effective and scalable way. In the experimental part of this work we study how merging can be tuned to avoid memory bottlenecks even across NUMA boundaries.

5. HASH-BASED JOINS

Having covered the optimized sort-based join implementations, we now look at hash-based joins. While efficient, hashing results in *random access* to memory, which can lead to cache misses. Shatdal et al. [24] identified that when the hash table is larger than the cache size, almost every access to the hash table results in a cache miss. As a result, a partitioning phase to the hash joins is introduced to reduce cache misses. The performance of the resulting join is largely dictated by this partitioning phase.

5.1 Radix Partitioning

Manegold et al. [19] refined the partitioning idea by considering as well the effects of *translation look-aside buffers (TLBs)* during the partitioning phase, leading to the multi-pass *radix partitioning* join. Conceptually, *radix partitioning* takes all input tuples one-by-one and writes them to their corresponding destination partition (pos[·] keeps the write location in each partition):

```
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]++;
```

Generally, partitions are far apart and on separate VM pages. If the *fan-out* of a partitioning stage is larger than the number of TLB entries in the system, copying each input tuple will cause another TLB miss. The number of TLB entries is thus treated as an upper bound to the partitioning fan-out.

5.2 Software-Managed Buffers

The TLB miss limitations on maximum fan-out can be reduced, when writes are *buffered* inside the cache first. The idea is to allocate a set of buffers, one for each output partition and each with room for up to N input tuples. Buffers are copied to final destinations only when full:

```
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
```

Buffering leads to additional copy overhead. However, for sufficiently small N, all buffers will fit into a single memory page and into L1 cache. Thus, a single TLB entry will suffice unless a buffer becomes full and the code enters the copying routine in line 6. Beyond the TLB entry for the buffer page, an address translation is required only for every Nth input tuple, significantly reducing the pressure on the TLB system. And as soon as TLB misses become infrequent, it is likely that the CPU can hide their latency through out-of-order execution mechanisms. This optimization follows the idea of Satish et al. [22] who used it to reduce the TLB pressure of *radix sort*.

In our implementation of *radix join* we utilize such *software-managed buffers* and configure N such that one buffer will exactly fill one cache line (64 bytes). This in turn allows for another low-level optimization. Since we are now always writing a full cache line at once to global memory, the CPU can take advantage of its *write combining* facilities together with *non-temporal writes*, thus avoiding to read the cache line before writing it back.

In practice, the advantage of software-managed buffers is two-fold: *(i)* for many situations, software-managed buffers offer better absolute performance, since fewer passes can usually achieve the same overall fan-out; *(ii)* it is possible to partition even larger data sizes in a single pass, which has not been considered previously.

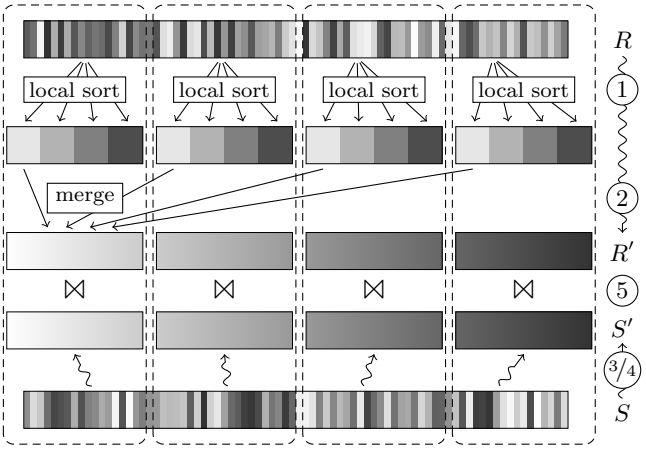


Figure 4: *m-way*: NUMA-aware sort-merge join with multi-way merge and SIMD.

6. JOIN ALGORITHMS ANALYZED

6.1 Sort-Merge Join Algorithm – *m-way*

The *m-way* algorithm is a highly parallel sort-merge join that relies on both data and thread parallelism and is carefully optimized toward NUMA. The general idea and the individual phases of the algorithm are presented in Figure 4 assuming a hypothetical machine with four NUMA regions and one thread per region.

Initially, input relations R and S are stored such that they are equally distributed across NUMA regions. In the first phase, each thread is assigned its NUMA-local chunk and all the threads range-partition their local chunks in parallel using the software-managed buffers technique (Section 5.2). The main intuition behind partitioning in the first place is allowing threads in the subsequent phases to work independently without any synchronization. In this phase, the partitioning fan-out is usually on the order of the number of threads (64–128) and can be done efficiently using a single pass at the speed of total memory bandwidth of the machine. Then, each local partition is sorted using the AVX sorting algorithm. In this phase, different threads can sort different partitions independently, again just reading from and writing to the NUMA-local memory. Partitioning and local sorting are indicated in Figure 4 as (1).

Phase (2) in Figure 4 is the only phase that requires shuffling data between different NUMA regions. Therefore, it is likely to be limited by the memory/interconnect bandwidth. Hence, we employ multi-way merging here as de-

scribed in Section 4.2. Multi-way merging successfully overlaps the data transfer and merging and brings computation and bandwidth into a balance. Outcome of this phase is a globally sorted copy of R, indicated as R' in Figure 4.

The same steps are also applied to relation S (indicated as Phases (3/4) in Figure 4). R' and S' are stored in the NUMA-local memory of each thread. Finally, each thread concurrently evaluates the join between NUMA-local sorted runs using a single-pass merge join (Phase (5)). This join amounts to an extremely fast linear scan of both sorted runs where matching pairs constitute the join result.

