Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, Franz Färber

Cache-Efficient Aggregation: Hashing Is Sorting

Erstveröffentlichung in / First published in:

SIGMOD/PODS'15: International Conference on Management of Data, Melbourne 31.05.2015. ACM Digital Library, S. 1123–1136. ISBN 978-1-4503-2758-9.

DOI: https://doi.org/10.1145/2723372.2747644

Cache-Efficient Aggregation: Hashing *Is* Sorting

Ingo Müller[1†‡], Peter Sanders[2†], Arnaud Lacurie[3‡]
Wolfgang Lehner[4*], Franz Färber[5‡]

[†]Karlsruhe Institute of Technology, [‡]SAP SE, [*]Dresden University of Technology

[1]ingo.mueller@kit.edu, [2]sanders@kit.edu, [3]arnaud.lacurie@sap.com,
[4]wolfgang.lehner@tu-dresden.de, [5]franz.faerber@sap.com

ABSTRACT

For decades researchers have studied the duality of hashing and sorting for the implementation of the relational operators, especially for efficient aggregation. Depending on the underlying hardware and software architecture, the specifically implemented algorithms, and the data sets used in the experiments, different authors came to different conclusions about which is the better approach. In this paper we argue that in terms of cache efficiency, the two paradigms are actually the same. We support our claim by showing that the complexity of hashing is the same as the complexity of sorting in the external memory model. Furthermore we make the similarity of the two approaches obvious by designing an algorithmic framework that allows to switch seamlessly between hashing and sorting during execution. The fact that we mix hashing and sorting routines in the same algorithmic framework allows us to leverage the advantages of both approaches and makes their similarity obvious. On a more practical note, we also show how to achieve very low constant factors by tuning both the hashing and the sorting routines to modern hardware. Since we observe a complementary dependency of the constant factors of the two routines to the locality of the input, we exploit our framework to switch to the faster routine where appropriate. The result is a novel relational aggregation algorithm that is cache-efficient—independently and without prior knowledge of input skew and output cardinality—, highly parallelizable on modern multi-core systems, and operating at a speed close to the memory bandwidth, thus outperforming the state-of-the-art by up to 3.7×.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.4 [**Database Management**]: Systems—*Parallel databases*

Keywords

Hashing, sorting, aggregation, grouping, cache-efficient, shared-memory, robust performance, “group by”, adaptive algorithm

1. INTRODUCTION

GROUPING with AGGREGATION is one of the most expensive relational database operators, maybe the most expensive one after the JOIN. It occurs in many analytical queries, e.g., in queries with a `GROUP BY` clause in SQL. The dominant cost of AGGREGATION is, as with most relational operators, the movement of the data. In the days of disk-based database systems, relational operators were designed to reduce the number of I/Os needed to access the disk whereas access to main memory was considered free. In today's in-memory database systems, the challenge stays more or less the same but moves one level up in the memory hierarchy [32]: How should an aggregation operator be designed such that it uses the CPU caches efficiently to overcome the bottleneck to the much slower main memory?

Traditionally, there are two opposite approaches to implement this operator: hashing and sorting. HASHAGGREGATION inserts the input rows into a hash table, using the grouping attributes as key and aggregating the remaining attributes in-place. SORTAGGREGATION first sorts the rows by the grouping attributes and then aggregates the consecutive rows of each group. The question about which approach is better has been debated for a long time and different authors came to different conclusions about which is the better approach in which context [3, 7, 19, 26]. The consensus is that HASHAGGREGATION is better if the number of groups is small enough such that the output fits into the cache, and SORTAGGREGATION is better if the number of groups is very large. Many systems implement both operators and decide a priori which one to use. In this paper we argue that in terms of data movement, the two paradigms are actually the same. By recognizing the fact that hashing *is* sorting, we can construct a single AGGREGATION operator with the advantages of both worlds.

As a first argument for our claim, we compare the number of cache line transfers of HASHAGGREGATION and SORTAGGREGATION. We reason in a general external memory model [1], which holds in the cache setting as well as in the disk-based setting. We confirm that if implemented naively, the two algorithms indeed exhibit a certain duality with respect to the number of groups. However with two simple, commonly known optimizations, the respective drawbacks of both algorithms can be removed. The two approaches have then exactly the same costs in terms of cache line transfers, matching the lower bound of MULTISETSORTING in the common case.

As a second argument for the similarity of the two approaches, we design an algorithmic framework that allows seamless switching between hashing and sorting during execution. It is based on the observation that hashing is in fact equivalent to *sorting by hash value*. Since hashing is a special form of sorting, intermediate results of a hashing routine can be further processed by a sorting routine and vice versa. This allows us to apply state-of-the-art optimizations to both routines separately, but still combine them to benefit from their respective advantages. Furthermore is sorting by hash value an *easy* instance of sorting, since hashing makes the key domain dense and eliminates value skew. We also discuss how to apply this framework in the context of column-store database systems and just-in-time compiled query plans, which are two most commonly used architectures for analytical workloads.

Furthermore we show how to achieve very low constant factors for both the hashing and the sorting routine by tuning them to modern hardware, thus making our analysis in the external memory model meaningful. The main techniques are wait-free parallelization, enabling super scalar instruction execution, and careful cache management. This is more important on modern in-memory database systems than on traditional disk-based systems, since much fewer CPU instructions can be executed in the time of a cache line transfer than in the time needed for loading a page from disk. With our careful tuning, however, we are able to leave the memory access costs as the main remaining performance bottleneck.

Despite all tuning, there is an intrinsic reason why hashing and sorting have complementary performance vis-à-vis the locality of keys in the input: Hashing allows for *early aggregation* while sorting does not. If several rows of the same group occur close to each other, hashing aggregates them immediately to a single row. By reducing very early and possibly by large factors the amount of subsequent work, hashing is much faster than sorting in this case. In the other case however, i.e., in case of an input with few repeating keys, the additional effort of *trying* to aggregate is in vain, so regular sorting without early aggregation is faster here. Our framework can exploit this complementarity by effectively detecting locality during execution and switching to the faster routine when appropriate, thus putting the insights of the theoretical analysis into practice.

Putting everything together, we obtain a novel relational aggregation algorithm that is optimal in terms of memory access complexity—independently and without prior knowledge of input skew and output cardinality—and has very low constant factors on modern hardware. It is cache-efficient, highly parallelizable on modern multi-core systems, and operating at a speed close to the memory bandwidth. With our single, robust aggregation operator, the possibly error-prone decision of the optimizer before the query execution is eliminated. We present extensive experiments on a variety of data sets comparing our implementation to the state-of-the-art, which we are able to outperform by up to factor 3.7.

The rest of the paper is organized as follows: In Section 2 we discuss the complexity of textbook algorithms for external aggregation, followed by a presentation of our algorithmic framework in Section 3. We continue with tuning our routines to modern hardware in Section 4 and show how their advantages can be combined in Section 5. The resulting AGGREGATION operator is evaluated in Section 6. Finally, we compare our contributions to prior work in Section 7, and make some concluding remarks in Section 8.

2. ANALYSIS OF EXTERNAL AGGREGATION

In this section we analyse the text book aggregation algorithms briefly mentioned in Section 1 in terms of the number of cache line transfers they incur. We show that with two simple optimizations, SORTAGGREGATION and HASHAGGREGATION have in fact the same costs. This insight will guide the design of our algorithm in the later sections.

2.1 Analysis of Sort-Based Aggregation

We do the analysis in the *external memory* model [1] because this model yields very general results that are applicable for both cache line transfers and I/Os. The external memory model has the following parameters for the input data and the cache:

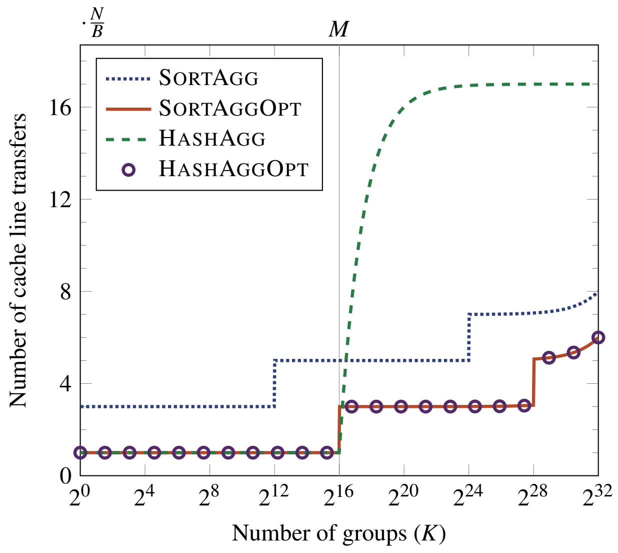

Figure 1: Comparison of aggregation algorithms in the External Memory model for $N = 2^{32}$, $M = 2^{16}$, and $B = 16$.

$$\begin{aligned} N &= \text{number of input rows} \\ K &= \text{number of groups in the input} \\ M &= \text{number of rows fitting into cache} \\ B &= \text{number of rows per single cache line} \end{aligned}$$

Note that the output will be of size K. The costs of an algorithm are the number of cache line transfers in the worst case—computations and access to the cache are free.

When applied to CPU caches, the external memory model does not allow to make as precise performance predictions as it does for the disk-based setup, for which it was originally conceived. It is still widely adopted [18, 15, 6, 41, 5] as a formal method to understand and minimize memory access costs of algorithms and data structures. In this case the implicit assumption holds that CPU costs can be reduced or hidden in large parts, for example by making out-of-cache memory access sequential, eliminating branches, and using out-of-order execution or vectorization. We will show in Section 4 how to achieve this with the algorithms we propose.

