On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning

Felix Martin Schuhknecht Pankaj Khanchandani Jens Dittrich

Information Systems Group
infosys.cs.uni-saarland.de

1. PROBLEM STATEMENT

Partitioning a dataset into ranges is a task that is common in various applications such as sorting [1,6,7,8,9] and hashing [3] which are in turn building blocks for almost any type of query processing. Especially radix-based partitioning is very popular due to its simplicity and high performance over comparison-based versions [6].

```
1  for i = 0 to num_elems - 1 do
2  |   ++histogram[input[i] >> (32 - R)];
3  offset = 0;
4  for i = 0 to num_partitions - 1 do
5  |   dest[i] = offset;
6  |   offset += histogram[i];
7  for i = 0 to num_elems - 1 do
8  |   bucket_num = input[i] >> (32 - R);
9  |   output[dest[bucket_num]] = input[i];
10 |   ++dest[bucket_num];
```

Figure 1: Original version

In its most primitive form, coined *original version* from here on, it partitions a dataset into 2^R (where $R \leq 32$) partitions as shown in Figure 1: in the first pass over the data, we count for each partition the number of entries that will be sent to it (lines 1-2). From this generated histogram, we calculate the start index of each partition (lines 3-6). The second pass over the data finally copies the entries to their designated partitions (lines 7-10).

Despite of its simple nature, several interesting techniques can be applied to enhance this algorithm such as *software-managed buffers* [3, 6, 7, 9], *non-temporal streaming operations* [3, 5, 6, 9], *prefetching*, and *memory layout* [3, 6] with many variables having an influence on the performance like *buffer sizes*, *number of partitions*, and *page sizes*. Although being heavily used in the database literature, it is unclear how these techniques individually contribute to the performance of partitioning. Therefore, in this work we will incrementally extend the original version by the mentioned optimizations to carefully analyze the individual impact on the partitioning process. As a result this paper provides a strong guideline on when to use which optimization for partitioning.

2. EXPERIMENTAL SETUP

The dataset used in this evaluation consists of $N = 100$ million entries, where each entry is composed of a 4B key (unsigned int) and a 4B payload[1]. The keys follow a uniform and random distribution and cover the full unsigned 4B space. We repeat each experiment five times and report the average runtime. All algorithms evaluated are purely single-threaded. We run all the experiments on a two-socket machine consisting of two quad-core Intel Xeon E5-2407 running at 2.2 GHz. The CPU neither supports hyper-threading nor turbo mode. The sizes of the L1, L2, and L3 caches are 32KB, 256KB, and 10MB respectively. The TLB can cache 64 (L1 dTLB) and 512 (L2 TLB) address translations for 4KB pages and 32 (L1 dTLB) translations for 2MB pages. The system is equipped with 48GB of main memory in total and runs a 64-bit openSuse 12. All programs are written in C++ and compiled using `g++4.7.1` with optimization level O3.

[1]We choose these sizes as typical numbers for main-memory indexes. In Section 3.6 we vary the number of entries and entry size.

Proceedings of the VLDB Endowment, Vol. 8, No. 9

3. EXPERIMENTAL EVALUATION

In the following evaluation, we incrementally extend and modify the *original version* of radix partitioning by applying both known techniques from the literature as well as new approaches to analyze the impact of the optimizations on the total runtime of partitioning. Figure 2 shows the paths we follow when incrementally optimizing the original version alongside with their appearances in literature.

