Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities

Zuyu Zhang, Harshad Deshmukh, Jignesh M. Patel
Computer Sciences Department
University of Wisconsin – Madison
{zuyu, harshad, jignesh}@cs.wisc.edu

ABSTRACT

Data partitioning is an important primitive for in-memory data processing systems, and in many cases it is the key performance bottleneck. This important primitive has been the focus of many studies in the past. However, as we argue in this paper, these previous studies have been narrow in their scope leaving many unanswered questions that are of paramount importance in practice. Consequently, to the best of our knowledge, there is no clear answer to the seemingly simple question of what is an efficient partitioning strategy for in-memory data systems. In this paper, we carefully consider this data partitioning primitive in the context of multi-core in-memory data settings. We look at past work in this area and note that many of these studies miss looking at many important aspects such as the impact of tuple size and the impact of the data formats (e.g. row-store vs. columnar-store). We build on this initial observation and examine a number of data partitioning strategies, leading to a better understanding of how data partitioning methods perform on modern multi-core large memory systems. We note a few interesting observations, including how relatively simple methods work quite well in practice across a broad spectrum of data parameters. To help future researchers, we propose a partitioning benchmark so that work in this area can take a broader and more realistic perspective when working on data partitioning methods. Overall, the key contribution of this paper is to separate the wheat from the chaff in previous research in this area, analyze the relative performance of various methods on a broad set of data parameters, and help provide a more systematic evaluation framework for future work in this area.

1. INTRODUCTION

Data partitioning or data shuffling is an important operator in analytic data processing systems. During query processing, this operation is sometimes called explicitly and sometimes called implicitly by other data processing operations such as join, aggregation, and sort operations. For example, partitioning can speed up join and aggregation operations as it trims unnecessary computation on tuples [4, 8, 16, 17, 20, 24, 26]. Partitioning is a prerequisite step in sort algorithms like radix sort [4, 10, 16, 23, 28]. Moreover, partitioning is often called explicitly as an operator in data platforms to exploit the benefits of partitioned parallelism [12, 14].

Given the importance of the partitioning operation, it is not surprising that this important primitive has been the focus of many studies in the past. The lineage of some of the key methods in this area is shown in Figure 1, and a detailed and comprehensive summary of this body of work is shown in Table 1. The bulk of this prior work focuses on improving partitioning performance using techniques such as cache-friendly algorithms [8, 17, 18, 26], leveraging hardware parallelism primitives (SIMD [9, 19], multi-core [3, 7, 16, 23, 28], many-core [6]), exploiting hardware features (non-temporal streaming store [25, 28]), custom software-based techniques (software prefetching [25], software-managed buffer [23, 25]), new data organizations (micro layout optimization [25]), and hardware accelerators (FPGA [15], GPU [23, 27]).

When we scrutinize this previous work, we quickly learn that many important aspects are simply missing from these studies, dramatically impacting the utility of these results in practice. For example, to the best of our knowledge, the impact of tuple size and the impact of the data formats (e.g. row-store vs. columnar-store) has not been considered in a systematic way in these previous works. Take the data format issue as another example. Do we know the performance impact of partitioning a dataset when we change the format of the dataset from a row-store to a columnar-store? Other related questions that are paramount to answer when considering implementing an efficient partitioning methods are: What methods work well when the tuple size is "small" and what methods work well when the tuple size is "large"? Are some partitioning methods "versatile" so that they can cover a wide range of data parameters and provide relatively good performance?

We set out to answer some of these questions, and in this paper detail our findings. First, we note the limitation in the prior research in this area, and take a small step in addressing this limitation by presenting an analysis of many previous methods. We point to the strengths and weakness of these methods.

Second, we propose that any work in partitioning should at least consider three data parameters: tuple size, key size, and data format. We propose a benchmark, simply called the *Partitioning Benchmark*, to capture these key parameters.

Third, we use the newly-proposed Partitioning Benchmark to examine some of the issues outlined above. A few key highlights of our results are: a) Columnar-stores and row-stores often have similar performance; b) A simple "textbook" radix-implementation with prefetching works quite well across a broad range of parameters; c) Increasing the number of partitions quickly leads to a big increase in partitioning costs. The last point has potential implications for techniques like over-partitioning, which are considered to be robust ways to deal with data skew.

The remainder of this paper is organized as follows. In the next

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

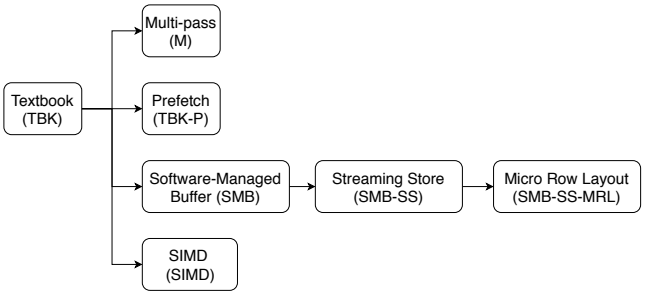

Figure 1: **Radix Partition Strategies**

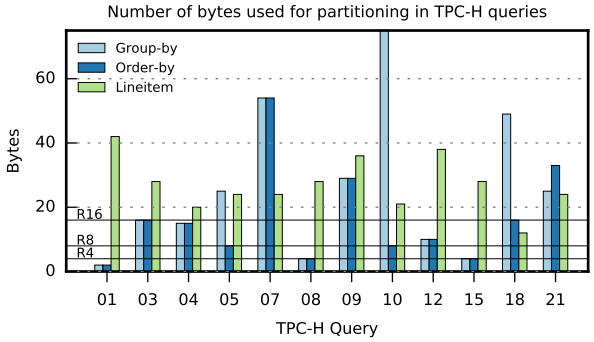

Figure 2: **Width of tuple required for partitioning to execute various TPC-H queries. The lines at 4, 8, and 16 bytes indicates the tuple width considered by related work. For this analysis, we assume that the *Decimal*, *Date*, and *year* columns occupy 8, 8, and 4 bytes, respectively, while a variable-length attribute uses its maximal length.**

section, we look at the parameters that have been used in previous work and compare those to the ones in an end-to-end benchmark, TPC-H. In Section 3 we propose the Partitioning Benchmark. Section 4 describes a few popular radix partitioning-based techniques, and Section 5 contains the experimental results. Section 6 briefly discusses previous work on partitioning. We present our concluding remarks in Section 7.