6.2 Sort-Merge Join Algorithm – *m-pass*

The second variant for sort-merge join is *m-pass*. The algorithm differs from *m-way* only in Phase (2) in Figure 4. Instead of applying a multi-way merge for merging NUMA-remote runs, *m-pass* 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. As a result of the first iteration, the number of runs reduces to 1/2 of the initial total number of runs. The rest of the merging continues in local memory, using the multi-pass merging technique (cf. Section 3.2.2) in an iterative manner.

6.3 Massively Parallel Sort-Merge Join – *mpsm*

The *mpsm* algorithm first *globally* range-partitions relation R (again as discussed in Section 5.2). This step ensures that different ranges of R are assigned to different NUMA-regions/threads. Next, each thread independently sorts its partition, resulting in a globally-sorted R'. In contrast, S is sorted only *partially*. Each thread sorts its NUMA-local chunk of S without a prior partitioning. Therefore, during the last phase, a run of R must be merge-joined with all the NUMA-remote runs of relation S. For cases where relation S is substantially larger than R, avoiding the global partitioning/sorting may pay off and the overall join may become more efficient. For further details, we refer to [2].

6.4 Radix Hash Join – *radix*

For *parallel radix-hash join*, we partition both input relations as discussed in Section 5.2. The goal is to break at least the smaller input into pieces that fit into caches. Then, we run a cache-local hash join on individual partition pairs. For a detailed discussion, refer to [4, 5, 15, 19].

6.5 No-Partitioning Hash Join – *n-part*

The *no-partitioning join* is a direct parallel version of the canonical hash join. Both input relations are divided into equi-sized portions that are assigned to a number of worker threads. In the build phase, all worker threads populate a *shared* hash table with all tuples of R. After synchronization via a *barrier*, all worker threads enter the probe phase and concurrently find matching join partners for their assigned S portions. For further details, we refer to [4, 5].

Table 1 summarizes the algorithms considered in the experiments and the shorthand notation used in the graphs.

short notation	algorithm
m-way	Sort-merge join with multi-way merging
m-pass	Sort-merge join with multi-pass naïve merging
mpsm	Our impl. of massively parallel sort-merge [2]
radix	Parallel radix hash join [15, 4]
n-part	No-partitioning hash join [5, 4]

Table 1: Algorithms analyzed.

	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

Table 2: Workload characteristics.

7. EXPERIMENTAL SETUP

7.1 Workloads

To facilitate comparisons with existing results, we use similar workloads to those of Kim et al. [15], Albutiu et al. [2] and Balkesen et al. [4]. They all assume a column-oriented storage model and joins are assumed to be evaluated over narrow (8- or 16-byte) $\langle key, payload \rangle$ tuple configurations. To understand the value of data parallelism using vectorized instructions, we assume *key* and *payload* are four bytes each. The workloads used are listed in Table 2. All attributes are integers, but AVX currently only supports floating point; therefore we treat integer keys as floats when operating with AVX.[1] There is a foreign key relationship from S to R. That is, every tuple in S is guaranteed to find exactly one join partner in R. Most experiments (unless noted otherwise) assume a uniform distribution of key values from R in S.

7.2 System

Experiments are run on an Intel Sandy Bridge with a 256-bit AVX instruction set. It has a four-socket configuration, with each CPU socket containing 8 physical cores and 16 thread contexts by the help of the hyper-threading. Cache sizes are 32 KiB for L1, 256 KiB for L2, and 20 MiB L3 (the latter shared by the 16 threads within the socket). The cache line size of the system is 64 bytes. TLB1 contains 64/32 entries when using 4 KiB/2 MiB pages (respectively) and 512 TLB2 entries (page size 4 KiB). Total memory available is 512 GiB (DDR3 at 1600 MHz). The system runs Debian Linux 7.0, kernel version 3.4.4-U5 compiled with transparent huge page support and it uses 2 MiB VM pages for memory allocations transparently. This has been shown to improve join performance up to ≈15% under certain circumstances [3, 4]. Therefore, we assume the availability of large page support in the system. The code is compiled with `gcc` 4.7.2 using `-O3`. Experiments using Intel's `icc` compiler did not show any notable differences, qualitatively or quantitatively.

8. ANALYSIS OF THE SORT PHASE

In a first set of experiments, we make sure that our single-thread sorting performs well compared to alternatives.

Figure 5 shows the performance of our SIMD sorting algorithm implemented using AVX instructions. As a baseline, we include the C++ STL sort algorithm. Overall, AVX sort runs between 2.5 and 3 times faster than the C++ sort. One expected result is also visible: whenever the size of the input increases, both algorithms suffer due to the increasing

[1] AVX2 will support vectorized integer operations, thus there will be no longer semantical differences for our code.

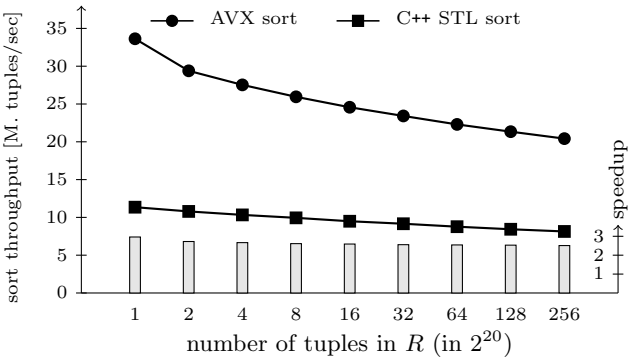


Figure 5: Single-threaded sorting performance where input table size varies from 8 MiB to 2 GiB.