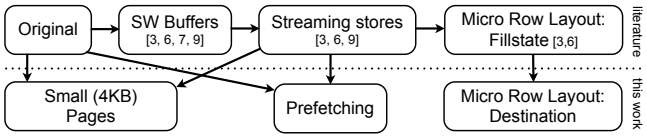


*Figure 2: **Overview** of the applied optimizations.*

3.1 Software-Managed Buffers

A well known optimization for accessing multiple data streams are software-managed buffers, that are used in a variety of works [3, 6, 7, 9]. As writing to p different streams means accessing p pages (if stream offset is larger than the page size), we have to cache p address translations in the TLB to avoid page walks. Unfortunately, as the TLB capacity is scarce, for a larger value of p, TLB-misses occur frequently [4]. Software-managed buffers try to reduce this problem by keeping a buffer of b entry slots for each partition, such that these buffers are filled first. Only if a buffer is full, its b entries are flushed to the final partition. Thus, the output array is now accessed at buffer granularity and no longer at entry granularity, reducing the amount of required address translations by a factor of b. In recent work [3, 6] a partition buffer is set to the size of a single cache-line to minimize cache-misses when filling the buffers. However, by choosing larger partition buffers, we might decrease the number of TLB-lookups even more (at the risk of more data cache misses as buffers are more likely to get evicted). Thus, we will test a variety of partition buffer sizes to analyze this tradeoff.

Let us start by looking at the partitioning time using software-managed buffers while varying the size of a partition buffer from one cache-line (64B, capacity for 8 entries) to 1248 cache-lines

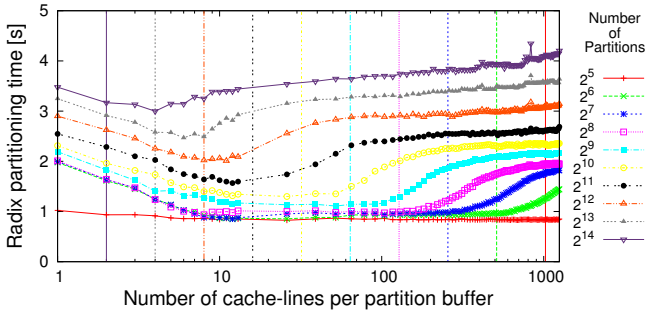


Figure 3: ***Partitioning time*** *of the* ***software-managed buffers*** *version for varying partition buffer sizes.*

(78KB, capacity for 9984 entries) and the number of partitions[2] from 2^5 to 2^{14} in Figure 3. As expected, the runtime increases with the requested number of partitions. However, we can also observe that for all tested numbers of partitions, the runtime improves with the buffer size until a certain optimal point is reached and then slowly degrades again. The more partitions we request, the smaller is this optimal partition buffer size. The point at which the performance degrades also heavily depends on the number of partitions (2048 elements per buffer for 128 partitions and around 512 elements per buffer for 1024 partitions). Thus, the performance degradation is related to the total space that the buffers occupy with a degradation point around 2MB (visualized by the vertical lines).

To get further insights into the behavior of the method, we measured the total amount of cache-misses for the software-managed buffers version using *perf*. Compared with the runtimes of Figure 3, we observe a clear correlation. The increase in runtime for buffers larger than 2MB is related to the increase in cache-misses. Furthermore, the fastest runtime is measured around the partition buffer size that triggers the smallest amount of misses. The question is now how much we gained over the original version by using software-managed buffers. Table 1 shows a direct comparison.

# Partitions	Original Version [s]	SW Buffers [s]	([cl])	Speedup
32	0.79	0.83	(26)	0.95x
64	1.91	0.85	(12)	2.25x
128	2.03	0.85	(12)	2.39x
256	2.12	0.94	(8)	2.25x
512	2.27	1.11	(52)	2.05x
1024	2.41	1.30	(26)	1.85x
2048	2.57	1.57	(12)	1.64x
4096	2.91	2.01	(11)	1.45x
8192	3.32	2.47	(6)	1.34x
16384	3.61	3.00	(4)	1.20x

Table 1: ***Partitioning time*** *of the* ***original version*** *and the* ***software-managed buffers version*** *(as shown in Figure 3). For software-managed buffers, we show the runtime and the buffer size per partition in number of cache-lines (cl) in brackets.*

In the case of software-managed buffers, we individually picked the best suitable partition buffer size (shown in the brackets) for the comparison. Obviously, the buffered version significantly improves the partitioning time for all numbers of partitions from 64 on, ranging from a speedup of 2.25x for 64 partitions to 1.20x for 16,384 partitions. The improvement decreases with the number of partitions, as the buffers run out of the private caches. Only for 32 partitions, the original version performs significantly better, as it does not yet suffer from TLB- and cache-misses.