We start with the analysis of SORTAGGREGATION. We use bucket sort because the analysis is simple, the result is valid for any other cache-efficient sort algorithm and because it can be easily turned into a state-of-the-art radix sort, the fastest known type of sort algorithms for dense domains [45, 25, 36]. Bucket sort recursively partitions the input into buckets until the data is sorted. Then a final pass over the data aggregates the rows of the same group, which reside in consecutive memory locations in the sorted input.

We use the tree representing the recursive calls of the algorithm for the analysis of SORTAGGREGATION, which we develop in three iterations. The first, simple iteration of the analysis works as follows: Since we can sort each cache line for free before we write it, the recursion stops when all partitions have size B, so there are as many leaves in the call tree as there are cache lines in the input: $\frac{N}{B}$. Furthermore the tree has degree $\frac{M}{B}$ since the number of partitions is limited by the number of buffers that fit into cache. This means that if we assume that the tree is somewhat balanced, it has a height of $\left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$. Since the input is read and written once per level of the

Final edited form was published in "SIGMOD/PODS'15: International Conference on Management of Data. Melbourne 2015", S. 1123–1136, ISBN 978-1-4503-2758-9
https://doi.org/10.1145/2723372.2747644

tree and the subsequent aggregation pass reads the input ($\frac{N}{B}$) and writes the output once ($\frac{K}{B}$), the overall costs of SORTAGGREGATION are roughly:

$$\textsc{SortAggStat}(N,K) = 2 \cdot \frac{N}{B} \cdot \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil + \frac{N}{B} + \frac{K}{B}$$

The analysis is slightly simplified because it assumes a static depth of the call tree independently of K. In a second iteration, we can make the analysis more precise by taking into account the fact that the keys form a multiset in the cases where $K < N$. In this case the recursion actually stops earlier than for the case where $K = N$. In fact, the call tree only has $\min(\frac{N}{B}, K)$ leaves, at most one for each partition, so SORTAGGREGATION needs the following number of cache line transfers:

$$\textsc{SortAgg}(N,K) = 2 \cdot \frac{N}{B} \cdot \left\lceil \log_{\frac{M}{B}} \left(\min\left(\frac{N}{B}, K\right)\right) \right\rceil + \frac{N}{B} + \frac{K}{B}$$

It is known that this is a lower bound for multiset sorting [33], i.e., no sort algorithm can do asymptotically less cache line transfers in the general case. Other than bucket sort, also distribution sort, sample sort, quicksort, and radix sort achieve this bound, with an analysis very similar to the above, first shown by Aggarwal and Vitter [1], as well as variants of heap sort [43] and sorting with buffer trees [4].

Figure 1 plots the number of cache line transfers as function of K for $N = 2^{32}$, $M = 2^{16}$, and $B = 16$, which are typical values for modern CPU caches. For small K, SORTAGGREGATION needs one pass for sorting and one pass for aggregating. For larger K, the number of passes only increases logarithmically. Due to the large base and the rounding, the logarithm has only values $\in \{1,2,3\}$ in our plot (corresponding to the steps) and is never larger than four in most realistic settings. Only for very large K, where K gets close to N, the $\frac{K}{B}$ cache line transfers for writing the output become noticeable.

In the third iteration of the analysis, we make a small modification to SORTAGGREGATION: we merge the last bucket sort pass with the final aggregation pass, i.e., instead of writing a cache-line to memory when the buffer of a partition runs full, we aggregate the elements of this cache-line to make space. Since there are few enough groups left in the last pass, this produces the final result of the current bucket in cache and thus completely eliminates one pass over the entire data. Furthermore it allows us to hold a factor B more partitions (M instead of $\frac{M}{B}$), so there are now only $\frac{K}{B}$ leaves in the call tree of the algorithm. This optimization requires that intermediate results are of size $O(1)$, which is true for *distributive* and *algebraic* aggregation functions [22], including the most common ones like `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`, but not for `MEDIAN`. With this analysis and a small reorganization of the formula, the number of cache line transfers made by SORTAGGREGATIONOPTIMIZED is the following:

$$\textsc{SortAggOpt}(N,K) = \frac{N}{B} + 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{K}{B}$$

The first and the last term correspond to reading the input and writing the output respectively. The second term corresponds to writing intermediate results and reading them again in the next level of recursion.

Figure 1 plots the costs of SORTAGGREGATIONOPTIMIZED. It shows that the optimization eliminates an entire pass and slightly delays the necessity of an additional pass due to better cache usage in the last pass. In particular, for $K < M$, the algorithm just reads the data once and calculates the result in cache.

2.2 Analysis of Hash-Based Aggregation

We now analyze the number of cache line transfers that HASHAGGREGATION needs. Apart from the $\frac{K}{B}$ cache lines for writing the result, the algorithm just needs the $\frac{N}{B}$ cache line transfers for reading the input—as long as the resulting hash table fits into the cache, i.e., $K < M$, and assuming that intermediate aggregates can be saved in a state of size $O(1)$ like above. In the other case, if $K > M$, even with a perfect cache and without hash collisions, only a fraction of $\frac{M}{K}$ rows can be in the cache at the same time, so every access to one of the other rows produces a cache miss ($= 1\,\text{write} + 1\,\text{read}$). The overall number of cache line transfers is therefore:

$$\textsc{HashAgg}(N,K) = \frac{N}{B} + \begin{cases} \frac{K}{B} & \text{if } K < M \\ 2 \cdot \left(1 - \frac{M}{K}\right) \cdot N & \text{otherwise} \end{cases}$$

Figure 1 shows the costs of HASHAGGREGATION: As long as the output K is small enough to fit into the cache, HASHAGGREGATION is really fast. However as soon as the cache cannot hold the output anymore, HASHAGGREGATION triggers a cache miss for almost every input row, so the number of cache line transfers explodes.

A common optimization to overcome this problem is to (recursively) partition the input by hash value and to apply HASHAGGREGATION on each partition separately. Since each partition contains only a part of the groups, i.e., since K is reduced, this makes the algorithm work in cache. However the partitioning also entails costs, which are the same as the partitioning of bucket sort. Consequently the analysis works the same way as the one for SORTAGGREGATIONOPTIMIZED: $\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1$ partitioning passes plus the reading ($\frac{N}{B}$) and writing ($\frac{K}{B}$) of the HASHAGGREGATION pass of the partitions. In total HASHAGGREGATIONOPTIMIZED needs the following number of cache line transfers:

$$\textsc{HashAggOpt}(N,K) = 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{N}{B} + \frac{K}{B}$$

Figure 1 shows that HASHAGGREGATIONOPTIMIZED has the same cost in terms of number of cache line transfers as the optimized sort-based aggregation above.

2.3 Conclusions from the Analysis

Our analysis shows that SORTAGGREGATION and HASHAGGREGATION have indeed a complementary behavior if implemented naively: HASHAGGREGATION performs better when the number of groups is small, while SORTAGGREGATION is more efficient in the other case. However the respective drawback of both algorithms can be removed if each of them includes a simple, well-known optimization: doing the aggregation pass together with the last sorting pass and recursive partitioning as preprocessing respectively. This suggests that—at least on this level of abstraction—there is no such thing as a duality between hashing and sorting. In terms of cache line transfers, the two approaches are actually the same. Note that since bucket sort matches the lower bound for sorting in the common case, this reasoning is valid for any other optimal sort algorithm as well.

Since no algorithm is known to date that requires fewer cache line transfers than the algorithms presented above, one might even wonder whether this is an intrinsic property of aggregation itself rather than being a property of specific algorithms, i.e., whether there is a lower bound on the cache line transfers needed to compute an aggregation query. We conjecture that there is indeed such a bound and that it matches the bound of multiset sorting for the most interesting parameter ranges, but leave the proof open for future research.

Final edited form was published in "SIGMOD/PODS'15: International Conference on Management of Data. Melbourne 2015", S. 1123–1136, ISBN 978-1-4503-2758-9
https://doi.org/10.1145/2723372.2747644

Algorithm 1 Algorithmic Building Blocks

```
1: func PARTITIONING(run: Seq. of Row, level)
2:     for each row in run do
3:         $R_h \leftarrow R_h \cup$ row with $h =$ HASH(row.key, level)
4:     return $(R_1, \dots, R_F)$
5: func HASHING(run: Seq. of Row, level)
6:     for each row in run do
7:         table.INSERTORAGGREGATE(row.key, row, level)
8:         if table.ISFULL() then
9:             tables ← tables ∪ table ; table.RESET(())
10:    return $(R_1, \dots, R_F)$ with $R_i \leftarrow \bigcup_{t \in \text{tables}}$ GETRANGE(t,$i$)
```

Algorithm 2 Aggregation Framework

```
1: AGGREGATE(SPLITINTORUNS(input), 0)                ▷ initial call
2: func AGGREGATE(input: Seq. of Seq. of Row, level)
3:     if |input| == 1 and ISAGGREGATED(input[0]) then
4:         return input[0]
5:     for each run at index $j$ in input do
6:         PRODUCERUNS ← HASHINGORPARTITIONING()
7:         $R_{j,1}, \dots, R_{j,F} \leftarrow$ PRODUCERUNS(run, level)
8:     return $\bigcup_{i=1}^{F}$ AGGREGATE($\bigcup_j R_{j,i}$, level + 1)
```

The insights of this analysis shall be the design principles of the following sections. We show how to engineer a single aggregation algorithm similar to the optimized versions of both SORTAGGREGATION and HASHAGGREGATION that has excellent practical performance characteristics independently of the value of K.

3. ALGORITHMIC FRAMEWORK

In this section we present an algorithmic framework for an AGGREGATION operator putting the theoretical insights of the previous section into practice. The main idea is to design the algorithm like an integer sort algorithm on the dense hash values with hashing as a special case used for early aggregation. Furthermore we show how to achieve wait-free parallelization of the algorithm. Finally we also discuss how to fit the framework into popular processing models of modern database systems.

