A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort

Orestis Polychroniou
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

ABSTRACT

Analytical database systems can achieve high throughput main-memory query execution by being aware of the dynamics of highly-parallel modern hardware. Such systems rely on partitioning to cluster or divide data into smaller pieces and thus achieve better parallelism and memory locality. This paper considers a comprehensive collection of variants of main-memory partitioning tuned for various layers of the memory hierarchy. We revisit the pitfalls of in-cache partitioning, and utilizing the crucial performance factors, we introduce new variants for partitioning out-of-cache. Besides non-in-place variants where linear extra space is used, we introduce large-scale in-place variants, and propose NUMA-aware partitioning that guarantees locality on multiple processors. Also, we make range partitioning comparably fast with hash or radix, by designing a novel cache-resident index to compute ranges. All variants are combined to build three NUMA-aware sorting algorithms: a stable LSB radix-sort; an in-place MSB radix-sort using different variants across memory layers; and a comparison-sort utilizing wide-fanout range partitioning and SIMD-optimal in-cache sorting. To the best of our knowledge, all three are the fastest to date on billion-scale inputs for both dense and sparse key domains. As shown for sorting, our work can serve as a tool for building other operations (e.g., join, aggregation) by combining the most suitable variants that best meet the design goals.

1. INTRODUCTION

The increasing main-memory capacity of contemporary hardware allows query execution to occur entirely in memory. If the entire database also fits in RAM, analytical query workloads that are typically read-only need no disk access after the initial load, setting the memory bandwidth as the only performance bound. Since analytics are at the core of business intelligence tasks today, the need for high-throughput main-memory query execution is apparent.

*This work was supported by National Science Foundation grant IIS-0915956 and a gift from Oracle Corporation.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610522.

To maximize memory bandwidth and capacity, a few CPUs can be combined in a shared-memory system using a fast interconnection. Such hardware combines the parallelism of multiple multi-core CPUs with a higher aggregate memory bandwidth. The shared-memory functionality is provided by the non-uniform-memory-access (NUMA) interconnection, adding an additional layer in the memory hierarchy.

In a modern multi-core CPU, the best performance is achieved when all cores work in a shared-nothing fashion and the working set is small enough to fit in the fast (and private per core) caches. The same approach was more efficient even before the advent of the multi-core era [11], since random RAM accesses are too expensive out-of-cache.

Query execution is decomposed into a series of operations, the most time consuming of which are typically joins and aggregations. To speed up these operations using hardware parallelism, we partition into small pieces using the keys, then process each piece independently. For instance, an efficient algorithm for joins is to hash partition in parallel until the input is split into cache resident pieces before we execute the join using a hash table [11]. In fact, even in-cache join can further partition into trivial parts with very few distinct items, before executing a nested loop to join them [7].

This paper considers a comprehensive menu of partitioning options across several dimensions. The three *types* of partitioning are hash, radix and range partitioning, depending on the function that takes the key as an input and outputs the destination partition. Partitioning also depends on the layer of the memory hierarchy that it targets, namely in-cache, out-of-cache and across NUMA regions. Finally, we distinguish partitioning variants based on whether they use auxiliary space that is linear to the input size or not.

Until recently, prior work used the in-cache versions of partitioning and, if parallel, the non-in-place variant, which can be trivially distributed across threads. In all cases, when the input is larger than the cache, the performance is throttled by TLB misses [11] and cache conflicts [14]. Satish et al. [14] suggested in-cache buffering to mitigate TLB misses and Wassenberg et al. [15] used non-temporal writes on cache line sized buffers to facilitate hardware write-combining.

Efficient out-of-cache partitioning [14, 15] assumes free access to linear auxiliary space, to write the output. We introduce several in-place out-of-cache partitioning variants utilizing the same crucial performance factors: an in-place method analogous to the shared-nothing non-in-place method; a modified non-in-place method that generates the output as a list of blocks that overwrites the input; and a parallel non-in-place method that combines the previous two.

To support scaling to multiple CPUs, we consider how the NUMA layer affects partitioning performance and modify both in-place and non-in-place out-of-cache partitioning to guarantee minimal transfers across NUMA boundaries, also ensuring sequential accesses so that hardware pre-fetching can hide the latency of the NUMA interconnection [1].

All of the above algorithms target the *data shuffling* part of partitioning and implicitly assume that the *partition function* is cheap and can be computed at virtually no cost. While this assumption holds for radix and hash partitioning given a suitable hash function choice, the cost of computing a range partition function is higher than the cost to transfer the tuple, especially when the number of partitions increases beyond the TLB capacity. The standard (and slow) implementation is a binary search in a sorted array of delimiters that define the partition ranges. The slowdown is caused by the logarithmic number of cache loads. We introduce a specialized SIMD-based cache-resident index that speeds up range function computation up to 6 times and makes range partitioning a practical choice for many applications.

All partitioning variants discussed in this paper are shown in Figure 1, where we also mark our contributions. We apply all variants to design and implement three large-scale NUMA-aware sorting algorithms. We use sorting, rather than joins or aggregations for two reasons. First, because it is a wider problem that can be a sub-problem for both join and aggregation. Second, we can apply *all* partitioning variants to build *unique* sorting algorithms, such that each is more scalable in distinct cases depending on input size, key domain size, space requirements, and skew efficiency.

The first sorting algorithm we propose is stable least-significant-bit (LSB) radix-sort based on non-in-place out-of-cache radix partitioning [14] where we add two innovations. We use hybrid range-radix partitioning to provide perfect load balancing, and guarantee that each tuple will cross NUMA boundaries at most once, even if the algorithm would by default re-organize the entire array in each pass.

The second sorting algorithm we propose is an in-place most-significant-bit (MSB) radix-sort that uses all variants of in-place partitioning that we introduce, one for each distinctive level in the memory hierarchy: shared out-of-cache in-place partitioning, shared-nothing out-of-cache partitioning, and in-cache partitioning. We reuse the range-radix idea of LSB radix-sort for NUMA optimality and load balancing.

The third sorting algorithm we propose is a comparison-sort that uses the newly optimized range partitioning. We perform very few out-of-cache range partitioning passes with a very wide fanout until we reach the cache, providing NUMA optimality. In the cache, we employ sorting [6] that scales to the SIMD length, modified to use the cache more effectively.

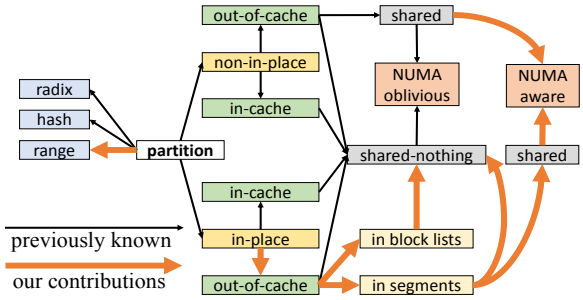

Figure 1: Partitioning variants and contributions

We use fixed length integer keys and payloads, typical of analytical database applications. We evaluate on both dense and sparse key domains. If order-preserving compression is used [12, 16], any sparse or dense domain with fixed or variable length data is compacted into a dense integer domain.

Skewed workload distribution can reduce parallelism in some naïve approaches. In our context, when we statically distribute partitions to threads, we ensure that the partitions are as balanced as possible. Specifically, we never use radix partitioning to any range of bits to divide the workload. Instead, we combine range with radix partitioning to guarantee that, if specific bit ranges have very few distinct values, we can find delimiters that split the workload equally among threads, independently of the key value range.

We summarize our contributions:

- We introduce several new variants for main-memory partitioning, most notably large-scale in-place partitioning and efficient range partitioning, and also guarantee minimal NUMA transfers across multiple CPUs.
- We combine partitioning variants to design three sorting algorithms, all the fastest of their class for billion-scale inputs, and evaluate the best options depending on input size, key domain, available space, and skew.

The rest of the paper is organized as follows. Section 2 outlines related work. In Section 3 we describe partitioning variants. In Section 4 we discuss sorting. Section 5 presents our experimental results, and we conclude in Section 6.

2. RELATED WORK

We first outline related work on partitioning. Manegold et al. [11] identified the TLB thrashing problem when naïvely partitioning to a large number of outputs. Satish et al. [14] introduced efficient out-of-cache partitioning and Wassenberg et al. [15] identified the significance of write-combining. Manegold et al. [11] proposed partitioning to cache-resident hash tables to join and Kim et al. [7] reused the same design on a multi-core CPU. Wu et al. [17] proposed hardware accelerated partitioning for performance and power efficiency.