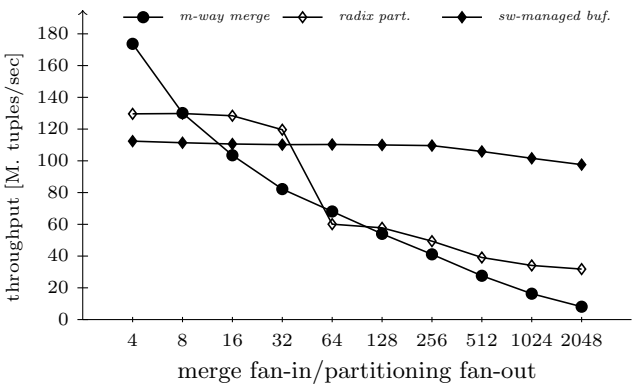


Figure 6: Impact of fan-in/fan-out on multi-way merging/partitioning (1-pass and single-thread).

number of trips to the main-memory. AVX sort mainly suffers from the multi-pass, pair-wise merging of cache-sized sorted runs. As the input size increases, the number of runs double at each data point. Accordingly, the number of trips to the memory also increases logarithmically in the input runs. Whenever used alone, this performance degradation of single-threaded AVX sort might be acceptable since it still runs 2.5 times faster than a serial of-the-shelf algorithm.

On the other hand, the excessive memory bandwidth consumption indicates that there will be a problem when multiple threads are active and contend for the same bandwidth. We look into this problem when discussing the merge phase.

We have also compared our sorting algorithm with those of Chhugani et al. [7] and Kim et al. [15]. For reasons of space, we omit the results. In summary, the sort algorithm behaves well in comparison to existing ones and it does not affect the relative performance of the join algorithms.

9. ANALYSIS OF THE MERGE PHASE

Increasing input sizes require multiple passes over the entire data if sub-lists are pair-wise merged. Multi-way merging remedies this problem by doing the merge in single pass. For efficiency, Kim et al. [15] assume that entire merge trees fit into a single last-level cache. However, as the degree of parallelism, data sizes, and merge fan-ins increase, this assumption no longer holds.

9.1 Modeling the Merge Phase

As described in Section 4.1, overall sorting starts with *in-cache sort* which generates 1/2-L2-cache-sized sorted runs. Therefore, the *fan-in* (F) of the multi-way merge operation equals to the number of 1/2-L2-sized sorted runs: Given an input size of N tuples, $F = {}^{N \times \text{tuple-size}}/_{1/2\text{-L2-size}}$. Secondly, efficient multi-way merging requires the merge tree to reside in the last level (L3) cache. A multi-way merge tree is essentially a binary tree and the number of total nodes (M) in a merge tree with fan-in of F equals to $M = 2^{\lceil log_2 F \rceil + 1} - 1$. Moreover, the shared L3 must be divided by the number of threads (T; *i.e.,* $T = 16$ for our system) in a CPU socket. Accordingly, in order to be L3 resident, each node in the merge tree must have at most B tuples as follows $B = {}^{\text{L3-size}}/_{(M \times \text{tuple-size}) \times T}$. Finally, and most importantly, B must be on the order of hundreds of tuples to ensure that multi-way merge will not degrade to single or a few item-at-a-time (scalar) merging.

9.2 Performance of the Merge Phase

The model we have outlined essentially suggests that increasing fan-in, and hence input sizes, will deteriorate the merge performance. To verify this finding, we performed the following experiment: We increased the number of fixed-size sorted runs from 4 to 2,048 and ran multi-way merging with this number as fan-in. To isolate the effect of concurrent threads, we ran our AVX multi-way merge on a single core using 1/16 of our 20 MiB L3 cache exclusively assigned to the merging buffer. The results are shown in Figure 6 denoted with -●-. The results confirm the model: merge performance decreases steeply with increasing fan-in. The main reason is the decreasing size of the L3 buffer per merge node in the merge tree. For instance, with fan-in of 256, the L3-resident FIFO buffer of each merge node holds just 321 tuples and decreases to 160 tuples with fan-in of 512. At higher fan-in values, multi-way merge becomes less effective as the number of tuples processed at once drops significantly.

Another reason for performance degradation can be observed through performance counters: The increasing depth of the merge tree means a logarithmically increasing number of executed instructions per tuple. At that point, multi-way merging becomes CPU bound as shown in Table 3. The instruction count per tuple increases significantly with increasing fan-in along with the instructions-per-cycle (IPC). On the Intel Sandy Bridge, a maximum of 2 IPC for AVX instructions is barely possible. Consequently, the thread is far from being memory bound (cf. column "Total Bytes/Cycle"). Other profiling metrics are also in line with the expectations: each tuple is read and written once where the write also causes another read for a total of 2 reads and 1 write of the 8-byte tuple (compare with "Read/Tup." and "Write/Tup."). In terms of the cache efficiency, the expected 1/8 (0.125) L3 misses per tuple is also in line with the measured L3 misses.

The cache contention problem in multi-way merging raises a question against the assumption that multi-way merge can be done efficiently. This is possible only for certain data sizes/fan-ins, however increasing data sizes will require multiple passes over the data, affecting the overall performance of the join operator.