2. PARAMETER SPACE

The data-related parameters that can affect the performance of data partitioning are large, and in our work we focus on a limited set of key parameters. These parameters are: tuple size, key size, storage format, and number of partitions.

2.1 Tuple size, Key size, and Data format

Let us consider the parameter *tuple size* first. For our purpose we consider a tuple to have a key and a payload. The internal structure of the payload is not important, and we are only concerned with its size. The *key size* though is important as some methods only work with certain key sizes. The tuple size represents how many bytes of data must be moved for each input tuple across the partitioning operation, but the input data format (we consider row-store and columnar-store) can also impact the cost of moving the data.

As shown in Table 1, most prior work consider fixed size (8 or 16 bytes) tuples with numeric columns, and limited payload sizes.

However, queries and datasets are much more complicated in practice. As a comparison point, consider the TPC-H benchmark [2]. Figure 2 shows three values for a selected set of queries in which the *Lineitem* table is involved. The first measure is the projected attributes from the *Lineitem* table, which represents the "effective tuple size" for that query. Conceptually this measure represents the size of the tuple on which that query operates. The other measures are the size of the keys in the `GROUP BY` and the `ORDER BY` clauses, and are the"key size" that a partitioning operation has to work with.

The solid lines show the common tuple sizes considered by related work (4, 8, and 16 bytes). Figure 2 highlights a gap between what has been considered in evaluating partitioning methods in prior work and what is needed by TPC-H queries. As we will show later, these sizes have a large impact on performance and applicability of partitioning methods.

Next, let us look at another dimension, which is the storage format of input tables. Many previous works represent the columnar-store as a Binary Association Table (BAT) [8, 17, 18], where each column consists of the values and the tuple id. Thus, only the key column is partitioned, and the other columns are untouched. Partitioning is considered "done" when a partition file is produced with tuple ids for the "value" component of the output. However, this leaves the open question of what happens if one needs the actual values in the output (as one often does when considering end-to-end query performance).

Now, consider a storage format in which actual data values are present (rather than tuple ids). Figure 3 shows the end-to-end partition time using a standard radix partitioning approach (described in detail later in Section 4.1) with 32 partitions, with data in both the row-store and the columnar-store formats, when running in a single thread. Although the columnar-store is generally faster than the row-store in the *Histogram* computation phase, the *Output* phase is much slower due to more TLB misses. For the `col-4-16` case, the large payloads (16 bytes) result in additional cache misses making that a more expensive format (compared to `row-4-16`) to partition. Thus, considering only BAT files misses the case when (even in a columnar database) actual payloads have to be outputted.

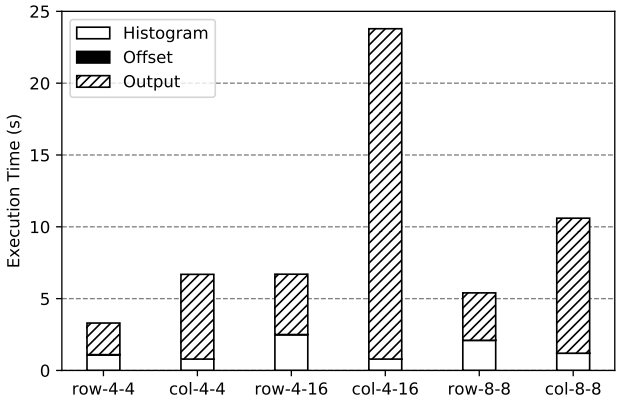

Figure 3: **Storage format impact on Radix Partitioning 1 billion tuples with 32 partitions using just 1 thread. The dataset names are coded as `format-keysize-payloadsize`; for example, `col-4-16` is a dataset with two columns, 4 and 16 bytes, stored in a columnar storage format. The three key phases of radix partitioning are shown, the initial histogram creation phase, creating offsets, and the final output. The second phase takes negligible amount of time.**

3. PARTITION BENCHMARK

Paper	Approaches	Radix bits	tuple size†	Number of Threads
Boncz et al. [8], Manegold et al. [17, 18]	M	1-23	4/4	1
Cieslewicz et al. [11]	M	1-18	8/8	1, 2, 4, 8, 16, & 32
Kim et al. [16]	M, SIMD	6-20	4/4	4, 8
Satish et al. [23]	M, SMB	8	4, 8, 16, 4/4	8
Wassenberg et al. [28]	SMB-SS	8	4, 4/4	1-8
Blanas et al. [7]	M	4, 6, 8-13, 15, 17	8/8	6/12 on Intel Nehalem, 8/64 on Sun UltraSPARC T2
Albutiu et al. [3]	TBK	5-11	8/8	2, 4, 8, 16, 32, 64
Balkesen et al. [5]	M	12-16	4/4, 8/8	1-8
Balkesen et al. [4]	SMB-SS	2-11	4/4	1, 2, 4, 8, 16, 32, 64
Polychroniou et al. [20]	SIMD, SMB-SS	2-13	4-4, 8-8	1-16
Schuhknecht et al. [25]	TBK, TBK-P, SMB, SMB-SS, SMB-SS-MRL	5-14	4/4, 8/8	1
Cho et al. [10]	TBK	5-12	8/8, 10/90	1-32
Polychroniou et al. [19]	TBK, SMB, SIMD	3-13	4, 4-4	1, 3, 7, 15, 30, 61, 122, 244
Schuh et al. [24]	M, SMB-SS	8-16	4/4	4, 8, 16, 32, 60, 120
Kara et al. [15]	SMB-SS	8-13	4/4	1, 2, 4, 8, 10
Stehle et al. [27]	M	8	4, 8, 4/4, 8/8	12
Barthels et al. [6]	M	N/A	8/8	128, 256, 512, 1024, 2048, 4096 cores