[2]We focus on typical numbers of partitions that are used in practice like 256 partitions (byte-wise radix sort) and 1024 partitions (initial range-partitioning step in database cracking algorithms [8]).

3.2 Non-temporal Streaming Stores

So far, we have flushed the buffers in the traditional way using `memcpy`. Internally, this triggers the fetching of the corresponding cache-lines from main memory to write the new data to them. While this is a valid strategy in general, where modified cache-lines are likely to be used subsequently, in the case of partitioning this behavior is wasteful. Instead, we want to directly write the data to main memory and bypass the caches entirely. This can be achieved by using *non-temporal streaming stores*, that are used widely in write-intensive situations [5] and partitioning tasks [3, 6, 9]. Since we write entries of size 8B, we can use the following AVX intrinsic to write 4 buffered entries to the partition at once:

```
_mm256_stream_si256(__m256i* mem, __m256i a)
```

Furthermore, the processor tries to apply write-combining [3,5,6,9] to fill a cache-line in its *write-combine buffer* before writing to main memory. As soon it is filled (after two subsequent calls to the stream intrinsic), it is flushed out without ever reading the corresponding cache-line from main memory. However, streaming stores also introduce difficulties: the address we flush to must be a multiple of 32B. One solution to this problem is padding the partitions of the output array such that they are cache-line aligned, but this increases the size of the array unnecessarily. In contrast to that, our implementation simply offsets the start of the partitioning filling to align it and corrects the few wrongly written entries afterwards. As there is no runtime difference observable between the padded and unpadded version, we will not distinguish them in the further investigation.

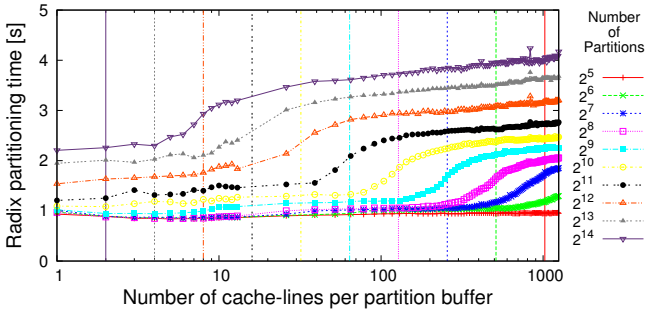
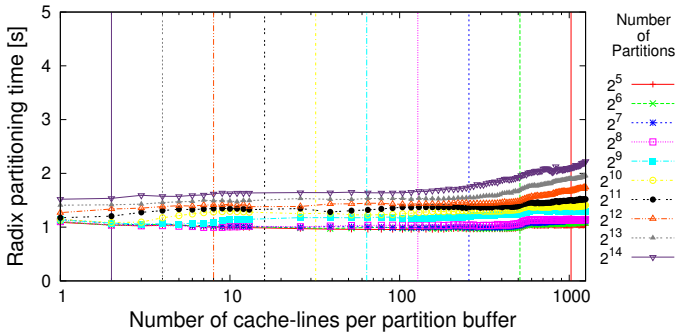
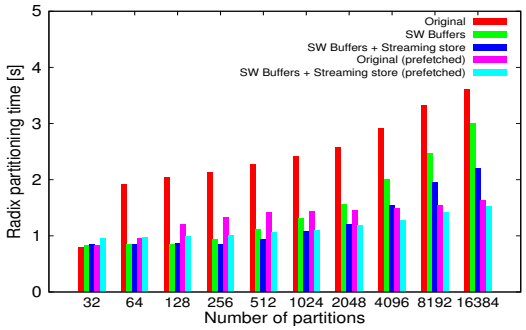


Figure 4: ***Partitioning time*** *of the* ***buffered & streamed version*** *for varying number of partitions and partition buffer sizes.*