10. OPTIMIZING THE MERGE PHASE

The previous results show the importance of the input size on the performance of sort-merge operations. The same

Merge Fan-in	IPC /Core	Instr. /Tup.	Tot.-Bytes. Cyc.	Read /Tup.	Write /Tup.	L3-Miss /Tup.
8	1.04	19.70	1.38	16.53	9.50	0.12
16	1.13	26.74	1.08	16.47	8.96	0.12
32	1.17	34.00	0.87	16.45	8.73	0.12
64	1.23	43.68	0.71	16.47	8.73	0.12
128	1.26	56.81	0.57	16.52	8.92	0.13
256	1.35	79.55	0.44	16.63	9.41	0.13
512	1.46	126.87	0.31	16.83	10.32	0.13
1024	1.64	245.51	0.20	17.32	12.16	0.14

Table 3: Performance counter monitoring results for multi-way merge with increasing merge fan-in.

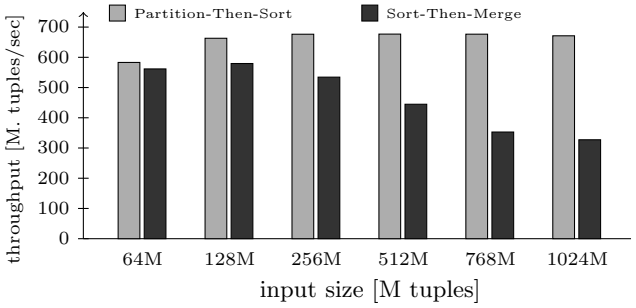


Figure 7: Impact of input size on different multi-threaded sorting approaches (using 64 threads).

problem occurs for the partitioning phase of hash-based joins. Interestingly, the solution for that problem can also be used for sort-merge joins as we show in this section.

10.1 Performance of the Partitioning Phase

Figure 6 shows the performance of partitioning with the same amount of total data as in the analysis for merging. This randomly generated data is partitioned with given fan-out. The results denoted with ⊸◇⊸ show that *radix partitioning* is sensitive to the TLB size, which is 32 in our system for 2 MiB pages. Therefore, partitioning throughput significantly decreases after a fan-out of 32 due to TLB misses. However, the *software-managed buffers* idea described in Section 5.2 is much better (cf. ⊸◆⊸). The robust performance of the software-managed buffers strategy for high fan-outs is clearly a big improvement over the normal radix partitioning previous authors have considered so far.

Partitioning and merging are duals of each other. However, the cost of these two operations differ as shown in Figure 6. When compared with the *software-managed buffering* technique, multi-way merging is significantly slower than partitioning. This raises the question of why not using partitioning with the software-managed buffers technique for sorting. If we also use this technique for hash-based joins, previous assumptions about the number of passes over the data for the different join approaches no longer hold. With the optimized technique, a significant amount of data can be partitioned in a single pass as opposed to the two or more passes previously assumed by Kim et al. [15]. This in turn calls for a more careful comparison of join algorithms under a wider range of data sizes.

10.2 Using Partitioning with Sort

The overall NUMA-aware sorting-based join algorithm, *m-way* is described in Section 6.1. The implementation of

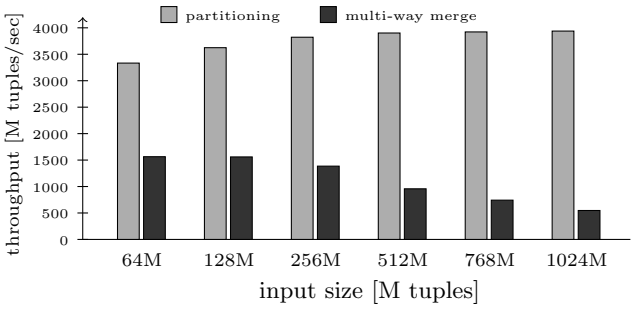


Figure 8: Trade-off between partitioning and merging (using 64 threads).

the first phase, *local sorting*, can choose either partitioning or merging:

Partition-then-Sort approach first range-partitions the input with efficient software-managed radix partitioning (using most-significant radix bits). Each of the partitions are individually sorted using the AVX sort routine, and concatenation of these naturally creates a sorted output.

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

In Figure 7, we compare the performance of the two approaches. The workload consists of 64 M to 1024 M 8-byte tuples. First, the Partition-then-Merge approach achieves a throughput of up to 680 million tuples per second. More importantly, it shows a stable performance with increasing input sizes, sorting 8 GB in less than 2 seconds. On the other hand, the performance of Sort-then-Merge approach drops significantly beyond table sizes of 256 M; mainly due to increasing fan-in of the multi-way merge. For instance with table size of 512 M, multi-way merge runs with a fan-in of 512. The main performance difference stems from partitioning vs. merging performance. Figure 8 shows the throughput achieved for merging and partitioning phases in isolation. Partitioning achieves up to 4 billion tuples per second by effectively utilizing all the available bandwidth in our machine ($4\,\text{billion} \times 8$ bytes $\times\, 4 \approx 128\,\text{GB/s}$; *i.e.*, 3 reads/1 write for radix partitioning), whereas merge performance drop significantly from 1.5 to 0.5 billion tuples per second.