3.1 Mixing Hashing and Sorting

As presented in the previous section, sort-based and hash-based aggregation have the same complexity in the external memory model. This is very intuitive considering the fact that the two algorithms have the same high-level structure: they recursively partition the input—either by the keys of the groups or by their hash values—until there are few enough groups left to process each partition in cache. However the two approaches are even more similar: the process of building up a hash table also partitions the input by hash value.

Consequently we can define the following two partitioning routines, which will be the main building blocks of our framework and which are shown in Algorithm 1: plain partitioning by hash value called PARTITIONING (Line 1) and a partitioning routine based on the creation of hash tables called HASHING (Line 5). Both routines produce partitions in form of "runs"[1]: PARTITIONING produces one run per partition by moving every row to its respective run. HASHING starts with a first hash table of the size of the cache and replaces its current hash table with a new one whenever it is full. Every full hash table is split into one run per partition—merely a logical operation since the hash values of one hash partition are stored in a consecutive range in the hash table.

Note that the working set of both HASHING and PARTITIONING is strictly limited to the CPU cache. The working set of PARTITIONING is limited through its partitioning fan-out, while HASHING has a limited working set because the hash table size is fixed to the size of the cache.

We can now combine these two building blocks into a recursive algorithm similar to both SORTAGGREGATIONOPTIMIZED and HASHAGGREGATIONOPTIMIZED. The algorithm is shown in Algorithm 2: The input is first split into runs. Then each run of the input is processed by one of the two routines selected by HASHINGORPARTITIONING (line 6), which produces runs for the different buckets. Once the entire input has been processed, the algorithm treats all runs of the same partition as a single bucket and recurses into the buckets one after each other. With every step of the recursive partitioning, more and more hash digits are in common within a bucket, thus reducing the number of groups per bucket more and more. The recursion stops when there is a single run left for each bucket and in that run, all rows with of same group have been aggregated to a single output row.

Using the hash values as partition criterion has the advantage that it solves the problem of balancing the number of groups in the buckets, which was an assumption of the analysis in Section 2. The hash function distributes the groups uniformly—it makes the key domain dense and hence the partitioning easier.

The way we use HASHING has also the advantage that it enables *early aggregation* [19, 27]. In contrast to the two illustrative algorithms from the previous section, we can now aggregate in *all* passes, not just the last. Since HASHING can aggregate rows from the same group, the resulting hash tables and hence the output runs are potentially much smaller than the input run, thus reducing the amount of data for subsequent passes by possibly large factors. It is important to understand that this does not change the number of cache lines needed in the worst case, but it is very beneficial in case of locality of the groups. So one might even wonder why we do not used HASHING all the time, similarly to recent work on disk-based systems [23]. As we show in the next section, on modern hardware, PARTITIONING can be tuned to a 4 times higher throughput than HASHING, which makes it the better routine in cases where early aggregation is not helpful.

The fact that our framework supports HASHING and PARTITIONING interchangeably not only makes the similarity between the two approaches obvious but also gives it the power to switch to the better routine where appropriate: In presence of locality or clusters, our framework can use HASHING, whose early aggregation reduces the amount of work for later passes. In absence of locality, we can switch to the faster PARTITIONING instead. The switching can happen spontaneously during runtime, without loosing work accomplished so far, and without coordination or planning. Algorithm 2 still uses HASHINGORPARTITIONING as a black box, but in Section 5, we discuss several switching strategies.

We argue that the design of our framework resembles that of a sort algorithm. While the similarity to bucket sort from the previous section is obvious, our framework has similarities with other sort algorithms as well: Since the bucket of an element is determined by some bits of a hash function, our algorithm is in fact a radix sort rather than a bucket sort. Thus hashing turns AGGREGATION into an instance of *integer sorting*, which is easier than sorting in general [45, 25, 36]. Furthermore it repeatedly merges intermediate

[1]We consciously use the term "run" from the disk-based era, which was commonly used to denote temporary files for intermediate processing results on disk [19].

runs from different parts of the input, so in a way the algorithm is also similar to merge sort. Finally, hashing can be seen as a variant of insertion sort, which is commonly used for the leaves of recursive sort algorithms [14].

An interesting interpretation is the following: The concatenation of the final runs of our algorithm is a hash table like HASHAGGREGATION would produce but it is built with a sorting algorithm—which is actually much faster. This suggests that the optimal way to do hashing *is* sorting.

Finally the fact that we mix hashing and sorting requires some technical attention: The buckets may contain rows that were just copied from the input, but also rows that are already aggregates of several rows from the input. In order to aggregate two aggregated values, one needs to use the so-called super-aggregate function [22], which is not always the same as the function that aggregates two values from the input. For example the super-aggregate function of `COUNT` is `SUM`. However it is easy to keep some meta-information associated with the intermediate runs indicating which aggregation function should be used.

3.2 Parallelization

Apart from cache-efficiency, the design of our framework is also influenced by the requirement for intra-operator parallelism of large analytical systems. It allows for full parallelization of all phases of the algorithm: First, the main loop that partitions the input in Line 5 of Algorithm 2 can be executed in parallel without further synchronization since neither input nor output are shared among the threads. Second, the recursive calls on the different buckets in Line 8 can also be done in parallel. Only the management of the runs between the recursive calls (the $\bigcup$-operations in our pseudocode) require synchronization, but this happens infrequently enough to be negligible.

We use user level scheduling to balance the two axes of parallelism as follows: we always create parallel tasks for the recursive calls, which are completely independent of each other, while we use work-stealing to parallelize the loop over the input. It is important to see that the latter form of parallelization implies that several runs per bucket are produced (at least one per thread), which in turn implies another level of recursion. By using work-stealing, our framework limits the creation of additional work to situations where no other form of parallelism is available. In particular, parallelizing the main loop is needed in the following two cases: First it is the only way to parallelize the initial call of the algorithm and second it allows for full parallelization even in presence of heavy skew: the buckets after the first call can be of arbitrarily different sizes because even an ideal hash function only distributes the *groups* evenly over the buckets, but does not affect the distribution of *rows* into these groups (which is given by the input). With work-stealing in the main loop however, our framework can schedule the threads to help with the large buckets once they have finished their own recursive call.

3.3 System Integration

We now want to discuss how to integrate our operator into the two prevailing processing models of analytical in-memory database systems, column-wise processing and just-in-time compiled (JiT) query plans.

Column-wise processing is the traditional processing model of column store database systems. It imposes some specific requirements for the implementation of aggregation operators. While our framework fulfills these requirements, this is not true for all aggregation algorithms proposed in the past. In the column store architecture, the question arises *when* and *how* the aggregate columns should be processed with respect to the grouping columns. The question has been discussed in the literature for the column store architecture in general [10, 11, 31], but there are some additional aspects specific to aggregation.

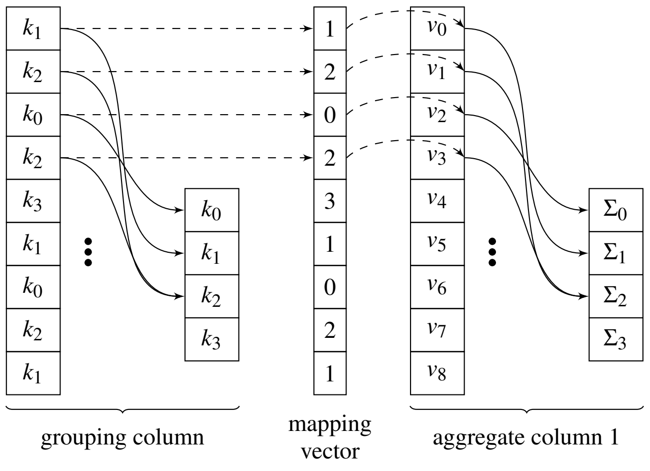

Figure 2: Column-wise processing.

One possibility for column-wise processing is to process all columns at the same time, similarly to a row store. An aggregation operator would read the values of a single row from one column after another, compute the aggregates, and then store them in their respective result columns one after each other. This approach is known to have the disadvantage that the data is not processed in tight loops [11], which results in considerable performance deterioration on modern hardware. Furthermore it effectively decreases the size of the cache during aggregation: Since all relevant attributes of a row together are larger than than just a single attribute, less elements fit into cache, so an aggregation operator needs more passes for cache-efficient processing.

Another possibility is to do the processing one column at a time, like it is done for example in MonetDB [10, 31]. With this approach, aggregation is split into two operators as illustrated by Figure 2: The first operator processes the grouping column and produces a vector with identifiers of the groups and a mapping vector, which maps every input row to the index of its group. The second operator applies this mapping vector by aggregating every input value with the current aggregate of the group as indicated by the mapping vector and is executed once for each aggregate column. This approach is known to have the disadvantage to require additional memory access to write and read the mapping vector. Furthermore it ignores the insights of our analysis for cache-efficient aggregation: if we aggregate the input values directly to their group in the output column, we get the same sub-optimal memory access pattern as HASHAGGREGATION, producing close to a cache miss for every input row for large outputs. Since there are often many more aggregate columns than grouping columns, this would even have a worse impact on performance than inefficient processing of the keys.

The state-of-the-art column-wise processing model was introduced in MonetDB/X100 [11] and combines the advantages of the above two possibilities by interleaving the processing of the different columns in blocks of the size of the cache. Applied to aggregation, this allows to process the columns in tight loops without materialization of the mapping vector to memory. However we also need to adopt this model *inside* the aggregation operator to make it compatible with our recursive run production. Consequently our framework operates as follows if used for column-wise processing:

While producing a run of the grouping column, both HASHING and PARTITIONING produce a mapping vector as depicted in Figure 2, but *only for this run*. This mapping is then applied to the corresponding parts of the aggregate columns. When the corresponding runs of all columns have been produced, the framework continues with the processing of the rest of the input.