†Tuple size notations: key column only (e.g., 4), key/payload sizes in the row-store (e.g., 4/4), key-payload size in the columnar-store (e.g., 4-4).

Table 1: **Previous work on radix-based data partitioning methods. The tags for the key approaches are introduced in Figure 1.**

We propose a partitioning benchmark to encourage a more holistic evaluation of data partitioning methods. The benchmark has four key parameters: 1) the tuple size, 2) the key size, 3) the storage format, and 4) the number of partitions. Note that these parameters are independent of each other and are important characteristics of the physical organization of data in a table. The default benchmark has 1 billion tuples, which is the only case we consider in this paper.

There are two key sizes in the benchmark (8 and 10 bytes), three payload sizes (8, 90, and 92 bytes), and two data formats (row-store and columnar-store). We aim to balance the complexity of the benchmark and the factors that impact partitioning methods. Thus, to run the entire benchmark one only uses the 5 combinations that are shown in Table 2. The names for each of the 5 datasets in the benchmark is simply the `storageformat-keysize-payloadsize` string. For example, `row-10-90` refers to the (100 GB) dataset in row-store with 10-byte keys and 90-byte payloads. For 100-byte tuples, both the keys and payloads are simply treated as character arrays, as in the Sort Benchmark [1, 13]. In fact, we use a modified version of the Sort Benchmark as the data generator for the Partition Benchmark. The keys generated follow both a uniform and a skew distribution.

In Table 1, we also show the key and payload sizes that have been used in previously proposed partitioning strategies. As one can quickly observe, many of the previous methods have used limited datasets to draw conclusions about partitioning performance.

The last component of the benchmark is reporting number for a varying **number of partitions**. For this parameter, we aim to cover a broad spectrum of hardware characters, including the TLB capacity, and L1/L2/L3 cache sizes. The benchmark requires reporting results for 8, 64, 512, 4K, and 32K partitions.

4. RADIX PARTITIONING TECHNIQUES

Radix partitioning is a starting point for partitioning in a modern processor with TLB, and we use this as the initial "textbook" implementation, called *TBK* in Figure 1. In this section, we also describe key previous implementation techniques of radix partitioning that improve on the canonical version, including prefetching, multi-pass, software-managed buffer, and non-temporal streaming store. (These methods are also shown in Figure 1.) We also discuss whether these techniques apply to the columnar-store and different tuple sizes cases. In the interest of space, we omit the SIMD approach, and refer the interested readers to the work by Polychroniou et al. [19, 20].

storage format	key size	payload size	TBK (-P/M)	SMB (-SS /-MRL)	SIMD
row	10	90	✓	✓	
	8	8	✓	✓	✓
column	10	90	✓	✓	
	8	92	✓	✓	✓†
	8	8	✓	✓	✓

†Only applicable to the key column.

Table 2: **Partition Strategies in the Partition Benchmark**

4.1 Textbook Radix Partitioning Algorithm

The basic not-in-place radix partitioning, which we abbreviate as TBK, operates in three phases as shown in Algorithm 1. By not-in-place, we mean that we do not modify the inputs, but instead have dedicated space for the partitioned output. All the partitions in the output are stored continuously.

In the first *Histogram* phase, a histogram is computed for all the partitions. The *Offset* phase then calculates the starting index position value in the output (per partition) by computing the prefix sum of the histogram. Finally, the *Output* phase writes the input tuples to the corresponding partitions in the output at the index calculated previously. Each partition keeps track of its index for the next tuple location. Note that the second pass dominates the overall execution time as it incurs "random" writes. (Even in the in-memory settings, the random write patterns are important, as we discuss below.)

Algorithm 1 The Textbook Radix Partitioning Algorithm

```
1:  procedure RADIXPARTITION(input, output, radix_bit)
2:      for i = 0 to num_tuples - 1 do
3:          ++histogram[getRadixPartitionId(input[i], radix_bit)]
4:      end for
5:      offset ← 0
6:      for i = 0 to num_partitions - 1 do
7:          dest[i] = offset
8:          offset += histogram[i]
9:      end for
10:     for i = 0 to num_tuples - 1 do
11:         partition_id ← getRadixPartitionId(input[i], radix_bit)
12:         output[dest[partition_id]] = input[i]
13:         ++dest[partition_id]
14:     end for
15: end procedure
16: procedure GETRADIXPARTITIONID(key, radix_bit)
17:     mask ← (1 « radix_bit) - 1
18:     return (key & mask)                          ▷ LSB version
19: end procedure
```

The radix partition has at least two ways to compute the partition id: use the most significant bits (MSB) or the least significant bits (LSB). Algorithm 1 shows the LSB implementation that uses the lowest *radix_bit* bits in the key column. Our partition function is based on the radix-cluster algorithm proposed by Manegold et al. [18]. We partition tuples based on the rightmost $\mathfrak{p}$ bits of the key column to $2^{\mathfrak{p}}$ partitions. Richter et al. [22] summaries more advanced hash schemes for different data distributions.