We briefly outline recent work on sorting for modern hardware, due to space constraints. Inoue et al. [6] proposed in-cache SIMD-vector comb-sort followed 2-way SIMD merging. Chhugani et al. [5] proposed in-cache sorting networks followed by cyclic merging in buffers to avoid being memory bound. Kim et al. [7] compared sort-merge-join against hash join, projecting that sort-merge-join will eventually outperform hash with wider SIMD. Satish et al. [14] compared radix-sort and merge-sort in CPUs and GPUs on multiple key domain sizes and concluded in favor of merge-sort. However, the result is based on small arrays and only LSB radix-sort is considered. Wassenberg et al. [15] improved over Satish et al. [14] and claimed that radix-sort is better. Kim et al. [9] studied network-scale sorting maximizing network transfer with CPU computation overlap. Albutiu et al. [1] studied the NUMA effects using sort-merge-join on multiple CPUs with billion-scale arrays. Balkesen et al. [4] claimed that non-partitioning hash joins are competitive, but Balkesen et al. [3] improved over Blanas et al. [4] and concluded that partitioning joins are generally faster, even without using fast partitioning [14, 15]. Balkesen et al. [2] further improved joins on multiple CPUs using fast partitioning. Thus, on the fastest CPUs [3], partitioning appears to be the best choice for both hash joins and radix-sort-merge-joins.

3. PARTITIONING

3.1 In-Cache

We start by considering the versions that best operate when the table fits in the cache. The non-in-place version (Algorithm 1) uses a separate array from the input to store the output, while the in-place version (Algorithm 2) uses one array for both input and output. Each partition is generated in a single segment. In-cache partitioning can be run in parallel, if the threads operate in a shared-nothing fashion.

Algorithm 1 Non-in-place in-cache partitioning

```
i ← 0                                   // P: the number of partitions
for p ← 0 to P-1 do
  offset[p] ← i          // point at the start of each partition
  i ← i + histogram[p]
end for
for i_in ← 0 to |T_in|-1 do
  t ← T_in[i_in]                        // T_in: the input table
  i_out ← offset[f(t.key)]++ // f: the partition function
  T_out[i_out] ← t                      // T_out: the output table
end for
```

The simplest non-in-place version does only two random accesses per item. When operating in the cache, we need the output and the offset array to be cache-resident. A slightly more complicated version of the algorithm allows the partitioning to happen in-place, by swapping items across locations. In short, we start by reading an item, find the correct partition and the output destination through the offset array, swap it with the item stored there, and continue for the new item until the cycle is closed. Each item is moved exactly once and we stop when the whole array is covered.

Item swaps are performed in cycles of transfers, defined as *swap cycles*. When the items are processed low-to-high [1], the cycle starts by doing a read and then swaps until it reaches the same location it initially read from, to write back. This case occurs $1/P$ of time on average but requires branching. In Algorithm 2 below, the partitions are written high-to-low and swap cycles close when all items of a partition have been placed, avoiding branching for every tuple.

Algorithm 2 In-place in-cache partitioning

```
i ← 0                                   // P: the number of partitions
for p ← 0 to P-1 do
  i ← i + histogram[p]
  offset[p] ← i          // point at the end of each partition
end for
p ← i_end ← 0
while histogram[p] = 0 do
  p++                                   // skip initial empty partitions
end while
repeat
  t ← T[i_end]                          // T: the input & output table
  repeat
    p ← f(t.key)                        // f: the partition function
    i ← --offset[p]
    T[i] ↔ t                                                  // swap
  until i = i_end
  repeat
    i_end ← i_end + histogram[p++]      // skip if empty
  until p = P or i_end ≠ offset[p]
until p = P
```

3.2 Out-of-Cache

Out-of-cache performance is throttled by increased cache conflicts [14] and cache pollution with output tuples [15]. TLB thrashing occurs when the number of partitions exceeds the TLB capacity [11], unless the entire dataset can be placed in equally few large OS pages to be TLB resident.

3.2.1 Non-in-place

To mitigate these problems, prior work [14] proposed using the cache as an intermediate buffer before writing back to memory. Also, when write backs occur, they bypass the higher cache levels entirely and avoid polluting the cache [15]. Recent work [2] uses the same basic technique for out-of-cache radix partitioning during hash join execution.

Buffering data for each partition reduces the working set size and eliminates the TLB problem when operating in the buffer. TLB misses still occur, but $1/L$ of the time, if L is the number of tuples buffered for each partition before writing to output. If the buffer for each partition is exactly as big as a cache line, writing the full cache line to memory is accelerated by write-combining and avoids polluting the higher cache levels with output data. The partitioning fanout is now bounded by the number of cache lines in the fast core-private cache, rather than the TLB entries. Buffer flushing is optimally done using wider registers [15]. To maximize the cache use, we use the last buffer slot to save the output offset and access one cache line per iteration (Algorithm 3).

To extend the above method to multiple columns stored in separate arrays, the standard case in RAM-resident database data, we use one cache line per column in the buffer of each partition. A generic implementation can use one cache line per column and flush it separately depending on the column width. We can also interleave the columns in a single tuple and de-interleave the columns when the buffer is flushed. For example, when partitioning arrays of 32-bit keys and 32-bit payloads, we store 64-bit tuples in the cached buffer. Tuple (de-)interleaving can be accelerated using SIMD.

Parallel execution of the non-in-place out-of-cache partitioning is trivial. The input can be split to equal pieces, one for each thread. By executing a prefix sum of all individual histograms, one can ensure that each partition output is written in a distinct location. Threads are only synchronized after individual histograms are built. This is the only known technique for parallel partitioning on shared segments.

Algorithm 3 Non-in-place out-of-cache partitioning

```
i_out ← 0                               // P: the number of partitions
for p ← 0 to P-1 do
  buffer[p][L-1] ← i_out    // L: # of tuples per cache line
  i_out ← i_out + histogram[p]
end for
for i_in ← 0 to |T_in|-1 do
  t ← T_in[i_in]          // T_in/T_out: the input/output table
  p ← f(t.key)                          // f: the partition function
  i_out ← buffer[p][L-1]++
  buffer[p][i_out mod L] ← t
  if i_out mod L = L-1 then
    for i_buf ← 0 to L-1 do
      T_out[i_out + i_buf - L] ← buffer[p][i_buf] // no cache
    end for
    buffer[p][L-1] ← i_out + 1
  end if
end for
```

3.2.2 In-place, Shared-Nothing Segments

Adapting the out-of-cache buffering technique to in-place partitioning requires a more complicated approach. The basic idea is to perform the swaps inside the buffer, so that the sparse RAM locations are accessed only $1/L$ of the time, reducing the overhead from TLB misses. Compared with non-in-place out-of-cache partitioning, which uses the buffer as an intermediate layer to group tuples before writing them, in-place out-of-cache partitioning performs all tuples swaps in the buffer, and accesses RAM one cache line at-a-time.

Before the main partitioning loop starts, we load cache lines from all starting partition locations. Item swaps between partitions occur using the last L tuples that are stored in the buffer. When a buffer has swapped all L items, the cache line is streamed to the RAM location it was loaded from and the buffer is re-filled with the next L items of the same partition. Thus, we operate in the buffer $(L-1)/L$ of the time and do not miss in the TLB. The offsets are stored inside the buffer and the last L items of each partition are handled differently to eliminate branching in the inner loop.

If having T contiguous segments per partition (T is the number of threads) is acceptable, then we can run in-place partitioning in parallel. However, unlike the non-in-place variant, generating one segment per partition across threads is impossible with coarse-grained synchronization.

Algorithm 4 In-place out-of-cache partitioning

```
i ← 0                                   // P: the number of partitions
for p ← 0 to P-1 do
  end[p] ← i
  i ← i + histogram[p]
  for i_buf ← 0 to L-1 do
    buffer[p][i_buf] ← T[i − (i mod L) + i_buf]
  end for
  item_0[p] ← buffer[p][0]      // save 1st item out of buffer
  buffer[p][0] ← i
  [...]     // special handling for partitions smaller than L
end for
p ← 0
while histogram[p] = 0 do
  p++                                   // skip initial empty partitions
end while
t ← T[0]                                // T: the input & output table
loop
  repeat
    p ← f(t.key)                        // f: the partition function
    i ← −−buffer[p][0]
    buffer[p][i mod L] ↔ t                                  // swap
  until i mod L = 0     // L: # of tuples per cache line
  [...]     // (rare) branch for end of partition (exits here)
  for i_buf ← 0 to L-1 do
    T[i] ← buffer[p][i_buf]                                 // no cache
  end for
  for i_buf ← 0 to L-1 do
    buffer[p][i_buf] ← T[i − L]                             // cache
  end for
  t ← item_0[p]
  item_0[p] ← buffer[p][0]
  buffer[p][0] ← i
  if i − end[p] < L then
    [...]       // (rare) branch for last L items of partition
  end if
end loop
```

3.2.3 In-place, List of Blocks

For large-scale out-of-cache partitioning, the requirement of producing all partitions in P non-splitting segments can be relaxed. Instead of writing each partition output sequentially, we can write large blocks that only contain items from a single partition. When the block is full, we get a new block at some new available location. The block size must be large enough to amortize sequential writes, but not too large, in order to avoid external fragmentation from non-full blocks.