10.3 Alternative Implementations for Merge

The new way of doing the merge seems promising but needs to be evaluated against alternative proposals from the literature. Chhugani et al. [7] claim that multi-way merging accommodates parallelism in a natural way and works well for multi-core processors. However, as the available degree of hardware parallelism increases, contention in large merge trees also increases. This contention is possibly not visible in four-core configurations considered by Chhugani et al. [7]. But it critically affects performance in modern systems with a few ten hardware threads. It turns out that the new approach we propose to merging, also addresses this problem. We have confirmed this experimentally by implementing the two different approaches to multi-way merging:

The *cooperative m-way* approach follows the original idea by Chhugani et al. [7] where there is a single multi-way merge tree and multiple threads cooperatively merge the

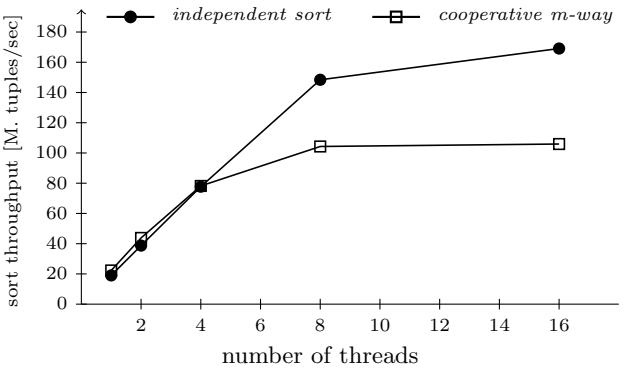


Figure 9: Scalability of cooperative multi-way merging vs. synchronization-free independent sorting. Using only one CPU socket.

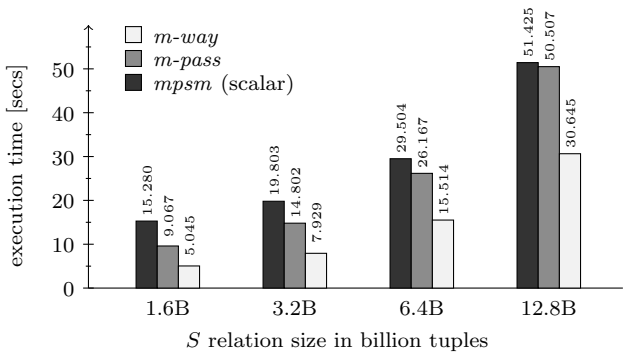


Figure 10: Execution time comparison of sort-merge join algorithms. Workload A, 64 threads.

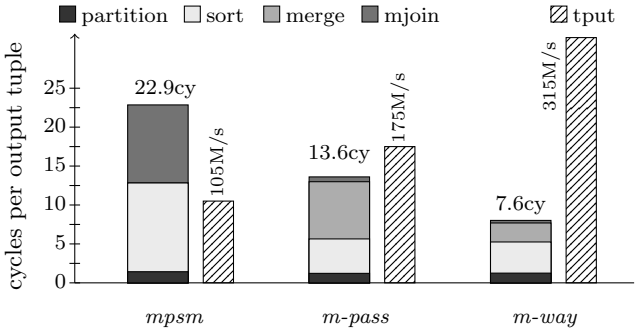


Figure 11: Performance breakdown for sort-merge join algorithms. Workload A. Throughput metric is output tuples per second, *i.e.* $|S|/\text{execution time}$.

data. Once the children of a node have enough data, the node becomes available for merging by any of the threads. This approach has a potential advantage: It increases the buffer space per merge node as there is a single merge tree resident in last-level cache (in Section 9 we showed that buffer space is important for merge performance).

The *independent sorting* approach follows the *Partition-then-Sort* idea discussed in Section 10.2. Each thread locally partitions its data and then sorts smaller individual partitions. In this case, the thread can independently merge the sorted runs without further interaction with other threads.

Figure 9 shows the throughput of the two approaches when sorting an input relation of 256 M tuples. We intentionally restrict to a single socket to ensure that all threads use the same last-level cache. As the figure shows, cooperative multi-way merging 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. The independent sorting approach, in contrast, scales linearly up to the physical number of cores as shown in Figure 9. However, the scalability in the hyper threads region remains limited. This is the common case for hardware-conscious algorithms where most of the physical hardware resources are shared among hyper-threads. As a conclusion, even though all the threads have a fraction of the last level cache for their multi-way merge, the synchronization-free nature of this approach shows its benefit and *independent sorting* proves to be better than the originally proposed cooperative multi-way merging.

11. SORT-MERGE JOINS

After identifying the factors affecting the performance of the components of a sort-merge join algorithm and choosing the best-possible implementations for the different phases, we now compare the performance of the resulting sort-merge join operator (*m-way*) with that of *mpsm* and *m-pass*.

11.1 Comparison with Different Table Sizes

We run first an experiment for different sizes of table S using Workload **A** shown in Figure 10. The results show that *m-way* runs significantly faster than the other options and is robust across different relation size ratios while producing fully sorted output. Algorithm *mpsm* draws its true benefit from only sorting the smaller of the input tables completely whereas the larger one is only partially sorted. Yet, *mpsm* can match the performance of *m-pass* only when the S relation is significantly large ($\gg$ 12.8 billion tuples (100 GiB)). Nonetheless, *mpsm* is a scalar algorithm applicable to wider keys as well. For reasons of space, we omit the results of scalar sorting-based joins that are also applicable to wider keys. In general, the scalar *m-way* has a good performance despite not using SIMD and performs better than *mpsm* even with 8-byte tuples and large S relations.

Figure 11 shows execution time breakdown and throughput for the equal-sized table case in Workload **A** using 64 threads. First, the merge phase in *m-way* is 3 times faster than *m-pass* with bandwidth-aware multi-way merging. Second, in contrast to *mpsm*, the "mjoin" phase is a linear merge-join operation on NUMA-local sorted runs in the other algorithms and overhead of that phase becomes negligible.