We can see in comparison to Figure 3 that for a high number of partitions, using the non-temporal streaming store clearly improves the runtime of partitioning. The higher the total number of partitions, the more space the buffers occupy and the more precious cache memory becomes. In this situation, avoiding cache-pollution caused by bringing in output cache-lines is clearly beneficial. However, one should use small buffers (less than 8 cache-lines) to make this technique pay off or else the cache-misses triggered by buffer filling overshadow the streaming benefit. In Table 2, we compare the streamed version with the plain software managed buffers version of Figure 3 to see the impact of this technique. In contrast to the previous optimization (Table 1), the positive impact of streaming stores increases with the number of partitions, up to a speedup of 1.36x for 16384 partitions. The optimal partition buffer size decreases again with the number of partitions, with the best size of a single-cache line from 2048 partitions on. We can see that the streamed version prefers in all cases smaller buffers than the unstreamed version. This shows that the avoidance of cache-misses when flushing a buffer has a higher priority when using streaming stores. Nevertheless, we also see that streaming introduces overhead for a small number of partitions.



(a) **Partitioning time** of the **buffered & streamed & prefeteched version**.



(b) **Comparison** of partitioning time of versions **with and without prefetching** (best buffer size shown).

*Figure 5: Analyzing the effect of **prefetching hints** on the original version and the buffered & streamed version.*

# Partitions	SW Buffers [s]	([cl])	SW Buffers + Streaming [s]	([cl])	Speedup
32	0.83	(26)	0.84	(5)	0.98x
64	0.85	(12)	0.85	(7)	1x
128	0.85	(12)	0.86	(6)	0.99x
256	0.94	(8)	0.84	(6)	1.12x
512	1.11	(52)	0.94	(4)	1.18x
1024	1.30	(26)	1.08	(2)	1.20x
2048	1.57	(12)	1.21	(1)	1.30x
4096	2.01	(11)	1.54	(1)	1.31x
8192	2.47	(6)	1.95	(1)	1.27x
16384	3.00	(4)	2.21	(1)	1.36x

*Table 2: **Partitioning time** of the **software-managed buffers version** (Figure 3) and the **software-managed buffers version using non-temporal streaming stores** (Figure 4). We show the best runtime and partition buffer size in number of cache-lines (cl).*

3.3 Prefetching

Writing to individual partitions respectively buffers resembles random access that can cause cache-misses. Idealy, we want to hide these by introducing prefetching hints to ensure that requested data is cached in time. To do so, we use the following intrinsic:

```
__builtin_prefetch(void* mem, int rw, int l)
```

This triggers the generation of data prefetch instructions that will prefetch the memory at address `mem`, such that it is (hopefully) already available in cache at access time. We set `rw` to 1 since we write and `l` to 0, indicating that the temporal locality of the memory is low (high eviction chance after first access). To the best of our knowledge, we are the first group extending both the original version and the version using buffers and streaming stores in a way that when writing to an output partition respectively a buffer, the subsequent slot is prefetched. Figure 5(a) shows the runtime of the version that extends the buffered and streamed version with previously mentioned prefetching hints. Obviously, the performance is much more stable with respect to the buffer size, indicating that the prefetching indeed hides the cache-miss latency nicely. If we compare that with the versions without prefetching (see Figure 5(b)), we can observe a high impact of the prefetching hints for both the unbuffered and buffered version. From 4096 partitions on, both prefetching versions improve over all previous ones, showing that prefetching can indeed mask the cache misses from random writes. However, we can also observe that prefetching adds overhead and decreases the performance if misses are not a major problem, e.g. for less than 1024 partitions.

3.4 Micro Row Layouts

In the trivial implementation using software-managed buffers, each insertion of an entry into a buffer triggers the access of two cache-lines. First, the buffer fillstate is read from an array (first cache-line) and then, the entry is inserted (second cache-line). Additionally, when a buffer is flushed, the destination index is read from another array, leading to the access of a third cache-line. However, when limiting the buffer size to a single cache-line, it is possible to guarantee the access of only one line per entry. To do so, the last slot of each buffer is used to temporarily store the working variables. The aforementioned fillstate is stored in that way in [3, 6]. We additionally store the write destination in that slot. Note that this technique does not sacrifice a slot in the partition buffer: the fillstate and the destination are overwritten by the last entry that is inserted and restored from a local variable after a flush. In Table 3 we present the impact of placing only the fillstate, or both fillstate and destination on the cache-line (basically a micro row layout) over the naive approach of storing them separately. Obviously,