One important aspect of this discussion is that not all techniques for aggregation proposed in prior work are compatible with column-wise processing. Our two routines, hashing and partitioning, move every input element directly to its final location, so it is easy to create the required mapping vector. However it is unclear how to fit some other routines into this scheme, for example vectorized sorting networks recently used for joins [7] or software-caching used for cache-efficient aggregation in row stores [13].

Recent work [35, 17, 34] promises to replace column-wise processing by a processing model based on just-in-time compilation (JiT). In this model, each pipeline of the execution plan of a query is compiled into a fragment of machine code just before execution, thus enabling processing in tight loops without decoupling the processing of different columns. It is straightforward to fit our framework into this model: The main part of the operator, i.e., the initial call to the AGGREGATE function of Algorithm 2, is compiled into the pipeline fragment including (and ending with) the aggregation operator. When this pipeline ends, all data resides in intermediate runs in the first level of the buckets. For the recursive function calls, a second code fragment is compiled, which only contains the code to process the buckets further. Both fragments contain the code path of both the PARTITIONING and the HASHING routine from Algorithm 1. Since the runtime decision between the two paths is the same for the entire run, it is easy to predict by the hardware and does hence not hurt performance.

4. MINIMIZING COMPUTATIONS

In this section we study the details of the two routines used in our algorithm: hashing and partitioning. The goal is to bring the behaviour of our implementation on real hardware as close as possible to the idealized external memory machine model. Our analysis and the algorithmic framework built upon it are only relevant if we turn the movement of the data into the dominant part of the execution time by reducing the computational overhead to a minimum (i.e., ideally by making it "free"). We show that this is much more difficult in the cache setting of modern hardware than it used to be in the disk setting, since the gap between fast and slow memory is much narrower and small differences in the implementation can change the performance by factors.

The microbenchmarks that we run in this section are run in the same experimental setup as the experiments in Section 6.

4.1 Minimizing CPU Costs of Hashing

We conducted a performance comparison of several hash table implementations. It turned out that the simplest approach has the lowest CPU overhead: a single level hash table with linear probing, similar to the state-of-the-art `dense_hash_map` of Google[2]. We fix the hash table to the size of the L3 cache and consider it full at a very low fill rate of 25%. With this configuration, collisions are very rare or even non-existing with high probability if the number of groups is more than two orders of magnitude smaller than the cache, so no CPU cycles are lost for collision resolution. The apparent waste of memory is in fact negligible because it is limited to one or very few hash tables per thread in our final algorithm presented in the next section. Interestingly this is the opposite of what Barber et

[2]https://code.google.com/p/sparsehash/

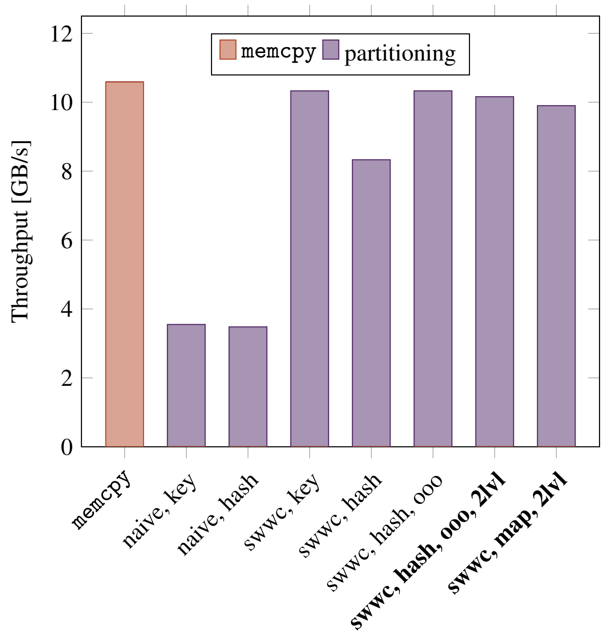

Figure 3: Microbenchmark of degenerated partitioning routines.

al. [8] recently proposed for JOINs, where denser storage increased performance. We tried out many different hash functions that are popular among practitioners and found that for small elements, MurmurHash2[3] is the fastest. On a technical note, we adapted the linear probing to work within blocks, such that we can cleanly split a table into ranges for the recursive calls. The final insertion costs of our implementation are below 6 ns per element. This is roughly 4 times more than an L1 cache access, but more than an order of magnitude faster than out-of-cache insertion, where CPU costs are dwarfed by the costs of cache misses and our reasoning in the external memory model is meaningful.

4.2 Minimizing CPU Costs of Partitioning

To minimize the CPU costs of the partitioning routine, we use a technique known from integer sorting for dense domains. A large body of prior work suggests that this kind of sort algorithms is faster than comparison based sorting [45, 25, 36], except special cases such as sorting almost sorted data [12]. With partitioning, integer sorting is made branch- and even comparison-free, which eliminates most CPU costs of comparison-based sort algorithm.

The key idea is to use a technique called "software write-combining", first described by Intel [24] and used by various other authors [7, 38, 42]. Software write-combining is designed to avoid the *read-before-write* overhead and to reduce the number of TLB misses inherent in partitioning [30], which writes to a high number of memory pages. It consists in buffering one cache line per partition, which is flushed when it runs full using a non-temporal store instruction that by-passes the cache. This scheme works best with 256 partitions, so we use this number to split runs into ranges in our framework. Since the final size of the partitions is unknown before processing, most authors start with a counting pass to determine output positions. Wassenberg et al. [42] eliminate this pass using a trick with virtual memory to over-allocate every partition so that it can hold the entire input. This is not possible with the memory

[3]https://code.google.com/p/smhasher/

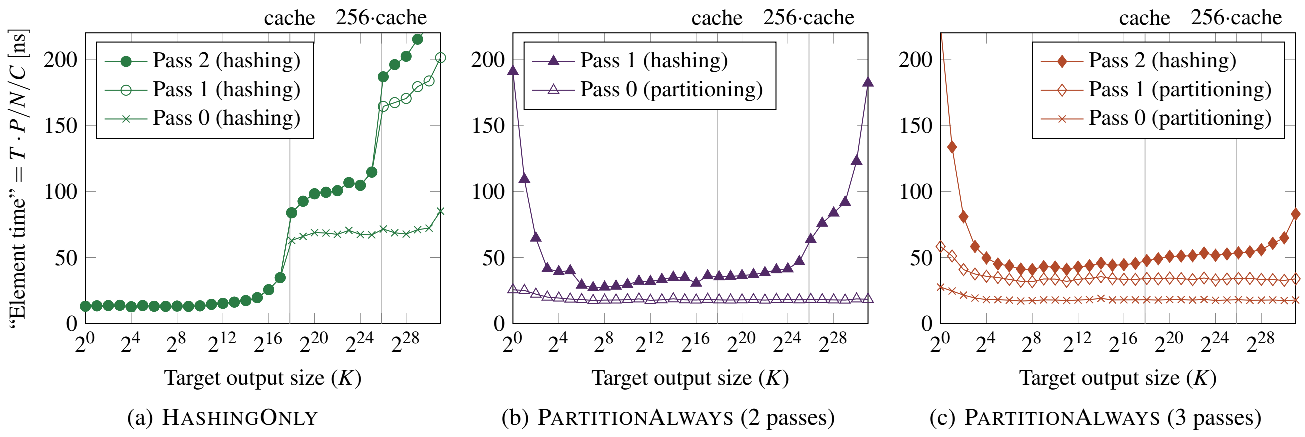

Figure 4: Breakdown of passes of illustrative aggregation strategies using $P = 20$ threads.

management of industry-grade database systems, but using a two-level data structure, a list of arrays, has the same benefit and only very low overhead, as we show below. We use this technique for processing not only for the grouping column but also for the aggregate columns as discussed in Section 3.3, since their memory access pattern is equivalent.

We conducted a series of microbenchmarks to measure the impact of our optimizations. Figure 3 shows the (payload) bandwidth of different versions of the partitioning routine on uniformly distributed random data. The first bar shows the bandwidth of a self-implemented `memcpy` using non-temporal store instructions as a reference. The next two bars show the throughput of a naive partitioning scheme, once partitioning according to some bits of the keys themselves (called *key*) once partitioning according to bits of the hash function (called *hash*). The difference between the two is only small as the throughput is mainly limited by the inherent TLB misses mentioned before. The next two bars show how software write-combining considerably improves performance (called *swwc*): The *key*-variant is 2.9 times faster than the naive counterpart. The variant partitioning by hash value seems to suffer from the computational overhead though. However we can benefit from out-of-order execution by manually unrolling the main loop into blocks of 16 elements, which are first all hashed and then all put into their partition buffers. The next bar (denoted *ooo*) shows that we gain 24% throughput with this optimization, thus achieving a 3.0 times higher throughput than the naive partitioning routine. Finally we replace over-allocated output partitions by the two-level data structure, which lowers performance by roughly 2%. This final routine runs at 97% of the bandwidth of our `memcpy`. The last bar shows the bandwidth of applying the mapping vector to an aggregate column using software write-combining and our two-level data structure (denoted *map*). Since reading the mapping vector adds memory traffic that we do not count as application bandwidth, the bandwidth here is slightly lower than that of the previous variant, namely 93% of our measured memory bandwidth.

To sum things up, like in the case of hashing, partitioning of both grouping and aggregate columns can be tuned to modern hardware such that the inevitable movement of the data remains the dominant part of the processing time.