The random writes in the *Output* phase have two performance issues: TLB misses and cache misses. In general, each partition in the output resides in its own memory page, and thus needs to use/occupy one TLB entry for the virtual address translation. If the number of partition is greater than the number of TLB entries (typically 64 on modern Intel processors), then the output phase will encounter a large number of TLB misses, impacting performance. (Future hardware that does not need TLBs and uses direct mapping will likely have a far larger tolerance to the partitioning parameter.)

With radix partitioning another technique that can help improve performance is using software prefetching hints, where by the programmer deliberately introduces prefetching instructions for anticipated *write locations*. This technique can result in better cache performance, as the the processor may fetch the memory address into the cache, and thus reduce the compulsory cache misses. This is a general optimization, and works across any tuple size.

4.2 Multi-pass (M)

Boncz et al. [8] observed the TLB thrashing problem, and proposed a multi-pass partition algorithm called "radix-cluster", where TBK (cf. Section 4.1) is invoked on each pass. In each pass the input is divided into B partitions, where B is less than the number of L1 TLB entries. For example, to partition a dataset into 2^{13} partitions, on a machine with 64 L1 TLB entries, 3 passes are used: the first pass partitions the data using the leftmost first 5 bits (32-way partitioning), the second uses the next 5 bits, and the last uses the next 3 bits.

4.3 Software-Managed Buffers (SMB)

Software-managed buffer [23] was proposed as a technique to improve both TLB and cache thrashing. It assigns each partition a fixed size region (at least one cache line size) to combine the output(s). Each partition in the buffer maintains a "fill-up" status. When a partition region fills up, the tuples in the region are written to the actual output destination.

This approach introduces additional overheads, since it writes the output tuples twice. Even worse, for the second write, the cache could require the processor to fetch the corresponding cache line of the output location (for the write).

4.4 Streaming Stores (SMB-SS)

Non-temporal streaming store was proposed [20, 25] to avoid the cache pollution resulting from the second write that is incurred by the software-managed buffer technique. It does this by using a special instruction that leverages dedicated "write-combining" hardware registers to bypass the cache and directly transfer to the memory destinations. This approach, however, requires that the output being written must align to a fixed size (e.g., 16 or 32 bytes on Intel processors). Thus, the start index of each partition in the output must be padded to this fixed size. In other words, this method requires a buffer size that is larger than the buffer size that SMB uses when tuples do not pack nicely inside a single cache line (e.g., a 10-byte column) or are large (e.g., 90-byte column in the benchmark).

Schuhknecht et al. [25] proposed a micro row layout optimization (SMB-SS-MRL) to optimize using the cache space, and to possibly reduce the number of cache accesses. Recall that when using the SMB approach, two states are maintained for each partition: the output destination for the next tuple to write, and the fill-up state of the buffer. Instead of storing these two pieces of information separately, the optimization stores them in the last entry of the buffer for each partition. Thus, in general it needs to access the cache once (and twice at most, but rarely).

4.5 Parallel Radix Partitioning

While the discussion above did not consider multi-threaded execution, all these methods can be run in multi-treaded mode. One simply divides the input into subsets/sub-arrays, and assigns each sub-array to a different thread. After all threads finish computing the histogram on its own sub-array, we calculate the global offset of the output buffer for all partitions and threads. Finally, in the output phase, each thread writes the output to its own pre-computed region with no contention. In the output phase, all optimizations described above apply.

For the multi-pass (M) approach the first pass does the same described above. For the remaining passes, each thread takes as input one partitioned segment from the previous, and repartitions it into sub-partitions using different bits as the previous one.

4.6 Implications for Columnar-Store

When the input data is in a columnar-store format, some of the techniques above do not apply. When partitioning data in row-store format, the tuple size is a key parameter. With a columnar-store foramt, however, the size of each of the columns also plays a critical role. The SIMD approach can only process data using the scatter instructions that can work efficiently only when the column width/size fits in the SIMD vector registers, which are 128-, 256-, or 512-bit wide. For column that are 10-, 90-, or 92-bytes wide (see in Table 2), it is not easy to load the tuples efficiently. On the other hand, the SMB-SS approach requires that the buffer size per partition aligns with the streaming write width (e.g., 128-, 256-, or 512-bit). For a column whose size can be divided by the streaming write width, choosing the streaming write width as the miminal buffer size per partition is optimal for cache locality. Otherwise, we need to pack multiple tuples into a buffer whose size is multiple times that of the streaming write width (For example, when using the 256-bit streaming store instruction, `_mm256_stream_si256`, and the