To access data from a single partition only, we create a small linked list that connects all blocks that contain data of the same partition. While the access is not entirely sequential as in the single segment case, the list hops after scanning each block are amortized by a sufficient block size.

This method can be done in place, if we remove $P \cdot B$ items from the start of the input and save it in private space (B is the block capacity in tuples). We start range partitioning the input from the $(P \cdot B)$-th tuple. By the time any partition is filled, the input pointer will have advanced enough for the output to safely use the space of input we read before, without overwriting tuples not yet read. At the end, we also add the data initially copied out back to the correct block lists. For each partition, only the last block of the block list can be non-full. Thus, unused space has an upper bound of $P \cdot B$ and is negligible compared to size of the input.

Block-based partitioning has a number of nice properties. First, it uses the fast non-in-place out-of-cache partitioning. Second, it does not require the pre-computation of a histogram. Third, it can be done in place by ensuring no overlap between input and output data. Finally thread parallelism is trivial; the only requirement is to connect the linked lists of blocks from all threads for each partition.

3.2.4 In-place, Shared Segments

In order to partition and shuffle data in parallel inside the same segment, we need fine-grain synchronization. Since using OS latches are overly expensive, we use atomic instructions. Atomic `fetch-and-add` reads a memory location, increments it by some value and returns its previous value. Imagine an array of items and multiple threads where each item must be processed by exactly one thread. Each thread can safely use item at index i which is returned by invoking `fetch-and-add`(c,1) on a shared counter c. When done, the thread asks for the next item to process or terminates if i exceeds the number of items. We apply the same idea to in-place partitioning using one shared counter for each partition to represent the number of tuples swapped so far.

We use `fetch-and-add` on the shared counter of partition p, to "lock" the cell of the next yet unread item of partition p. We store the first index that initiates the cycle. After swapping an arbitrary number of keys, when we return to the original partition p, we store the last tuple in the initial location. Only the P counters are shared across threads.

As mentioned in Section 3.1, we define a *swap cycle* as a sequence of swaps that starts by reading a key from a specific partition and after a number of swaps, returns to the same partition to write a key in the initially read location. We cannot know in advance how large a swap cycle will be, thus we cannot lock all locations the cycle will go through before actually moving tuples. Imagine a scenario where the first partition has only one item found in the last cell of the array. Then, one thread would perform a single swap cycle covering all items before the last cell is reached and the cycle is closed.

To solve this first problem, threads lock only one location at a time for one swap. However, when close to completion, multiple threads may compete for swap cycles, creating deadlocks. For example, assuming one item per partition, if thread t_1 reads item k_x (must go to l_x) from location l_y (must bring k_y here) and thread t_2 reads k_z (must go to l_z) from l_x (must bring k_x here), t_1 will find no space for k_x, because the offset of partition X was incremented by t_2 when it read k_z. If t_1 waits, t_2 will reach k_y. Then, a deadlock will occur, since t_1 holds (k_x, l_y) and t_2 holds (k_y, l_x).

To solve this second problem and avoid waiting for others, when a thread finds a partition to be full, it records both the current key and the locked location that the swap cycle started from. In the above example, t_1 records (k_x, l_y) and t_2 records (k_y, l_x). A final fix step occurs "offline" and takes trivial time, as the number of such pairs is upper bounded by the number of partitions P, times the number of threads.

So far, we presented a way for multiple threads to partition items in-place concurrently, but this solution is impractical if used as is. First, we make no use of buffering to improve out-of-cache performance and second, for each key we move to its destination, we update a shared variable triggering cache invalidations on every tuple move. To make this approach practical, we change the unit of transfer from tuples to blocks. Each block must have a fixed size and all tuples must belong to the same partition. We generate such blocks using the technique described previously (see Section 3.2.3). Out-of-cache accesses are amortized by the block size, as is the synchronization cost of accessing shared variables.

Algorithm 5 Synchronized in-place partitioning

```
P_active ← {}   // P_active: set of yet unfinished partitions
T_deadlock ← {}   // T_deadlock: set of tuple & location pairs
i ← 0                                  // P: the number of partitions
for p ← 0 to P-1 do
  P_active ← P_active + {p}
  offset[p] ← i
  i ← i + histogram[p]
end for
while |P_active| > 0 do
  p ← any ∈ P_active
  i ← used[p]++                                  // atomic fetch-and-add
  if i ≥ histogram[p] then
    P_active ← P_active− {p}
    goto loop-end
  end if
  i_beg ← i + offset[p]
  t ← T[i_beg]                          // T: the input & output table
  p_next ← f(t.key)                       // f: the partition function
  while p ≠ p_next do
    i ← used[p]++                                // atomic fetch-and-add
    if i ≥ histogram[p_next] then
      T_deadlock ← T_deadlock + {t, i_init}
      goto loop-end
    end if
    i ← i + offset[p_next]
    T[i] ↔ t                                                    // swap
    p_next ← f(t.key)
  end while
  T[i_beg] ← t
  loop-end:
end while
[...] // handle tuples that could cause deadlock (T_deadlock)
```

3.3 Across NUMA

Moving RAM-resident data across multiple CPUs raises questions about the effectiveness of NUMA RAM transfers. Accessing remote memory locations goes through an interconnection channel that issues operations to remote RAM modules, increasing the latency. Normally, random accesses are much slower than sequential access and the gap increases when the accesses reference remote RAM regions and go through the CPU interconnection. Prior work [1] proposed doing sequential accesses to remote memory, since hardware pre-fetching hides the latency. To avoid imbalanced use of the NUMA layer when all transfers are directed to a subset of CPUs, we can pre-schedule the transfers and supervise them via synchronization to ensure load balancing [10].

One way to make NUMA-oblivious code scale on multiple CPUs is to allocate both arrays to be physically interleaved across all RAM regions. The OS can support interleaved allocation, where the physical locations of a single array are interleaved across all NUMA regions. Randomization of page placement balances accesses across the NUMA interconnection, but precludes NUMA locality. Thus, if we do random accesses, we pay the extra NUMA latency. Cache-line buffering, used by out-of-cache partitioning to avoid TLB misses and facilitate write-combining, also mitigates the NUMA overhead. Still, we measured out-of-cache partitioning to be up to 55% slower on four NUMA regions on interleaved space. The overhead for single tuple random access is higher.

A more NUMA-friendly allocation is to split space into large segments bound to a specific region. We can have one segment per thread or one segment per NUMA region. We use the second approach for sorting (see Section 4.1).

3.3.1 Non-in-place

Using NUMA-bound segmented allocation for threads or CPUs and if extra space is allowed, we can ensure that all tuples will cross the NUMA boundaries at most once. We use shared-nothing partitioning locally and then use a separate step to shuffle across CPUs. We can use the NUMA interconnection in a balanced way without manual schedules [10]. We distribute each segment across all threads of the destination CPU, and do the transfers in a per thread random order. Since some tuples are already on destination, the expected number of transfers is $(x-1)/x$ for x regions.

The NUMA-oblivious partitioning might perform faster than the two step method of shared-nothing partitioning followed by NUMA shuffling, since out-of-cache partitioning mitigates latencies. The decision to guarantee minimal transfers by incurring shuffling, depends on the hardware.

3.3.2 In-place

Assuming NUMA-bound segmented allocation, the only in-place variant where threads do not work in a shared-nothing fashion is during block shuffling (see Section 3.2.4). During the phase of block shuffling on multiple NUMA regions, threads can read and write blocks from all regions, but all accesses are sequential, since the block is large enough to amortize the random access penalty. In the worst case, the operating thread CPU, the source, and the destination of the swapped block, will be on three different regions. Thus, the tuples can cross the interconnection at most twice. Since collocations occur, the expected number of transfers is $(2x^2-3x+1)/x^2$ for x regions, given $x > 2$. On 4 regions, we do 1.3125 transfers, 75% more than when not in-place.

3.4 Radix/Hash Histogram

Hash and radix partitioning are similar because the time required to compute the partition function is trivial. Complicated hash functions are used to decrease the collisions in hash tables. In our context, we need partitions of almost equal size that separate the keys randomly. Thus, employing hash functions designed to minimize hash table collisions, would waste CPU cycles without offering any advantage.

The radix function is a shift operation followed by a logical `and`. To isolate bit range $[x,y)$, we shift by x, then mask with $2^{y-x}-1$. Using multiplicative hashing and $P=2^k$ partitions, we multiply with an odd factor and then shift by $B-\log P$ for B-bit data. In both cases, computing the partition function only marginally affects performance.