5. ADAPTATION TO LOCALITY

In the previous sections, we describe an algorithmic framework for designing an aggregation operator similar to a sort algorithm and how to reduce the CPU costs of two possible subroutines. In this section, we answer the remaining question of when to select which of the two.

To that aim we present a series of experiments with naive strategies for selection of one of the two routines that illustrate their respective performance characteristics. Figure 4 shows the results. In HashingOnly the only subroutine used is Hashing (Figure 4(a)), whereas with PartitionAlways, the input is always preprocessed by one or two passes of Partitioning before a final Hashing pass (Figure 4(b) and 4(c)). To keep our implementation simple, we only allow a single Hashing pass by exceptionally letting its hash tables grow larger than the cache. This prevents full parallelization for very small K and cache misses for very large K, but these effects do not occur in our final algorithm. The experiments are run on uniformly distributed data.

The first observation that can be made in this experiment is the fact that HashingOnly automatically does the right number of passes: If $K <$ cache, it computes the result in cache. The subsequent merging of the runs of the different threads is insignificant due to their small size and thus not visible in the plot. Once $K >$ cache, Hashing recursively partitions the input until the result is computed in cache and the recursion stops automatically. For PartitionAlways this is not the case. Since it does not aggregate during partitioning, it can only be used as preprocessing and external knowledge is necessary to find the right depth of recursion before the final Hashing pass.

The second observation is that Partitioning is *much* faster than Hashing if $K >$ cache, i.e., if the latter produces more than one run (by more than factor 4 in our experiments). In this case Hashing suffers from its non-sequential memory access and wasted space and hence wasted memory transfers intrinsic to hash tables. Furthermore, as discussed in Section 2, as soon as there are only slightly more groups than fit into one hash table, chances are very low to find two elements with the same key, so the amount of data is not reduced significantly. In contrast Partitioning achieves a high throughput independently of K thanks to the tuning from the previous section.

In the case of uniformly distributed data, the best strategy is obvious: use Partitioning until the number of groups per partition is small enough such that Hashing can do the rest of the work in cache. However it is not clear how to find out when this is the case if K is not known. Furthermore the best strategy is less obvious with other distributions: consider a clustered distribution with

Final edited form was published in "SIGMOD/PODS'15: International Conference on Management of Data. Melbourne 2015", S. 1123–1136, ISBN 978-1-4503-2758-9
https://doi.org/10.1145/2723372.2747644

a high locality where each key mostly occurs in one narrow region of the input. HASHING is then able to reduce the amount of data significantly *although* the entire partition has more groups than fit into cache. Hence HASHING can be the better choice even before the last pass if the ratio of input data size to output data size high enough.

This leads us to defining an ADAPTIVE strategy. The algorithm starts with HASHING. When a hash table gets full, the algorithm determines the factor $\alpha := \frac{n_{in}}{n_{out}}$ by which the input (run) has been reduced, where n_{in} is the number of processed rows and n_{out} the size of the hash table. If $\alpha > \alpha_0$ for some threshold α_0, HASHING was the better choice as the input was reduced significantly, so the algorithm continues with HASHING. Otherwise it switches to PARTITIONING. The parameter α_0 balances the performance penalty of HASHING compared to PARTITIONING with its benefit of reducing the work of later passes. We show how we determine this machine constant in Appendix A.1. When enough data was processed with the faster PARTITIONING routine such that the overhead of the (inadequate) HASHING is amortized, i.e., when $n_{in} = c \cdot \text{cache}$ for some c, the algorithm switches back to its initial mode in case the distribution has changed. In experiments shown in Appendix A.2, we found $c = 10$ to be a good compromise between amortization effect and reactivity to distribution changes.

Figure 5 shows the performance of ADAPTIVE compared to the illustrative strategies from Figure 4. It shows that ADAPTIVE automatically partitions the input using PARTITIONING until HASHING can process each partition in cache—without knowing K in advance. Consequently its performance corresponds piecewise to the best of the other strategies. If $K <$ cache (or $K/256^i <$ cache for pass i), each thread only works with a single hash table, which never runs full. If however the input does not fit into one hash table, input is partitioned with the faster PARTITIONING routine first. The fact that PARTITIONING is interleaved with occasional HASHING to check whether the distribution has changed has only low overhead and is barely noticeable for $K < 256 \cdot$ cache. In the following section, we show that ADAPTIVE not only works well on uniform data, but has a robust performance on many distributions.

We see the main advantage of our approach in the fact that it combines the respective advantages of two complementary routines by switching between them based on a simple, local criterion. No completed work is ever thrown away; no extra work or preprocessing is necessary; no potentially unprecise information from the optimizer is needed; no synchronization is needed among the threads. In fact the different threads do not even need to take the same decision: they can benefit from changing locality or clusteredness in the input by aggregating where the locality is high and partitioning first where it is low.

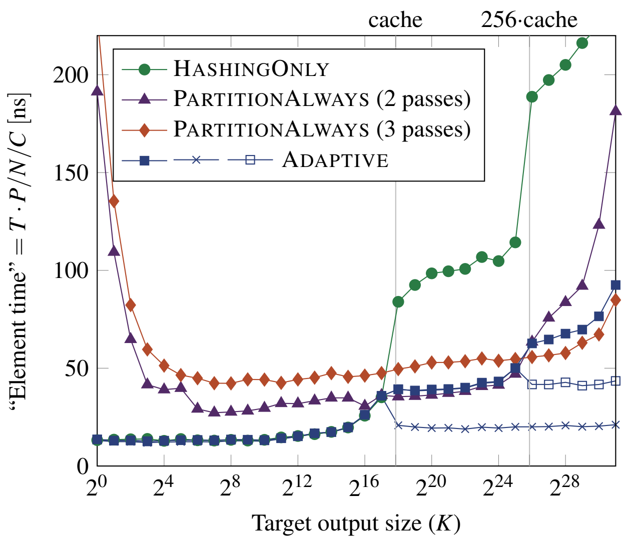

Figure 5: ADAPTIVE strategy in comparison with HASHINGONLY and PARTITIONALWAYS (2 and 3 passes) using $P = 20$ threads.

6. EVALUATION

In this section we evaluate the effectiveness of our algorithm design, assess the quality of our implementation, and compare the performance of our operator with previous work.

We implemented our algorithm for column-wise processing, but argue that the experiments have a certain validity for the JiT processing model as well: Where not otherwise mentioned, the experiments are run just on the grouping column, so the inner loops of both processing models are exactly equivalent.

6.1 Test Setup

We run the experiments on two Intel Xeon E7-8870 CPU[4] with 256 GB of main memory. They run at 2.4 GHz and have 10 cores each. Each core has 64 kB of private L1 cache, 256 kB of private L2 cache, and access to a shared 30 MB on-chip L3 cache (3 MB per core). The TLB of the CPUs have two levels, the first of which have 64 entries for data and 128 for instructions and the second 512 entries for both combined. The operating system is SLES 11.3 with Linux kernel 3.0.101 for x86_64. We use GCC 4.8.3 as compiler using `-O3 -march=native` optimizations.

Our data sets consist of $N = 2^{31}$ rows where all columns are 64-bit integers. If not otherwise mentioned we report run times as "Element Time" $= T \cdot P/N/C$, where T is the total run time, P the number of cores, and C the number of columns (grouping and aggregate columns combined). This metric represents the time each core spends to process one element and makes numbers of different configurations easily comparable—among themselves and to known machine constants such as the time of a cache miss.

All presented numbers are the median of 10 runs.

6.2 Scalability with the Number of Cores

We first assess the parallelization mechanisms of Section 3.2 and the quality of our implementation in terms of scalability with the number of cores. Figure 6 shows the speedup of ADAPTIVE for different numbers of groups K compared to its respective performance on a single core. As the plot shows, the speedup is around 16 on our 20 CPU cores no matter K, which is as close to optimal speedup as practical implementations usually get. Section 6.5 also shows the experiments with other distributions, where we found our algorithm to perform just as well. Finally we also ran this experiment with concurrent dummy threads on the idle cores in order to simulate a real system under load: If the dummy threads loop over their respective 3 MB of cache to keep the cache warm, the performance of our algorithm is not influenced since its threads only rely on their respective part of the cache. However if the dummy threads run an out-of-cache `memcpy`, the performance of our algorithm deteriorates by up to factor two, confirming that memory bandwidth is the main bottleneck our algorithm faces. The good scalability in all situations does not come to a surprise, since the threads of our algorithm do not share any resources and synchronize only at a very coarse granularity.

[4]http://ark.intel.com/products/53580

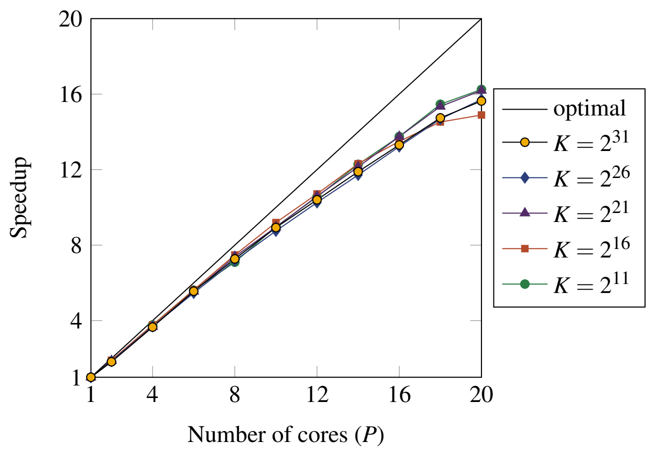

Figure 6: Speedup of ADAPTIVE compared to single core performance.