# Partitions	Nothing [s]	Fillstate [s]	Fillstate & Destination [s]
32	0.84	0.91	0.87
64	0.85	0.93	0.89
128	0.86	0.90	0.88
256	0.84	0.88	0.87
512	0.94	0.94	0.92
1024	1.08	0.99	0.96
2048	1.21	1.02	0.99
4096	1.54	1.18	1.11
8192	1.95	1.42	1.31
16384	2.21	1.59	1.45

*Table 3: **Partitioning time** of the **buffered & streamed version** with a **partition buffer size of one cache-line**. We distinguish on whether only the buffer fillstate, destination and fillstate, or nothing is stored on the partition buffer cache-line.*

these tiny changes have a surprising and significant impact on the runtime. For 16384 partitions, we see an improvement of factor 1.52x just due to this effect! Interestingly, we also observe that for less than 1024 partitions, storing that data in separate arrays (a micro column layout) shows the best runtime as these structures are small (e.g. 1KB for 128 partitions) and remain in L1 cache.

3.5 Small vs. Huge Pages

The initial motivation in using software-managed buffers was to reduce TLB-misses when writing to a large number of partitions. In this context, the size of the memory page is directly connected to the amount of required address translations when writing to the output array. In current linux kernels, pages of size 4KB and 2MB (*huge pages*) are supported. Huge pages reduce the amount of pages needed to allocate a consecutive memory area by a factor of 512, therefore decreasing the chance of TLB misses at access time. By default, our system was set up such that *transparent huge*

pages were considered for all allocations, so we were actually using them in the previous experiments already. Let us now see how a smaller page size affects the runtime of the algorithms in Figure 6.

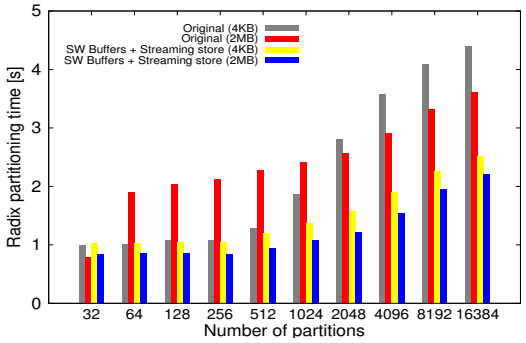


Figure 6: ***Partitioning time*** *of the* ***original version*** *and the* ***buffered & streamed version*** *for* ***4KB*** *and* ***2MB pages****.*

For 32 partitions, the impact of the page size is negligible as the translations can be cached in any case. From that on till 1024 partitions, we see a clear advantage of 4KB over 2MB pages for the original version, since for a small number of partitions, a single partition is still larger than a huge page and thus, we benefit from the higher number of 4KB page TLB slots. This turns around as soon as multiple partitions fit on a single huge page. Interestingly, in combination with software-managed buffers and streaming stores, the 4KB pages do never pay off. By buffering, the potential TLB miss rate decreased already by factor b (see Section 3.1) and the higher allocation costs for 4KB pages render the advantage void.

3.6 Varying Experimental Setup

So far, we have evaluated all methods over an array of 100 million (key-payload) pairs of 8B each. Let us now see how the algorithms scale with a ten-fold increase of the number of entries to 1 billion and when doubling the entry size to 16B for a pair. Figure 7 shows the average slowdown factors over all tested numbers of partitions. Obviously, the tested methods scale almost linearly with the number of entries. A doubling of the entry size results in an average slowdown between only 1.31x for Original (prefetched) and 1.69x for SW Buffers + Streaming (prefetched), although the amount of data to move is twice as large as before. This indicates again, that not data transfer is the limiting factor but the processing of individual elements in terms of random access costs.