3.5 Range Histogram

For hash and radix partitioning, computing the destination partition comes almost at zero cost, whereas range partitioning is more expensive [17]. To split into P ranges, we need P-1 delimiters that define the lower and upper bounds of each partition. We sort the P-1 delimiters and do a binary search for each input key. The difference from textbook binary search is that we search ranges rather than keys. Thus, we omit equality comparisons and do not exit the loop in less than $\log P$ steps even if an exact match is found earlier.

Scalar binary search is one example where being cache-resident is not fast enough to give good performance. The delimiter array of P-1 items always remains L1-cache-resident. However, the delays from waiting to load a new delimiter to proceed to the next comparison and the logarithmic number of loads that have to be performed, slow down range function computation almost by an order of magnitude compared to hash and radix. Replacing branches with conditional moves performs even worse, proving that the problem here is not the branches, but the logarithmic number of cache accesses that are tightly coupled due to data (or conditional control flow) dependencies and thus incur their full latency cost.

3.5.1 *In SIMD registers*

We first show range function computation can be accelerated using SIMD instructions, since every SIMD register can hold a number of values and do all comparisons at the same time. In the simplest case, we try to do small fanout range partitioning using register-resident delimiters.

The first approach holds different delimiters in each SIMD register and a single input key broadcast to all cells of another SIMD register. We can do a SIMD comparison, convert the result to a bit-mask and search the least-significant-set-bit to see which delimiter is actually larger than the key.

To extend the fanout, we can store $W \times R$ delimiters in $R \cdot W$-wide SIMD registers. We perform R SIMD comparisons and R-1 SIMD packs, then we extract and append the masks. The number of range partitions we can cover depends on the number of available registers. In practice, we do not have that many registers before stack (thus cache) spilling starts happening and, unlike binary search, this *horizontal* approach is also linear in the number of delimiters.

We now briefly describe the Intel SSE SIMD intrinsics used in the paper. `_mm_shuffle_epi32` forms the 4 32-bit lanes of the result from the 4 lanes of the input register according to a constant mask. `_mm_cmpgt_epi32(x,y)` (or `_mm_cmpeq_epi32`) compares 32-bit integers and outputs a -1/0 mask per lane if $x_i > y_i$ (or $x_i = y_i$). `_mm_packs_epi32` (or `_mm_packs_epi16`) packs 8X 32-bit (or 16X 16-bit) integers from 2 registers into 8X 16-bit (or 16X 8-bit) integers. The `_mm_movemask_epi8` (or `_mm_movemask_ps`) generates a bit-mask from the high bits of 16 bytes (or 4 32-bit words). In `_mm_blendv_epi8(m,x,y)`, each byte of the result is equal to y_i, if m_i<0, or x_i otherwise. `_mm_add_epi32`, `_mm_sub_epi32`, `_mm_min_epu32`, and `_mm_max_epu32` perform 4 32-bit add, subtract, min, and max. `_mm_load_si128` and `_mm_store_si128` perform 16-byte reads and writes.[1]

We show an example of SIMD code that does 17-way range partitioning for 32-bit integer keys. The 16 32-bit range delimiters are sorted and stored in four 128-bit SIMD registers: `del_ABCD`, `del_EFGH`, `del_IJKL`, and `del_MNOP`.

```
// broadcast 1 32-bit key to all SIMD lanes
key = _mm_loadl_epi32(input_keys++);   // asm: movd
key = _mm_shuffle_epi32(key, 0);
// compare with 16 delimiters stored in 4 registers
cmp_ABCD = _mm_cmpgt_epi32(key, del_ABCD);
cmp_EFGH = _mm_cmpgt_epi32(key, del_EFGH);
cmp_IJKL = _mm_cmpgt_epi32(key, del_IJKL);
cmp_MNOP = _mm_cmpgt_epi32(key, del_MNOP);
// pack results to 16-bytes in a single SIMD register
cmp_A_to_H = _mm_packs_epi32(cmp_ABCD, cmp_EFGH);
cmp_I_to_P = _mm_packs_epi32(cmp_IJKL, cmp_MNOP);
cmp_A_to_P = _mm_packs_epi16(cmp_A_to_H, cmp_I_to_P);
// extract the mask the least significant bit
mask = _mm_movemask_epi8(cmp_A_to_P);
res = _bit_scan_forward(mask | 0x10000);   // asm: bsf
```

The transposed approach is to broadcast each delimiter to a different SIMD register and compare W keys from the input at the same time. We use the results from earlier comparisons to blend delimiters into new custom delimiters. For instance, suppose we want to do 4-way range partitioning using 3 delimiters $A<B<C$. We first compare the keys with (the SIMD register with broadcast) B, then blend A with C to create a new delimiter D. Each lane will either have A or C based on the comparison of the input key with B. Then, we compare the keys with the custom D and combine the results. This *vertical* approach can be seen as a binary tree structure of depth D with 2^D-1 delimiters. On each comparison, the first half nodes of each level are blended with the other half, creating a new tree of depth D-1. The result is generated by bit-interleaving the D comparisons into ranges $\in [0, 2^D)$. We now show an 8-way range function for 4 32-bit keys using 7 delimiters `del_1`, ..., `del_7` ($D=3$).

```
// load 4x 32-bit keys from the input
keys = _mm_load_si128(input_keys);  input_keys += 4;
// perform 4x 3-level binary tree comparisons
cmp_L1 = _mm_cmpgt_epi32(keys, del_4);
del_15 = _mm_blendv_epi8(keys, del_1, del_5);
del_26 = _mm_blendv_epi8(keys, del_2, del_6);
del_37 = _mm_blendv_epi8(keys, del_3, del_7);
cmp_L2 = _mm_cmpgt_epi32(keys, delim_26);
del_1357 = _mm_blendv_epi8(keys, del_15, del_37);
cmp_L3 = _mm_cmpgt_epi32(keys, del_1357);
// bit-interleave 4x the 3 binary comparison results
res = _mm_sub_epi32(_mm_setzero_si128(),     cmp_L1);
res = _mm_sub_epi32(_mm_add_epi32(res, res), cmp_L2);
res = _mm_sub_epi32(_mm_add_epi32(res, res), cmp_L3);
```

The vertical version does fewer partitions than $R \times W$ of the horizontal, but is faster in small fanout. We use both in the following section. We also use the vertical version to combine range with radix partitions (see Section 4.2.1).

[1]For a complete guide to Intel SIMD intrinsics, refer here: http://software.intel.com/sites/landingpage/IntrinsicsGuide/

3.5.2 In-cache

Register-resident range partitioning is very fast, but the number of partitions we generate is not enough to saturate the partitioning fanout. Fast tree-index search has been studied using SIMD [8]. The speed-ups are very good because scalar index search performs very poorly. To search each tree level, [8] performs a horizontal search with one register. The fanout increases from 2 to W making the tree shorter, while node accesses remain almost equally fast.

We extend and optimize the idea of larger fanout to optimize range partitioning function computation. First, we remove the pointers entirely. A cache-resident index tree does not need pointers; each level of the tree can be a single sorted array. For example, a tree with 24 delimiters can be represented in 2 levels as follows: the first level holds delimiters 5,10,15, and 20; the second level of the tree holds delimiters (1,2,3,4), (6,7,8,9), (11,12,13,14), (16,17,18,19), and (21,22,23,24). Nodes are accessed using one SIMD register if $W \geq 4$. The node distinction implied by the parentheses is not explicit; accessing the arrays at different offsets works as accessing a different node. No delimiters are repeated across tree levels. The index does not (need to) support updates.

The array pointers representing the tree levels are kept in scalar registers and all tree node accesses are "hard-coded" with ad-hoc code per tree level. Thus, we can design trees with different fanout on each level based on the total number of range partitions. We generate a menu of tree fanout configurations in order to be able to choose the best partitioning fanout, similar to picking the number of bits for radix or hash partitioning. The sensible configurations are the ones where we load and horizontally search k SIMD registers, giving tree node fanout of the form $k \cdot W + 1$. We can hold the zero level (root) of the tree in registers during the operation using horizontal register search, but we found that using vertical SIMD search to access the root is faster.