11.2 Dissecting the Speedup of *m-way*

In order to understand the efficiency of *m-way*, we calculated the speedup ratios of *m-way* over the other algorithms (Figure 12). The bars denoted with "speedup from merge" shows the speedup of *m-way* attained over *m-pass*. This metric reflects the actual benefit of multi-way merging alone. As seen in the figure, up to 16 threads the speedup from multi-way merge is $\approx$1.5X in which case there is enough aggregate memory bandwidth for that number of threads. However, once the number of threads go beyond 16, memory bandwidth per thread becomes limited and multi-way

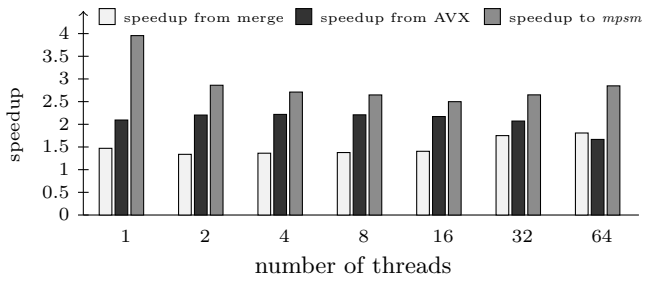


Figure 12: Speedup of *m-way* due to parallelism from AVX and efficiency from multi-way merge.

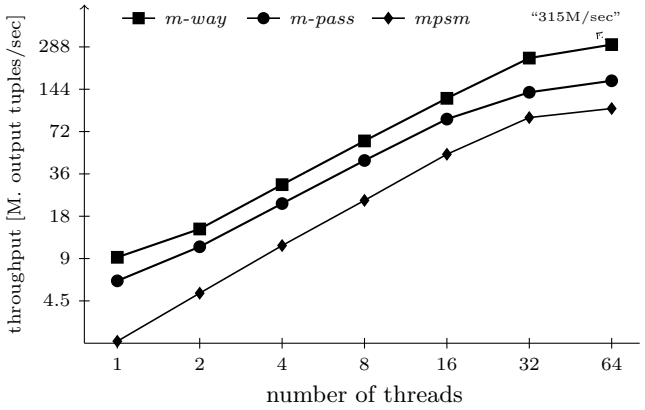


Figure 13: Scalability of sorting-based joins. Workload A, (11.92 GiB ⋈ 11.92 GiB). Throughput metric is output tuples per second, *i.e.* $|S|/\text{execution time}$.

merge benefit goes up to a factor of 2. The bars denoted with "speedup from AVX" show the speedup of *m-way* attained over the same algorithm's scalar variant. Generally, speedup from AVX is between 2X and 2.3X. Lastly, the overall speedup of *m-way* over *mpsm* is ≈3X.

11.3 Scalability of Sort-based Join Algorithms

Figure 13 shows the scalability of sorting-based join algorithms with increasing number of threads where both axes are in logarithmic scale. All the algorithms exhibit linear scaling behavior up to 16 physical CPU cores. However, as all of these algorithms are cache- and CPU resource-sensitive, the scaling with hyper threads is rather limited.

12. SORT OR HASH?

In this section, we present the best sort and hash join algorithms side-by-side under a wide range of parameters.

12.1 Comparison with Different Workloads

The results of best sorting-based join algorithm *m-way* and best hash join algorithm *radix* in our study are summarized in Figure 14 over various workloads.

In Figure 14 we observe that hash-based join algorithms still maintain an edge over sort-based counterparts. 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.

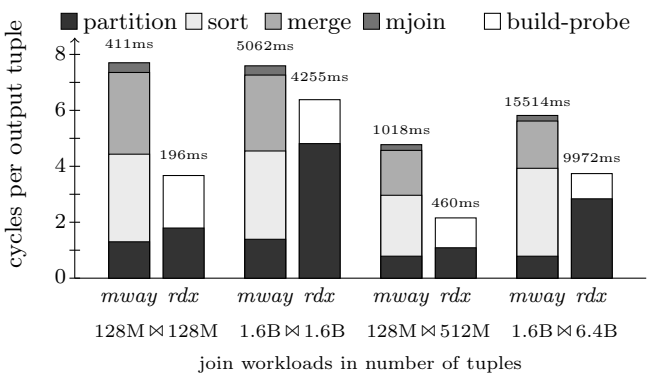


Figure 14: Comparison of best sort vs. best hash join algorithms with cycles per output tuple metric under different workloads. Using 64 threads.

Despite all the optimizations discussed in this paper and the performance boost through 256-bit vector instructions, sort-merge joins cannot match the performance of hash joins on existing recent hardware and for these workloads.

For significantly larger workloads (as suggested by Albutiu et al. [2]) the picture becomes more favorable for sort-merge joins. Input table sizes in the range of billions of tuples make *m-way* more competitive with *radix*. Since *m-way* produces fully sorted output, it could be a good choice for large data volumes that must be further processed.

12.2 Effect of Input Size

The effect of the input relation size can be better captured by the following experiment: we vary the size of each equi-sized R and S tables from 128M tuples (≈ 1 GB) to 1,920M tuples (≈ 15 GB) and run *m-way* and *radix* at each data point using 64 threads. The results are shown in Figure 15.

The conclusion is clear: sort-merge join approaches the performance of radix only when the input tables become significantly large. Radix join performance degrades with the increasing input size due to the increasing number of passes for the partitioning phase. To illustrate, radix configurations vary from 1 pass/12 bits up to 1 billion tuples and from that point on resorts back to 2 pass/18 bits optimal configurations. The optimized radix partitioning with software-managed buffers is 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). Consequently, the partitioning performance degrades gracefully up to L2 cache size. Once the partitioning requirement goes above 16,384, 2-pass partitioning, each pass with 9-bits, and fully L1-resident buffers become the best option. Further, sort-merge join demonstrates robust performance due to the optimizations previously discussed in this paper. Finally, sorted output in query plans is an attractive argument to make sort-merge joins even more favorable in this range of the spectrum of input sizes.