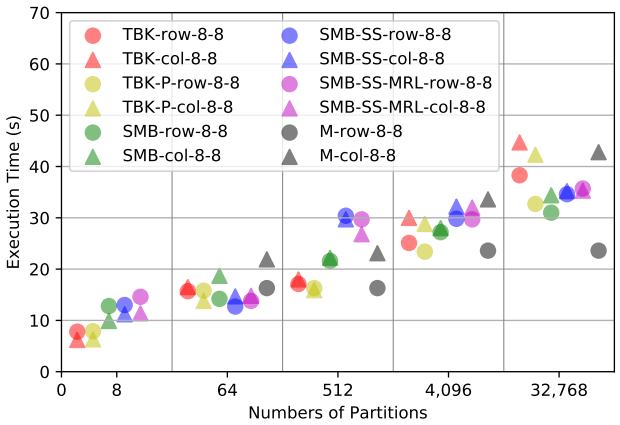

Figure 4: **Partitioning 1 billion 16-byte tuples using a single thread.**

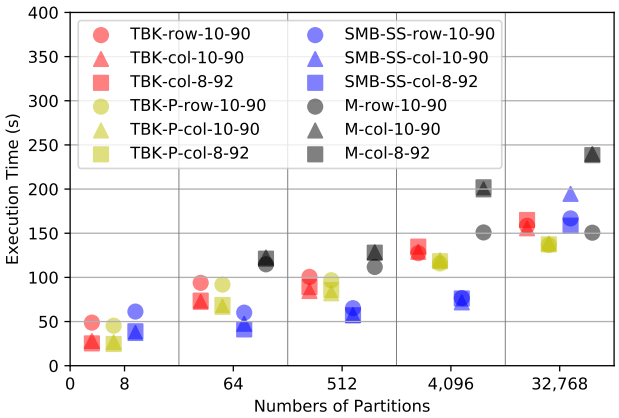

Figure 5: **Partitioning 1 billion 100-byte tuples using a single thread.**

`col-10-90` format, we have to group 16 tuples per partition as the minimal buffer size, while `col-8-92` just needs to group 8). Note that the larger the buffer size, the poor the cache locality. Thus, we should choose the minimal buffer size for best performance.

5. EVALUATION

In this section, we present results comparing various partitioning methods. We focus our study on the following methods: textbook radix partitioning with and without prefetching (cf. Section 4.1), with software-managed buffer (cf. Section 4.3), with streaming store (cf. Section 4.4), with streaming store and a micro layout optimization (cf. Section 4.4), and with multi-pass (cf. Section 4.2).

We have implemented these in a stand-alone C++ program, and use the Partition Benchmark datasets (cf. Section 3).

Our experiments were conduct on a "bare-metal" CloudLab server [21]. The machine has 384 GB of main memory and two 2.6 GHz Intel Xeon Gold (Skylake) processors, each with 16 physical cores. Each core has a dedicated 32 KB 8-way L1 instruction cache, a dedicated 32 KB 8-way L1 data cache, a dedicated 1 MB 16-way set L2 unified cache, and it shares a 22 MB 11-way L3 cache with other cores on the same processor. In 4KB page mode, each core has 64 entries in a 4-way L1 TLB. (There is a separate TLB for data and instructions). The measured memory bandwidth for 16 threads is 96.2 GB/s. The machine runs Ubuntu 18.04 LTS, and we compiled our programs using Clang 6.0 with -O3 flag. We ran each experiment five times, and present the average of the last four. The difference in the runtime for each experiment was typically less than 10%.

5.1 Single-threaded Execution

Figure 4 compares the six partitioning approaches on the two `8-8` datasets (one dataset is in a row-store format and the other in a columnar-store format). (Thoughout this paper we use a compact representation to show performance results, as exemplified by this figure. In each figure, we use a different color for each partitioning algorithm, and we use a different icon shape to distinguish between the row-store and the column-store cases. The x-axis is partitioned into buckets as separated by vertical lines, and in each bucket, we show the results for each algorithm and for each storage format.)

From Figure 4, we observe that the storage formats do not have a large impact on performance. However, we note that are certain "cliff" points around the TLB sizes and there is a difference in behavior when the number of partitions is above or below that point. This can be seen in Figure 3 (shown earlier), where the storage formats have a large impact with 32 partitions. But, when the number of partitions is above the number of TLB entries, the storage formats have a small impact as shown in Figure 4, except for the multi-pass (M) approach. (We note that in our dataset the payload is treated as a single column, and if that were to be treated as multiple columns, then we expect the performance of partitioning columnar-stores will decrease.)

Overall from Figure 4, we also note that the TBK-P (textbook radix-partitioning with prefetching) performs well, except in the region where the number of partitions is just above the number of TLB entries and below the number of L1 cache lines (i.e. between 64 and 512). In this region for large tuples the SMB-SS methods generally outperforms TBK-P by about 20%. For 16-byte tuples in the row-store format, the multi-pass (M) approach works well when the number of partitions is large (i.e., between 4K and 32K).

This experiment also reveals that there appear to be three zones of performance based on the number of partitions, below the number of TLB entries, above the number of L1 cache lines, and in between those two. This behavior is important in picking the number partitions, especially if there is flexibility in choosing the number of partitions. From Figure 4, we also see that increasing the number of partitions dramatically increases the cost to partition the data, but the multi-pass (M) approach has an interesting behavior for the row-store case as its performance stays the same across a broad range of partitions. For example in Figure 4 the M-row (the grey circular dot) performance is the same for 64 and 512 partitions, 4K and 32K partitions, respectively.

Next, we switch to using 100 byte records. Note in this case the SIMD partitioning methods mostly cannot be used. The TBK and SMB-based methods however are quite versatile and can deal with this dataset too. These results are shown in Figure 5. Compared to Figure 4, we note that overall partitioning is about 5X slower on this dataset which is about 6X larger. The SMB-SS method performs well for a medium numbers of partitions when the total buffer size mostly fits in the L2 cache. On the other hand, once the buffer size reaches the L3 cache size, the simply TBK-P is better performant than SMB-SS. Surprisingly, contrast to the 16-byte tuples case, the multi-pass (M) approach does not perform well in 100-byte ones, and we believe the poor cache behavior contributes to that, and the columnar-store is even worse.