For 32-bit keys and 128-bit SIMD registers, we use 5-way or 9-way fanout on tree levels. Having a 17-way fanout is outperformed by two consecutive 5-way levels that also give a total fanout of 25. The best configurations that work well with out-of-cache partitioning fanout, are a 360-way range function using a 3-level $(8 \times 5 \times 9)$-way tree search, a 1000-way function using a 4-level $(8 \times 5 \times 5 \times 5)$-way tree, and a 1800-way function using an $(8 \times 5 \times 5 \times 9)$-way tree. We show the 360-way tree access code omitting the root.

```
// access level 1 (non-root) of the index (5-way)
lvl_1 = _mm_load_si128(&index_L1[r_0 << 2]);
cmp_1 = _mm_cmpgt_epi32(lvl_1, key);
msk_1 = _mm_movemask_ps(cmp_1);   // ps: epi32
r_1 = _bit_scan_forward(r_1 ^ 0x1FF);
r_1 += (r_0 << 2) + r_0;
// access level 2 of the index (9-way)
lvl_2_A = _mm_load_si128(&index_L2[ r_1 << 3]);
lvl_2_B = _mm_load_si128(&index_L2[(r_1 << 3) + 4]);
cmp_2_A = _mm_cmpgt_epi32(lvl_2_A, key);
cmp_2_B = _mm_cmpgt_epi32(lvl_2_B, key);
cmp_2 = _mm_packs_epi32(cmp_2_A, cmp_2_B);
cmp_2 = _mm_packs_epi16(cmp_2, _mm_setzero_si128());
msk_2 = _mm_movemask_epi8(cmp_2);
r_2 = _bit_scan_forward(r_2 ^ 0x1FFFF);
r_2 += (r_1 << 3) + r_1;
```

To maximize parallel cache accesses and boost performance, we loop-unroll all tree level accesses to 4 keys at-a-time and the root vertical search to 16 keys at-a-time. The optimal loop-unrolling degree was evaluated experimentally.

4. SORTING

4.1 Setting

Like most prior work, we operate on fixed-length keys and payloads. In read-only workloads typical of data analytics, more complex types can be dictionary coded into compact integer types that preserve the key order [12, 16]. Keys and payloads are assumed to reside in separate arrays, as would be typical of a column-store analytical database. When the algorithm starts, we assume that our input is evenly divided among the C NUMA regions in contiguous segments. All columns of the same tuple reside in the same NUMA region.

The requirement for C separate segments is because the OS either interleaves allocated memory across regions, or allocates space from one region exclusively. To maximize performance, we use C segments, one per NUMA region. The output is also a collection of C segments of sorted data, each residing in a distinct NUMA region. All keys in region i have a smaller key than all keys in region j when $i < j$. The sorted output is thus a concatenation of the segments.

4.2 Radix-Sort

Radix-sort is a very efficient sorting method for analytical databases that use order-preserving compression to the data [12, 16]. The number of passes done is dependent on the number of bits required to encode the distinct values in the array. If we store n distinct values using $\log n$ bits, we still need $n \log n$ operations, same as traditional comparison sorts. The speed-up of radix-sort over comparison sorting is not primarily due to decreased complexity, but due to the (as of now) faster radix compared to range partitioning.

4.2.1 Stable LSB

The simplest algorithm we describe in this paper is the least-significant-bit (LSB) radix-sort. The algorithm is well known and performs a fixed number of passes over the data, depending on their width. If all bit ranges are processed, the array is sorted and needs no further processing. Building on the partitioning techniques described in Section 3, we can readily present how a NUMA-aware LSB radix-sort works.

In order for LSB radix-sort to work, we must ensure that during partitioning, we never swap items with equal keys. The order of the items is defined by the bit range of the radix function. All passes (except for the first) must be *stable* (i.e. preserve the ordering of equal items). In-place partitioning is not stable, thus can only be used for the first pass.

If we overlook NUMA transfers entirely, each tuple will cross NUMA regions several times. This design is clearly suboptimal. We ensure that each tuple is transferred through the NUMA interconnection network exactly once. In the first partitioning step, we do C-way range partitioning. The C-1 delimiters are acquired by uniform sampling. Since C is small and the $\log C$ bits cannot saturate the partitioning fanout, we fill up with low order radix bits. In steps:

1. Sample and determine C-1 range delimiters that equally split the data across the C NUMA regions.
2. Range-radix partition locally on each NUMA using a range (C-way) & radix function (Section 3.2.1) and use enough bits to optimize the partitioning fanout.
3. Shuffle across C NUMA nodes (Section 3.3.1).
4. Until all bits of the key domain are covered continue radix-partitioning locally per NUMA (Section 3.2.1) using enough bits to optimize the partitioning fanout.

All data movement during any partitioning pass are based on parallel out-of-cache partitioning (see Section 3.2.1) between the threads of the *same* NUMA region only. Data transfers across NUMA nodes happen only during the shuffling of the C ranges (see Section 3.3.1). Threads operating on the same NUMA nodes always split the workload in equal pieces, regardless of partition sizes and skew. The C range delimiters ensure equalized segment sizes across the C NUMA regions. Finally, to compute the mixed range-radix partitioning function of the first phase, we use register-resident delimiters (see Section 3.5.1) and concatenate the range function result with the low order radix bits.

The number of passes is the same as before, only adding a minor $\log C$ range partition bits. This design can also support balancing of workload across non-uniform CPUs, by dividing the data to C non-equal range partitions, if we can estimate the relative performance between CPUs.

When the key domain is dense, LSB radix-sort does the minimum number of passes. When the key domain is large, however, and the value domain is significantly larger than the number of items, the LSB approach does more passes than required over the data, eliminating its advantage over other approaches. The total number of passes is determined by $\log D$, where D is the size of the key domain.

4.2.2 In-place MSB

The second variant we present is the most-significant-bit (MSB) radix-sort. In this version, the bits are processed high-to-low, rather than low-to-high as in LSB. The partitioning passes are dividing the data to non-overlapping domains, creating a radix-based analog to quick-sort. The outputs of each pass do not need to be interleaved. Since each pass does not need to maintain the ordering of previous passes, stable partitioning is not useful for MSB radix-sort.

The MSB radix-sort in this paper is done entirely in-place. Alongside in-place partitioning, our work first describes efficient large-scale in-place sorting. LSB is faster than MSB in compressed data, but the small performance gap justifies paying a small price in time and in exchange save space and use it in some other concurrent operator more effectively.

In the first pass, we use shared segment in-place partitioning across all NUMA regions (see Section 3.3.2). If the initial bits do not divide the data in a balanced fashion, the later phases will either operate on unbalanced sub-array sizes, or transfer data across NUMA regions more than once to rebalance work. Mixing radix with range partitioning during the first pass is more robust. Balancing is unaffected even if the upper key bits are almost always the same. This case is not uncommon for databases that use fixed data types, or schemas with larger types than necessary. In steps:

1. Sample and determine T-1 range delimiters that equally split the data across T threads. Add T' more range delimiters ($T' \geq T$) generated from $\log T'$ high bits.
2. Range partition $T + T'$-way in-blocks (Section 3.2.3).
3. Shuffle across NUMA regions using synchronized in-place partitioning of blocks (Sections 3.2.4 and 3.3.2).
4. Sort each part recursively in a shared-nothing fashion:
   (a) If (sub-)array is larger than the cache, employ out-of-cache shared-nothing radix partitioning (Section 3.2.2) and make a recursive call for each part.
   (b) Else, radix partition in-cache (Section 3.1). If not all radix bits are covered, either make recursive call or use insert-sort for trivial part sizes.

As in LSB, we split our range-split data on T ranges, where T is the number of total threads (not the number of NUMA regions) and acquire the T-1 delimiters using sampling. After this phase, threads can proceed in a shared-nothing fashion. We add $\log T$ bits to the total key length processed, in the same way LSB added $\log C$ bits. We hide the extra bits by overlapping with $2^{\lceil \log T' \rceil}$ radix partitions generated from the high $\lceil \log T' \rceil$ radix bits $T' \geq T$. This process adds at most one bit to the total key domain size.

After the T-way range split across T threads and NUMA shuffling, we work shared-nothing. Depending on the sub-array size, we use both out-of-cache (see Section 3.2.2) and in-cache (see Section 3.1) variants. When a sub-array of size n reaches the cache, using $\approx \log n$ radix bits generates parts of trivial size [7]. We use $\lceil \log n \rceil - 2$ bits and generate partitions of average size 4–8 that are insert-sorted ignoring the remaining bits. Thus, unlike LSB that covers $\log D$ bits (D: domain size), MSB can cover $\log n$ bits (n: array size).

4.3 Comparison-Sort

Traditional algorithms for sorting are based on item comparisons, providing perfect balancing and skew immunity. The standard variants are quick-sort [1] and merge-sort [5, 14]. The latter has been extensively optimized using SIMD.

4.3.1 In-cache

Prior work [2, 5, 6, 14] has suggested SIMD-aware sorting. The proposed method [2, 5, 14] starts from sorting in SIMD registers, then combines multiple registers using sorting networks or bitonic sort, then switches to merging. Outside the database community, an approach that scales to the SIMD length has been proposed [6], initially using W-way comb-sort in-cache without comparing keys across lanes, then transpose and merge all parts out-of-cache.