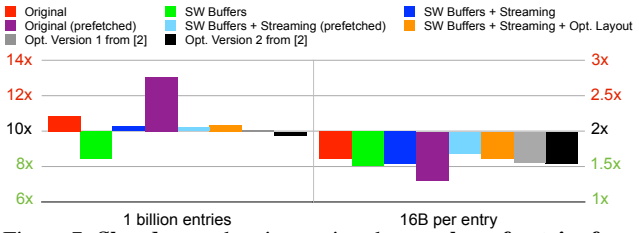


Figure 7: ***Slowdown*** *when increasing the* ***number of entries from 100 million to 1 billion*** *and the* ***entry size from 8B to 16B****.*

4. CONCLUSION

In this mini paper, we saw that even a trivial algorithm such as radix partition can be significantly improved by making it aware of current hardware features, leading to an improvement over the original version of factor 2.5x. Figure 8 gives a complete overview over all evaluated methods in comparison to two state-of-the-art implementations [2] used in [3]. As a side-effect of our evaluation, we were even able to improve over the existing methods in all tested configurations. For smaller number of partitions, using larger partition buffers without micro layouts shows the best performance. For 32 partitions, we recommend using the original version without any optimizations. For larger partition numbers, the layout-optimized single cache-line version pays off the most. Further, prefetching hints have a high impact on the runtime and are an option as soon as cache-space becomes the limiting factor. Additionally to the identification of an absolute winner in terms of runtime, we carefully investigated the individual positive and negative impact of each optimization. Figure 9 summarizes for all tested optimization paths of Figure 2 those, that improved over the original version. This final overview showing the influenceability of the techniques by the partition count and their varying impact on the runtime clearly demonstrates a surprising difficulty of simple things.

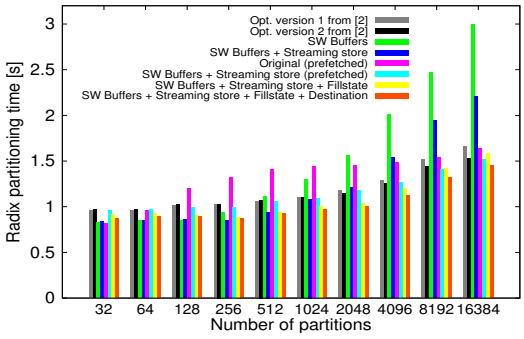


Figure 8: ***Overview of partitioning time*** *of* ***all optimized version*** *in comparison with a* ***state-of-the-art partitioning from [2,3]****.*

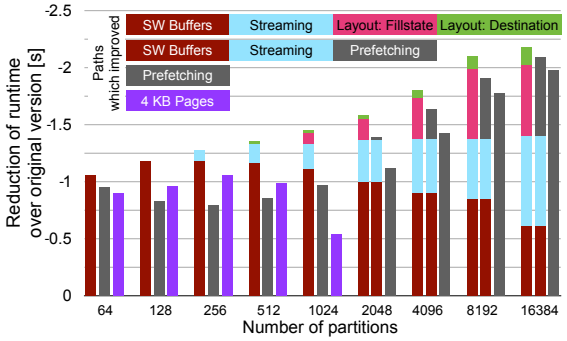


Figure 9: ***Reduction of runtime*** *of the individual techniques over the original version. Each stacked bar corresponds to a path of optimizations in Figure 2.*

Research partially supported by BMBF.

5. REFERENCES

[1] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main memory adaptive indexing for multi-core systems. DaMoN '14, Snowbird, UT, USA, 2014.

[2] C. Balkesen. `http://www.systems.ethz.ch/sites/default/files/file/sort-merge-joins-1_4_tar.gz`, January 2015.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[4] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *PVLDB*, pages 54–65, 1999.

[5] U. Drepper. What every programmer should know about memory, 2007.

[6] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD 2014, Snowbird, UT, USA, June 22-27*, pages 755–766, 2014.

[7] 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, Indianapolis, Indiana, USA, June 6-10*, pages 351–362, 2010.

[8] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *PVLDB*, 7(2):97–108, 2013.

[9] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par, Bordeaux, France, August 29 - September 2*, pages 160–169, 2011.