5.2 Multi-threaded Single-socket Execution

Next, we expand our evaluation to the multi-threaded case. We

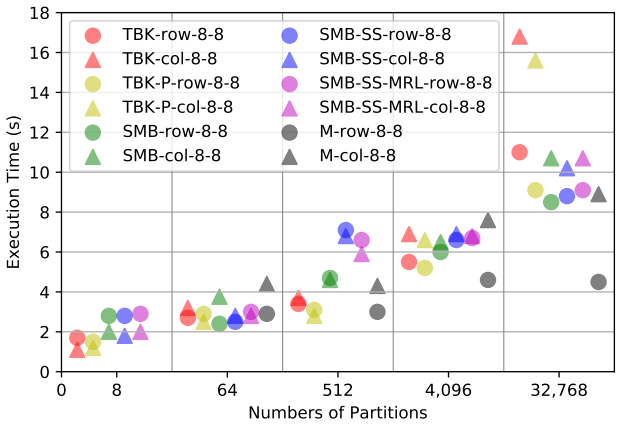

Figure 6: **Partitioning 1 billion 16-byte (8-8) tuples using 4 threads.**

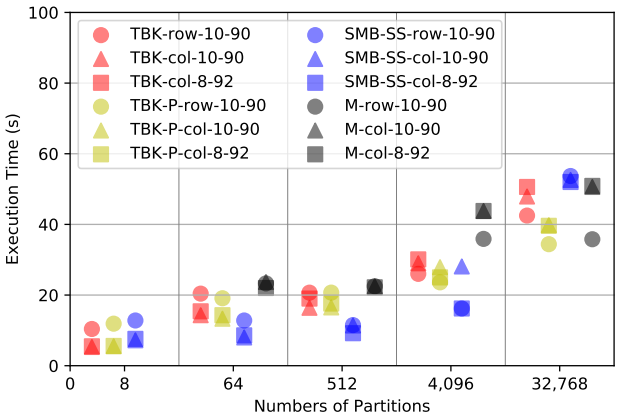

Figure 7: **Partitioning 1 billion 100-byte tuples using 4 threads.**

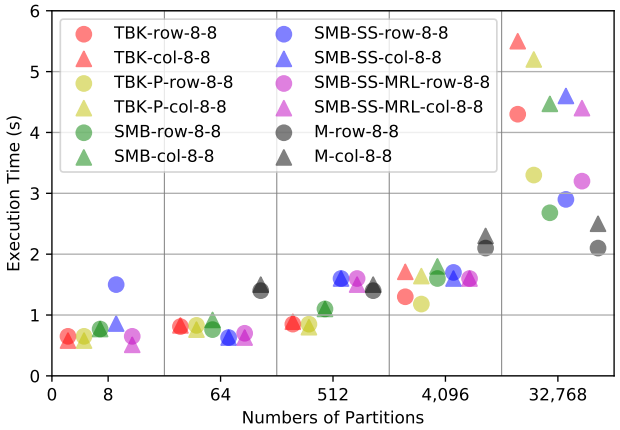

Figure 8: **Partitioning 1 billion 16-byte (8-8) tuples using 16 threads.**

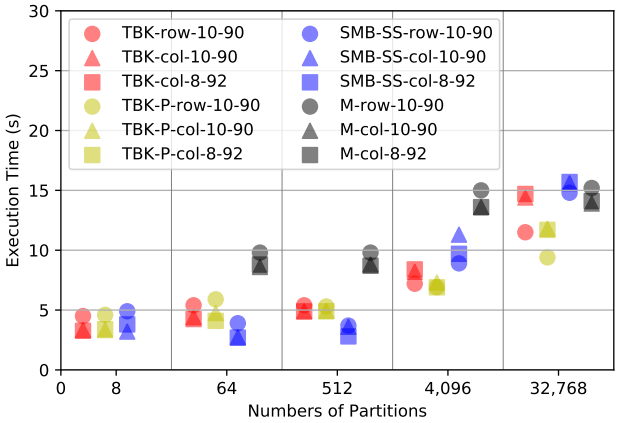

Figure 9: **Partitioning 1 billion 100-byte tuples using 16 threads.**

have run the experiments with 4 (cf. Figures 6 and 7) and 16 threads (cf. Figures 8 and 9).

Overall, the observations from the single-threaded experiments hold. We also note that 1 to 4 threads performance scales nicely across all methods by about 4X (these results are omitted in the interest of space), but from 1 to 16 the overall performance scales by about 12X, as the memory bus saturates in most cases.

For the 100-byte columnar-store, we find in some case (e.g., 4K partitions with 4-thread) the `col-10-90` format is much slower compared to the `col-8-92` counterpart. As described before (cf. Section 4.6), the former needs to pack twice as many tuples as the latter. We note that for the 256-bit version of streaming writes, in both the 10-byte and the 90-byte column cases, we need to group 16 tuples to align the boundary of the streaming write width, while the 8-byte and 92-byte needs to group just 8 tuples, and thus may encounter many more unaligned output addresses that can not leverage the streaming writes. This situation gets worse as the number of threads increase.

5.3 Data Skew

In this section, we present the results with skewed data. The skewed datasets for 16-byte tuples (cf. Figure 10, 12, and 14) were generated according to the Zipf distribution ($\alpha = 1.0$), while the dataset with 100-byte tuples (cf. Figure 11, 13, and 15) was generated according to the skew settings in the Sort Benchmark, where 3 out of 4 tuples are skewed.

Compared with the uniform dataset case, we observe that radix partition is resilient to data skew, due to its technique of using prefix sum calculations to determine the output destinations. The behavior when using single-thread (Figure 10 and Figure 11) is similar for the 4 threads (Figure 12 and Figure 13) and 16 threads cases (Figure 14 and Figure 15).