We have also analyzed the potential impact of relative table sizes (partly shown in Figure 14). The experiments show that both algorithms are insensitive to the variance in relation size ratios, being affected only by the size of the output as the amount of data grows. We omit the graphs

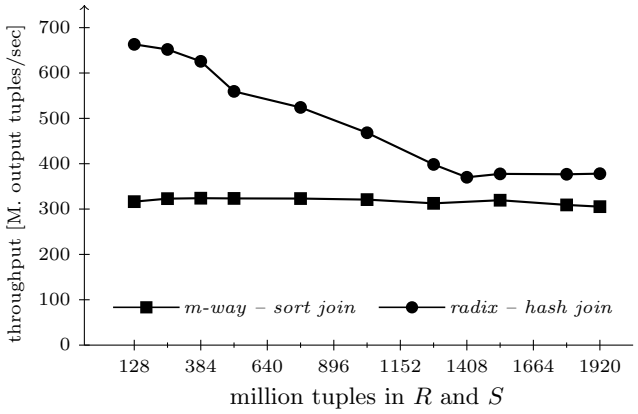


Figure 15: Sort vs. hash with increasing input table sizes ($|R| = |S|$). Throughput metric is total output tuples per second, *i.e.* $|S|/\text{execution time}$.

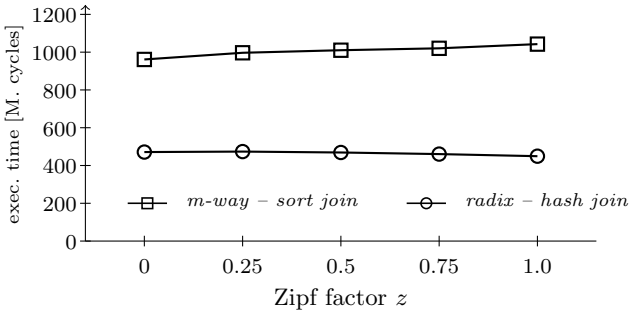


Figure 16: Join performance when foreign key references follow a Zipfian distribution. Workload B.

and more detailed discussions for reasons of space.

12.3 Effect of Skew

In this section, we take a look at the effect when the foreign key column in table S follows a Zipf distribution.

For handling skew in parallel radix hash join, we previously proposed a *fine-granular task decomposition* method. The key idea is to parallelize the processing of larger partitions through a refined partitioning. We refer to [4, 15] for details.

The first and second phases in *m-way* are not vulnerable to skew since all the threads have an equal share of the data. Phase ② in Figure 4, multi-way merging, is prone to skew.

We handle the skew in two steps: ① When creating a multi-way merge task for the thread, if the total size of the merge exceeds an expected average size, we create multiple merge tasks by finding boundaries within each of the sorted runs with binary search. These tasks are then inserted into a NUMA-local task queue shared by the threads in the same NUMA region. ② For extremely large tasks, we identify heavy hitters by computing an equi-depth histogram over sorted runs (which is a fast operation) in a similar approach to Albutiu et al. [2]. Then heavy hitters are directly copied to their output targets without a need for merging.

Figure 16 illustrates the behavior of *m-way* and *radix* with increasing Zipf skew factor. The enhancements to *m-way*, explicit skew and heavy-hitter handling mechanisms, result in a robust performance against skewed distribution of keys while showing less than 10 % overhead. On the other hand, radix join is also robust against skew with the fine-granular task decomposition technique. Overall, the results in Figure 16 show that the comparison of sort vs. hash joins does not significantly change due to skew.

12.4 Scalability Comparison

We compare the algorithms in terms of scalability by varying the number of threads up to the available 64 threads in our system. Threads are assigned to NUMA regions in a round-robin fashion. Between 32 and 64 threads, algorithms use SMT (hyper-)threads. Results for two different workloads are shown in Figure 17.

First, both algorithms show almost linear scalability up to the physical number of cores. Consequently, *radix hash join* achieves approximately 650 million whereas *m-way sort-merge join* achieves approximately 315 million output tuples per second. As mentioned earlier, performance gap between algorithms closes only with significant input sizes as in Figure 17(b), where the number of passes for radix partitioning increases to two. On the other hand, only within the SMT region, algorithms scale poorly. This is an inevitable, well-known situation for hardware-conscious algorithms: SMT provides the illusion of running 2 threads on a single core where almost all of the resources are shared.