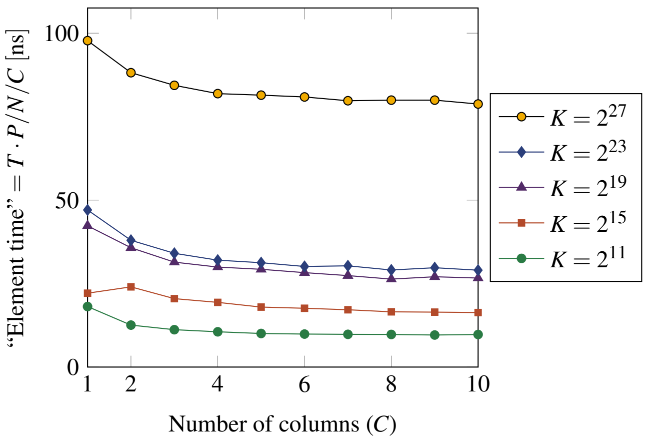

Figure 7: Scalability of ADAPTIVE with the number of columns using $P = 20$ threads.

6.3 Scalability with the Number of Columns

Figure 7 shows how the number of aggregate columns affects the performance of ADAPTIVE for different output cardinalities K. Just for this plot, we use $N = 2^{28}$ input elements to compensate the memory increase due to the additional columns. The experiment evaluates the effectiveness of the column-wise processing presented in Section 3.3, which is designed to process the different columns independently. Indeed the plot indicates that the run time per element is almost the same for any number of columns. As discussed in Section 4, the processing of the grouping column is a bit more expensive as the processing of the other columns because of the hashing and collision resolution, which explains the slightly higher costs per element with lower number of aggregates.

We also confirmed the scalability of our operator with the number of columns in experiments with other data distributions omitted here due to space constraints. Since the number of columns does not affect the processing cost per element, we run all other experiments only with a grouping column, i.e., without aggregate columns or $C = 1$.

6.4 Comparison with Prior Work

We now show an experimental analysis of several state of the art algorithms for in-memory aggregation from the work of Cieslewicz and Ross [13] and Ye et al. [46] and compare them to ADAPTIVE. Since their work targets the row store architecture (implicitly assuming JiT query compilation) while our implementation targets the column store architecture, we use a `DISTINCT` query with no aggregate columns ($C = 1$) for the comparison. In this type of query, the input and output data structure are equivalent in both architectures, and our algorithm does not need to produce a mapping vector for column-wise processing, so the experiments abstracts from all architectural differences.

We used the original implementations, but made the following modifications to tune them to this experiment: First we changed the minimal output data structure size to the size of the L3 cache, which effectively eliminates collision resolution for small K and consequently reduces the run time in these cases by up to 25%. Second we removed padding and redundant fields in the intermediate and output data structures, in order to reduce tuple size and hence memory traffic. This reduces the run time by roughly 20% for large K and even up by to 50% where the reduction in size makes the output just fit into cache. The padding originally improved the collision resolution in the high-throughput cases of small K, but our first modification improves theses cases even more. Third we replaced system mutexes by much smaller spin locks, again to reduce memory traffic, which also reduces the time by roughly 20% for the variants using them. Finally we replaced the multiplicative hashing by MurmurHash2, which we use in our algorithms as well. This has the same effect on the algorithms from prior work than on ours: a more predictable performance with up to 20% run time reduction due to less collisions, but noticeable overhead for small K. Furthermore we exceptionally provide ADAPTIVE with the output size, which is an information that all algorithms from the shown competitors rely on and which makes our algorithms somewhat faster due to implementation details (the benefit is only noticeable for large output cardinalities, i.e., if $K \approx N$, and always less than 10%).

Figure 8 shows a comparison of run times on data with uniform distribution. As we analyze in the following, all algorithms from prior work have an intrinsic limit in terms of K. They all consist of a fixed number of passes (either one or two) over the data, which means that they work well until a certain number of groups, but are penalized by a high number of cache misses beyond this limit. This is in line with our analysis of Section 2.

We describe the different algorithms and analyze their performance in more detail:

HYBRID (1 pass): Each thread aggregates its part of the input into a private hash table with a size fixed to its part of the shared L3 cache. When this table is full, old entries are evicted similarly to an LRU cache and inserted into a global, shared hash table. This becomes inefficient as soon as most of the output does not fit into the private tables, which happens for $K > 2^{16}$ (marked with L3 in Figure 8).

ATOMIC (1 pass): All threads work on a single, shared hash table protected by atomic instructions. This approach can suffer from contention due to concurrent updates as discussed by the original authors [13], but in the present `DISTINCT` query without aggregate columns, virtually no updates occur, so the problem is inexistent. It reaches its cache efficiency limit when the shared hash table exceeds the cache size, namely at around $K = 2^{19}$ (marked with ΣL3). This gives ATOMIC an advantage over all other algorithms for numbers of groups where it can fit its output into the *combined* cache while the shared-nothing approaches cannot.

INDEPENDENT (2 passes): In a first pass, every thread produces a hash table of his part of the input. In a second pass, the hash tables

Final edited form was published in "SIGMOD/PODS'15: International Conference on Management of Data. Melbourne 2015", S. 1123–1136, ISBN 978-1-4503-2758-9
https://doi.org/10.1145/2723372.2747644

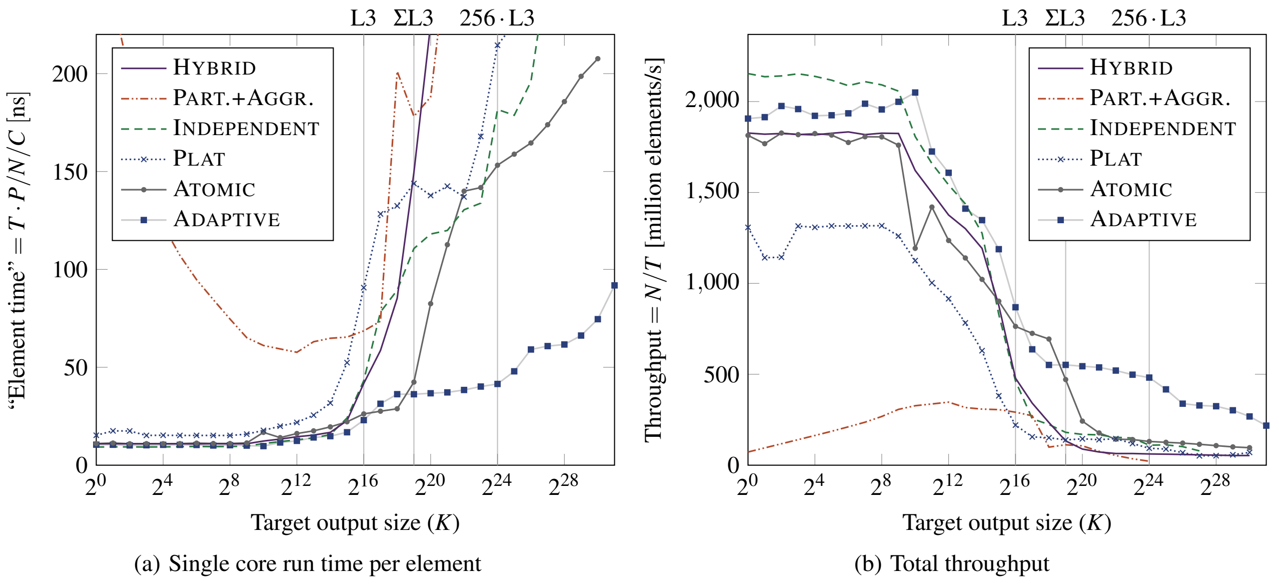

Figure 8: Comparison with prior work of Cieslewicz and Ross [13] and Ye et al. [46] using $P = 20$ threads.

are split and merged in parallel[5]. This makes the algorithm similar to HASHINGONLY with two passes, but since the hash tables of the first pass can be larger than the cache, both passes can trigger close to a cache miss per row. This limit is reached when the working set exceeds the L3 cache fraction *corresponding to each thread*, namely roughly at $K = 2^{16}$ for the first pass (marked with L3) and $K = 2^{24}$ for the second (marked with $256 \cdot$ L3).

PARTITION-AND-AGGREGATE (2 passes): Similarly to PARTITIONALWAYS with two passes, this algorithm first partitions the entire input by hash value and then merges each partition into its part of a hash table. Like our algorithm if limited to two passes as shown in Figure 4(b), this algorithm cannot do the merging in a cache-efficient manner if $K > 256 \cdot \text{cache} = 2^{24}$ ($256 \cdot$ L3 in the plot). Furthermore its partitioning uses the naive implementation as presented in Section 4 and is therefore slower than ours.

PLAT (Partition with Local Aggregation Table, 2 passes): Similarly to HYBRID, in this algorithm each thread aggregates into a private, fixed-size hash table. When it is full, new entries are overflown into hash partitions, which are merged in a subsequent pass like in the previous algorithm. This entails the same limit: the merging becomes inefficient if $K > 256 \cdot \text{cache} = 2^{24}$ (marked with $256 \cdot$ L3 in the plot). The partitioning in itself is also less efficient than ours, but in contrary to the previous algorithm, our optimization could not be applied here, since the private hash tables would destroy the explicit L1 cache management of software write-combining.

ADAPTIVE (variable number of passes): Our algorithm is the only algorithm that gracefully degrades with larger K thanks to the efficient additional passes. Compared to the fastest of the other algorithms, it achieves a speedup of at least factor 2.7 for all $K \leq 2^{21}$. The peak speedup factor of 3.7 is achieved at $K = 2^{24}$ where ADAPTIVE needs only 41 ns/element while ATOMIC needs 153 ns/element. Note that this speedup is higher than one usually hopes for for such a fundamental operator like aggregation, where improvements of several tens of percent are already worth some effort.