5.4 Discussion

Table 3 describes the recommended partition strategies, depending on the number of partitions. When the number of partitions is less than that of TLB entries, the textbook approach with prefetching is a good choice. If we have at least one L1 cache line for each partition, i.e. the number of partitions is between the number of TLB entries and the number of L1 cache lines, then the streaming store approach is often the best performer (by about 20% over the textbook method). When the number of partitions is more than the number of L1 cache lines, TBK-P starts to become the algorithm of choice in general, except when partitioning dataset that are in the row-store format and have "small" tuple sizes (16 bytes or smaller). This exceptional cases is better handled using the multi-pass (M)

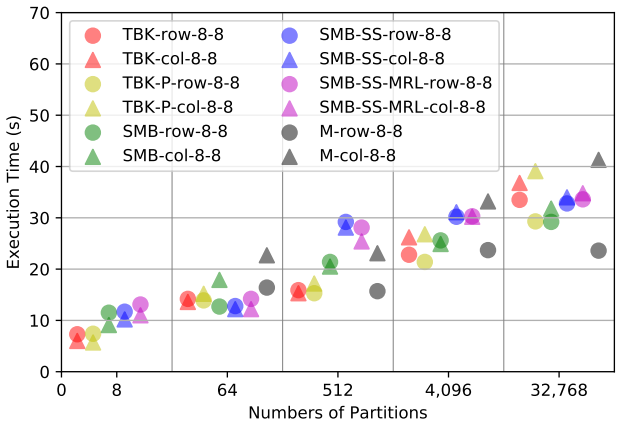

Figure 10: **Partitioning 1 billion 16-byte tuples with skew using a single thread.**

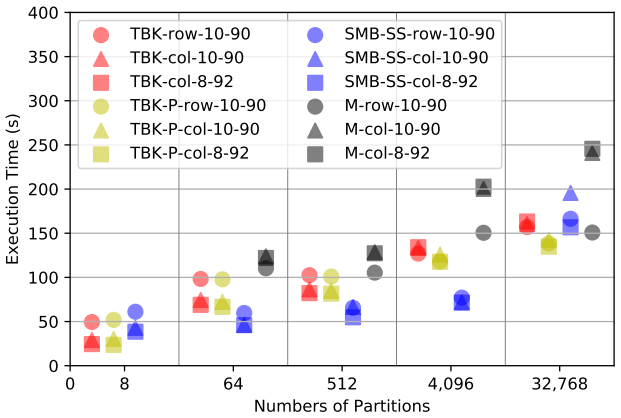

Figure 11: **Partitioning 1 billion 100-byte tuples with skew using a single thread.**

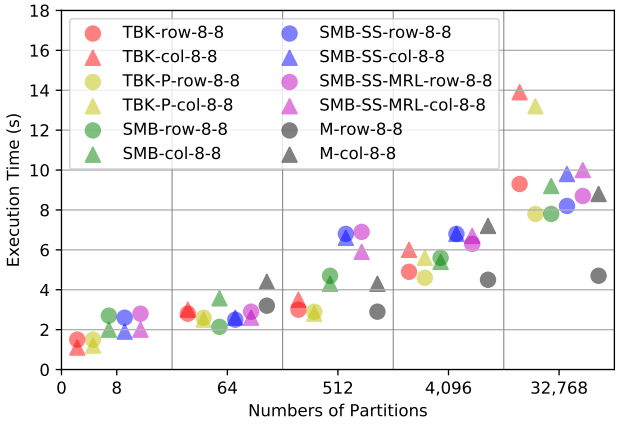

Figure 12: **Partitioning 1 billion 16-byte (8-8) tuples with skew using 4 threads.**

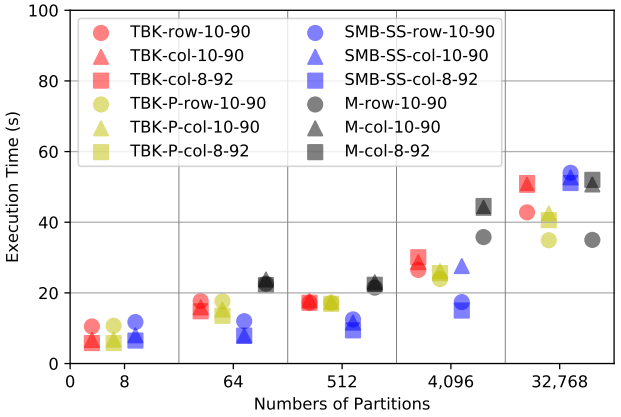

Figure 13: **Partitioning 1 billion 100-byte tuples with skew using 4 threads.**

method. Thus, for the algorithms that we compare in this paper, the simple TBK-P appears to be the algorithm of choice if a database implementer had to pick just one methods to implement.

# partitions	TBK-P	SMB-SS	M
<= 64 (# TLB entries)	✓		
<= 512 (L1 cache lines)		✓	
> 512	✓		✓†

†Choose for small (no larger than 16 bytes) tuples in the row-store format.

Table 3: **Partitioning Strategy (for the Intel Xeon Skylake processors).**

We also suspect that there is another performance "cliff" point around the number of caches lines in the L2 cache (16K), and the use of a SIMD method may be worth investigating. But, we leave that investigation as part of future work.