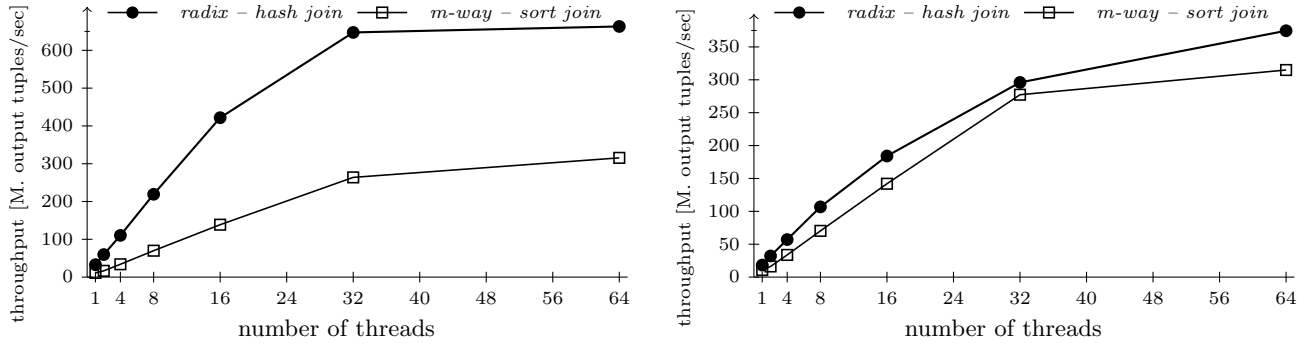
Finally, SMT scalability for radix join are different in the two workloads mainly because of the optimal partitioning configuration: In Figure 17(a), radix join runs with single-pass partitioning with fan-out of 4,096 that fully stresses the L2 cache. Whenever the other thread starts to contend for the L2 cache, SMT shows no benefit apart from hiding the self-caused misses. However, in Figure 17(b), the partitioning buffer is L1 resident with a fan-out of 512 in each of the two passes. Therefore, a limited benefit can be observed where certain amount of L1 misses are hidden by SMT.

12.5 Sort *vs.* Hash with All Algorithms

In this section, we bring all of the state-of-the-art join strategies together for a side-by-side comparison using common workloads. The results are shown in Figure 18. Radix hash join comes out as the fastest algorithm in this comparison. Albutiu et al. [2] previously compared their *massively-parallel sort-merge join* (*mpsm*) algorithm to *no-partitioning join* (*n-part*) implementation of Blanas et al. [5] and found out that sort-merge joins are already faster than hash joins. However, in our recent work [4], we have optimized the no-partitioning join idea further (nevertheless it is still not fully NUMA-aware as the hash table is spread over NUMA regions). Therefore, we extend the comparison of Albutiu et al. [2] with our optimized implementations of the algorithms. The results in Figure 18 indicate that *mpsm* and *n-part* algorithms in fact achieve similar performance. Optimized *n-part* is only faster by ≈10-15 % while lacking the partially sorted output benefit of *mpsm*. Nevertheless, all the results clearly indicate that hash joins are still faster than sort-merge join counterparts.

13. CONCLUSIONS

As hardware changes, the "Sort *vs.* Hash" question needs to be revisited regularly over time. In this paper, we look



(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.* $^{|S|}/_{\text{execution time}}$.

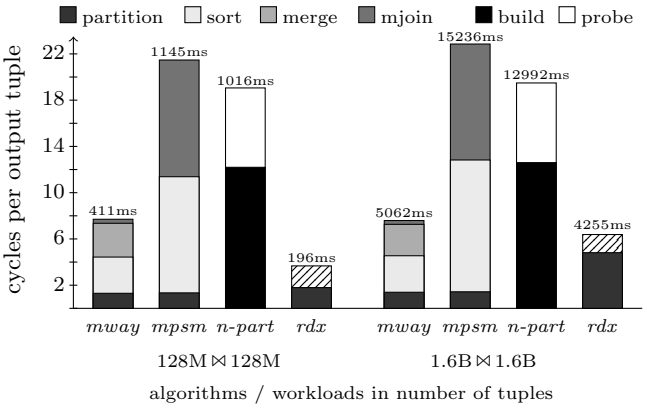


Figure 18: Sort vs. hash join comparison with extended set of algorithms. All using 64 threads.

at it in the context of in-memory data processing for recent multi-core machines. Our results provide the fastest in-memory join processing algorithms using sorting (2–3 times faster than available results) and hashing. Moreover, we show that hash-based join algorithms still have an edge over sort-merge joins despite the advances on the hardware side such as 256-bit SIMD. Finally, sort-merge join turns out to be more comparable in performance to radix-hash join with very large input sizes.

All the code used to obtain results in this paper is available at http://www.systems.ethz.ch/projects/paralleljoins.

Acknowledgements

This work was supported by the Swiss National Science Foundation (Ambizione grant; project Avalanche), by the Enterprise Computing Center (ECC) of ETH Zurich, and by Deutsche Forschungsgemeinschaft (DFG; SFB 876 "Providing Information by Resource-Constrained Analysis")

14. REFERENCES

[1] Intel architecture code analyzer. `http://software.intel.com/en-us/articles/intel-architecture-code-analyzer`. Online, accessed February 2013.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. Technical report, ETH Zurich, Nov. 2012.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*, pages 37–48, 2011.

[6] K. Bratbergsengen. Hashing methods and relational algebra operations. In *VLDB*, pages 323–333, 1984.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.

[8] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986.

[9] S. Fushimi et al. An overview of the system software of a parallel relational database machine grace. In *VLDB*, 1986.

[10] B. Gedik, R. Bordawekar, and P. S. Yu. Cellsort: High performance sorting on the cell processor. In *VLDB*, 2007.

[11] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *ICDE*, pages 406–417, 1994.

[12] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *IEEE Trans. Knowl. Data Eng.*, 6(6):934–944, 1994.

[13] H. Inoue et al. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *PACT*, pages 189–198, 2007.

[14] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. Cloudramsort: fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012.

[15] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[16] M. Kitsuregawa et al. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1), 1983.

[17] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[18] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[19] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[20] T. H. Merrett. Why sort-merge gives the best implementation of the natural join. *SIGMOD Rec.*, 13(2):39–51, Jan. 1983.

[21] R. Müller, J. Teubner, and G. Alonso. Sorting networks on fpgas. *VLDB J.*, 21(1), 2012.

[22] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.

[23] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD '89, pages 110–121, 1989.

[24] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.