Optimally, we need $(n/W) \log n$ instructions to do $n \log n$ comparisons on W-way SIMD. In practice, only sorting networks and bitonic sort optimally scale to the length of SIMD giving $O((n/W) \log^2 n)$. If we view the array as n/W vectors rather than n keys and never compare keys across lanes, we need $O((n/W) \log (n/W))$ SIMD comparisons. Afterwards, we merge the W arrays with $n \log W$ comparisons. In total, we need $O((n/W) \log (n/W) + n \log W)$ comparisons, which is very close to $O((n/W) \log n)$ and better than bitonic sort. However, for most average $O(n \log n)$ algorithms, the location of the next key read is dependent on the previous key comparisons. The hardware must gather-load W keys as fast it loads one, which is impractical. Thus, we use comb-sort that has no such dependency. The key exchanges need a pair of `min`/`max` instructions, as shown below (Intel SSE):

```
j = i + comb_sort_gap;
x = _mm_load_si128(&input_keys[i]);
y = _mm_load_si128(&input_keys[j]);
min = _mm_min_epu32(x, y);
max = _mm_max_epu32(x, y);
_mm_store_si128(&input_keys[i], min);
_mm_store_si128(&input_keys[j], max);
```

After W-wide comb-sort finishes, we have to merge the W lanes. For this task, we keep the "last" key from the W arrays in a SIMD register. We do the same for the merge pointer and payloads. On each iteration, we find the lane of the minimum key, extract the payloads and the index, write the minimum tuple to the output, read one new tuple from the min key index + W, and replace the min key lane.

We show one loop of 4-way merging for 32-bit keys on 128-bit SIMD below, omitting payloads for simplicity. The keys are read in the transposed form that is the output from the W-way comb-sort. All phases are shown in Figure 2.

```
// find the min key and output it
min_key = min_across(keys);
k = _mm_cvtsi128_si32(min_key);
_mm_stream_si32(output_keys++, k);
// find the location (index) of min key
min_loc = _mm_cmpeq_epi32(keys, min_key);
min_loc = _mm_xor_si128(min_loc, all_bits_1);
min_loc = _mm_or_si128(min_loc, locs);
min_loc = min_across(min_loc);
// load the next key and update the index
i = _mm_cvtsi128_si32(min_loc);
new_key = _mm_loadl_epi32(&input_keys[i + 4]);
new_key = _mm_shuffle_epi32(new_key, 0);
new_loc = _mm_add_epi32(min_loc, all_words_4);
// insert new key and location
mask = _mm_cmpeq_epi32(min_loc, locs);
keys = _mm_blendv_epi8(keys, new_key, mask);
locs = _mm_blendv_epi8(locs, new_loc, mask);
```

In order to compute (and broadcast) the minimum key across a SIMD cell holding W keys we need $\log W$ comparisons and $\log W$ shuffle instructions. We show the implementation of min-across on Intel SSE below for 4 32-bit keys (X, Y, Z, W, where $A = \min(X, Y)$ and $B = \min(Z, W)$):

```
__m128i min_across(__m128i XYZW)   {
  __m128i YXWZ = _mm_shuffle_epi32(XYZW, 177);
  __m128i AABB = _mm_min_epu32(XYZW, YXWZ);
  __m128i BBAA = _mm_shuffle_epi32(AABB, 78);
  return       _mm_min_epu32(AABB, BBAA);   }
```

The W-way merging phase takes significantly less time than the W-way comb-sort step and skips the need for 2-way merging [6] or bitonic sort [5]. The interleaved arrays are not transposed before merging, we avoid using extra space *during* sorting, and the output can bypass the cache. The algorithm does $O((n/W)\log(n/W)) + n \log W$ comparisons and works for any SIMD length (W) with no adjustment.

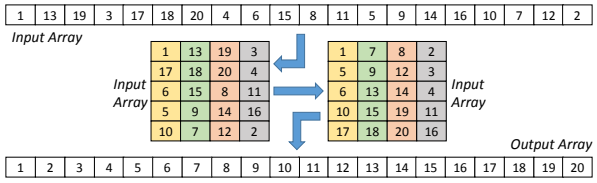

Figure 2: In-cache sorting on 4-wide SIMD

4.3.2 *Out-of-cache*

Prior work [2, 5, 14] has suggested merge-sort as the most effective comparison-based sort for databases. Since merging two streams on each pass is bounded by the RAM bandwidth, merging a larger number of arrays is a better approach. Quick-sort splits the array using 2-way range partitioning optimized to avoid histograms, but is also bounded by the RAM bandwidth when running out-of-cache.

We introduce a new variant for comparison sorting based on range partitioning that uses a large fanout on each pass. Since radix-sort can do a lot of radix partitions on each pass, we could do the same for range partitioning. Until recently, this idea hit the wall of slow out-of-cache partitioning, and, until now, the wall of slow range function computation.

The comparison-based algorithm we propose is quite similar to radix-sort, but replaces radix with range partitioning. The version we present here uses non-in-place partitioning, thus, uses linear auxiliary space, similarly to merge-sort. The algorithm decides on a number of range partitioning passes to range-split the data to cache-resident range parts.

To compute the range function, we build a specialized index in the cache (see Section 3.5.2) and, alongside computing the histogram, we store the range partition for each tuple to avoid re-computing it. While the tuples are actually moved, we (sequentially) read the range partitions from the array where they are stored during range histogram generation. The data movement part of partitioning performs almost as fast as radix partitioning, since scanning an additional short range array has a small cost. The boost in performance is due to the fast range partition function. In recursive steps:

1. If in-cache, sort using SIMD comb-sort and exit.
2. Sample the data and gather range delimiters.
3. Generate histogram using in-cache ranges index.
4. If not 1st pass, employ shared-nothing range partition.
5. Else range partition locally per NUMA (Section 3.2.1) and shuffle across the C NUMA regions (Section 3.3.1).
6. Call sorting recursively for each range part.

Using the large fanout of partitioning is essential to avoid doing many passes bounded by the RAM bandwidth. The alternative would be to use merging. *Incremental merging* proposed in [5] is CPU-bound, but we transfer tuples $\log P$ times between small intermediate buffers. More recent work that compares merging with partitioning [2], shows 16-way merging to be as fast as non-in-place out-of-cache radix partitioning for up to 11-bits. As we will see experimentally, this advantage holds also for range partitioning, using the efficient cache-resident range index that we propose.

In order to avoid skew and imbalance problems, before shuffling across NUMA regions, we create more range partitions than the total threads. Thus, the threads can fine-grain share the workload and the C segment sizes can have almost equal sizes. Furthermore, to achieve good splitting [13] and avoid repeating keys that could cause in-cache sorting to work out-of-cache, when X is sampled twice or more as a delimiter, we use only X and X-1 and switch to a smaller range index if too many delimiters are discarded. The single-key range partitions need no further processing.

5. EXPERIMENTAL EVALUATION

All our experiments are run on the same platform, which has 4 Intel Xeon E5-4620 8-core CPUs at 2.2 GHz based on the Sandy Bridge micro-architecture and a total of 512 GB quad-channel DDR3 ECC RAM at 1333 MHz. Each CPU supports 16 hardware threads through 2-way simultaneous multi-threading. Each core has 32 KB private L1 data cache and 256 KB L2 cache. Each CPU has 8 MB L3 cache shared across the 8 cores. The bandwidth we measured is 122 GB/s for reading, 60 GB/s for writing, and 37.3 GB/s for copying.

The operating system is Linux 3.2 and the compiler is GCC 4.8 with `-O3` optimization. We use SSE (128-bit SIMD) instructions on AVX registers (256-bit), in order to use non-destructive 3-operand instructions (Intel VEX) that improve pipeline performance. We cannot fully use 256-bit SIMD because our platform (AVX 1) supports 256-bit operations for floating point operations only. Unless stated otherwise, experiments use 64 threads and data are uniform random.