6. RELATED WORK

Data partitioning as an important step for parallel data processing, and has been well studied in the community. Boncz et al. [8] identified the TLB thrashing problem, and proposed multi-pass radix-cluster algorithms where in each pass the number of partitions is bound by the TLB capacity. Satish et al. [23] optimizes the partition algorithm by introducing software-managed buffers. Kim et al. [16] leverages SIMD for partitioning data. Polychroniou et al. [20] conduct a comprehensive analysis on data partitioning in multiple dimensions, including multiple partitioning approaches (i.e., hash, radix, and range) memory hierarchy (in-cache or not), memory usage (in-place or not), and NUMA awareness. They later proposed the SIMD approach for partitioning tuples in the columnar-store [19]. Schuhknecht et al. [25] compared various implementations of radix partitioning using row-store formats when using a single thread, including software prefetching, software-managed buffer, non-temporal streaming store, and proposed the micro layout optimization. Kara et al. [15] proposed fast FPGA accelerated partitioning techniques. The applicability of these and other methods is shown in Table 1. No previous work takes a comprehensive look at data partitioning.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we examine the crucial data partitioning primitive and highlight the need to take a broader perspective considering a range of data characteristics when designing and evaluating algorithms for this operation. A high-level summary of which methods

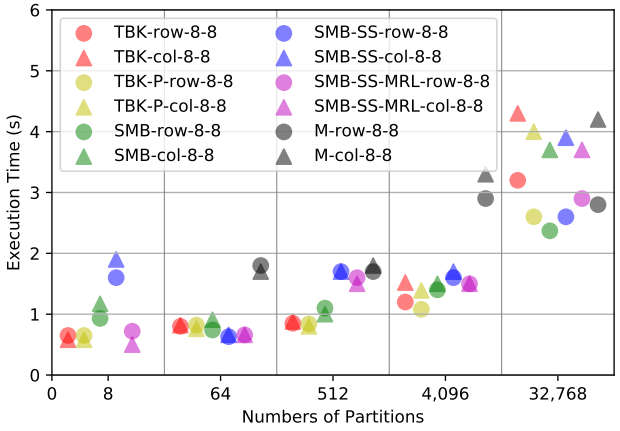

Figure 14: **Partitioning 1 billion 16-byte (8-8) tuples with skew using 16 threads.**

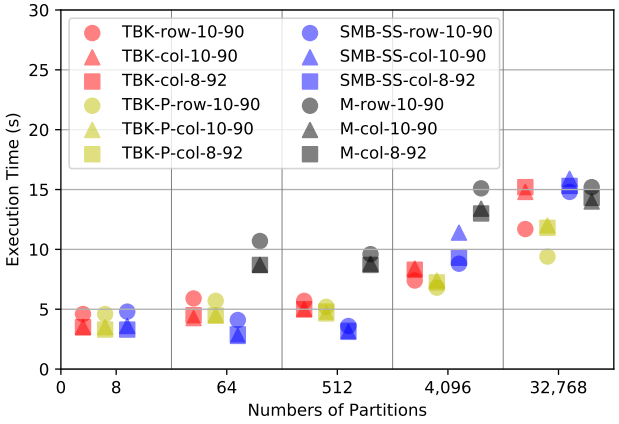

Figure 15: **Partitioning 1 billion 100-byte tuples with skew using 16 threads.**

work well overall is shown in Table 3, which surprisingly shows that a simple textbook implementation and the software-managed buffer one are quite versatile. We also propose the Partitioning Benchmark for broader research use in this area. As part of future work, we plan to extend our study to multi-socket settings, and also consider the impact of future hardware with dramatically different TLB and cache architectures.

Acknowledgments

This work was supported in part by CRISP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and by a gift donation from Google.

References

[1] Sort benchmark - Homepage. http://sortbenchmark.org/.

[2] TPC-H - Homepage. http://www.tpc.org/tpch/.

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

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

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. ICDE, pages 362–373, April 2013.

[6] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, Jan. 2017.

[7] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. SIGMOD, pages 37–48, New York, NY, USA, 2011. ACM.

[8] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. VLDB, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[10] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri. Paradis: An efficient parallel algorithm for in-place radix sort. *Proc. VLDB Endow.*, 8(12):1518–1529, Aug. 2015.

[11] J. Cieslewicz and K. A. Ross. Data partitioning on chip multiprocessors. DaMoN, pages 25–34, New York, NY, USA, 2008. ACM.

[12] D. J. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, 1990.

[13] A. et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, Apr. 1985.

[14] G. Graefe, R. L. Cole, D. L. Davison, W. J. McKenna, and R. H. Wolniewicz. Extensible query optimization and parallel execution in volcano. In *Query Processing for Advanced Database Systems, Dagstuhl*, pages 305–335. Morgan Kaufmann, 1991.

[15] K. Kara, J. Giceva, and G. Alonso. Fpga-based data partitioning. SIGMOD, pages 433–445, New York, NY, USA, 2017. ACM.

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

[17] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, July 2002.

[18] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? dissecting cpu and memory optimization effects. VLDB, pages 339–350, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[19] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. SIGMOD, pages 1493–1508, New York, NY, USA, 2015. ACM.

[20] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. SIGMOD, pages 755–766, New York, NY, USA, 2014. ACM.

[21] R. Ricci, E. Eide, and C. Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[22] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *Proc. VLDB Endow.*, 9(3):96–107, Nov. 2015.

[23] 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. SIGMOD, pages 351–362, New York, NY, USA, 2010. ACM.

[24] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. SIGMOD, pages 1961–1976, New York, NY, USA, 2016. ACM.

[25] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. On the surprising difficulty of simple things: The case of radix partitioning. *Proc. VLDB Endow.*, 8(9):934–937, May 2015.

[26] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[27] E. Stehle and H.-A. Jacobsen. A memory bandwidth-efficient hybrid radix sort on gpus. SIGMOD, pages 417–432, New York, NY, USA, 2017. ACM.

[28] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. Euro-Par, pages 160–169, Berlin, Heidelberg, 2011. Springer-Verlag.