It is also worth noting that the second best algorithm for large values of K is actually the simplest in terms of cache management: While the cache-efficiency mechanisms of the other algorithms take extra time even though they do not work outside the range of K they were designed for, ATOMIC "just" pays single cache miss per row.

As Figure 8(b) shows, ADAPTIVE is also as least as fast as almost all other algorithm for other values of K[6]. The similarity of all algorithms for small K does not come as a surprise, since all hash-based algorithms do effectively exactly the same in these scenarios. It is interesting to see however that the throughput starts dropping slightly later for ADAPTIVE than for the other algorithms. The reason is that the linear probing scheme our algorithms use can store more elements in the same amount of space than the chaining scheme used by the other algorithms, which need to store an additional pointer[7]. Only for $K = 2^{18}$, ATOMIC can fit the output just into its shared L3 cache, and is therefore slightly faster than ADAPTIVE, which already needs a partitioning pass. Since the partitioning is so fast though, the difference is only very small.

6.5 Skew Resistence

We now extend the experiments on uniform data to other data sets in order to test the skew resistance of our ADAPTIVE operator. We use the synthetic data generators of Cieslewicz et al. [13], which generate input data for any combination of N and K for a series of distributions with different characteristics (since data cannot have $K = N$ groups *and* be skewed at the same time, K is only approximated). The distributions are namely heavy-hitter, moving-cluster, self-similar, sorted, uniform, and zipf[8]. In short, in heavy-hitter, 50% of all records have the key 1, the others are distributed uniformly between 2 and K. In moving-cluster, the keys are chosen uniformly from a sliding window of size 1024. Self-similar is the Pareto distribution with an 80–20 proportion (also known as 80–20 rule) and zipf is the Zipfian distribution with exponent 0.5.

[5]Note that the time of the second pass was not taken into account in the original paper [13].

[6]We leave the narrow gap to INDEPENDENT as small open problem.
[7]This is also the reason why the cache sizes indicated in Figure 8 are off by factor two for ADAPTIVE.
[8]We omit the distribution sequence because of its similarity to uniform.

Final edited form was published in "SIGMOD/PODS'15: International Conference on Management of Data. Melbourne 2015", S. 1123–1136, ISBN 978-1-4503-2758-9
https://doi.org/10.1145/2723372.2747644

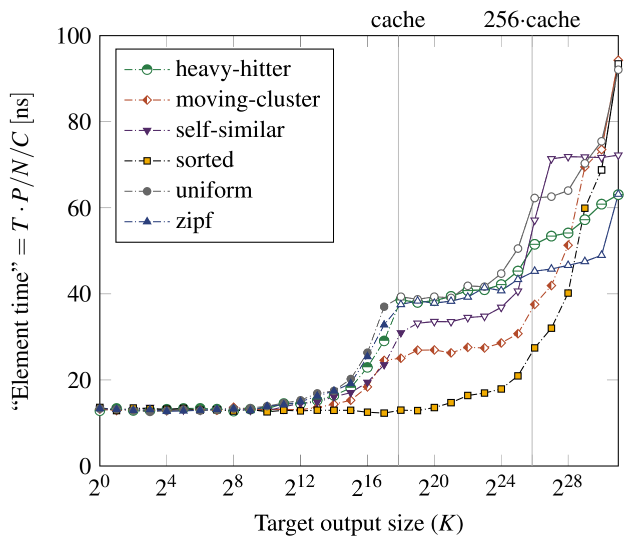

Figure 9: ADAPTIVE on different data sets using $P = 20$ threads.

Figure 9 shows the performance of ADAPTIVE on all data sets. The first and most important observation is that ADAPTIVE is not slower on the other distribution than uniform. In this sense, uniform is the hardest distribution for our operator and skew only improves its performance. Since skew means that some keys occur more often than others, our operator can benefit from skew by using hashing for early aggregation of these values.

To show the mechanics of *how* our algorithm adapts to the skew, we plot those cases of each distribution with solid markers where our algorithm uses only hashing in the first phase, while we plot with hollow markers the cases where it switches to partitioning in the first phase at least once. This way we can see that on the sorted data set, ADAPTIVE switches to partitioning only where $K \geq N/2$. Before that point every values is repeated at least 4 times, so every run of hashing reduces the data by factor $\alpha = 4 \geq \alpha_0$ or more, so our algorithm uses hashing for the next run as well. Since all hash tables are produced in cache and there are only very few of them except for the largest K, the second pass remains negligible and makes the run time only grow slowly with larger K. The behaviour with moving-cluster is very similar, except that the hashing is more costly due to inferior locality. Since self-similar only exhibits relatively mild skew, the same effect is less pronounced on this data set: the reduction factor α is only high enough where there are just more groups that would fit into cache. The run time is consequently very similar to that of uniform data, except that our algorithm switches to partitioning slightly later, namely for $K \geq 2^{19}$. Heavy-hitter even switches to partitioning at the same point as uniform, so for our algorithm this distribution does not have noticeable skew. In particular it does not cause contention since there are no shared data structures in our algorithm. It rather seems like the non-hitter keys are the hard part of this distribution. The zipf data set is so little skewed that it is processed just like uniform data. The fact that it takes less time for the largest K is an artifact intrinsic to the generation of skewed data as discussed above.

As discussed and shown experimentally in the original papers, some of the algorithms of Cieslewicz and Ross [13] and Ye et al. [46] also adapt to skew, but to a lesser degree than ours. All of them are based on hashing and therefore profit from locality. However only HYBRID can adapt to changes in locality as occurring in data sets like sorted or moving-cluster since it maintains a set of "hot" groups similarly to an LRU cache. Furthermore HYBRID can be complemented with ATOMIC, which has the best performance of the algorithms known at the time for larger K, as shown by Cieslewicz and Ross [13]. By using sampling during execution, they can choose the best of the two algorithms, which is quite robust but has considerably higher constants than our ADAPTIVE. All other shown competitors have no mechanism to adapt to changing locality.

Maybe even more importantly, the authors of above algorithms do not give mechanisms to adapt to unknown K and rely on a prediction of the optimizer instead. This could be fixed by growing the data structures on demand, but would be highly non-trivial (if at all possible efficiently) for the shared data structure of ATOMIC, which would decrease the performance of their best algorithm for large K. In contrast, our recursive, run-based algorithm handles any K transparently.

7. RELATED WORK

We now give a brief overview about other work on AGGREGATION. The problem has been studied extensively in the disk-based setting. Many of the algorithms presented in the last decades were built with an analysis in mind that is similar to ours, thus often consisting of recursive processing with some optimization for the case of small output. Examples of early work include a merge-sort with early aggregation by Bitton and DeWitt [9] and a hybrid aggregation by DeWitt et al. [16], which does as much hash aggregation as possible during recursive partitioning of the input. Shatdal and Jeffrey [39] propose to use an algorithm similar to INDEPENDENT for small results and an algorithm similar to PARTITIONALWAYS for large results. They either sample to decide on the algorithm beforehand or switch from hashing to partitioning when the first hash table is full. This is similar to our switching criterion, but somewhat more naive and without mechanism to switch back. Helmer et al. [23] maintain a cache with LRU replacement with "hot" groups for early aggregation and partition the rest recursively similarly to PLAT. Finally Graefe [20] designed a sort-based aggregation algorithm through clever scheduling of disk pages such that it behaves like hash-based aggregation for small cardinalities while keeping its advantage of recursive processing for large cardinalities. The algorithm was later implemented by Albutiu et al. [2, 3]. Other techniques can be found in surveys on the topic [19, 21, 44].

As argued throughout the paper, not all of the techniques from the disk-based world can be applied directly in the in-memory setting because computations are relatively more expensive in the latter scenario. However we think that the basic ideas to reduce data movement are still relevant, so it comes to a surprise that none of the in-memory algorithms we found is recursive. As dissected in detail in Section 6, Cieslewicz and Ross [13] and Ye et al. [46] studied many algorithms in the in-memory setting, but all of them with a fixed number of passes and the consequent short-comings. DB2 BLU [37] and HyPer [28] both implement algorithms similar to PLAT, where aggregation is done in private hash tables if possible and additional data is overflown to partitions for later merging, but both of them do not mention recursive processing, so they are most likely limited as well in the number of groups they can process efficiently.

Finally it is interesting to note that the Reduce phase of MapReduce is very similar to aggregation. It is therefore not surprising that Vernica et al. [40] study cache-efficiency and adaptation mechanisms for the MapReduce framework remotely similar to ours.

8. CONCLUSION

In summary, our work starts with the assumption that even in the in-memory setting, the movement of data is the hard part of relational operators such as aggregation. We use an external memory model to show that HASHAGGREGATION and SORTAGGREGATION are equivalent in terms of the number of cache lines transfers they incur. Consequently we design an algorithmic framework based on sorting by hash value that allows to combine hashing for early aggregation and state-of-the-art integer sorting routines depending on the locality of the data. We tune both the hashing and the sorting routine to modern hardware and devise a simple, yet effective criterion of locality to switch between the two. We show extensive experiments on different data sets and a comparison with several algorithms from prior work. Thanks to the combination of optimal high-level design guided by our theoretical analysis and low-level tuning to modern hardware, we are able to outperform all our competitors by up to factor 3.7. We expect work like ours to become of increasing importance in the near future, since memory bandwidth is developing at a lower rate than processing speed of multi-core CPUs. In particular we believe that other cache-efficient sort algorithms can be augmented with early aggregation similar to what we do with bucket sort, so we invite more work on the duality of hashing and sorting.

Acknowledgments

We would like to thank SAP, the SAP HANA development team, and the students from the SAP HANA Campus for the fruitful work environment and helpful discussions and feedback. We would also like to express our gratitude to Cieslewicz and Ross [13] and Ye et al. [46] for providing the code of their work.

9. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] M.-c. Albutiu. *Scalable Analytical Query Processing*. PhD thesis, 2013.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. In *PVLDB*, volume 5, pages 1064–1075, 2012.

[4] L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. *BRICS*, pages 334–345, 1996.

[5] L. Arge, G. S. l. Brodal, and R. Fagerberg. Cache-Oblivious Data Structures. In *Handbook of Data Structures and Applications*, pages 38/1–38/28. 2005.

[6] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, page 197, 2008.

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

[8] R. Barber, G. Lohman, I. Pandis, G. Attaluri, N. Chainani, S. Lightstone, V. Raman, R. Sidle, and D. Sharpe. Memory-Efficient Hash Joins. In *PVLDB*, pages 353–364, 2015.

[9] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *TODS*, 8(2):255–265, 1983.

[10] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, 1999.

[11] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[12] B. Chandramouli and J. Goldstein. Patience is a Virtue: Revisiting Merge and Sort on Modern Processors. In *SIGMOD*, pages 731–742, 2014.

[13] J. Cieslewicz and K. Ross. Adaptive Aggregation on Chip Multiprocessors. In *PVLDB*, pages 339–350, 2007.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[15] E. D. Demaine. Cache-Oblivious Algorithms and Data Structures. *BRICS*, page 29, 2002.

[16] D. J. DeWitt, R. H. Katz, et al. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, pages 1–8, 1984.

[17] C. Freedman, E. Ismert, and P.-A. k. Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.*, 37(1):22–30, 2014.

[18] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, 1999.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[20] G. Graefe. New algorithms for join and grouping operations. *Computer Science – R&D*, 27(1):3–27, 2011.

[21] G. Graefe, R. Bunker, and S. Cooper. Hash Joins and Hash Teams in Microsoft SQL Server. In *PVLDB*, pages 86–97, 1998.

[22] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *ICDE*, pages 152–159, 1996.

[23] S. Helmer, T. Neumann, and G. Moerkotte. Early Grouping Gets the Skew. Technical report, 2011.

[24] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2009.

[25] D. Jimenez-Gonzalez, J. Navarro, and J.-L. Larriba-Pey. CC-Radix: a cache conscious sorting based on Radix sort. In *PDP*, pages 101–108. IEEE, 2003.

[26] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, et al. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[27] P.-k. Larson. Grouping and duplicate elimination: Benefits of early aggregation. Technical report, 1997.

[28] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism. In *SIGMOD*, pages 743–754, 2014.

[29] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding Up Queries in Column Stores – A Case for Compression. In *DaWaK*, pages 117–129, 2010.

[30] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, 2002.

[31] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *PVLDB*, volume 30, pages 684–695, 2004.

[32] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[33] Y. Matias, E. Segal, and J. S. Vitter. Efficient Bundle Sorting. *SIAM Journal on Computing*, 36(2):394, 2006.

[34] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for

efficient query processing in managed runtimes. In *PVLDB*, pages 1095–1106, 2014.

[35] T. Neumann. Efficiently compiling efficient query plans for modern hardware. volume 4, pages 539–550, 2011.

[36] 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*, pages 755–766, 2014.

[37] V. Raman, G. Attaluri, R. Barber, N. Chainani, et al. DB2 with BLU Acceleration: So Much More than Just a Column Store. In *PVLDB*, page 773, 2013.

[38] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs. In *SIGMOD*, page 351, 2010.

[39] A. Shatdal and J. F. Naughton. Adaptive Parallel Aggregation Algorithms. In *SIGMOD*, pages 104–114, 1995.

[40] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *EDBT*, page 420, 2012.

[41] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[42] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par*, pages 160–169, 2011.

[43] L. Wegner and J. Teuhola. The External Heapsort. *TSE*, 15(7):917–925, 1989.

[44] J. Wen. *Revisiting aggregation techniques for data intensive applications*. PhD thesis, 2013.

[45] Wikipedia. Integer sorting — Wikipedia, the free encyclopedia, 2015. [Online; accessed 22-January-2015].

[46] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable Aggregation on Multicore Processors. In *DaMoN*, pages 1–9, 2011.

APPENDIX

A. TUNING OF ALGORITHM CONSTANTS

In this section we show how we empirically determine the constants of the ADAPTIVE algorithm described in Section 5.

A.1 Switching Threshold α_0

The parameter α_0 balances the performance penalty of HASHING compared to PARTITIONING with its benefit of reducing the work of later passes. In order to determine its value for our system, we run both HASHINGONLY and PARTITIONALWAYS on data sets with varying skew using our parameterized data generators from Section 6.5. For every data set, we observe the value of $\alpha = \frac{n_{in}}{n_{out}}$ in the traces of HASHINGONLY. For unclustered distributions like uniform, the transition from $\alpha = \infty$, where all elements fit into the a single hash table, to the other extreme, $\alpha = 1$, where all keys are distinct, is very sharp and only small values of α occur at all. Almost any value of α_0 works in these cases.

However the three distributions moving-cluster, self-similar, and heavy-hitter can be parameterized to a large range of degrees of spatial locality. In Figure 10, we plot the run times of HASHINGONLY and PARTITIONALWAYS on different data sets with these distributions as function of the observed values of α. As expected, for high values of α and any distribution, HASHINGONLY outperforms PARTITIONALWAYS. In these cases the input of the first pass can be reduced by large factors, so it exhibits enough spatial locality for the HASHING routine to be beneficial. For values of α approaching 1, the order of the routines is inverted: it is better to ignore low locality and use PARTITIONING instead. The desired threshold α_0

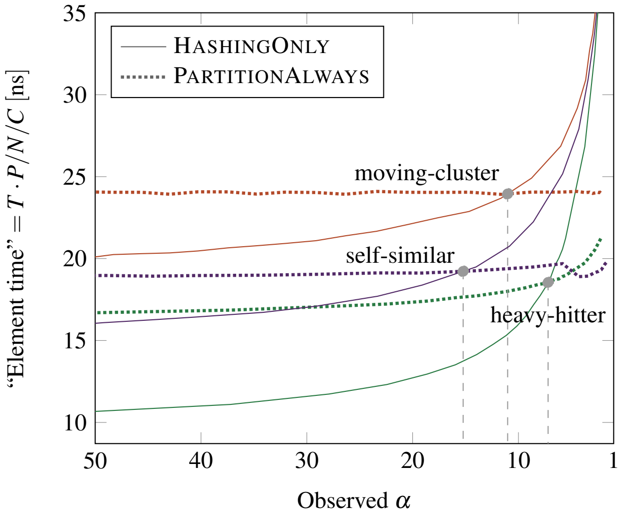

Figure 10: Determining the cross-over of HASHINGONLY and PARTITIONONLY.

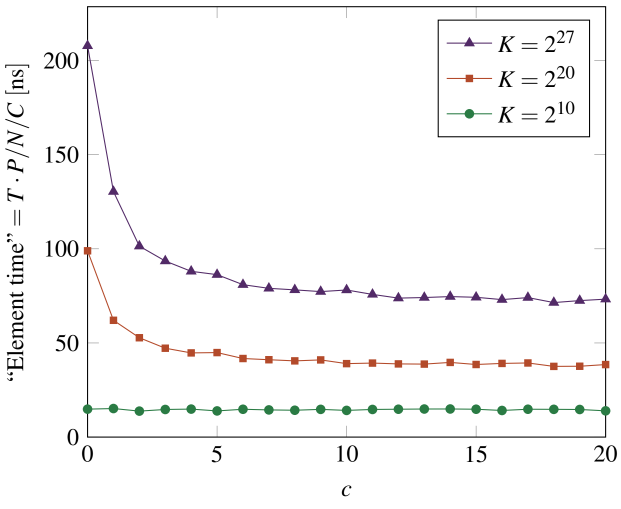

Figure 11: Impact of tuning constant c on the run time of ADAPTIVE.

should separate the first case from the latter. We observe that the respective lines of a particular distribution all intersect in the range of $\alpha \in [7, 16]$ and use the α with the smallest overall error as value for α_0 in our system, which is roughly 11.

A.2 Ratio of HASHING VS PARTITIONING (c)

The constant c in the ADAPTIVE algorithm controls how long the PARTITIONING routine is run before the algorithm switches back to HASHING, which happens after $c \cdot$ cache processed rows. Figure 11 shows the impact of c on the run time of ADAPTIVE for different K and the uniform data set. For $K <$ cache, such as $K = 2^{10}$ in the plot, the algorithm never switches to PARTITIONING in the first place, so c does not have any impact. In the extreme case of $c = 0$, the algorithm degenerates into HASHINGALWAYS, which is quite slow for $K >$ cache (cf. Figure 5). The larger c gets, the more data is processed with the faster PARTITIONING in the non-leaf recursive calls, so the more the performance of ADAPTIVE approaches that of PARTITIONALWAYS. This suggests that $c = \infty$ is the best choice, i.e., never to switch back.

However, smaller c have the benefit to be able to adapt to changing distributions, which are likely to occur after `UNION ALL` operators or as an artifact of reordering of rows for compression purposes [29]. This benefit is hard to quantify because it depends on the database system and typical workloads. Because with growing c the benefit diminishes, a smaller c seems affordable anyway: For $c = 5$, the difference to PARTITIONALWAYS is 17% for $K = 2^{20}$ (19% for $K = 2^{27}$), while it is 5% (11%) for $c = 10$ and still 4% (5%) for $c = 20$.

In summary c allows to choose a trade-off between robustness to changing distributions and maximum throughput. We choose a rather performance oriented value of $c = 10$ for the experiments of this paper, which are all done on data sets of a single distribution, but suggest a slightly lower value for productive systems.