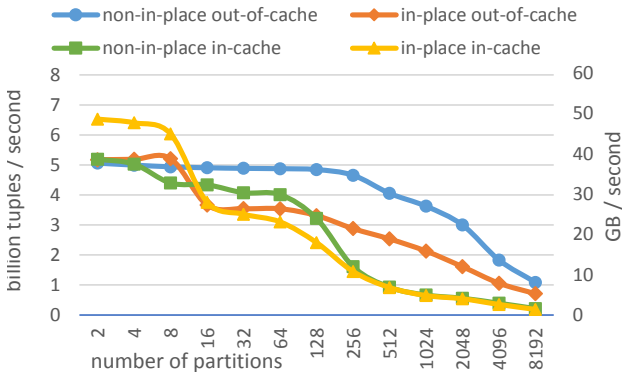

Figure 3: Shared-nothing partitioning for 10^{10} tuples (32-bit key and 32-bit payload on separate arrays)

In Figure 3 we show the performance of the four variants of partitioning using 32-bit keys and 32-bit payloads run in a shared-nothing fashion. The in-cache variants are bound by the TLB capacity, thus, have poor performance for large fanout. When running out-of-cache, the optimal cases are 5–6 bits (32–64 partitions). The out-of-cache variants increase the optimal fanout to 10–12 bits, when non-in-place, and 9–10 bits, when in-place. Using out-of-cache variants for small cache-resident array sizes incurs unnecessary overheads. The optimal fanout is the one with the highest performance per partitioning bit ($= \log P$, for P-way partitioning) ratio.

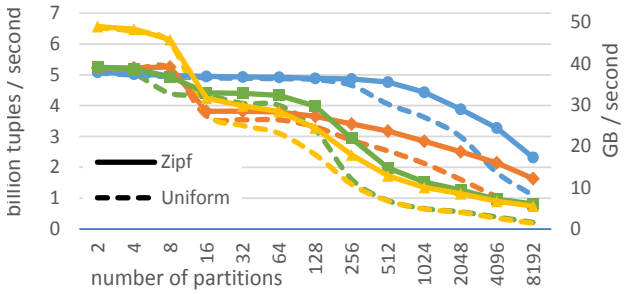

Figure 4: Partitioning as in Figure 3 using uniform and skewed data under the Zipf distribution ($\theta = 1.2$)

In Figure 4, we repeat the experiment of Figure 3 including runs with data that follow the Zipf distribution with $\theta = 1.2$. Under skew, some partitions are accessed more often than others. Implicit caching of these partitions decreases the probability of cache misses and TLB thrashes, improving performance. With less skew ($\theta < 1$), we found no significant difference in partitioning performance.

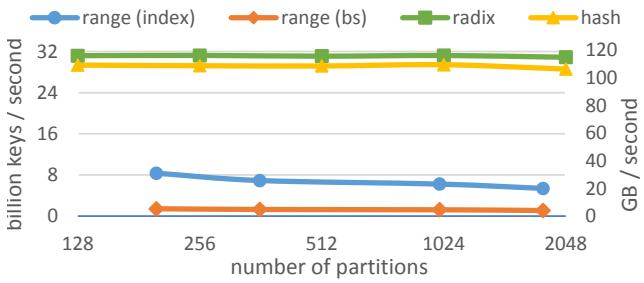

Figure 5: Histogram generation for 10^{10} 32-bit keys

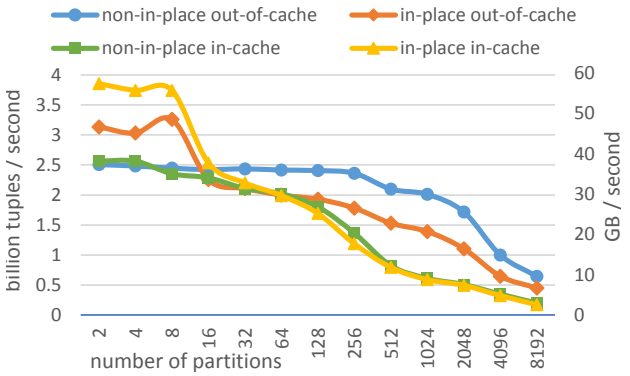

Figure 6: Shared-nothing partitioning for 10^{10} tuples (64-bit key and 64-bit payload on separate arrays)

In Figure 5 we show the performance of histogram generation for all variants of partitioning. Radix and hash partitioning operate roughly at the memory bandwidth. Range partitioning using the configured range function index (see Section 3.5.2) improves 4.95–5.8X compared to binary search.

Figure 6 shows the performance of partitioning for 64-bit keys and 64-bit payloads. Compared to the 32-bit case, partitioning is actually slightly faster (in GB/s), since RAM accesses and computation overlap more effectively.

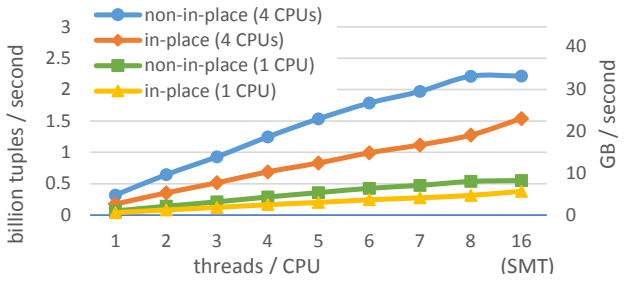

Figure 7: Shared-nothing out-of-cache partitioning 1024-way for 10^9 tuples (64-bit key, 64-bit payload)

Figure 7 shows the scalability of out-of-cache partitioning variants with a 1024-way fanout. The in-place variant gets a noticeable benefit from SMT compared to the non-in-place.

Figure 8 shows the performance of 64-bit histogram generation. Radix and hash histogram generation still run at the memory bandwidth. Using the range function index speeds-up the process 3.17–3.4X over scalar code, despite the limitation of only 2-way 64-bit operations. The speed-up decreases because scalar binary search doubles its performance (in GB/s), fully shadowing the RAM accesses.

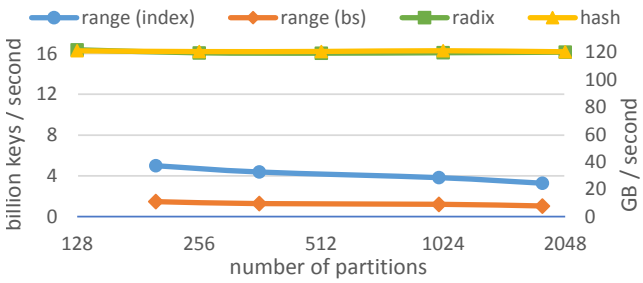

Figure 8: Histogram generation for 10^{10} 64-bit keys

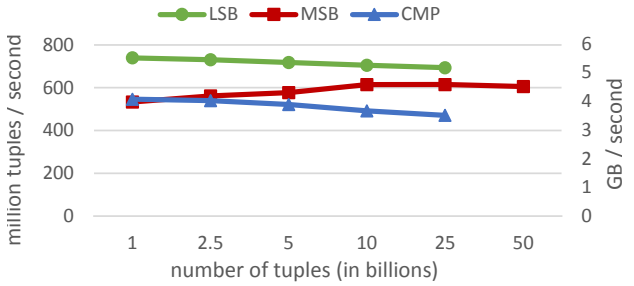

Figure 9: Sort performance (32-bit key, 32-bit rid)

In Figure 9, we show the performance of the three sorting variants that we propose, using 32-bit key, 32-bit payload tuples. LSB denotes LSB radix-sort, MSB denotes MSB radix-sort and CMP denotes comparison-sort. MSB is 10–20% slower than the fastest LSB and maximizes when the array exceeds the domain size, using only radix partitioning in the cache. CMP is slower but comparable to radix-sorts.

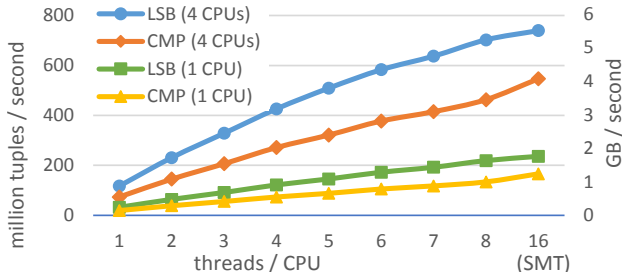

Figure 10: NUMA & non-NUMA sorts for 10^9 tuples

Figure 10 shows the scalability of both NUMA and non-NUMA versions of LSB and CMP (32-bit key, 32-bit rid). CMP benefits more from SMT than LSB. Using all hardware threads, the speed-up of 4 CPUs over 1 is 3.13X for LSB and 3.29X for CMP. The 1-CPU variants are entirely NUMA-local and omit the NUMA shuffling step. Closer to 4X speed-up is not achieved for 4 CPUs due to the required extra step.

Figure 11 shows the time distribution across phases. We double the maximum size using in-place MSB. As shown, MSB outperforms LSB when no pre-allocated memory is available. CMP needing only two range partition passes to reach the cache and spends ≈ 40% of the time to sort there.

LSB radix-sort and especially comparison-sort are adaptive to skew. For 10^{10} tuples following the Zipf distribution,

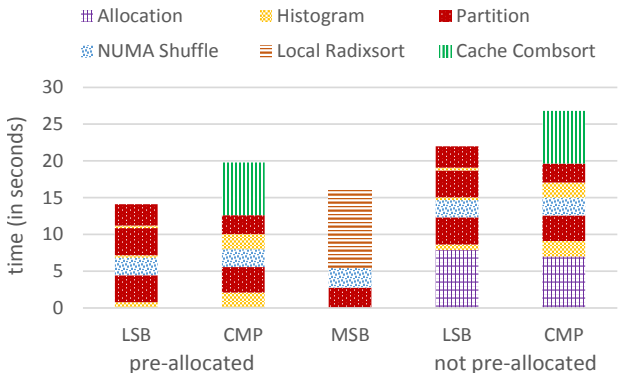

Figure 11: Sorting 10^{10} tuples (32-bit key, 32-bit rid)

Figure 12: Sort performance (64-bit key, 64-bit rid)

LSB is 5% faster for $\theta = 1$ and 15% faster for $\theta = 1.2$. CMP is 30% faster for $\theta = 1$ and 80% faster for $\theta = 1.2$ outperforming all other methods. MSB can be affected negatively, but not until high skew ($\theta \geq 1.2$). If excessive repeats from a key affect the balance of the C-way split across C NUMA regions, we identify these keys during sampling and create single key partitions [13] shared across NUMA regions.

Figure 12 shows the speed of all variants for 64-bit keys and 64-bit payloads and Figure 13 shows the time distribution. These algorithms are most useful in cases where data-to-query time is important and compression is too expensive, or when tuples change rapidly and cannot be compressed.

The MSB radix-sort is faster than LSB, since it stops partition passes earlier. When we reach the cache, we create $n/4$ to $n/8$ partitions and then sort 4–8 items using insert-sort. Our advantage over hybrid approaches, such as MSB switching to LSB [15], is that the latter will do more passes for sparse keys that are not free, even if cache-resident.

Comparison-sort, like MSB radix-sort, does fewer passes than LSB. Range partitioning performs closer to radix, since the data movement costs twice, while the range histogram is less affected. On the other hand, in-cache SIMD sorting is more expensive, since each 128-bit SIMD register can hold two 64-bit keys and cannot be much faster than scalar code.

The effect of the NUMA layer is dependent on the number of partitioning passes. Figure 14 compares NUMA-aware and NUMA-oblivious versions. A pass can be more than 50% slower on NUMA-interleaved memory. Using an extra pass for NUMA shuffling always helps. LSB radix-sort is ≈ 25% faster when NUMA-aware, even if only 3 passes are required for 32-bit keys. When using 64-bit keys, being NUMA-aware is more than 50% faster. Finally, the effect on comparison sort is smaller (10–15%), since range histograms are CPU-bound and the number of passes is minimal.

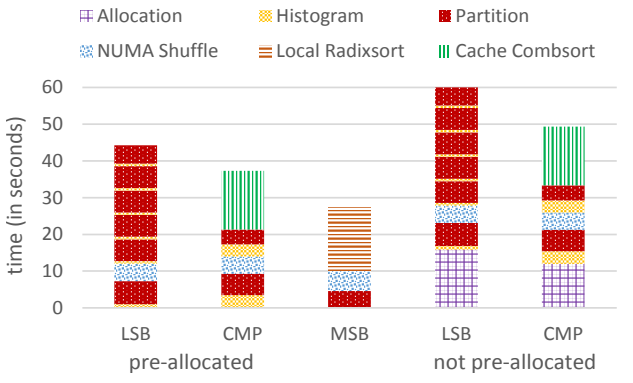

Figure 13: Sorting 10^{10} tuples (64-bit key, 64-bit rid)

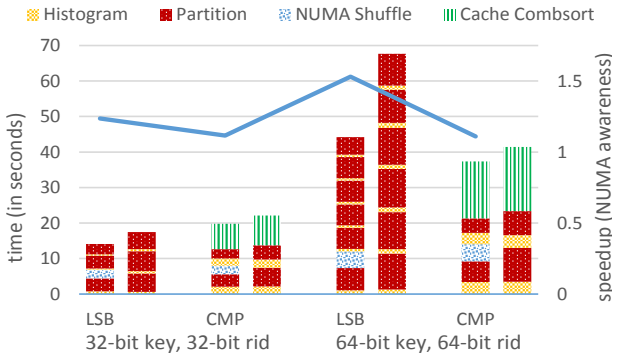

Figure 14: NUMA-aware vs. NUMA-oblivious (interleaved) comparison- and radix-sort for 10^{10} tuples

In Figure 15, we show the performance of SIMD sorting for 32-bit keys and payloads using small array sizes in-cache, as done in the last phase of CMP. The speed-up over scalar code is 2.9X on average with 4-wide SIMD (SSE) comb-sort.

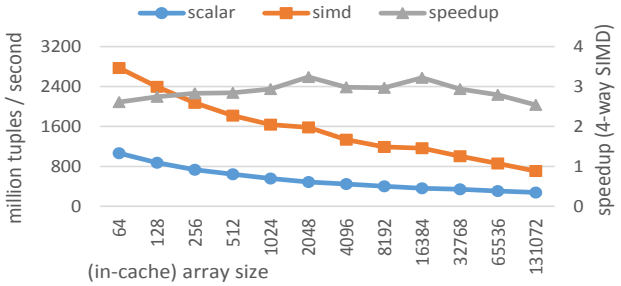

Figure 15: Shared-nothing in-cache performance of scalar and SIMD comb-sort (32-bit key, 32-bit rid)

Billion-scale sorting has been recently used for sort-merge-joins [1, 2]. Albutiu et al. [1] does one in-place MSB radix partition pass using the in-cache variant, then uses intro-sort to sort the partitions. This hybrid approach operates in-place, but is neither a comparison- nor a radix-sort. In-place radix-sort is 2–3X faster than intro-sort on 32-bit keys, and one radix partitioning step still leaves most of the work for intro-sort. Balkesen et al. [2], that improves over Albutiu et al. [1], uses similar hardware to ours (4 Intel Sandy Bridge CPUs, 32 cores with SMT, NUMA memory). For 1 billion tuples, Balkesen et al. [2] sorts 350 million tuples per second using merging, compared with our comparison-sort that achieves 540 million. Again for 1 billion tuples, Balkesen et al. [2] sorts 675 million tuples per second using non-in-place MSB radix partitioning out-of-cache, compared with our non-in-place LSB radix-sort that achieves 740 million. Thus, we outperform prior work on both radix- and comparison-based sorting. Furthermore, prior work [2] implicitly assumes that the key value range covers the entire key domain, as MSB radix partitioning is not combined with range partitioning to guarantee load balancing across threads. Neither work [1, 2] considers large key domains.

Merge-sort performance in Chhugani et al. [5] drops by 12.4% when sorting 2^{28} keys (1 GB) compared to 2^{27} and in Balkesen et al. [2] by 25% when sorting 2^{30} tuples (8 GB) compared to 2^{29}. Our comparison-sort performance drops by 13% when sorting 25 billion tuples (186.26 GB) compared to 1 billion. Thus, our optimized range partitioning is more scalable than merging on large-scale comparison sorting.

6. CONCLUSION

We studied a wide menu of partitioning variants across all layers of the main-memory hierarchy, introduced large-scale in-place partitioning, and provided guarantees for minimal transfers across NUMA boundaries on multiple processors. We made range partitioning comparably fast with hash or radix by designing a SIMD-based cache-resident range index.

By combining all partitioning variants, we designed three sorting algorithms, the fastest to date: a stable LSB radix-sort, an in-place MSB radix-sort, and a comparison-sort that uses range partitioning. Our evaluation on billion-scale arrays suggests using LSB radix-sort on dense key domains, MSB radix-sort on sparse key domains or to save space, and comparison-sort for load balancing and skew efficiency.

Our work can serve as a tool for designing other operations by combining the most suitable partitioning variants. In-place versions offer a trade-off between space and time. Range partitioning offers optimally balanced parts regardless of skew or domain. Minimal NUMA transfers guarantee efficiency and scalability on future more parallel hardware.

7. REFERENCES

[1] 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, June 2012.

[2] C. Balkesen et al. Multicore, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, Sept. 2013.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

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

[5] J. Chhugani et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[6] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[7] C. Kim et al. Sort vs. hash revisited: fast join implementation on modern multicore CPUs. In *VLDB*, pages 1378–1389, 2009.

[8] C. Kim et al. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[9] C. Kim et al. CloudRAMsort: fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In *SIGMOD*, pages 841–850, 2012.

[10] Y. Li et al. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[11] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[12] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, Aug. 2013.

[13] K. A. Ross et al. Optimal splitters for database partitioning with size bounds. In *ICDT*, pages 98–110, 2009.

[14] N. Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[15] J. Wassenberg and P. Sanders. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.

[16] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.

[17] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, pages 249–260, 2013.